Program Verification

Matthias Heizmann

Summer Term 2021
Program Verification
Summer Term 2021
Lecture 1: Introduction

Matthias Heizmann

19th April
Section 1

Introduction
Outline

- Introduction
- Propositional Logic
- First-Order Logic
- First-Order Theories
- SMT-LIB
- Boogie and Boostan
- Hoare Proof System
- Ultimate Referee
- Arrays
Outline of the Section on Introduction

Some Motivation
Program Verifier
Motivation
Challenges
Content of this Course
Quick answer: No. Program does one iteration too much, array accessed beyond its bounds in the last iteration. Typical bug.

Well-considered answer: Maybe. What is the definition of “correctness”? What is the programming language?
Computers are very good in detecting syntax errors. (Here, Eclipse complains about a missing semicolon). It would be great if tools could also underline bugs that we have seen before.
Is this program correct?

```plaintext
var year, days : int;

procedure main()
modifies year, days;
{
var leapYear : bool;
assume year >= 1980;
assume days >= 0 && days <= 366;
while (days > 365) {
call leapYear := isLeapYear(year);
if (leapYear) {
  if (days > 366) {
    days := days - 366;
    year := year + 1;
  }
} else {
  days := days - 365;
  year := year + 1;
}
}
```

Code similar to the code that caused the bug in Microsoft's Zune player.
Program Verification
Motivation
Challenges
Course outline
Outline of the Section on Introduction

- Some Motivation
- Program Verifier
- Motivation
- Challenges
- Content of this Course
Typical specifications:
- No division by zero
- Array only accessed within its bounds
- Termination
- Memory safety
- No assert statement is violated
According to the international competition on software verification SV-COMP \(^1\) one of the best verification tools.

Mainly developed by our group \(^2\) at the University of Freiburg. Many student projects and theses improved the tool.

Source code available at GitHub \(^3\).
Some program written in the C language. A specification written in ACSL is given by assert statement in line 6.

```c
unsigned int foo(unsigned int x, unsigned int y) {
    if (x < 1000 || y < 1000) {
        return 1000;
    }
    unsigned int z = x + y;
    //@ assert z >= 1000;
    return z;
}
```

- Naive proposition: The program satisfies the specification.
- Naive justification: If we add two large numbers the result is a large number.

Ultimate Automizer rightly disagrees: Since the type of \( x \) and \( y \) is `unsigned int` the value of \( z \) is 0 if \( y \) was 4294966296 and \( z \) was 0. (If we follow the ISO C11 standard and assume that an `unsigned int` can store values from 0 to 4294967295.)
Let’s try to fix the program as follows.

```c
unsigned long foo(unsigned int x, unsigned int y) {
    if (x < 1000 || y < 1000) {
        return 1000;
    }
    unsigned long z = x + y;
    //@ assert z >= 1000;
    return z;
}
```

▶ Naive proposition: Now, the program satisfies the specification.

▶ Naive justification: The range of values of z is now large enough.

Ultimate Automizer rightly disagrees: The type of the expression \( x+y \) is still unsigned int and hence the preceding counterexample applies again.
Let's finally fix the program as follows.

```c
unsigned long foo(unsigned int x, unsigned int y) {
    if (x < 1000 || y < 1000) {
        return 1000;
    }
    unsigned long z = (long) x + y;
    //@ assert z >= 1000;
    return z;
}
```

Ultimate Automizer confirms that the program satisfies its specification.
Outline of the Section on Introduction

Some Motivation
Program Verifier
Motivation
Challenges
Content of this Course
Motivation

- More and more devices in our lives are controlled by software.
- Software usually has bugs.
- Some bugs make the software directly dysfunctional, some bugs affect security and are exploited secretly.
- Software is getting more and more complex.
- Higher complexity, more bugs.
Testing is not Always Sufficient

Unless we test all inputs, we cannot use testing to prove correctness.

```c
int foo(int x, int y) {
    return y / (myHash(x) - 23);
}
```
Motivation

- Find more software bugs
- Get mathematical proof of correctness
- Speed up software development
Outline of the Section on Introduction

Some Motivation
Program Verifier
Motivation
Challenges
Content of this Course
Challenge 1: Undecidability

The program verification problem is undecidable.

Do not try to develop algorithms that solve the problem for all programs. Algorithms that solve the problem for some programs are also helpful.
Challenge 2: Ambiguities

Example: What are the values of \( x \) and \( y \)?

\[
\begin{align*}
x & := -7 \div 5; \\
y & := -7 \mod 5;
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>( x )</th>
<th>( y )</th>
<th>makes sense because</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>-1</td>
<td>-2</td>
<td>((-1) \cdot 5 + (-2) = 7)</td>
</tr>
<tr>
<td>Python</td>
<td>-2</td>
<td>3</td>
<td>((-2) \cdot 5 + 3 = 7)</td>
</tr>
<tr>
<td>Javascript</td>
<td>-1.4</td>
<td>-2</td>
<td>((-1.4) \cdot 5 = 7)</td>
</tr>
</tbody>
</table>

Use mathematical logic to give programming languages a precise semantics.
In this course: We develop a small programming language whose semantics can be defined in a couple of slides.
Challenge 3: Correctness Proofs are Hard to Find

```c
int main(void) {
    unsigned int x = 1;
    unsigned int y = 1;
    while (1) {
        if (user_input()) {
            x = 3 * x;
            y = -2 * y + 1;
        } else {
            unsigned int tmp = x;
            x = y;
            y = tmp;
        }
        //@ assert y != 4;
    }
    return 0;
}
```

We cannot track all executions.

Simple argument for correctness:
The values of x and y are always odd.
Outline of the Section on Introduction

- Some Motivation
- Program Verifier
- Motivation
- Challenges
- Content of this Course
Content of this Course

- Mathematical logic
  Propositional logic, First-order logic, SMT-LIB

- Boostan
  A small programming with a precisely defined semantics

- Hoare proof system
  A proof system for programs

- Algorithms for program verification
  Develop algorithms that analyze if a program satisfies a specification
Program Verification

Summer Term 2021

Lecture 2: Propositional Logic

Matthias Heizmann

Wednesday 21st April
Section 2

Propositional Logic
We presume that all of you know propositional logic. Propositional logic is one of the basic concepts in computer science, it has applications in many areas but there exist several terminology and several notations.

Goals of this section are

- recall the basic ideas of propositional logic
- fix the notation and terminology that we use in this lecture
- ease the presentation of first order logic (next section)
- introduce the idea of a proof system
Syntax of Propositional Logic

Definition

Let $\mathcal{V}_{PL}$ be a nonempty set whose elements we call *propositional logical variables*. We define *propositional logic (PL) formulas* inductively as follows.

1. **false** is a PL formula.
2. For each $X \in \mathcal{V}_{PL}$, $X$ is a PL formula.
3. If $F$ is a PL formula, then $\neg F$ is a PL formula.
4. If $F_1$ and $F_2$ are PL formulas, then $(F_1 \land F_2)$ is a PL formula.

Abbreviations

\[
\begin{align*}
\text{true} & := \neg \text{false} \\
F_1 \lor F_2 & := \neg(\neg F_1 \land \neg F_2) \\
F_1 \rightarrow F_2 & := (\neg F_1 \lor F_2) \\
F_1 \leftrightarrow F_2 & := (F_1 \rightarrow F_2) \land (F_2 \rightarrow F_1)
\end{align*}
\]
Terminology

We call **true**, **false** atoms. 
If $X \in \mathcal{V}_{PL}$, we call $X$ an atom. 
If $F$ is an atom, we call $F$ and $\neg F$ a literal. 
We call the symbols $\neg$, $\land$, $\lor$, $\rightarrow$, $\leftrightarrow$ logical connectives.

Notation

We may omit parentheses.

- Use the following order of precedence for logical connectives: $\neg$, $\land$, $\lor$, $\rightarrow$, $\leftrightarrow$
- Use the convention that binary operators are right-associative.

Right-associativity means e.g. that $F_1 \rightarrow F_2 \rightarrow F_3$ is $F_1 \rightarrow (F_2 \rightarrow F_3)$.
Semantics

We call true and false truth values and we call a mapping $\rho : \mathcal{V}_{PL} \rightarrow \{\text{true}, \text{false}\}$ a variable assignment.

Definition

The evaluation is a mapping $[\cdot]$ that takes a PL formula $F$ and a variable assignment $\rho$, and returns a truth value. It is defined as follows.

1. $[\text{false}]_{\rho}$ is false.
2. For each $X \in \mathcal{V}_{PL}$, $[X]_{\rho}$ is $\rho(X)$.
3. $[\neg F]_{\rho}$ is \begin{cases} true & \text{if } [F]_{\rho} \text{ is false} \\ false & \text{if } [F]_{\rho} \text{ is true.} \end{cases}$
4. $[F_1 \land F_2]_{\rho}$ is \begin{cases} true & \text{if } [F_1]_{\rho} \text{ is true and } [F_2]_{\rho} \text{ is true} \\ false & \text{otherwise.} \end{cases}$

Definition

1. We call a PL formula $F$ satisfiable if there is a variable assignment $\rho$ such that $[F]_{\rho}$ is true.
2. We call a PL formula $F$ valid if for all variable assignments $\rho$ the evaluation $[F]_{\rho}$ is true.
Examples: Satisfiability and Validity

Which of the following formulas is satisfiable, which is valid?

- $F_1 : P \land Q$
  - satisfiable, not valid
- $F_2 : \neg(P \land Q)$
  - satisfiable, not valid
- $F_3 : P \lor \neg P$
  - satisfiable, valid
- $F_4 : \neg(P \lor \neg P)$
  - unsatisfiable, not valid
- $F_5 : (P \rightarrow Q) \land (P \lor Q) \land \neg Q$
  - unsatisfiable, not valid (see next slides)

Is there a formula that is unsatisfiable and valid?
Truth tables

Functions that have finite domain are sometimes visualized or defined via a table. The truth table is a table that visualizes the evaluation mapping $[\cdot]$ for a given formula $F$, i.e., the input is a variable assignment, the output is a truth value. In the truth table (an example is depicted on the next slides), every column is assigned to some subformula of $F$. The columns are partitioned into two parts. On the left hand side, there is one column for each propositional variable, on the right hand side there is a column for $F$ and sometimes there are also columns for subformulas of $F$. A row of the table represents one variable assignment. The rows for subformulas can help to compute the entries for the formula $F$. 
Truth Table: Example

Truth table for the formula $F_5 : (P \rightarrow Q) \land (P \lor Q) \land \neg Q$

<table>
<thead>
<tr>
<th>$P$</th>
<th>$Q$</th>
<th>$(P \rightarrow Q)$</th>
<th>$(P \lor Q)$</th>
<th>$\neg Q$</th>
<th>$F_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

We conclude that $F_5$ is neither satisfiable nor valid.
Truth Tables: Limited Applicability
Unfortunately, the applicability of truth tables is rather limited. A truth table has one row per variable assignment, and there are $2^n$ variable assignments for $n$ variables.
Deciding satisfiability of a PL formula is an NP-complete problem. However, there are many algorithms that work well in practice and that are known to be polynomial on relevant subclasses of PL formulas. Some of these algorithms are discussed in other lectures, e.g., Decision Procedures, and we do not want to discuss the problem in this lecture.
Excursus: Using SMT Solvers for SAT Problem

<table>
<thead>
<tr>
<th>SAT solver</th>
<th>propositional logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMT solver</td>
<td>first order logic modulo theories</td>
</tr>
</tbody>
</table>

Example:

\[(P \rightarrow Q) \land (P \lor Q)\]
Tools for checking satisfiability of PL formulas
Finding satisfying assignments for PL formulas can be a time-consuming task. In practice, we use tools to solve this task. Tools that are specialized in finding satisfying assignments for PL formulas are called *SAT solvers*.

Later in this lecture, we will use tools that are called *SMT solvers*. Every SMT solver is also able to find satisfying assignments for PL formulas, but SMT solvers are typically not highly optimized for this task. Since performance is not an issue for us, we will not learn how to use a SAT solver and start to use SMT solvers right now.

Users communicate with an SMT solver via so-called *SMT scripts*. An SMT script is a file that contains a list of commands. In order to get a satisfying assignment for a PL formula $F$, we need only the following four commands.

1. First, we write `(define-fun X Bool)` for each propositional variable $X$ in our formula $F$.
2. Then, we write `(assert $F$)` and have to write the formula $F$ using the prefix (or Polish) notation that is defined at the following URL.
   
   [http://smtlib.cs.uiowa.edu/theories-Core.shtml](http://smtlib.cs.uiowa.edu/theories-Core.shtml)
   
   E.g., for PL formulas $F_1, F_2$ we write `(and $F_1$ $F_2$)` instead of $(F_1 \land F_2)$
3. Next, we write `(check-sat)`.
4. Finally, if the formula is satisfiable and we want to see a satisfying assignment, we can write `(get-model)`.

There are several SMT solvers available, we propose to use Z3 because it is also available via a web interface. [https://rise4fun.com/z3/](https://rise4fun.com/z3/)
Implications

Definition

Given a set of PL formulas $\Gamma := \{F_1, \ldots, F_n\}$ and a PL formula $F'$, we say that $\Gamma$ *implies* $F'$ if for all variable assignments $\rho$ we have that if $[F_i]_\rho = \text{true}$ holds for all $i \in \{1, \ldots, n\}$ then also $[F']_\rho = \text{true}$ holds. We use $\vdash$ to denote this binary implication relation and we say that the implication $\Gamma \vdash F'$ holds if $\Gamma$ implies $F'$.

Example

\[ \{A, A \to B\} \vdash A \land B \quad \{A \to B\} \vdash \neg B \to \neg A \]

How can we prove that $\{F_1, \ldots, F_n\}$ implies $F'$?

1. Truth table. (Not doable if number of variables is high)
2. Prove that the PL formula $F_1 \land \ldots \land F_n \to F'$ is valid. (Requires algorithm for checking validity)
3. Prove that the PL formula $\neg (F_1 \land \ldots \land F_n \to F')$ is not satisfiable. (Theorem on next slide – requires algorithm for checking satisfiability – implemented in SMT solvers)
4. Use a proof system (next subchapter)
Satisfiability and Validity

**Theorem**

The PL formula $F$ is valid iff the PL formula $\neg F$ is not satisfiable.

**Proof.**

- $F$ valid
  - iff for all variable assignments $\rho$ we have $\llbracket F \rrbracket_\rho = \text{true}$
    (def of validity)
  - iff for all variable assignments $\rho$ we have $\llbracket \neg F \rrbracket_\rho = \text{false}$
    (def of negation $\neg$)
  - iff there is no variable assignment $\rho$ such that $\llbracket \neg F \rrbracket_\rho = \text{true}$
  - iff $\neg F$ not satisfiable
    (def of satisfiability)
Equivalence

**Definition**

We call two PL formulas $F_1$ and $F_2$ *equivalent*, denoted $F_1 \equiv F_2$, if they evaluate to the same truth value under every variable assignment.

**Note**

$$F_1 \equiv F_2 \quad \text{iff} \quad \{F_1\} \models F_2 \quad \text{and} \quad \{F_2\} \models F_1$$
Proof System (Informally)

- template for giving a proof
- reasoning according to a fixed number of rules
- prove once that every rule is “correct”
- find a proof \(\leadsto\) find a sequence of rules
Proof system $\mathcal{N}_{PL}$

- Proof system for implications between PL formulas.
- Proof rules of $\mathcal{N}_{PL}$ are $(n + 1)$-ary relations over implications denoted as follows:

\[
\frac{\Gamma_1 \models F_1 \quad \ldots \quad \Gamma_n \models F_n}{\Gamma_{n+1} \models F_{n+1}}
\]

Idea: the rule represents a step in a proof with the following meaning. If $\Gamma_i$ implies $F_i$ for $i \in \{1, \ldots n\}$ then $\Gamma_{n+1}$ implies $F_{n+1}$. 
Proof rules of $\mathcal{N}_{PL}$

\[ (Ax) \quad \frac{\Gamma \cup \{F\} \models F}{\Gamma \models F} \quad \text{(RAA)} \quad \frac{\Gamma \cup \{\neg F\} \models \text{false}}{\Gamma \models F} \]

Introduction rules:

\[ (I_\land) \quad \frac{\Gamma \models F_1 \quad \Gamma \models F_2}{\Gamma \models F_1 \land F_2} \quad \text{(I_\lor)} \quad \frac{\Gamma \models F_i \quad i \in \{1, 2\}}{\Gamma \models F_1 \lor F_2} \]

\[ (I_\rightarrow) \quad \frac{\Gamma \cup \{F_1\} \models F_2}{\Gamma \models F_1 \rightarrow F_2} \quad \text{(I_\neg)} \quad \frac{\Gamma \cup \{F\} \models \text{false}}{\Gamma \models \neg F} \]

Elimination rules:

\[ (E_\land) \quad \frac{\Gamma \models F_1 \land F_2 \quad i \in \{1, 2\}}{\Gamma \models F_i} \quad \text{(E_\lor)} \quad \frac{\Gamma \models F_1 \lor F_2 \quad \Gamma \cup \{F_1\} \models F_3 \quad \Gamma \cup \{F_2\} \models F_3}{\Gamma \models F_3} \]

\[ (E_\rightarrow) \quad \frac{\Gamma \models F_1 \quad \Gamma \models F_1 \rightarrow F_2}{\Gamma \models F_2} \quad \text{(E_\neg)} \quad \frac{\Gamma \models F_1 \quad \Gamma \models \neg F_1}{\Gamma \models F_2} \]

The letters $F, F_1, F_2, F_3$ denote PL formulas.
Proof system $\mathcal{N}_{\text{PL}}$

**Definition**

A *derivation* is a tree whose nodes are labelled by implications such that the following holds. If a node labelled by implication $\Gamma_{n+1} \models F_{n+1}$ has children that are labelled by implications $\Gamma_1 \models F_1 \ldots \Gamma_n \models F_n$ then

$$\Gamma_1 \models F_1 \ldots \Gamma_n \models F_n$$

$$\Gamma_{n+1} \models F_{n+1}$$

must be an instance of some rule.

**Example**

Let $A, B$ be PL variable, define $\Gamma := \{A, A \rightarrow B\}$

$$\Gamma \models A \quad \Gamma \models A \rightarrow B$$

$$\Gamma \models A \quad \Gamma \models B$$

$$(\text{Ax}) \quad \Gamma \models A \quad \Gamma \models A \rightarrow B$$

$$(I_{\land}) \quad \Gamma \models A \land B$$

$$(I_{\rightarrow}) \quad \Gamma \models A \quad \Gamma \models B$$

$$(\text{Ax}) \quad \Gamma \models A \land B$$
For derivations: We do not use the typical graph representation of a tree (left, striked out). Instead, we use horizontal lines together with the names of proof rules (right).

We conclude from the preceding definition that a leaf of the derivation can only be labelled by a implication $\Gamma \vdash F$ such that $\Gamma \vdash F$ is an instance of some (unary) rule.
By now, we saw several definitions (proof rules of $\mathcal{N}_{\text{PL}}$, derivation) but we may still wonder whether $\mathcal{N}_{\text{PL}}$ is good for something. The following theorem shows us an application: $\mathcal{N}_{\text{PL}}$ can be used to prove implications. Whenever we want to prove an implication, we can find a derivation and conclude that the implication holds.

A consequence: If we have to implement a tool for proving implications, we can solve the task by developing an algorithm for finding derivations.\footnote{Please note however that we introduce $\mathcal{N}_{\text{PL}}$ mainly to get familiar with proof systems. State-of-the-art SAT solver implement completely different algorithms \texttt{jsat/HeuleJS19}}
Proof system $\mathcal{N}_{PL}$

**Theorem (Soundness of $\mathcal{N}_{PL}$)**

If a node in a derivation is labelled by $\Gamma \models F_{n+1}$, then the implication $\Gamma \models F_{n+1}$ holds.

**Proof.**

(Sketch) Show for each rule that the implication below the line holds if all implications above the line hold. Use induction to conclude that the theorem holds.

**Theorem (Completeness of $\mathcal{N}_{PL}$)**

If the implication $\Gamma \models F_{n+1}$ holds then there exists some derivation in which the root is labelled by $\Gamma \models F_{n+1}$,

Proof difficult, not in the scope of this lecture.
Let’s prove the implication \( \{A, A \rightarrow B\} \models A \land B \)

\[ \vdash \Gamma \]

Introduction rules:

\[
\begin{align*}
(Ax) & \quad \Gamma \cup \{F\} \models F \\
(I_\land) & \quad \Gamma \models F_1 \quad \Gamma \models F_2 \quad \Gamma \models F_1 \land F_2 \\
(I_\lor i) & \quad \Gamma \models F_i \quad i \in \{1, 2\} \\
(I_{\neg}) & \quad \Gamma \models F \quad \Gamma \models \neg F \\

(\text{RAA}) & \quad \Gamma \cup \{\neg F\} \models \text{false} \\
\end{align*}
\]

Elimination rules:

\[
\begin{align*}
(E_\land i) & \quad \Gamma \models F_1 \land F_2 \quad i \in \{1, 2\} \\
(E_\lor) & \quad \Gamma \models F_1 \quad \Gamma \models F_1 \rightarrow F_2 \\
(E_{\neg}) & \quad \Gamma \models F_1 \quad \Gamma \models \neg F_1 \\
\end{align*}
\]

\[
\begin{align*}
(Ax) & \quad \Gamma \models A \\
(Ax) & \quad \Gamma \models A \rightarrow B \\
(Ax) & \quad \Gamma \models A \land B \\
\end{align*}
\]
A guide for proving implications.

1. Goal: Try to construct a derivation whose root node is labelled by the implication that we want to prove.

2. Start building the tree at the root (bottom).

3. For each node in the tree, use the rules to determine the number of children and their labels, because we must never violate the definition of a derivation.

4. Use the Rule $Ax$ as soon as possible because it allow us to construct a leaf of the tree without violating the definition of a derivation.

In the example form the preceding slide we start with a root node that is labelled by $\Gamma \models A \land B$. Since the right hand side of this implication is a conjunction we cannot use the rules $I_\lor 1, I_\lor 1, I_\to, I_\neg$ to construct the children of our root node. Since $\Gamma$ does not contain $A \land B$ we cannot use the rule $Ax$. Without looking a few steps ahead, we cannot exclude any other rule and have to try all of them. Luckily, this example is rather simple and by looking a few steps ahead we see that the rule $I_\land$ is a good choice.
We note that it is not allowed to replace formulas by equivalent formulas in a derivation. E.g., the following tree is **not a derivation** (according to the definition), because it is not allowed to swap the operands of the logical connective $\land$.

\[
\begin{array}{c}
(Ax) \\
(I_\land)
\end{array}
\quad
\begin{array}{c}
(Ax) \\
(E\to)
\end{array}
\quad
\begin{array}{c}
(Ax) \\
\Gamma \vdash A
\end{array}
\quad
\begin{array}{c}
(Ax) \\
\Gamma \vdash A \to B
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash B
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash B \land A
\end{array}
\]

Rationale: In a proof system, it should be possible to find (and check) derivations by mechanically applying rules without a need for understanding the semantics of formulas. We can e.g., implement a proof system on a computer without teaching the computer to understand the semantics of formulas.
Program Verification

Summer Term 2021

Lecture 3: First-Order Logic

Matthias Heizmann

Monday 26th April
Section 3

First-Order Logic
Like propositional logic, first-order logic (also known as predicate logic) is a basic concept in computer science that has applications in many areas, but there exist several terminology and several notations.

Goals of this section are

- recall the basic ideas of first-order logic
- fix the notation and terminology that we use in this lecture
- get more familiar with proof systems / see proof rules with side-conditions
- learn to formalize statements in first-order logic
Before we introduce first-order logic formally, we will have a look at three examples.

On the next two slides you will see three “famous” theorems and a formalization in first-order logic.
Famous Theorems in FOL

- The length of one side of a triangle is less than the sum of the lengths of the other two sides.

\[ \forall x, y, z. \ triangle(x, y, z) \rightarrow length(x) < length(y) + length(z) \]

- Fermat’s Last Theorem.

\[ \forall n. \ integer(n) \land n > 2 \rightarrow \forall x, y, z. \]
\[ \quad \land \ integer(x) \land integer(y) \land integer(z) \]
\[ \quad \land x > 0 \land y > 0 \land z > 0 \]
\[ \quad \rightarrow x^n + y^n \neq z^n \]
Famous Theorems in FOL

For every regular Language $L$ there is some $n \geq 0$, such that for all words $z \in L$ with $|z| \geq n$ there is a decomposition $z = uvw$ with $|v| \geq 1$ and $|uv| \leq n$, such that for all $i \geq 0$: $uv^iw \in L$.

\[ \forall L. \text{regularlanguage}(L) \rightarrow \exists n. \text{integer}(n) \land n \geq 0 \land \forall z. z \in L \land |z| \geq n \rightarrow \exists u, v, w. \text{word}(u) \land \text{word}(v) \land \text{word}(w) \land z = uvw \land |v| \geq 1 \land |uv| \leq n \land \forall i. \text{integer}(i) \land i \geq 0 \rightarrow uv^iw \in L \]

Predicate symbols: regularlanguage, integer, word, $\cdot \in \cdot$, $\cdot \leq \cdot$, $\cdot \geq \cdot$, $\cdot = \cdot$

Constant symbols: 0, 1

Function symbols: $|\cdot|$ (word length), concatenation, iteration
Syntax of First-order Logic

Definition
Let a vocabulary $\mathcal{V}$ be a tuple $(\mathcal{V}_{\text{Var}}, \mathcal{V}_{\text{Const}}, \mathcal{V}_{\text{Fun}}, \mathcal{V}_{\text{Pred}})$ such that

- $\mathcal{V}_{\text{Var}}$ is a countable set whose elements we call *variables*.
- $\mathcal{V}_{\text{Const}}$ is a countable set whose elements we call *constant symbols*.
- $\mathcal{V}_{\text{Fun}}$ is a countable set whose elements we call *function symbols*. Each function symbol $f$ has a natural number $\geq 1$ that we call the *arity* of $f$.
- $\mathcal{V}_{\text{Pred}}$ is a countable set whose elements we call *predicate symbols*. Each predicate symbol $p$ has a natural number $\geq 0$ that we call the *arity* of $p$.

For the following definitions, we fix a vocabulary $\mathcal{V} = (\mathcal{V}_{\text{Var}}, \mathcal{V}_{\text{Const}}, \mathcal{V}_{\text{Fun}}, \mathcal{V}_{\text{Pred}})$.

Definition
We define *first-order logic (FOL) terms* inductively as follows.

1. For each $x \in \mathcal{V}_{\text{Var}}$, $x$ is a *term*.
2. For each $c \in \mathcal{V}_{\text{Const}}$, $c$ is a *term*.
3. If $t_1, \ldots, t_n$ are terms, $f \in \mathcal{V}_{\text{Fun}}$, $f$ has arity $n$, then $f(t_1, \ldots, t_n)$ is a *term*.
Syntax of First-order Logic

Definition

We define *first-order logic (FOL) formulas* inductively as follows.

1. **false** is a formula.
2. If \( t_1, \ldots, t_n \) are terms, \( p \in \mathcal{V}_{\text{Pred}} \), \( p \) has arity \( n \), then \( p(t_1, \ldots, t_n) \) is a formula.
3. If \( \varphi \) is a formula, then \( \neg \varphi \) is a formula.
4. If \( \varphi_1 \) and \( \varphi_2 \) are formulas, then \( (\varphi_1 \land \varphi_2) \) is a formula.
5. If \( \varphi \) is a formula and \( x \in \mathcal{V}_{\text{Var}} \) then \( \exists x. \varphi \) is a formula.

Abbreviations, Terminology, and Notation

- Analogously to propositional logic we use the abbreviations \( \lor, \rightarrow, \leftrightarrow \).
- Additionally, we introduce \( \forall x. \varphi := \neg \exists x. \neg \varphi \)
- We call the symbols \( \exists \) and \( \forall \) quantifiers. We call formulas of the form \( \text{true}, \text{false} \), and \( p(t_1, \ldots, t_n) \) *atoms*.
- Analogously to propositional logic we may omit parentheses. The precedence of quantifiers is lower than the precedence of logical connectives. We may abbreviate \( \exists x_1. \exists x_2. \varphi \) to \( \exists x_1, x_2. \varphi \).
Definition

A *model* $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ is a pair where $\mathcal{D}$ is a set that we call *interpretation domain* and $\mathcal{I}$ is a function that we call *interpretation function* and that has the following properties.

- The domain of $\mathcal{I}$ is $\mathcal{V}_{\text{Const}} \cup \mathcal{V}_{\text{Fun}} \cup \mathcal{V}_{\text{Pred}}$.
- $\mathcal{I}$ maps every constant symbol to an element of $\mathcal{D}$.
- $\mathcal{I}$ maps every $n$-ary function symbol to an $n$-ary function whose domain is $\mathcal{D}^n$ and whose range is $\mathcal{D}$.
- $\mathcal{I}$ maps every $n$-ary predicate symbol to an $n$-ary relation over $\mathcal{D}$.

We call a function $\rho : \mathcal{V}_{\text{Var}} \to \mathcal{D}$ that maps variable symbols to elements of the interpretation domain a *variable assignment*.

Notation

Let $f : X \to Y$ be a function whose domain is some set $X$ and whose range is some set $Y$. Let $\tilde{x} \in X$ and $\tilde{y} \in Y$, then we use $f \triangleleft \{ \tilde{x} \to \tilde{y} \}$ to denote the function that maps all $x \in X \setminus \{\tilde{x}\}$ to $f(x)$ and that maps $\tilde{x}$ to $\tilde{y}$.
Definition

The *evaluation of terms* is a mapping $\llbracket \cdot \rrbracket_{\mathcal{M},\rho}$ that takes a formula $\varphi$, a model $\mathcal{M} = (\mathcal{D},\mathcal{I})$, and a variable assignment $\rho$, and returns an element of $\mathcal{D}$. It is inductively defined as follows.

1. For each $x \in \mathcal{V}_{\text{Var}}$, $\llbracket x \rrbracket_{\mathcal{M},\rho}$ is $\rho(x)$.
2. For each $c \in \mathcal{V}_{\text{Const}}$, $\llbracket c \rrbracket_{\mathcal{M},\rho}$ is $\mathcal{I}(c)$.
3. If $t_1, \ldots, t_n$ are terms, $f \in \mathcal{V}_{\text{Fun}}$, $f$ has arity $n$, then $\llbracket f(t_1, \ldots, t_n) \rrbracket_{\mathcal{M},\rho}$ is $\mathcal{I}(f)(\llbracket t_1 \rrbracket_{\mathcal{M},\rho}, \ldots, \llbracket t_n \rrbracket_{\mathcal{M},\rho})$. 
Semantics of First-order Logic

Definition

The \textit{evaluation of formulas} is a mapping $\llbracket \cdot \rrbracket_{\mathcal{M}, \rho}$ that takes a formula $\varphi$, a model $\mathcal{M} = (\mathcal{D}, \mathcal{I})$, and a variable assignment $\rho$, and returns a truth value. It is inductively defined as follows.

1. $\llbracket \text{false} \rrbracket_{\mathcal{M}, \rho}$ is \textbf{false}.

2. $\llbracket p(t_1, \ldots, t_n) \rrbracket$ is
   \[
   \begin{cases} 
   \text{true} & \text{if } (\llbracket t_1 \rrbracket_{\mathcal{M}, \rho}, \ldots, \llbracket t_n \rrbracket_{\mathcal{M}, \rho}) \in \mathcal{I}(p) \\
   \text{false} & \text{otherwise.}
   \end{cases}
   \]

3. $\llbracket \neg \varphi \rrbracket_{\mathcal{M}, \rho}$ is
   \[
   \begin{cases} 
   \text{true} & \text{if } \llbracket \varphi \rrbracket_{\mathcal{M}, \rho} \text{ is false} \\
   \text{false} & \text{if } \llbracket \varphi \rrbracket_{\mathcal{M}, \rho} \text{ is true.}
   \end{cases}
   \]

4. $\llbracket \varphi_1 \land \varphi_2 \rrbracket_{\mathcal{M}, \rho}$ is
   \[
   \begin{cases} 
   \text{true} & \text{if } \llbracket \varphi_1 \rrbracket_{\mathcal{M}, \rho} \text{ is true and } \llbracket \varphi_2 \rrbracket_{\mathcal{M}, \rho} \text{ is true} \\
   \text{false} & \text{otherwise.}
   \end{cases}
   \]

5. $\llbracket \exists x. \varphi \rrbracket_{\mathcal{M}, \rho}$ is
   \[
   \begin{cases} 
   \text{true} & \text{if there exists } v \in \mathcal{D} \text{ such that } \llbracket \varphi \rrbracket_{\mathcal{M}, \rho \leftarrow \{x \mapsto v\}} \text{ is true} \\
   \text{false} & \text{otherwise.}
   \end{cases}
   \]
Satisfiability and Validity

Definition (Satisfiability)

We call a formula $\varphi$ **satisfiable** if there exists a model $\mathcal{M}$ and a variable assignment $\rho$ such that $[\varphi]_{\mathcal{M},\rho}$ is true.

Definition (Validity)

We call a formula $\varphi$ **valid** if $[\varphi]_{\mathcal{M},\rho}$ is true for all models $\mathcal{M}$ and for all variable assignments $\rho$.

Note

$\varphi$ is valid iff $\neg \varphi$ is unsatisfiable
Implications

Definition

Given a (possibly infinite) set of FOL formulas $\Gamma$ and a FOL formula $\psi$, we say that $\Gamma$ implies $\psi$ if for all models $M$ and for all variable assignments $\rho$ we have that if $[\varphi]_{M,\rho} = \text{true}$ holds for all $\varphi \in \Gamma$ then also $[\psi]_{M,\rho} = \text{true}$ holds. We use $\vdash$ to denote this binary implication relation and we say that the implication $\Gamma \vdash \psi$ holds if $\Gamma$ implies $\psi$. 
Definition (Free Variables, Bound Variables, Closed Formulas)

Given a FOL term $t$, we define the set of free variables inductively as follows.

$$\text{freevars}(t) = \begin{cases} \{x\} & \text{if } t \text{ is } x \in \mathcal{V}_{\text{Var}} \\ \emptyset & \text{if } t \text{ is } c \in \mathcal{V}_{\text{Const}} \\ \text{freevars}(t_1) \cup \ldots \cup \text{freevars}(t_n) & \text{if } t \text{ is } f(t_1, \ldots, t_n) \end{cases}$$

Given a FOL formula $\psi$, we define the set of free variables inductively as follows.

$$\text{freevars}(\psi) = \begin{cases} \emptyset & \text{if } \psi \text{ is false} \\ \text{freevars}(t_1) \cup \ldots \cup \text{freevars}(t_n) & \text{if } \psi \text{ is } p(t_1, \ldots, t_n) \\ \text{freevars}(\varphi) & \text{if } \psi \text{ is } \neg \varphi \\ \text{freevars}(\varphi_1) \cup \text{freevars}(\varphi_2) & \text{if } \psi \text{ is } \varphi_1 \land \varphi_2 \\ \text{freevars}(\varphi) \setminus \{x\} & \text{if } \psi \text{ is } \exists x. \varphi \end{cases}$$

We call a variable that occurs in $\psi$ but is not free bound.

We call a formula that does not contain free variables closed.

Note: For a closed formula $\varphi$ the evaluation $[\varphi]_{\mathcal{M}, \rho}$ is independent of the variable assignment $\rho$. 
**Notation**

- Given a function \( f \), we use \( \text{dom}(f) \) to denote the domain of \( f \).
- Given a function \( f \) that maps variables to terms, we use \( \text{vars}(f) \) to denote the set that contains \( \text{dom}(f) \) and all variables of all terms in the range of \( f \). I.e., \( \text{vars}(f) = \text{dom}(f) \cup \bigcup_{x \in \text{dom}(f)} \text{freevars}(f(x)) \).

**Definition (Substitution)**

Given a function \( \sigma \) from variable symbols to terms we define the substitution for FOL terms \( t \) and FOL formulas \( \psi \) as follows.

\[
\begin{align*}
t\sigma &= \begin{cases} 
\sigma(x) & \text{if } t \text{ is } x \in \mathcal{V}_{\text{Var}} \text{ and } x \in \text{dom}(\sigma) \\
t & \text{if } t \text{ is } c \in \mathcal{V}_{\text{Const}} \text{ or if } t \text{ is } x \in \mathcal{V}_{\text{Var}} \text{ and } x \notin \text{dom}(\sigma) \\
f(t_1\sigma, \ldots, t_n\sigma) & \text{if } t \text{ is } f(t_1, \ldots, t_n)
\end{cases} \\
\psi\sigma &= \begin{cases} 
\text{false} & \text{if } \psi \text{ is false} \\
p(t_1\sigma, \ldots, t_n\sigma) & \text{if } \psi \text{ is } p(t_1, \ldots, t_n) \\
\neg(\varphi\sigma) & \text{if } \psi \text{ is } \neg\varphi \\
\varphi_1\sigma \land \varphi_2\sigma & \text{if } \psi \text{ is } \varphi_1 \land \varphi_2 \\
\exists x.\varphi\sigma & \text{if } \psi \text{ is } \exists x.\varphi \text{ and } x \notin \text{vars}(\sigma) \\
\exists x'.(\varphi\sigma')\sigma & \text{if } \psi \text{ is } \exists x.\varphi \text{ and } x \in \text{vars}(\sigma)
\end{cases}
\end{align*}
\]

where \( \sigma' \) is the function that maps \( x \) to \( x' \) and \( x' \) is a fresh variable (i.e., a variable that neither occurs in \( \psi \) nor in \( \text{vars}(\sigma) \)).
Notation

If we do not want to specify the substitution function $\sigma$ separately, we write $\varphi[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$ instead of $\varphi\sigma$ if $\sigma$ is the function that maps $x_i$ to $t_i$ for $i \in \{1, \ldots, n\}$.

Notation

We sometimes use $\varphi[x]$ to refer to a formula and a variable. We may then use in this context $\varphi[t]$ to denote $\varphi[x \mapsto t]$. 
Analogously to $\mathcal{N}_{\text{PL}}$ for propositional logic there is a proof system for proving implications $\Gamma \vdash \varphi$ of FOL formulas. We call this proof system *natural deduction for first order logic* and denote it by $\mathcal{N}_{\text{FOL}}$. Analogously to $\mathcal{N}_{\text{PL}}$ we define the term *derivation* and use this tree as a proof.

For each rule of $\mathcal{N}_{\text{PL}}$ there is an analogous rule in $\mathcal{N}_{\text{FOL}}$. Additionally we have the four rules that are shown on the next slide. Two of these rules have additional *side conditions* that are written right beneath the rule. A tree is only a *derivation* if all side conditions are satisfied.
Proof rules of $\mathcal{N}_{\text{FOL}}$

For each rule of $\mathcal{N}_{\text{PL}}$ there is an analogous rule in $\mathcal{N}_{\text{FOL}}$. Additionally we have the following four rules, where $\varphi$ is some a FOL formula, $t$ is some FOL term and $x, y$ are variables.

\[(I_\forall) \quad \frac{\Gamma \models \varphi[x \mapsto y]}{\Gamma \models \forall x. \varphi}\]  
\[(E_\forall) \quad \frac{\Gamma \models \forall x. \varphi}{\Gamma \models \varphi[x \mapsto t]}\]

\[(I_\exists) \quad \frac{\Gamma \models \varphi[x \mapsto t]}{\Gamma \models \exists x. \varphi}\]  
\[(E_\exists) \quad \frac{\Gamma \models \exists x. \varphi \quad \Gamma \cup \{\varphi[x \mapsto y]\} \models \psi}{\Gamma \models \psi}\]

(a) $y \notin \text{freevars}(\Gamma)$ and either $x = y$ or $y \notin \text{freevars}(\varphi)$

(b) $y \notin \text{freevars}(\Gamma \cup \psi)$ and either $x = y$ or $y \notin \text{freevars}(\varphi)$
Example

\[ \Gamma = \{ \forall x, y, z. p(x, y) \land p(y, z) \rightarrow p(x, z), \forall x, y. p(x, y) \rightarrow p(y, x) \} \]

Task: prove that the implication \( \Gamma \models p(a, b) \land p(b, c) \rightarrow p(c, a) \) is valid.

In this derivation, we use \( \Gamma' \) as a shorthand for \( \Gamma \cup \{p(a, b) \land p(b, c)\} \).

\[
\begin{array}{c}
\Gamma' \models p(a, b) \land p(b, c) \\
\hline
(Ax) \quad \frac{\Gamma' \models \forall x. \forall y. \forall z. p(x, y) \land p(y, z) \rightarrow p(x, z)}{}
\end{array}
\]

\[
\begin{array}{c}
\Gamma' \models p(a, z) \\
\hline
(E_{\forall}) \quad \frac{\Gamma' \models \forall x. \forall y. \forall z. p(x, y) \land p(y, z) \rightarrow p(a, z)}{}
\end{array}
\]

\[
\begin{array}{c}
\Gamma' \models p(a, c) \\
\hline
(E_{\forall}) \quad \frac{\Gamma' \models p(a, b) \land p(b, c) \rightarrow p(a, c)}{}
\end{array}
\]

\[
\begin{array}{c}
\Gamma' \models p(a, c) \\
\hline
(Ax) \quad \frac{\Gamma' \models \forall x. \forall y. p(x, y) \rightarrow p(y, x)}{}
\end{array}
\]

\[
\begin{array}{c}
\Gamma' \models p(a, c) \\
\hline
(E_{\forall}) \quad \frac{\Gamma' \models \forall y. p(a, y) \rightarrow p(y, a)}{}
\end{array}
\]

\[
\begin{array}{c}
\Gamma' \models p(a, c) \\
\hline
(E_{\forall}) \quad \frac{\Gamma' \models p(a, c) \rightarrow p(c, a)}{}
\end{array}
\]

\[
\begin{array}{c}
\Gamma' \models p(a, c) \\
\hline
(I_{\rightarrow}) \quad \frac{\Gamma \models p(a, b) \land p(b, c) \rightarrow p(a, c)}{}
\end{array}
\]

\[
\begin{array}{c}
\Gamma' \models p(c, a) \\
\hline
(E_{\rightarrow}) \quad \frac{\Gamma \models p(a, b) \land p(b, c) \rightarrow p(c, a)}{}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \models p(a, b) \land p(b, c) \\
\hline
(G) \quad \frac{\Gamma \models p(a, b) \land p(b, c) \rightarrow p(c, a)}{}
\end{array}
\]
Program Verification

Summer Term 2021

Lecture 5: First-Order Theories

Matthias Heizmann

Monday 3rd May
Section 4

First-Order Theories
Outline

- Introduction
- Propositional Logic
- First-Order Logic
- First-Order Theories
- SMT-LIB
- Boogie and Boostan
- Hoare Proof System
- Ultimate Referee
- Arrays
In practice, symbols like e.g., the constant symbol “0”, the function symbol “+”, or the relation symbol “=” come with a fixed predefined meaning. In this section we will see how we can give symbols in first-order logic a meaning.

What we learn in this section:

- Our intuitive understanding of satisfiability and validity does not always coincide with the classical definition from the last section.
- Finding all axioms that are needed to define the meaning of a symbol is an error-prone and difficult task.
- The expressiveness of an apparently simple theory can be surprisingly high.
- An apparently simple theory can be undecidable.
- There are not only theories for classical arithmetic but also for arithmetic of CPUs.
Outline of the Section on First-Order Theories

Motivation

- $T$-Validity and $T$-Satisfiability
- Theory of Equality
- Theory of Rock-Paper-Scissors
- Decidability
- Natural Numbers and Integers
- Rationals and Reals
- Arrays
- Combination of Theories
- Decidability
We do not only want to use abstract constant symbols $c, d, e, \ldots$ function symbols $f, g, h, \ldots$ and predicate symbols $p, q, \ldots$ but also the symbols $0, 1, +, \cdot, /, =, \leq, \ldots$. If we use these symbols, we use an infix notation. E.g., we write $\exists x. \ y = 2 \cdot x$ instead of $\exists x. \ = (y, \cdot(2, x))$

Warning: symbols might not have the expected meaning.
Is the following program correct?

```c
void copyAtoBandC(int a) {
    int b = a;
    int c = b;
    assert (c == a);
}
```

In order to check correctness, we would like to check validity of the following FOL formula.

\[(a = b \land b = c) \rightarrow c = a\]

Problem:
Formula not valid. Counterexample: model \( \mathcal{M} = (\mathcal{D}, \mathcal{I}) \) where \( \mathcal{D} = \{\♣, \♠\} \) and \( \mathcal{I} \) maps the 2-ary predicate symbol \( = \) to the binary relation \( \{((♣, ♠), (♠, ♣))\} \subseteq \mathcal{D} \times \mathcal{D} \).
Problem: We do not want to check if $\varphi$ is valid. We want to check if $\varphi$ holds for some (partial) model $\mathcal{M}$.

Solution: Find a set of formulas $A_T$ such that only $\mathcal{M}$ (and “similar” models) can make all these formulas valid. Check if $A_T$ implies $\varphi$.

Example

We will not check if $\varphi : (a = b \land b = c) \rightarrow c = a$ is valid. Instead we consider the set $A_T$ that contains the following three formulas:

\[
\forall x. \ x = x, \quad \text{(reflexivity)}
\]
\[
\forall x, y. \ x = y \rightarrow y = x, \quad \text{(symmetry)}
\]
\[
\forall x, y, z. \ x = y \land y = z \rightarrow x = z, \quad \text{(transitivity)}
\]

and check if $A_T$ implies $\varphi$. 
Outline of the Section on First-Order Theories

Motivation

$T$-Validity and $T$-Satisfiability

Theory of Equality

Theory of Rock-Paper-Scissors

Decidability

Natural Numbers and Integers

Rationals and Reals

Arrays

Combination of Theories

Decidability
First-Order Theories: Definition

Definition (First-order theory)

A *first-order theory* $T$ consists of

- A *signature* $\Sigma$ - set of constant, function, and predicate symbols
- A set of *axioms* $A_T$ - set of *closed* (no free variables) $\Sigma$-formulae

A $\Sigma$-*formula* is a formula constructed of constants, functions, and predicate symbols from $\Sigma$, and variables, logical connectives, and quantifiers.

Idea:

- The symbols of $\Sigma$ are *just symbols* without prior meaning.
- The axioms of $T$ provide their meaning.
**T-Validity and T-Satisfiability**

**Definition (T-model)**

A model $\mathcal{M}$ is a **T-model**, if $\models_{\mathcal{M},\rho} \varphi = \text{true}$ for all $\varphi \in A_T$ and for all variable assignments $\rho$.

**Definition (T-valid)**

A $\Sigma$-formula $\varphi$ is **valid in theory T (T-valid)**, if for every $T$-model $\mathcal{M}$, it holds that $\models_{\mathcal{M}} \varphi = \text{true}$.

**Definition (T-satisfiable)**

A $\Sigma$-formula $\varphi$ is **satisfiable in T (T-satisfiable)**, if there is a $T$-model $\mathcal{M}$ such that $\models_{\mathcal{M}} \varphi = \text{true}$.

**Definition (T-equivalent)**

Two $\Sigma$-formulae $\varphi_1$ and $\varphi_2$ are **equivalent in T (T-equivalent)**, if $\varphi_1 \leftrightarrow \varphi_2$ is $T$-valid.
Program Verification

Summer Term 2021

Lecture 6: First-Order Theories

Matthias Heizmann

Wednesday 5th May
Outline of the Section on First-Order Theories

Motivation

$T$-Validity and $T$-Satisfiability

Theory of Equality

Theory of Rock-Paper-Scissors

Decidability

Natural Numbers and Integers

Rationals and Reals

Arrays

Combination of Theories

Decidability
**Question:** Are the following axioms sufficient for defining the usual meaning of the equality symbol?

\[
\begin{align*}
\forall x. & \ x = x, \quad \text{(reflexivity)} \\
\forall x, y. & \ x = y \rightarrow y = x, \quad \text{(symmetry)} \\
\forall x, y, z. & \ x = y \land y = z \rightarrow x = z, \quad \text{(transitivity)}
\end{align*}
\]

**Hint:** Is the following formula implied by the axioms?

\[a = b \land f(a) = c \rightarrow f(b) = c\]

**Answer:** These axioms are sufficient if there are no other predicate symbols or function symbols. Otherwise these axioms are not sufficient because we expect that functions return the same outputs for the same inputs.
Theory of Equality $T_E$

**Signature** $\Sigma_=: \{=, a, b, c, \cdots, f, g, h, \cdots, p, q, r, \cdots\}$

- $=,$ a binary predicate, *interpreted* by axioms.
- all constant, function, and predicate symbols.

**Axioms of $T_E$:**

1. $\forall x. x = x$ (reflexivity)
2. $\forall x, y. x = y \rightarrow y = x$ (symmetry)
3. $\forall x, y, z. x = y \land y = z \rightarrow x = z$ (transitivity)

4. for each positive integer $n$ and $n$-ary function symbol $f$,
   $\forall x_1, \ldots, x_n, y_1, \ldots, y_n. \ \land_i x_i = y_i \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$ (congruence)

5. for each positive integer $n$ and $n$-ary predicate symbol $p$,
   $\forall x_1, \ldots, x_n, y_1, \ldots, y_n. \ \land_i x_i = y_i \rightarrow (p(x_1, \ldots, x_n) \leftrightarrow p(y_1, \ldots, y_n))$ (equivalence)
Axiom Schemata

Congruence and Equivalence are *axiom schemata*.

4. for each positive integer $n$ and $n$-ary function symbol $f$,
   \[
   \forall x_1, \ldots, x_n, y_1, \ldots, y_n. \ \bigwedge_i x_i = y_i \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)
   \]  
   (congruence)

5. for each positive integer $n$ and $n$-ary predicate symbol $p$,
   \[
   \forall x_1, \ldots, x_n, y_1, \ldots, y_n. \ \bigwedge_i x_i = y_i \rightarrow (p(x_1, \ldots, x_n) \iff p(y_1, \ldots, y_n))
   \]  
   (equivalence)

For every function symbol there is an instance of the congruence axiom schema.

*Example*: Congruence axiom for binary function $f_2$:
\[
\forall x_1, x_2, y_1, y_2. \ x_1 = y_1 \land x_2 = y_2 \rightarrow f_2(x_1, x_2) = f_2(y_1, y_2)
\]

$A_{T_E}$ contains an infinite number of these axioms.
Outline of the Section on First-Order Theories

- Motivation
- $T$-Validity and $T$-Satisfiability
- Theory of Equality
- Theory of Rock-Paper-Scissors
- Decidability
- Natural Numbers and Integers
- Rationals and Reals
- Arrays
- Combination of Theories
- Decidability
On the next slide, we will next define our own theory and see that this is a difficult task.

**Question 1:** Are axioms 1-3 sufficient?

**Hint 1:** Is the formula \( \neg \exists x. \forall y. x \succ_{\text{win}} y \) (no element wins against all others) valid with respect to these axioms?

**Answer 1:** Axioms 1-3 are not sufficient. A model in which \( \succ_{\text{win}} \) is mapped to a relation that contains all pairs would satisfy the axioms.

**Question 2:** Are axioms 1-9 sufficient?

**Hint 2:** Is the formula \( \neg \exists x. \forall y. x \succ_{\text{win}} y \) valid with respect to axioms 1-9?

**Answer 2:** Axioms 1-9 are not sufficient. A model in which the domain contains also an element \textbf{Well} that wins against all others would satisfy the axioms.

As a solution, we also add axiom 10 which however requires that we also add axioms that define the semantics of the equality symbol.
Exercise: Define Theory of Rock-Paper-Scissors

- **Signature \( \Sigma_{RPS} \)**
  - Constant symbols: Rock, Paper, Scissors
  - Relation symbol: \( \succ_{\text{win}} \)

- **Axioms** \( A_{T_{RPS}} \)
  1. Rock \( \succ_{\text{win}} \) Scissors
  2. Scissors \( \succ_{\text{win}} \) Paper
  3. Paper \( \succ_{\text{win}} \) Rock
  4. \( \neg \) Rock \( \succ_{\text{win}} \) Rock
  5. \( \neg \) Rock \( \succ_{\text{win}} \) Paper
  6. \( \neg \) Scissors \( \succ_{\text{win}} \) Scissors
  7. \( \neg \) Scissors \( \succ_{\text{win}} \) Rock
  8. \( \neg \) Paper \( \succ_{\text{win}} \) Paper
  9. \( \neg \) Paper \( \succ_{\text{win}} \) Rock
  10. \( \forall x. \ x = \text{Rock} \lor x = \text{Paper} \lor x = \text{Scissors} \)

Are the following formulas T-valid?

- \( \neg \exists x. \ \forall y. \ x \succ_{\text{win}} y \)
- \( \forall x. \ \exists y. \ x \succ_{\text{win}} y \)
Outline of the Section on First-Order Theories

- Motivation
- $T$-Validity and $T$-Satisfiability
- Theory of Equality
- Theory of Rock-Paper-Scissors
- Decidability
- Natural Numbers and Integers
- Rationals and Reals
- Arrays
- Combination of Theories
- Decidability
Decidability

Reminder

We call a problem *decidable* if there exists an algorithm that terminates on all instances of the problem and gives a correct yes/no answer.

We call a problem *semi-decidable* if there exists an algorithm that terminates at least on all “yes”-instances of the problem and gives a correct answer if it terminates.

Example of an undecidable problem: halting problem for Turing machines.

Typical way to prove decidability: give an algorithm and prove its correctness.

Typical way to prove undecidability: proof via a diagonal argument (e.g., Cantor’s diagonal argument) or proof via reduction.
Decidability

Theorem

*Satisfiability of PL formulas is decidable.*

Proof not given in this course.

Decision procedure: truth table.

Theorem

*Satisfiability of FOL formulas is undecidable.*

Proof not given in this course.

Theorem

*Validity of FOL formulas is semi-decidable.*

Proof not given in this course.

Decision procedure: enumerate trees to find a derivation, semi-decidability follows from soundness and completeness of $\mathcal{N}_{FOL}$
Decidability of $T_E$

Is it possible to decide $T_E$-validity?

**Theorem**

$T_E$-validity is undecidable.

Proof not given in this course.

If we restrict ourselves to quantifier-free formulae we get decidability:

**Theorem**

*For a quantifier-free formula $T_E$-validity is decidable.*

Proof not given in this course.
Outline of the Section on First-Order Theories

Motivation
T-Validity and T-Satisfiability
Theory of Equality
Theory of Rock-Paper-Scissors
Decidability
Natural Numbers and Integers
Rationals and Reals
Arrays
Combination of Theories
Decidability
Natural Numbers and Integers

Natural numbers \( \mathbb{N} = \{0, 1, 2, \cdots \} \)
Integers \( \mathbb{Z} = \{\cdots, -2, -1, 0, 1, 2, \cdots \} \)

Three variations:

- **Peano arithmetic** \( T_{PA} \): natural numbers with addition and multiplication
- **Presburger arithmetic** \( T_{\mathbb{N}} \): natural numbers with addition
- **Theory of integers** \( T_{\mathbb{Z}} \): integers with +, −, >
Peano Arithmetic $T_{PA}$ (first-order arithmetic)

**Signature:** $\Sigma_{PA} : \{0, 1, +, \cdot, =\}$

**Axioms of $T_{PA}$:** axioms of $T_E$,

1. $\forall x. \neg(x + 1 = 0)$ (zero)
2. $\forall x, y. x + 1 = y + 1 \rightarrow x = y$ (successor)
3. $\varphi[0] \land (\forall x. \varphi[x] \rightarrow \varphi[x + 1]) \rightarrow \forall x. \varphi[x]$ (induction)
4. $\forall x. x + 0 = x$ (plus zero)
5. $\forall x, y. x + (y + 1) = (x + y) + 1$ (plus successor)
6. $\forall x. x \cdot 0 = 0$ (times zero)
7. $\forall x, y. x \cdot (y + 1) = x \cdot y + x$ (times successor)

Line 3 is an axiom schema.
Expressiveness of Peano Arithmetic

$3x + 5 = 2y$ can be written using $\Sigma_{PA}$ as

\[
x + x + x + 1 + 1 + 1 + 1 + 1 = y + y
\]

We can define $>$ and $\geq$:

\[
\begin{align*}
3x + 5 & > 2y \quad \text{write as} \quad \exists z. \ z \neq 0 \land 3x + 5 = 2y + z \\
3x + 5 & \geq 2y \quad \text{write as} \quad \exists z. \ 3x + 5 = 2y + z
\end{align*}
\]

Examples for valid formulae:

- Pythagorean Theorem is $T_{PA}$-valid

\[
\exists x, y, z. \ x \neq 0 \land y \neq 0 \land z \neq 0 \land xx + yy = zz
\]

- Fermat’s Last Theorem is $T_{PA}$-valid (Andrew Wiles, 1994)

\[
\forall n. \ n > 2 \rightarrow \neg \exists x, y, z. \ x \neq 0 \land y \neq 0 \land z \neq 0 \land x^n + y^n = z^n
\]
Expressiveness of Peano Arithmetic (2)

In Fermat’s theorem we used $x^n$, which is not a valid term in $\Sigma_{PA}$.
However, there is the $\Sigma_{PA}$-formula $EXP[x, n, r]$ with

1. $EXP[x, 0, r] \iff r = 1$
2. $EXP[x, i + 1, r] \iff \exists r_1. \ (EXP[x, i, r_1] \wedge r = r_1 \cdot x)$

$$EXP[x, n, r] : \exists d, m. \ (\exists z. \ d = (m + 1)z + 1) \wedge$$
$$\ (\forall i, r_1. \ i < n \wedge r_1 < m \wedge (\exists z. \ d = ((i + 1)m + 1)z + r_1)) \rightarrow$$
$$r_1x < m \wedge (\exists z. \ d = ((i + 2)m + 1)z + r_1 \cdot x)) \wedge$$
$$r < m \wedge (\exists z. \ d = ((n + 1)m + 1)z + r)$$

Fermat’s theorem can be stated as:

$$\forall n. \ n > 2 \rightarrow \neg \exists x, y, z, rx, ry. \ x \neq 0 \wedge y \neq 0 \wedge z \neq 0 \wedge$$
$$EXP[x, n, rx] \wedge EXP[y, n, ry] \wedge EXP[z, n, rx + ry]$$
Decidability of Peano Arithmetic

Gödel showed that for every recursive function \( f : \mathbb{N}^n \rightarrow \mathbb{N} \) there is a \( \Sigma_{PA} \)-formula \( \varphi[x_1, \ldots, x_n, r] \) with

\[
\varphi[x_1, \ldots, x_n, r] \leftrightarrow r = f(x_1, \ldots, x_n)
\]

\( T_{PA} \) is undecidable. (Gödel, Turing, Post, Church)

The quantifier-free fragment of \( T_{PA} \) is undecidable. (Matiyasevich, 1970)

Remark: Gödel’s first incompleteness theorem

Peano arithmetic \( T_{PA} \) does not capture true arithmetic:
There exist closed \( \Sigma_{PA} \)-formulae representing valid propositions of number theory that are not \( T_{PA} \)-valid.
The reason: \( T_{PA} \) actually admits nonstandard interpretations.

For decidability: no multiplication.
Presburger Arithmetic \( T_\mathbb{N} \)

Signature: \( \Sigma_\mathbb{N} : \{0, 1, +, =\} \) \hspace{1cm} \text{no multiplication!}

Axioms of \( T_\mathbb{N} \): axioms of \( T_E \),

1. \( \forall x. \neg(x + 1 = 0) \) \hspace{1cm} \text{(zero)}
2. \( \forall x, y. x + 1 = y + 1 \rightarrow x = y \) \hspace{1cm} \text{(successor)}
3. \( \varphi[0] \land (\forall x. \varphi[x] \rightarrow \varphi[x + 1]) \rightarrow \forall x. \varphi[x] \) \hspace{1cm} \text{(induction)}
4. \( \forall x. x + 0 = x \) \hspace{1cm} \text{(plus zero)}
5. \( \forall x, y. x + (y + 1) = (x + y) + 1 \) \hspace{1cm} \text{(plus successor)}

3 is an axiom schema.

\( T_\mathbb{N} \)-satisfiability and \( T_\mathbb{N} \)-validity are decidable. (Presburger 1929)
Theory of Integers $T_Z$

**Signature:**

$\Sigma_Z : \{\ldots, -2, -1, 0, 1, 2, \ldots, -3, -2, 2, 3, \ldots, +, -, =, >\}$

where

- $\ldots, -2, -1, 0, 1, 2, \ldots$ are constants
- $\ldots, -3, -2, 2, 3, \ldots$ are unary functions (intended meaning: $2 \cdot x$ is $x + x$)
- $+, -, =, >$ have the usual meanings.

**Relation between $T_Z$ and $T_N$**

$T_Z$ and $T_N$ have the same expressiveness:

- For every $\Sigma_Z$-formula there is an equisatisfiable $\Sigma_N$-formula.
- For every $\Sigma_N$-formula there is an equisatisfiable $\Sigma_Z$-formula.

$\Sigma_Z$-formula $\varphi$ and $\Sigma_N$-formula $G$ are *equisatisfiable* iff:

\[
    \varphi \text{ is } T_Z\text{-satisfiable} \quad \text{iff} \quad G \text{ is } T_N\text{-satisfiable}
\]
Example: \( \Sigma_N \)-formula to \( \Sigma_Z \)-formula.

\[
\forall x. \exists y. x = y + 1
\]
is equisatisfiable to the \( \Sigma_Z \)-formula:

\[
\forall x. x > -1 \rightarrow \exists y. y > -1 \land x = y + 1.
\]
Example: $\Sigma_\mathbb{Z}$-formula to $\Sigma_\mathbb{N}$-formula

Consider the $\Sigma_\mathbb{Z}$-formula

$F_0 : \forall w, x. \exists y, z. x + 2y - z - 7 > -3w + 4$

Introduce two variables, $v_p$ and $v_n$ (range over the nonnegative integers) for each variable $v$ (range over the integers) of $F_0$

$F_1 : \forall w_p, w_n, x_p, x_n. \exists y_p, y_n, z_p, z_n. (x_p - x_n) + 2(y_p - y_n) - (z_p - z_n) - 7 > -3(w_p - w_n) + 4$

Eliminate $>$ by moving to the other side of $>$

$F_2 : \forall w_p, w_n, x_p, x_n. \exists y_p, y_n, z_p, z_n. x_p + 2y_p + z_n + 3w_p > x_n + 2y_n + z_p + 7 + 3w_n + 4$

Eliminate $>$ and numbers:

$F_3 : \forall w_p, w_n, x_p, x_n. \exists y_p, y_n, z_p, z_n. \exists u. \neg (u = 0) \land x_p + y_p + y_p + z_n + w_p + w_p + w_p = x_n + y_n + y_n + z_p + w_n + w_n + w_n + u + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1$

which is a $\Sigma_\mathbb{N}$-formula equisatisfiable to $F_0$. 
Reducing $\mathcal{T}_\mathbb{Z}$ to $\mathcal{T}_\mathbb{N}$.

To decide $\mathcal{T}_\mathbb{Z}$-validity for a $\Sigma_\mathbb{Z}$-formula $\varphi$:

- transform $\neg \varphi$ to an equisatisfiable $\Sigma_\mathbb{N}$-formula $\neg \psi$,
- decide $\mathcal{T}_\mathbb{N}$-validity of $\psi$. 

Matthias Heizmann
Program Verification
Summer Term 2021
Rationals and Reals

\[ \Sigma = \{0, 1, +, -, \cdot, =, \geq\} \]

- **Theory of Reals** \( T_R \) (with multiplication)

\[ x \cdot x = 2 \quad \Rightarrow \quad x = \pm \sqrt{2} \]

- **Theory of Rationals** \( T_Q \) (no multiplication)

\[ 2x x \overbrace{+x}^{+x} = 7 \quad \Rightarrow \quad x = \frac{2}{7} \]

*Note: Strict inequality*

\[ \forall x, y. \exists z. x + y > z \]

can be expressed as

\[ \forall x, y. \exists z. \neg (x + y = z) \land x + y \geq z \]
Theory of Reals $T_R$

**Signature:** $\Sigma_R : \{0, 1, +, −, ·, =, \geq\}$ with multiplication.

**Axioms of $T_R$:** axioms of $T_E$,

1. $\forall x, y, z. \ (x + y) + z = x + (y + z)$ (+ associativity)
2. $\forall x, y. \ x + y = y + x$ (+ commutativity)
3. $\forall x. \ x + 0 = x$ (+ identity)
4. $\forall x. \ x + (−x) = 0$ (+ inverse)
5. $\forall x, y, z. \ (x \cdot y) \cdot z = x \cdot (y \cdot z)$ (· associativity)
6. $\forall x, y. \ x \cdot y = y \cdot x$ (· commutativity)
7. $\forall x. \ x \cdot 1 = x$ (· identity)
8. $\forall x. \ x \neq 0 \rightarrow \exists y. \ x \cdot y = 1$ (· inverse)
9. $\forall x, y, z. \ x \cdot (y + z) = x \cdot y + x \cdot z$ (distributivity)
10. $0 \neq 1$ (separate identities)
11. $\forall x, y. \ x \geq y \land y \geq x \rightarrow x = y$ (antisymmetry)
12. $\forall x, y, z. \ x \geq y \land y \geq z \rightarrow x \geq z$ (transitivity)
13. $\forall x, y. \ x \geq y \lor y \geq x$ (totality)
14. $\forall x, y, z. \ x \geq y \rightarrow x + z \geq y + z$ (+ ordered)
15. $\forall x, y. \ x \geq 0 \land y \geq 0 \rightarrow x \cdot y \geq 0$ (· ordered)
16. $\forall x. \ \exists y. \ x = y \cdot y \lor x = −y \cdot y$ (square root)
17. for each odd integer $n$,
$\forall x_0, \ldots, x_{n−1}. \ \exists y. \ y^n + x_{n−1}y^{n−1} + \cdots + x_1y + x_0 = 0$ (at least one root)
Decidability of $T_R$

$T_R$ is decidable (Tarski, 1930)
High time complexity: $O(2^{kn})$
Theory of Rationals $T_\mathcal{Q}$

**Signature:** $\Sigma_\mathcal{Q} : \{0, 1, +, −, =, \geq\}$ no multiplication!

**Axioms of $T_\mathcal{Q}$:** axioms of $T_E$,

1. $\forall x, y, z. \ (x + y) + z = x + (y + z)$  
   (+ associativity)
2. $\forall x, y. \ x + y = y + x$  
   (+ commutativity)
3. $\forall x. \ x + 0 = x$  
   (+ identity)
4. $\forall x. \ x + (−x) = 0$  
   (+ inverse)
5. $1 \geq 0 \land 1 \neq 0$  
   (one)
6. $\forall x, y. \ x \geq y \land y \geq x \rightarrow x = y$  
   (antisymmetry)
7. $\forall x, y, z. \ x \geq y \land y \geq z \rightarrow x \geq z$  
   (transitivity)
8. $\forall x, y. \ x \geq y \lor y \geq x$  
   (totality)
9. $\forall x, y, z. \ x \geq y \rightarrow x + z \geq y + z$  
   (+ ordered)
10. For every positive integer $n$: \[ \forall x. \ \exists y. \ x = \underbrace{y + \cdots + y}_n \]  
   (divisible)
Expressiveness and Decidability of $T_\mathbb{Q}$

Rational coefficients are simple to express in $T_\mathbb{Q}$

Example: Rewrite

$$\frac{1}{2}x + \frac{2}{3}y \geq 4$$

as the $\Sigma_\mathbb{Q}$-formula

$$x + x + x + y + y + y + y \geq 1 + 1 + \cdots + 1$$

$T_\mathbb{Q}$ is decidable.
Efficient algorithm for quantifier free fragment.
Outline of the Section on First-Order Theories

- Motivation
- $T$-Validity and $T$-Satisfiability
- Theory of Equality
- Theory of Rock-Paper-Scissors
- Decidability
- Natural Numbers and Integers
- Rationals and Reals
- Arrays
- Combination of Theories
- Decidability
Theory of Arrays $T_A$

**Signature:** $\Sigma_A : \{\cdot[\cdot], \cdot\langle\cdot\triangleright\cdot\rangle, =\}$, where

- $a[i]$ binary function – read array $a$ at index $i$ (“read($a,i$)”) 
- $a\langle i \triangleright v \rangle$ ternary function – write value $v$ to index $i$ of array $a$ (“write($a,i,e$)”) 

**Axioms**

1. the axioms of (reflexivity), (symmetry), and (transitivity) of $T_E$
2. $\forall a, i, j. \ i = j \rightarrow a[i] = a[j]$ \hspace{1cm} (array congruence)
3. $\forall a, v, i, j. \ i = j \rightarrow a\langle i \triangleright v \rangle[j] = v$ \hspace{1cm} (read-over-write 1)
4. $\forall a, v, i, j. \ i \neq j \rightarrow a\langle i \triangleright v \rangle[j] = a[j]$ \hspace{1cm} (read-over-write 2)
Equality in $T_A$

*Note:* $=$ is only defined for array elements

$$a[i] = e \rightarrow a\langle i \triangleleft e \rangle = a$$

not $T_A$-valid, but

$$a[i] = e \rightarrow \forall j. a\langle i \triangleleft e \rangle[j] = a[j]$$

is $T_A$-valid.

Also

$$a = b \rightarrow a[i] = b[i]$$

is not $T_A$-valid: We only axiomatized a restricted congruence.

$T_A$ is undecidable.
Quantifier-free fragment of $T_A$ is decidable.
Theory of Arrays $T^\equiv_A$ (with extensionality)

Signature and axioms of $T^\equiv_A$ are the same as $T_A$, with one additional axiom

$$\forall a, b. \ (\forall i. \ a[i] = b[i]) \iff a = b \ \text{(extensionality)}$$

*Example:*

$$F : \ a[i] = e \rightarrow a\langle i \triangleleft e \rangle = a$$

is $T^\equiv_A$-valid.

$T^\equiv_A$ is undecidable.

Quantifier-free fragment of $T^\equiv_A$ is decidable.
Outline of the Section on First-Order Theories

Motivation

\( T \)-Validity and \( T \)-Satisfiability

Theory of Equality

Theory of Rock-Paper-Scissors

Decidability

Natural Numbers and Integers

Rationals and Reals

Arrays

Combination of Theories

Decidability
Combination of Theories

How do we show that

\[ 1 \leq x \land x \leq 2 \land f(x) \neq f(1) \land f(x) \neq f(2) \]

is \((T_E \cup T_\mathbb{Z})\)-unsatisfiable?

Or how do we prove properties about

- an array of integers, or
- a list of reals . . . ?

Given theories \(T_1\) and \(T_2\) such that

\[ \Sigma_1 \cap \Sigma_2 = \{=\} \]

The *combined theory* \(T_1 \cup T_2\) has

- signature \(\Sigma_1 \cup \Sigma_2\)
- axioms \(A_1 \cup A_2\)
Nelson & Oppen showed that

if satisfiability of qff of $T_1$ is decidable,
satisfiability of qff of $T_2$ is decidable, and
certain technical requirements are met
then satisfiability of qff of $T_1 \cup T_2$ is decidable.
Theory of Bit-vectors

Idea: theory for low-level arithmetic on computer hardware

- Domain: sequences of bits
e.g., 11111111 (which represents the natural number 255 or the integer -1 in two’s complement representation)

- Functions: arithmetic and logical operations on FixedSizeBitvectors
  \[ \text{bvadd}_8(11111101, 000000100) = 00000001 \]
  \[ \text{bvand}_8(11111101, 000000100) = 00000100 \]
  \[ \text{bvshl}_8(11111101, 000000001) = 11111010 \]

- Predicates: comparisons
  \[ \text{bvult}_8(11111101, 000000100) \text{ is } \text{false} \]
  \[ \text{bvslt}_8(11111101, 000000100) \text{ is } \text{true} \]
  Meaning of bit-vector as number only given by operator.

Signature \( \Sigma \)

- Constant symbols: 0, 1, 01, 10, 11, 001, \ldots
- Function symbols: \( \text{bvadd}_1, \text{bvadd}_2, \text{bvadd}_3, \ldots, \text{bvmul}_1 \ldots \)
- Predicate symbols: \( \text{bvult}_1, \text{bvult}_2, \text{bvult}_3, \ldots, \text{bvslt}_1 \ldots \)

Axioms \( A_T \)

Many
Theory of Bit-vectors

```c
1  signed char s = 400;
2  unsigned char u1 = 250;
3  unsigned char u2 = 250;
4  if (s >= u1 + u2) {

    bvsge32(    signExtendFrom8To32(s),
    bvadd32(signExtendFrom8To32(u1), signExtendFrom8To32(u1))
}
```
Does the following loop terminate?

```cpp
1    for(double d = 0; d != 0.3; d += 0.1) {
2     }
```
Outline of the Section on First-Order Theories

Motivation

T-Validity and T-Satisfiability

Theory of Equality

Theory of Rock-Paper-Scissors

Decidability

Natural Numbers and Integers

Rationals and Reals

Arrays

Combination of Theories

Decidability
# First-Order Theories

<table>
<thead>
<tr>
<th>Theory</th>
<th>Decidable</th>
<th>QFF Dec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_E$ Equality</td>
<td>—</td>
<td>✓</td>
</tr>
<tr>
<td>$T_{PA}$ Peano Arithmetic</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>$T_N$ Presburger Arithmetic</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$T_Z$ Linear Integer Arithmetic</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$T_R$ Real Arithmetic</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$T_Q$ Linear Rationals</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$T_A$ Arrays</td>
<td>—</td>
<td>✓</td>
</tr>
<tr>
<td>$T_A^=$ Arrays with Extensionality</td>
<td>—</td>
<td>✓</td>
</tr>
<tr>
<td>$T_{BV}$ Bitvectors</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$T_{Float}$ FloatingPoint</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Section 5

SMT-LIB
Goals of this section:

- Use a tool to check T-satisfiability (resp. T-validity) of a formula.
  
  In detail:
  - Get familiar with sorted logics
  - Get familiar with the syntax of the SMT-LIB standard.
  - Translate the syntax of the preceding sections into SMT-LIB and vice versa.

- Fix a semantics for symbols like, e.g., $=$, $+$, $-$, $\cdot$, $\text{mod}$, $\leq$, $>$ for the remaining course.
SMT-LIB is an international initiative aimed at facilitating research and development in Satisfiability Modulo Theories (SMT). Since its inception in 2003, the initiative has pursued these aims by focusing on the following concrete goals.

- Provide standard rigorous descriptions of background theories used in SMT systems.
- Develop and promote common input and output languages for SMT solvers.
- Connect developers, researchers and users of SMT, and develop a community around it.
- Establish and make available to the research community a large library of benchmarks for SMT solvers.
- Collect and promote software tools useful to the SMT community.
SMT Script

- File format that allows you to write commands for SMT solvers.
- File ending .smt2
- Prefix notation

Example:
(set-logic QF_LIA) ← use quantifier-free linear integer arithmetic
(declare-fun x () Int) ← announce that constant x has sort Int
(declare-fun y () Int)
(assert (< x 2)) ← put formula on “assertion stack”
(assert (> x 0))
(check-sat) ← check satisfiability of conjunction of all formulas on assertion stack
(get-model) ← get satisfying assignment
(assert (= x (* y 2)))
(check-sat)
Theories defined by SMT-LIB standard:

- **Integer**
  - -, +, *, div, mod, abs, \(\leq, <, >\) =, >
- **Reals**
  - -, +, *, /, \(\leq, <, >\) =, >
- **Arrays** *(will be introduced later in this course)*
  - select, store
- **FixedSizeBitvectors** *(not relevant in this course)*
  - bvadd, bvmul, bvand, bvshl, bvult, ...
- **FloatingPoint** *(not relevant in this course)*
  - fp.add, fp.mul, fp.sqrt, fp.min, fp.leq, fp.isNaN, ...

Conventions

- From now on we use the SMT-LIB definitions for theories.
- Let $T$ be the combination of all theories listed on the preceding slide. Instead of $T$-satisfiability (resp. $T$-validity) we will just use the term *satisfiability* (resp. validity).
SMT-LIB: Logics

SMT-LIB logics:

- Describe syntactically and semantically restricted classes of sorted FOL with equality.
- Specify background theories, restrict to quantifier-free formulas, . . .
- Allow solvers to use efficient, specialized techniques.

Examples:

- **QF_LIA**: Quantifier-Free Linear Integer Arithmetic
- **QF_AX**: Quantifier-Free formulas over Arrays with eXtensionality
- **UFLRA**: Linear Real Arithmetic with Uninterpreted sort and Function symbols
What is a logic?

We have seen propositional logic and first-order logic, and the previous slide talked about different SMT-LIB logics. So what is a logic?

In general, a logic consists of two parts:

1. a language of logical formulas,
2. and an implication relation $\vdash$ between sets of formulas $\Gamma$ and formulas $\varphi$.

For instance:

- We have defined the syntax of propositional logic formulas, and the corresponding implication relation is defined based on the satisfying assignments.

- Similarly, we defined the syntax of FOL formulas. The implication relation is defined via models and satisfying assignments.

- In an SMT-LIB logic with background theory $T$, the formulas are a syntactically restricted subset of the FOL formulas over the signature of $T$. The implication relation is $T$-implication: $\Gamma \vdash_T \psi$ if and only if for every $T$-model $M$ and every assignment $\rho$ we have that if $[\varphi]_{M,\rho} = \text{true}$ for all $\varphi \in \Gamma$, then $[\psi]_{M,\rho} = \text{true}$ also holds.

- Many other logics exist: You may have heard of temporal logics, higher-order logics, intuitionistic logic, ...
SMT-LIB Terms

In the lecture, we defined a (FOL) term inductively to be a variable symbol, a constant symbol, or the application of a function symbol to terms. We defined a (FOL) formula to be the application of a predicate to terms, the negation of a formula, the conjunction of two formulas, and the application of a quantifier to a formula.

In SMT-LIB, every term has a sort. Constants are 0-ary functions, predicates are functions of sort Bool, and logical connectives are functions with argument sort Bool and return sort Bool. Therefore, a formula is just a term of sort Bool as it is an application of a function symbol to terms.
SMT-LIB: Terms

Terms as defined in the lecture:

- Constant symbol.
- Variable symbol.
- Application of a function symbol to terms.

Terms as defined in SMT-LIB:

- Constant symbol, variable symbol, function symbol (applied to terms), variable binders applied to terms, annotations on terms.
- Only well-sorted terms allowed.
- Constant symbols are nullary function symbols.
- Predicates are function symbols of sort Bool.
- Logical connectives are function symbols, and formulas are terms of sort Bool.
On the previous slide, we have seen an overview about the conceptual differences between (FOL) terms as defined in the lecture and SMT-LIB terms as defined by the SMT-LIB standard. There are also some differences in the notation of terms and formulas. We show how to write terms and formulas as defined in the lecture as SMT-LIB terms on the next slide.
## SMT-LIB: Terms

<table>
<thead>
<tr>
<th>Term or formula</th>
<th>SMT-LIB term</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$x$</td>
</tr>
<tr>
<td>$c$</td>
<td>$c$</td>
</tr>
<tr>
<td>$f(t_1 \ldots t_n)$</td>
<td>$(f \ t_1 \ldots \ tn)$</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>$\neg F$</td>
<td>(not $F$)</td>
</tr>
<tr>
<td>$F \land G$</td>
<td>(and $F \ G$)</td>
</tr>
<tr>
<td>$\exists x. F$</td>
<td>(exists ((x Sort)) (F))</td>
</tr>
</tbody>
</table>
SMT solvers are tools that execute SMT scripts.

- **Z3**\(^8\) \texttt{tacas/MouraB08}
  
  Often used in this course because there is a [Z3 web interface](https://github.com/Z3Prover/z3)

- **SMTInterpol**\(^9\) \texttt{spin/ChristHN12}
  
  Developed in our group at the University of Freiburg by Jochen Hoenicke and Tanja Schindler.

- Many more are available. Check the [list of SMT solvers at the SMT-LIB website](https://www.smt-lib.org/) or the [list of SMT solvers at Wikipedia](https://en.wikipedia.org/wiki/List_of_SMT_solvers).

You can submit SMT scripts to the [SMT-LIB benchmark repository](https://www.smt-lib.org/) and the annual [SMT competition](https://www.smtcompetition.org/) evaluates how SMT solver perform on these benchmarks.
SMT-LIB Commands
We have already seen an example for an SMT script. It consists of several commands that allow us, for instance, to tell the solver which logic to use, which function symbols exist, which formulas to check for satisfiability, and so on.

Communicating with the solver via commands allows to flexibly make use of several functionalities of the solver.

Most solvers provide more functionalities than just checking a formula for satisfiability. In the example script, we have seen the \texttt{(get-model)} command that tells the solver to provide a model for a satisfiable formula. If a formula is unsatisfiable, some solvers can also provide a proof for unsatisfiability (but usually, this requires to set an option that tells the solver to keep track of the proof, as this may be expensive).
Important commands to communicate with the solver:

- Set solver parameters:
  ```lisp
  (set-option :produce-models true)
  (set-logic QF_LIA)
  ```

- Declare sorts and symbols:
  ```lisp
  (declare-sort U 0)
  (declare-fun x () Int)
  ```

- Assert formulas:
  ```lisp
  (assert (> x 0))
  ```

- Check satisfiability:
  ```lisp
  (check-sat)
  ```

- Get models:
  ```lisp
  (get-model)
  ```
Program Verification

Summer Term 2021

Lecture 7: Boogie, Boostan

Matthias Heizmann

Monday 10th May
Section 6

Boogie and Boostan
In this section we introduce the programming languages that are most relevant for this course: Boogie and Boostan.

Goals of this section are:

▶ understand that real-world programming languages (C, Java, Python) are not a good choice for presenting the material of this course
▶ recall the basic ideas of context-free grammars
▶ define the syntax of a new programming language
▶ define the semantics of this programming language
▶ define the meaning of “correctness” for programs written in that language
Outline of the Section on Boogie and Boostan

Boogie and Boostan
Context-Free Grammars
Syntax of Boostan
Excursus: The semantics of C
Relational Semantics of Boostan
Precondition-Postcondition Pairs
Which programming language should we choose for our introduction to program verification?

At a first glance it seems reasonable to pick a language that is used by many programmers like e.g., C, Java, or Python. However, if we would do so we would face the following problems.

- The syntax of these languages is very rich and (together with an explanation of its meaning) could not be introduced within a few hours.

- The semantics of these languages is not defined very formally but in hundreds of pages of prose. TODO cite examples We would have to formalize these definitions which is a time-consuming task even if we restrict ourselves to a small fragment of the languages syntax. TODO cite some research

In this subsection we present the languages that we choose is this course.
Boogie and Boostan

Boogie

- Existing “programming language” optimized for verification.
- Devised by Rustan Leino.
- We will use Boogie for practical examples where we use tools.

Boostan

- Fragment of Boogie.
- (Will be) devised by the participants of this course.
- We will formally define the semantics of Boostan.
- We will use Boostan to formally introduce, discuss and analyze verification techniques.
Boogie

- Developed by Rustan Leino at Microsoft Research
- Programming language vs. verification language
- Intermediate language
- Supported by tools
- Limited features (scopes, side-effects, types, memory allocation, concurrency)

TODO Write down what was said in the lecture on each bullet
Boogaloo is an interpreter for Boogie developed by Nadia Polikarpova.

- Available via web interface\(^\text{10}\)
- Displays possible executions of a Boogie program
- Use option \(-o\) to control number of executions, e.g. \(-o 5\) for 5 executions.
- To get more diverse executions, use \(-n\), e.g. \(-n 3\) for at most 3 executions with the same sequence of statements.
- Other interesting options: \(-c=0\) turns off "concrete mode", \(-p\) specifies entry procedure.
- Output with assume \{ :print "text" \} true
- User Manual available\(^\text{11}\)

\(^\text{10}\) http://comcom.csail.mit.edu/comcom/#Boogaloo
Running the following program through Boogaloo with option $-o$ 3 produces the output below, listing arguments, output, and return value.

```plaintext
procedure Square(a : int) returns (square: int) {
    square := a * a;
    if (square == 0) {
        assume {: print "a is zero" } true;
    } else {
        assume {: print "a = ", a } true;
    }
}
```

Execution 0: Square(0) passed
  a is zero
  Outs: square $\rightarrow$ 0

Execution 1: Square(−1) passed
  a = −1
  Outs: square $\rightarrow$ 1

Execution 2: Square(1) passed
  a = 1
  Outs: square $\rightarrow$ 1
Running the following program through Boogaloo with options \(-o\ 4\ \ -n\ 1\ \ -c=0\) produces the output below.

```plaintext
procedure ZeroInit (a : [int]int, lo : int, hi : int) returns (b : [int]int)
{
    var i : int;
    b := a;
    i := lo;
    while (i <= hi) {
        b[i] := 0;
        i := i + 1;
    }
}
```

Execution 0: ZeroInit([], 0, -1) passed
Outs: b -> []

Execution 1: ZeroInit([0 -> 0], 0, 0) passed
Outs: b -> [0 -> 0]

Execution 2: ZeroInit([0 -> 0, 1 -> 0], 0, 1) passed
Outs: b -> [0 -> 0, 1 -> 0]

Execution 3: ZeroInit([0 -> 0, 1 -> 0, 2 -> 0], 0, 2) passed
Outs: b -> [0 -> 0, 1 -> 0, 2 -> 0]
You can try experimenting with the previous program and different options:

- If you only pass `-o`, Boogaloo will only produce executions with `lo > hi`.

  This is because it first chooses a sequence of statements (go through the loop once), and then searches variable values to fit that sequence. Because there are infinitely many (unlike in the first example), it will never consider another sequence.

- Additionally passing `-n` fixes this problem: It allows only the given number of executions per sequence of statements. However, only 2 instead of 4 executions will be found.

  This is because the number of possible values for the input parameters is restricted (Boogaloo calls this the *concrete mode*).

- Additionally passing `-c=0` turns off this concrete mode, finally showing the diverse executions on the previous slide.

Different combinations of these options can often help get the desired test cases for a program. However, always using all of them is not necessarily the solution in every case.
The specification of Boogie\textsuperscript{12} \texttt{leino\_this\_2016} has 52 pages and is not written with the formal rigor that we would like to have in this course.

Idea: let us define a (new) language Boostan

- syntax is a fragment of Boogie
- restricted to the needs of this course
- syntax and semantics defined very rigorously using terminology that we know from computer science lectures (context-free grammar, first-order logic)
- semantics compatible to Boogie

For our formal definitions, algorithms, theorems and proofs we will use Boostan. For demonstrations with tools we use Boogie. We will not establish a formal connection between Boogie and Boostan and resort to our intuition to get the connection.

\textsuperscript{12}https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/
Outline of the Section on Boogie and Boostan

Boogie and Boostan
Context-Free Grammars
Syntax of Boostan
Excursus: The semantics of C
Relational Semantics of Boostan
Precondition-Postcondition Pairs
Motivation: How can we Formalize Programs?

Sequence of characters vs. tree

```
while (i<x) {
  x := x + i;
  i := i + 1;
}
```
The syntax of a programming language is typically defined via a context-free grammar or via a closely related concept.

We will define the syntax of Boostan via a context-free grammar and use a notation that is typically used in lectures on theoretical computer science.

In order to make you (again) familiar with context-free grammars and in order to fix a notation for this course we give a formal definition on the next slides.
Definition

A context-free grammar is a 4-tupel $G = (\Sigma, N, P, S)$ such that

- $\Sigma$ is an alphabet, whose elements we call *terminal symbols*,
- $N$ ist a finite set whose elements we call *nonterminal symbols*,
- $P \subseteq N \times (N \cup \Sigma)^*$ ist a finite relation whose elements we call *derivation rules*,
- $S \in N$ is a nonterminal symbol that we call *start symbol* and $\Sigma \cap N \neq \emptyset$.

Example

Consider $G = (\Sigma, N, P, S)$ with $\Sigma = \{a, b\}$, $N = \{S\}$ and

$$P = \{ \begin{array}{c} S \rightarrow aSbS, \\ S \rightarrow bSaS, \\ S \rightarrow \varepsilon \end{array} \}.$$
Definition

A derivation tree is an ordered tree together with a labelling function \( \lambda : V \rightarrow (N \cup \Sigma \cup \{\varepsilon\}) \) such that

- a node \( v \in V \) may only have children \( v_1, \ldots, v_n \in V \) if \( \lambda(v) \rightarrow \lambda(v_1) \cdots \lambda(v_n) \) is a rule in \( P \) and
- all leaves are labelled by terminal symbols or by \( \varepsilon \).

Example

Consider \( G = (\Sigma, N, P, S) \) with \( \Sigma = \{a, b\}, N = \{S\} \) and

\[
P = \{ S \rightarrow aSbS, \\
    S \rightarrow bSaS, \\
    S \rightarrow \varepsilon \}.
\]
Definition

The derived word $dw$ of a node $v$ is inductively defined as follows.

$$
\begin{align*}
    dw(v) = \begin{cases} 
        dw(v_1) \ldots dw(v_n) & \text{if } v \text{ has children } v_1, \ldots, v_n \\
        \lambda(v) & \text{otherwise}
    \end{cases}
\end{align*}
$$

We say that a word $w \in \Sigma^*$ can be derived from a nonterminal symbol $A \in N$ if there is a derivation tree whose root node $v$ is labelled by $A$ and $dw(v) = w$.

We call the set of all words that can be derived from the start symbol $S$ the language of $G$, denoted $L(G)$.

Example

Derived word of the tree from preceding slide: $abba$

$$
L(G) = \{ w \in \Sigma^* \mid \text{The number of } a \text{'s in } w \text{ is the same as the number of } b \text{'s in } w \}$$
Example

See Exercise 3 on Exercise Sheet 05 for another context-free grammar and a derivation tree.
Exercise: Construct a context-free grammar
\[ G_{\text{Int}} = (\Sigma_{\text{Int}}, N_{\text{Int}}, P_{\text{Int}}, S_{\text{Int}}) \] that generates the language of all FOL terms for the vocabulary \((\mathcal{V}_{\text{Var}}, \mathcal{V}_{\text{Const}}, \mathcal{V}_{\text{Fun}}, \mathcal{V}_{\text{Pred}})\) such that

- \(\mathcal{V}_{\text{Const}}\) is the set of all non-empty words over the alphabet 0–9.
- \(\mathcal{V}_{\text{Var}}\) is the set of all non-empty words over the alphabet a–zA–Z0–9 that are not constant symbols.
- \(\mathcal{V}_{\text{Fun}}\) is the set that contains
  - the unary minus symbol \(-\) and
  - the binary symbols \(+, -, *, \text{div}, \text{mod}, \text{abs}\).
Outline of the Section on Boogie and Boostan

Boogie and Boostan
Context-Free Grammars
Syntax of Boostan
Excursus: The semantics of C
Relational Semantics of Boostan
Precondition-Postcondition Pairs
In this subsection we use context-free grammars to define the syntax of Boostan.

We start with a grammar for numbers and a grammar for variables and extend these grammars incrementally until we have a grammar for statements.

Please note that this is not the final version of Boostan. In the next sections we will extend this section’s definition by arrays, assumptions and nondeterministic assignments. **Todo add link**
Grammar for Numbers

Problem: We would like to be able to represent every integer, but an alphabet has to be finite.

Solution: Like SMT-LIB, we use digits 0 to 9, a decimal encoding and (later) a unary minus to obtain negative numbers.

Additional requirement: We can tolerate leading zeros, but a number should not be the empty word.

\[ G_{\text{num}} = (\Sigma_{\text{num}}, N_{\text{num}}, P_{\text{num}}, S_{\text{num}}) \]

\[ \Sigma_{\text{num}} = \{0, \ldots, 9\} \]

\[ N_{\text{num}} = \{X_{\text{num}}, X_{\text{num}}'\} \]

\[ P_{\text{num}} = \{X_{\text{num}} \rightarrow 0X_{\text{num}}'| \ldots | 9X_{\text{num}}' \}
\]

\[ S_{\text{num}} = X_{\text{num}} \]
Grammar for Variables

Requirements: Every alphanumeric sequence should be a variable but we do not want to allow the empty word and the set of variables should be disjoint from the set of numbers.

\[ G_{\text{var}} = (\Sigma_{\text{var}}, N_{\text{var}}, P_{\text{var}}, S_{\text{var}}) \]

\[ \Sigma_{\text{var}} = \Sigma_{\text{num}} \cup \{a, \ldots, z, A, \ldots Z\} \]

\[ N_{\text{var}} = \{X_{\text{var}}, X_{\text{var}}'\} \]

\[ P_{\text{var}} = \{X_{\text{var}} \rightarrow aX_{\text{var}}' | \ldots | zX_{\text{var}}' | AX_{\text{var}}' | \ldots | ZX_{\text{var}}' \}
\]

\[ X_{\text{var}}' \rightarrow aX_{\text{var}}' | \ldots | zX_{\text{var}}' | AX_{\text{var}}' | \ldots | ZX_{\text{var}}' | 0X_{\text{var}}' | \ldots | 9X_{\text{var}}' | \varepsilon \} \]

\[ S_{\text{var}} = X_{\text{var}} \]
Grammar for Integer Expressions

Requirements: We would like to have integer expressions that are very similar to integer terms in SMT-LIB. We want an infix notation, we would like to use the symbol / instead of \texttt{\texttt{\texttt{div}}} and we would like to use the symbol \% instead of \texttt{\texttt{mod}}.

\[
\mathcal{G}_I = (\Sigma_I, N_I, P_I, S_I)
\]

\[
\Sigma_I = \{-, +, *, /, \%, (, )\} \cup \Sigma_{\text{var}} \cup \Sigma_{\text{num}}
\]

\[
N_I = \{X_{iexpr}\} \cup N_{\text{var}} \cup N_{\text{num}}
\]

\[
P_I = \begin{align*}
X_{iexpr} & \rightarrow (X_{iexpr}) \\
X_{iexpr} & \rightarrow -X_{iexpr} \\
X_{iexpr} & \rightarrow X_{iexpr} + X_{iexpr} | X_{iexpr} - X_{iexpr} | X_{iexpr} \times X_{iexpr} \\
X_{iexpr} & \rightarrow X_{iexpr} / X_{iexpr} | X_{iexpr} \% X_{iexpr} \\
X_{iexpr} & \rightarrow X_{\text{var}} \\
X_{iexpr} & \rightarrow X_{\text{num}} \}
\end{align*} \cup P_{\text{var}} \cup P_{\text{num}}
\]

\[
S_I = X_{iexpr}
\]
Example

See Exercise 2 on Exercise Sheet 07 for a derivation tree of $G_1$.
Grammar for Boolean Expressions

Requirements: We would like to have Boolean expressions that are very similar to Boolean terms in SMT-LIB (resp. formulas in FOL). We want an infix notation, we would like to use the symbol `!` instead of `not` (resp. `¬`) and we would like to use the symbol `&&` instead of `and` (resp. `∧`) and we would like to use the symbol `||` instead of `or` (resp. `∨`) and we would like to use the symbol `==>` instead of `=>` (resp. `→`).

\[
\mathcal{G}_B = (\Sigma_B, N_B, P_B, S_B)
\]

\[
\Sigma_B = \{!, \&\&, ||, ==>, <, >, <=, >=\} \cup \Sigma_I
\]

\[
N_B = \{X_{bexpr}\} \cup N_I
\]

\[
P_B = \{X_{bexpr} \rightarrow (X_{bexpr})\}
\]

\[
X_{bexpr} \rightarrow !X_{bexpr}
\]

\[
X_{bexpr} \rightarrow X_{bexpr} \&\& X_{bexpr} | X_{bexpr} || X_{bexpr} | X_{bexpr} == > X_{bexpr}
\]

\[
X_{bexpr} \rightarrow X_{iexpr} < X_{iexpr} | X_{iexpr} > X_{iexpr} | X_{iexpr} <= X_{iexpr} | X_{iexpr} >= X_{iexpr}
\]

\[
X_{bexpr} \rightarrow X_{bexpr} == X_{bexpr} | X_{iexpr} == X_{iexpr}
\]

\[
X_{bexpr} \rightarrow X_{var}
\]

\[
X_{bexpr} \rightarrow \text{true} | \text{false}
\]

\[
S_B = X_{bexpr}
\]
Exercise 1 on Exercise Sheet 07
Terminology

We call

- a subword that is derived from $X_{\text{var}}$ a *(program) variable*,
- a subword that is derived from $X_{\text{ieexpr}}$ or $X_{\text{bexpr}}$ an *expression*,
- a subword that is derived from $X_{\text{stmt}}$ a *(program) statement*.
Definition

A Boostan program is a triple \( P = (V, \mu, T) \) where,

- \( V \) is a set of (program) variables,
- \( \mu \) is a map that assigns each variable either \( \mathbb{Z} \) or \{true, false\}
- \( T \) is a derivation tree for the start symbol \( S_{\text{Boo}} \) in the Boostan grammar such that the translation of each expression/type to an SMT term/sort is well-sorted wrt. the map \( \mu \).

Given a variable \( v \in V \) we call \( \mu(v) \) the domain of \( v \).

Example

\( P_{ab} = (V_{ab}, \mu_{ab}, T_{ab}) \) where

- \( V_{ab} = \{a, b\} \),
- \( \mu(a) = \mathbb{Z}, \mu(b) = \mathbb{Z} \), and
- \( T_{ab} \) is the derivation tree for the text on the right.

```plaintext
1 while (!(b == 0)) {
2   if (b >= 0) {
3      b := b - 1;
4   } else {
5      b := b + 1;
6   }
7   a := a + 1;
8 }
```
Outline of the Section on Boogie and Boostan

Boogie and Boostan
Context-Free Grammars
Syntax of Boostan
Excursus: The semantics of C
Relational Semantics of Boostan
Precondition-Postcondition Pairs
Question: Do we really have to define all this stuff formally? Isn’t the meaning of a statement intuitively clear to all of us?

Answers:

▶ Maybe. Depends on your intuition.
▶ A group of programmers has a problem if at least one programmer has a different intuition.
▶ Let’s make up our own mind by looking at the following C code.

In all these examples we presume that x is a global variable.
I would guess that non-experts have to study the C standard\textsuperscript{13} for several hours in order to give definite answers.

\textsuperscript{13}E.g., ISO/IEC 9899:2011 informally called C11
Program Semantics: Motivation

Puzzle 1:

```c
1 int x;
2 ...
3 x = 5;
4 int y = x++;
```

What is the value of y? 5? 6?

Puzzle 2:

```c
1 int x;
2 ...
3 x = 5;
4 int y = f(x++);
```

```c
1 int f(int a) {
2 return a + x;
3 }
```

What is the value of y? 10? 11? 12?
Program Semantics: Motivation

Puzzle 3:

```c
int x;
...
int y = 23;
x = 5;
if (x++ >= 5 && x++ >= 6) {
    y = 42;
}
```

What is the value of y? 23? 42?

Puzzle 4:

```c
int x;
...
int y = 23;
x = 5;
if (x++ >= 6 && x++ >= 6) {
    y = 42;
}
```

What is the value of x? 5? 6? 7?
Puzzle 5:

```c
1 int f(int a) {
2     return a + x--; 
3 }
```

```c
1 int g(int a, int b) {
2     return a * b; 
3 }
```

```c
1 int x;
2 ...  
3     x = 5;  
4     int y = g(x++, f(x));
```

What is the value of y? 40? 60?
Outline of the Section on Boogie and Boostan

- Boogie and Boostan
- Context-Free Grammars
- Syntax of Boostan
- Excursus: The semantics of C
- Relational Semantics of Boostan
- Precondition-Postcondition Pairs
There are various ways to define the semantics of a programming language\textsuperscript{14}. We will define the semantics of Boostan via relations. This definition of semantics is sometimes called *relational semantics*.

\textsuperscript{14}see \url{https://en.wikipedia.org/wiki/Semantics_(computer_science)}
Idea: assign each statement a binary relation over program states.

Example

We would like to assign to the program $P_{ab}$ a relation that says “Variable $a$’s new value is the sum of the old $a$ and the absolute value of the old $b$. The new value of $b$ is zero.”

Before we can define these relations we have to formally define a program state.
Program State

Definition (Program State)

Given a program $P = (V, \mu, T)$, a program state is a map that assigns each variable $v \in V$ a value of the variable’s domain. We use $S_{V, \mu}$ to denote the set of all program states.

Example

The map that assigns the variable $a$ to 23 and the variable $a$ to 42 is an element of $S_{V_{ab}, \mu_{ab}}$.

Notation

There are several notations for maps. We can e.g. write the state above as a set of pairs $\{(a, 23), (b, 42)\}$.

- Alternatively, we can write the pairs using an arrow symbol: $\{a \mapsto 23, b \mapsto 42\}$.
- Furthermore, we can give that state a name, e.g., $s$ and define the state via the equalities $s(a) = 23$ and $s(b) = 42$. 
Sets of Program States

Notation/Convention

We will use FOL formulas to denote sets of program states.

▶ The set of variables in our formulas will be the program variables.
▶ The constant symbols, function symbols, and predicate symbols are given by the SMT theories.
▶ The model $\mathcal{M}$ is defined by the SMT theories.
▶ A formula $\varphi$ denotes that set of all program states $s$ such that for $s = \rho$ the evaluation $\llbracket \varphi \rrbracket_{\mathcal{M},\rho}$ is true.
▶ We will introduce the notation for the set of states denoted by a formula later.

Example

▶ The formula $a = 23 \land b = 42$ denotes the singleton set $\{\{a \mapsto 23, b \mapsto 42\}\} \subseteq S_{\nu_{ab},\mu_{ab}}$
▶ We will define a program semantics such that the set of states in which $P_{ab}$ can be after executing the while loop “is” $b = 0$. 
Semantics of Expressions

Idea: assign each expression an SMT formula.

Given an expression $expr$, we define the semantics of the expression, denoted $\llbracket expr \rrbracket$ as the SMT formula that is denoted by the same string.

Exception: The symbols that are not identical in Boostan and SMT formulas: integer division and modulo.

The binary division function $/$ of Boostan will be mapped to the binary division function $div$ of SMT.
The binary modulo function $\%$ of Boostan will be mapped to the binary modulo function $mod$ of SMT.

Example: $\llbracket 2 \cdot (x \mod 16) + 42 \rrbracket$ is $2 \cdot (x \mod 16) + 42$.

Convention

Since Boostan expressions and SMT formulas are so closely related, we may omit the double brackets and will often write $expr$ instead of $\llbracket expr \rrbracket$. 
Semantics of the Assignment Statement

Given a program $P = (V, \mu, T)$ we define the semantics of an assignment statement $[x := \text{expr}]$ as the following binary relation over program states.

$$\{(s_1, s_2) \in S_{V,\mu} \times S_{V,\mu} \mid [x' = [\text{expr}] \land \bigwedge_{v \in V, v \neq x} v' = v]_{M,\rho} \text{ is true} \land \rho = s_1 \cup \text{prime}(s_2)\}$$

Here, prime is the function that takes a state $s$ and returns a map where every variable $x$ in the domain of $s$ is replace by $x'$. E.g., prime($\{a \mapsto 23, b \mapsto 42\}$) is $\{a' \mapsto 23, b' \mapsto 42\}$.

Example

$[a := a + 1]$ is $\{(s_1, s_2) \mid [a' = a + 1 \land b' = b]_{M,\rho} \text{ and } \rho = s_1 \cup \text{prime}(s_2)\}$
Semantics of the Assignment Statement

Example (continued)

\[ a := a + 1 \] is \( \{(s_1, s_2) \mid \llbracket a' = a + 1 \land b' = b \rrbracket_{\mathcal{M}, \rho} \text{ and } \rho = s_1 \cup \text{prime}(s_2)\} \)

E.g., the pair of states \((s_1, s_2)\) where \(s_1 = \{a \mapsto 5, b \mapsto 1\}\) and \(s_2 = \{a \mapsto 6, b \mapsto 1\}\) is an element of this relation, because for \(\rho = s_1 \cup \text{prime}(s_2) = \{a \mapsto 5, b \mapsto 1, a' \mapsto 6, b' \mapsto 1\}\) the evaluation \(\llbracket a' = a + 1 \land b' = b \rrbracket\) is true.

Alternatively, we could write this relation as follows.
\(\{(s_1, s_2) \mid s_2(a) = s_1(a) + 1 \text{ and } s_2(b) = s_1(b)\}\).
The \textit{relational composition} of two binary relations $R_1, R_2$ over a set $X$ is defined as follows.

$R_1 \circ R_2 := \{ (x, z) \mid \text{there exists } y \in X \text{ s.t. } (x, y) \in R_1 \text{ and } (y, z) \in R_2 \}$

\textbf{Example}

Let $R_1$ and $R_2$ be the “strictly smaller” relation over $\mathbb{Z}$ (i.e., $R_i = \{ (a, b) \in \mathbb{Z} \times \mathbb{Z} \mid a < b \}$) then we have

$R_1 \circ R_2 = \{ (a, b) \in \mathbb{Z} \times \mathbb{Z} \mid a + 1 < b \}$. 
Semantics of the Concatenation of Statements

Let \( st1 \) and \( st2 \) be two statements.

We define \([st1 \ st2]\) as the relational composition \([st1] \circ [st2]\)

Example

See Exercise Sheet 08
Reminder: (Convention)

We defined the formula/term $[\text{expr}]$ for an expression expr. Since expressions and formulas are very similar we will often omit the double brackets.

Notation

Given a program $P = (V, \mu, st)$ and a formula $\varphi$ whose free variables are a subset of $V$, then we will use $\{\varphi\}$ to denote the set of states that are a satisfying assignment for $\varphi$.

$$\{\varphi\} := \{s \in S_{V,\mu} \mid [\varphi]_{M,\rho} \text{ and } \rho = s\}$$

Warning

A formula in braces like e.g., $\{\varphi\}$ denotes

- the set that contains the formula $\varphi$ (you learned that notation in school) and
- a set of states (as defined above).

We have to conclude from the context which meaning is meant.
Semantics of the If-then-else Statement

Let $expr$ be an expression and let $st1$ and $st2$ be two statements.

We define

$$\left[ \text{if}(expr)\{st1\}\text{else}\{st2\} \right] \text{ as } \left( \{expr\} \times SV,\mu \right) \cap \left[ st1 \right] \cup \left( \{!expr\} \times SV,\mu \right) \cap \left[ st2 \right]$$

Example

$$\left[ \text{if } (b \geq 0)\{b:=b-1\} \text{ else } \{b:=b+1\} \right]$$

$$\left( \{b \geq 0\} \times SV,\mu \right) \cap \left[ b:=b-1 \right] \cup \left( \{!b \geq 0\} \times SV,\mu \right) \cap \left[ b:=b+1 \right]$$

$$\left\{ (s,s') | \begin{array}{ll} (s(b) \geq 0) & (s,b') \cap \{s'(b) = s(b) - 1 \text{ and } s'(a) = s(a)\} \\ (s,b') \cap \{s(b) < 0 \text{ and } s'(b) = s(b) + 1\} \end{array} \right\}$$

$$\left\{ (s,s') | \begin{array}{ll} (s(b) \geq 0 \text{ and } s'(b) = s(b) - 1) \text{ or } (s(b) < 0 \text{ and } s'(b) = s(b) + 1) \end{array} \text{ and } s'(a) = s(a) \right\}$$
On Exercise Sheet 06 we recalled the definitions of a binary relation, reflexivity, transitivity and the reflexive transitive closure.

On these slides we will only repeat the definition of the reflexive transitive closure.
Reminder: Reflexive Transitive Closure

Given a binary relation $R$ over the set $X$, the *reflexive transitive closure*, denoted $R^*$, is the smallest relation such that $R \subseteq R^*$, $R^*$ is reflexive and $R^*$ is transitive.

Example

Let $R_1$ and $R_2$ be the “strictly smaller” relation over $\mathbb{Z}$ (i.e., $R_i = \{(a, b) \in \mathbb{Z} \times \mathbb{Z} \mid a < b\}$) then we have $R_1 \circ R_2 = \{(a, b) \in \mathbb{Z} \times \mathbb{Z} \mid a + 1 < b\}$.

We define the *identity relation* $id := \{(x, x) \mid x \in X\}$ and for $i \in \mathbb{N}$ we define

$$R^i = \begin{cases} id & \text{if } i = 0 \\ R \circ R^{i-1} & \text{otherwise} \end{cases}$$

Theorem

*The reflexive transitive closure $R^*$ is $\bigcup_{i \in \mathbb{N}} R^i$.*

(Proof not given in this course.)
Semantics of the While Statement

Let \( expr \) be an expression and let \( st \) be a statement.
We define \( \llbracket \text{while } (expr)\{st\} \rrbracket \) as
\[
((\{expr\} \times S_{V,\mu}) \cap \llbracket st \rrbracket)^* \cap (S_{V,\mu} \times \{!expr\})
\]

Example

\[
\llbracket \text{while } (x\geq 0)\{x:=x-1;y:=y+1;\} \rrbracket
\]
Let us use \( R \) to denote
\[
\{(s,s')|s(x)\geq 0 \land s'(x) = s(x) - 1 \land s'(y) = s(y) + 1\}
\]
\[
R^0 = \text{id}
\]
\[
R^1 = \{(s,s') | s(x) \geq 0 \text{ and } s'(x) = s(x) - 1 \text{ and } s'(y) = s(y) + 1\}
\]
\[
R^2 = \{(s,s') | s(x) \geq 1 \text{ and } s'(x) = s(x) - 2 \text{ and } s'(y) = s(y) + 2\}
\]
\[
\vdots
\]
\[
R^* = \{(s,s') | s = s' \text{ or } (s(x) > s'(x) \geq -1 \text{ and } s'(y) - s(y) = s(x) - s'(x)) \}
\]
\[
R^* \cap (S_{V,\mu} \times \{!x\geq 0\}) = \{(s,s') | (s = s' \text{ and } s'(x) < 0)\text{ or } (s(x) > s'(x) = -1 \text{ and } s'(y) - s(y) = s(x) + 1)\}
\]
Reminder

Idea: assign each statement a binary relation over program states.

Example

```plaintext
while (!b == 0)) {
    if (b >=0) {
        b := b - 1;
    } else {
        b := b + 1;
    }
    a := a + 1;
}
```

We would like to assign to the program $P_{ab}$ a relation that says “Variable $a$’s new value is the sum of the old $a$ and the absolute value of the old $b$. The new value of $b$ is zero.”

On the next slide we compute the relation of the example above.
\[ b := b - 1 \] = \{ (s, s') \mid [b' = b - 1 \land a' = a]_{\mathcal{M}, \rho} = \text{true} \text{ and } \rho = s \cup \text{prime}(s') \}
\quad = \{ (s, s') \mid s'(b) = s(b) - 1 \text{ and } s'(a) = s(a) \}

\[ b := b + 1 \] = \{ (s, s') \mid s'(b) = s(b) + 1 \text{ and } s'(a) = s(a) \}

\[ a := a + 1 \] = \{ (s', s'') \mid s''(a) = s'(a) + 1 \text{ and } s''(b) = s'(b) \}

\[ \text{if/else} \] = \{ b >= 0 \} \times S_{\mathcal{V}, \mu} \cap [b := b - 1] \cup \{ !b >= 0 \} \times S_{\mathcal{V}, \mu} \cap [b := b + 1]
\quad = \{ (s, s') \mid s'(a) = s(a) \text{ and } ((s(b) \geq 0 \text{ and } s'(b) = s(b) - 1) \text{ or } (s(b) < 0 \text{ and } s'(b) = s(b) + 1)) \}

\[ \text{loop body} \] = \{ (s, s'') \mid \text{ex. } s' \text{ s.t. } (s, s') \in [\text{if/else}], (s', s'') \in [a := a + 1] \}
\quad = \{ (s, s'') \mid s''(a) = s(a) + 1 \text{ and } ((s(b) \geq 0 \text{ and } s''(b) = s(b) - 1) \text{ or } (s(b) < 0 \text{ and } s''(b) = s(b) + 1)) \}

\[ P_{ab} \] = \left( (\{ !(b == 0) \} \times S_{\mathcal{V}, \mu}) \cap [\text{loop body}] \right)^* \cap (S_{\mathcal{V}, \mu} \times \{ !!(b == 0) \})
\quad = \{ (s, s') \mid s(b) \neq 0 \text{ and } s'(a) = s(a) + 1 \text{ and } |s'(b)| = |s(b)| - 1 \}
\quad \quad \quad \quad \quad \quad \cap (S_{\mathcal{V}, \mu} \times \{ !!(b == 0) \})
\quad = \{ (s, s') \mid s'(a) + |s'(b)| = s(a) + |s(b)| \text{ and } |s'(b)| \leq |s(b)| \}
\quad \quad \quad \quad \quad \quad \cap (S_{\mathcal{V}, \mu} \times \{ !!(b == 0) \})
\quad = \{ (s, s') \mid s'(a) = s(a) + |s(b)| \text{ and } s'(b) = 0 \}
Example

See Exercise 1 on Exercise Sheet 07 for another example where we compute the relation of a program.
Outline of the Section on Boogie and Boostan

Boogie and Boostan
Context-Free Grammars
Syntax of Boostan
Excursus: The semantics of C
Relational Semantics of Boostan
Precondition-Postcondition Pairs
How can we specify correctness of a Boostan program?

- Now: precondition-postcondition pairs.
- Later: extend Boostan by assert statements.
Given a program $P = (V, \mu, st)$ and a pair of sets of states ($\{\varphi_{\text{pre}}\}, \{\varphi_{\text{post}}\}$) that we call precondition-postcondition pair, we want to define the following formally. Whenever we run $st$ in some state where $\varphi_{\text{pre}}$ holds and the execution of $st$ has come to an end, then we are in some state where $\varphi_{\text{post}}$ holds.

**Definition**

We say that program $P$ satisfies the precondition-postcondition pair ($\{\varphi_{\text{pre}}\}, \{\varphi_{\text{post}}\}$) if the inclusion $\text{post}(\{\varphi_{\text{pre}}\}, \llbracket st \rrbracket) \subseteq \{\varphi_{\text{post}}\}$ holds.

**Example**

```java
while (!(b == 0)) {
  if (b >= 0) {
    b := b - 1;
  } else {
    b := b + 1;
  }
  a := a + 1;
}
```

Does $P_{ab}$ satisfy the precondition-postcondition pair ($\{a \cdot b \geq 0\}, \{a \geq 0\}$)?
Post Image

**Definition**
Given a binary relation \( R \) over the set \( X \) and a subset of \( Y \subseteq X \), the *postimage of \( Y \) under \( R \)*, denoted \( \text{post}(Y, R) \), is the set 
\[
\{ x \in X \mid \text{exists } y \in Y \text{ such that } (y, x) \in R \}
\]

**Example**
Let \( R \) be the “strictly smaller” relation over \( \mathbb{Z} \) (i.e., 
\( R = \{(a, b) \in \mathbb{Z} \times \mathbb{Z} \mid a < b\} \)) and \( Y = \{y \in \mathbb{Z} \mid y \geq 5\} \) then 
\[
\text{post}(Y, R) = \{y \in \mathbb{Z} \mid y \geq 6\}
\]
Precondition-Postcondition Pairs

Example

1 while (!(b == 0)) {
2   if (b >= 0) {
3     b := b - 1;
4   } else {
5     b := b + 1;
6   }
7   a := a + 1;
8 }

Does $P_{ab}$ satisfy the precondition-postcondition pair ($\{a \geq 0\}$, $\{a \geq 0\}$)?

Check $\text{post}((\{a \geq 0\}, [st])) ? \subseteq \{a \geq 0\} !$

$[st] = \{(s, s') | s'(a) = s(a) + |s(b)| \text{ and } s'(b) = 0\}$

Example

See Exercise 2 on Exercise Sheet 08 for more examples.
Program Verification

Summer Term 2021

Lecture 10: Hoare Proof System

Matthias Heizmann

Wednesday 19th May
Section 7

Hoare Proof System
In this section we will learn to prove correctness of programs.

In more detail:

- We set up a proof system that helps us to show that a program satisfies a given precondition-postcondition pair.
- We will give a formal proof that the proof system is suitable for this task.
Outline of the Section on Hoare Proof System

Introduction
Rules of the Hoare Proof System
Soundness of the Hoare Proof System
At the beginning of this course, we used the $\mathcal{N}_{\text{PL}}$ proof system to derive valid implications of the form $\Gamma \models F$. In this section we will see a proof system that allows us analogously to derive program statements together with a precondition-postcondition pair such that the program satisfies this precondition-postcondition pair.

This proof system was proposed by the computer scientist Tony Hoare and hence we call it “Hoare proof system”.

Next,

1. we will first define the term Hoare triple,
2. see all rules of the Hoare proof system,
3. define the term “derivation” (analogously to a derivation in $\mathcal{N}_{\text{PL}}$),
4. and discuss then each rule in more detail.
Hoare Triple

**Definition (Hoare Triple)**

Given a set of states \( \{ \varphi \} \), a program statement \( st \) and a set of states \( \{ \psi \} \), we call the triple \( \{ \varphi \} st \{ \psi \} \) a **Hoare triple**.

We call a Hoare triple \( \{ \varphi \} st \{ \psi \} \) **valid** if \( st \) satisfies the precondition-postcondition pair \( (\{ \varphi \}, \{ \psi \}) \).

**TODO** Example of a Hoare triple that is valid
**TODO** Example of a Hoare triple that is not valid
Proof Systems of this Course

$\mathcal{N}_{PL}$
proof system for deriving valid PL implications
$\Gamma \vdash F$

$\mathcal{N}_{FOL}$
proof system for deriving valid FOL implications
$\Gamma \vdash \varphi$

Hoare proof system
proof system for deriving valid Hoare triples
$\{\varphi\} st \{\psi\}$
Outline of the Section on Hoare Proof System

Introduction

Rules of the Hoare Proof System

Soundness of the Hoare Proof System
Rules of the Hoare Proof System – Overview

Assignment axiom

\[(\text{assign})\) \{\varphi[x \mapsto \text{expr}]\} x := \text{expr}; \{\varphi\}\]

Composition rule

\[(\text{compo})\) \{\varphi_1\} \text{st}_1\{\varphi_2\} \{\varphi_2\} \text{st}_2\{\varphi_3\} \Rightarrow \{\varphi_1\} \text{st}_1\text{st}_2\{\varphi_3\}\]

Strengthen precondition rule

\[(\text{strepre})\) \{\varphi\} \text{st}\{\psi\} \Rightarrow \{\varphi'\} \text{st}\{\psi\} \text{ if } \varphi' \models \varphi\]

Weaken postcondition rule

\[(\text{weakpos})\) \{\varphi\} \text{st}\{\psi\} \Rightarrow \{\varphi\} \text{st}\{\psi'\} \text{ if } \psi \models \psi'\]

Conditional rule

\[(\text{condi})\) \{\varphi \land \text{expr}\} \text{st}_1\{\psi\} \{\varphi \land \neg\text{expr}\} \text{st}_2\{\psi\} \Rightarrow \{\varphi\} \text{if}(\text{expr})\{\text{st}_1}\text{else}\{\text{st}_2\}\{\psi\}\]

While rule

\[(\text{while})\) \{\varphi \land \text{expr}\} \text{st}\{\varphi\} \Rightarrow \{\varphi\} \text{while}(\text{expr})\{\text{st}\}\{\varphi \land \neg\text{expr}\}\]
We define a *derivation* as a tree whose nodes are labelled by Hoare triples such that the following holds.

If a node that is labelled by a Hoare triple \(\{\varphi_{n+1}\}st_{n+1}\{\psi_{n+1}\}\) has children that are labelled by Hoare triples \(\{\varphi_1\}st_1\{\psi_1\}\), \(\ldots\), \(\{\varphi_n\}st_n\{\psi_n\}\), then

\[
\frac{\{\varphi_1\}st_1\{\psi_1\} \quad \ldots \quad \{\varphi_n\}st_n\{\psi_n\}}{\{\varphi_{n+1}\}st_{n+1}\{\psi_{n+1}\}}
\]

must be an instance of some rule.

Note that this means in particular that leafs of the tree may only be labelled instances of the assignment axiom.

**Theorem (Soundness of the Hoare Proof System)**

*If there is a derivation whose root is labelled by \(\{\varphi\}st\{\psi\}\), then the statement st satisfies the precondition-postcondition pair \((\{\varphi\}, \{\psi\})\).*

Proof. Later, in the last subsection of this section.
The following four rules are sufficient to solve Exercise 3 of Exercise Sheet 09.

- Assignment axiom
- Composition rule
- Strengthen precondition rule
- Weaken postcondition rule

We next discuss the remaining two rules in more detail.
Conditional Rule

\[ (\text{condi}) \frac{\{ \varphi \land \text{expr} \} \ st_1 \ {\psi} \quad \{ \varphi \land \neg\text{expr} \} \ st_2 \ {\psi} }{\{ \varphi \} \ 	ext{if}(\text{expr})\{st_1\}\text{else}\{st_2\} \ {\psi}} \]

Example

\[
\{ x=y \land y\geq 0 \} y:=y-1; \{ (x\geq 0 \rightarrow y=x-1) \land (x<0 \rightarrow y=x+1) \} \\
\{ x=y \land \neg(y\geq 0) \} y:=y+1; \{ (x\geq 0 \rightarrow y=x-1) \land (x<0 \rightarrow y=x+1) \}
\]

\[
\{ y = x \} \text{if}(y>0)\{y:=y-1;\}\text{else}\{y:=y+1;\}\{ (x\geq 0 \rightarrow y=x-1) \land (x<0 \rightarrow y=x+1) \}
\]

Note that for both Hoare triples above the line the postcondition contains one conjunct that seems to be useless. Indeed, these conjuncts are “only” needed to obtain the postcondition for the Hoare triple below the line.
While Rule

\[
(\text{while}) \quad \begin{cases} \varphi \land \text{expr} \rightarrow \text{st} \rightarrow \varphi \\ \varphi \rightarrow \text{while}(\text{expr}) \rightarrow \text{st} ightarrow \varphi \land \neg \text{expr} \end{cases}
\]

We call the formula \( \varphi \) an \textit{inductive loop invariant}.

\begin{example}

Task: Show that the while loop \texttt{while(x>0)}\{\texttt{x:=x-1;y:=y+1;}\} satisfies the precondition-postcondition pair (\{\texttt{z = x + y \land x \geq 0}\}, \{\texttt{z = y}\}).

Solution:

\[
\begin{align*}
\{ \varphi \land \text{expr} \} & \xrightarrow{\text{while}} \text{st} \rightarrow \varphi \\
\{ \varphi \} & \xrightarrow{\text{weakpos}} \text{while}(\text{expr}) \rightarrow \text{st} \rightarrow \varphi \land \neg \text{expr}
\end{align*}
\]

Typical for a derivation in which we use the while rule:

- We have to combine the while rule with the rules (strepre) and (weakpos).
- The conjunction of the negated condition and the inductive loop invariant restrict some variable to a certain value (here \( x = 0 \)).
\end{example}
**Hoare Proof System – Example**

**Task:** prove that $P_{ab}$ satisfies the precondition-postcondition pair $(\{ a \geq 42 \land b \leq -23 \}, \{ a \geq 53 \})$.

We use $\varphi_l$ as an abbreviation for the formula $b \leq 0 \rightarrow a - b \geq 53$.

\[
\begin{array}{ll}
\{ b \leq 0 \rightarrow a - (b-1) \geq 52 \} & \text{(assig)} \\
\{ a - (b-1) \geq 52 \} & \\
\{ \varphi_l \land b \geq 0 \} & \text{(stpre)} \\
\{ \varphi_l \land -b \geq 0 \} & \text{(condi)} \\
\{ \varphi_l \} & \text{if } (b \geq 0) \{ \text{b:=b-1;} \} \text{ else } \{ \text{b:=b+1;} \} \\
\{ b \leq 0 \rightarrow a - b \geq 52 \} & \text{if } (b \geq 0) \{ \text{b:=b-1;} \} \text{ else } \{ \text{b:=b+1;} \} a:=a+1; \{ \varphi_l \} & \text{(compo)} \\
\{ (\varphi_l) \land -(b=0) \} & \text{if } (b \geq 0) \{ \text{b:=b-1;} \} \text{ else } \{ \text{b:=b+1;} \} a:=a+1; \{ \varphi_l \} & \text{(stpre)} \\
\{ a \geq 42 \land b \leq -23 \} & \text{while}(!(b==0)) \{ \text{if } (b \geq 0) \{ \text{b:=b-1;} \} \text{ else } \{ \text{b:=b+1;} \} a:=a+1; \} \{ (\varphi_l) \land -(b=0) \} & \text{(weakpos)} \\
\end{array}
\]

where (*) is the following subtree

\[
\begin{array}{ll}
\{ b \leq 0 \rightarrow a - b \geq 53 \} & \text{(assig)} \\
\{ a - b \geq 53 \} & \\
\{ b \leq 0 \rightarrow a - b \geq 52 \} & \text{(stpre)} \\
\{ a - b \geq 52 \} & \\
\end{array}
\]
Example

See Exercise 5 on Exercise Sheet 09 for another examples of a derivation.
Outline of the Section on Hoare Proof System

Introduction
Rules of the Hoare Proof System
Soundness of the Hoare Proof System
This last subsection is dedicated to the proof of the theorem that states that every derived Hoare triple is indeed valid.

We follow the typical approach for proving a theorem about derivations of a proof system:

- we state a property of proof rules (here: soundness)
- we prove that each proof rule has this property
- we conclude via induction that the theorem holds
Reminder: **Theorem (Soundness of the Hoare Proof System)**

If there is a derivation whose root is labelled by \{φ\}st{ψ} then the statement st satisfies the precondition-postcondition pair (\{φ\}, \{ψ\}).

Reminder: **Definition (Derivation)**

We define a *derivation* as a tree whose nodes are labelled by Hoare triples such that the following holds. If a node that is labelled by a Hoare triple \{φ_{n+1}\}st_{n+1}\{ψ_{n+1}\} has children that are labelled by Hoare triples \{φ_1\}st_1\{ψ_1\} \ldots \{φ_n\}st_n\{ψ_n\}, then

\[
\frac{\{φ_1\}st_1\{ψ_1\} \ldots \{φ_n\}st_n\{ψ_n\}}{\{φ_{n+1}\}st_{n+1}\{ψ_{n+1}\}}
\]

must be an instance of some rule.

**Definition (Sound Rule)**

We call a rule of the form

\[
\frac{\{φ_1\}st_1\{ψ_1\} \ldots \{φ_n\}st_n\{ψ_n\}}{\{φ_{n+1}\}st_{n+1}\{ψ_{n+1}\}}
\]

*sound* if the following holds. If for all \(i \in \{1, \ldots n\}\) the Hoare triple \{φ_i\}st_i\{ψ_i\} is valid, then the Hoare triple \{φ_{n+1}\}st_{n+1}\{ψ_{n+1}\} is also valid.
Soundness of the Assignment Axiom

Lemma (Soundness of the Assignment Axiom)

The Hoare triple \( \{\varphi[x \mapsto \text{expr}]\} \ x := \text{expr}; \ \{\varphi\} \) is valid.

Reminder

\[ [x := \text{expr}] \text{ is } \{(s_1, s_2) \in S_{\nu, \mu} \times S_{\nu, \mu} \mid \begin{align*} &\\\ &\end{align*} ] \begin{aligned} &\begin{array}{c} [x' = [\text{expr}] \land \land_{v \in \nu, v \neq x} v' = v]_{M, \rho} \text{ is true} \end{array} \\
\end{aligned} \]

and \( \rho = s_1 \cup \text{prime}(s_2) \)\]

Proof. Let \( s' \in \text{post}(\{\varphi[x \mapsto \text{expr}]\}, [x := \text{expr};]) \)

\( \Rightarrow \) There exists \( s \) such that \( s \in \{\varphi[x \mapsto \text{expr}]\} \) and \((s, s') \in [x := \text{expr};]\) \( \Rightarrow \) There exists \( s \) such that for \( \rho = s \cup \text{prime}(s') \)

\[ [\varphi[x \mapsto \text{expr}] \land x' = [\text{expr}] \land \land_{v \in \nu, v \neq x} v' = v]_{M, \rho} \text{ is true} \]

\( \Rightarrow \) for \( \rho = s' \) the evaluation \( [\varphi]_{M, \rho} \text{ is true} \)

\( \Rightarrow s' \in \{\varphi\} \)
Soundness of the Composition Rule

Lemma (Soundness of the Composition Rule)

If the Hoare triple $\{\varphi_1\} st_1 \{\varphi_2\}$ is valid and the Hoare triple $\{\varphi_2\} st_2 \{\varphi_3\}$ is valid, then the Hoare triple $\{\varphi_1\} st_1 st_2 \{\varphi_3\}$ is valid.

Proof. See Exercise 1 on Exercise Sheet 10.
Soundness of the Strengthen Precondition Rule

\[(strepre) \quad \frac{\{\varphi\}st\{\psi\}}{\{\varphi'\}st\{\psi\}} \text{ if } \varphi' \models \varphi\]

**Lemma (Soundness of the Strengthen Precondition Rule)**

If the Hoare triple \(\{\varphi\}st\{\psi\}\) is valid and the side condition \(\varphi' \models \varphi\) is valid, then the Hoare triple \(\{\varphi'\}st\{\psi\}\) is valid.

**Proof.** Let \(s' \in post(\{\varphi'\}, \llbracket st \rrbracket)\)

\[\Rightarrow \text{ There exists } s \text{ such that } s \in \{\varphi'\} \text{ and } (s, s') \in \llbracket st \rrbracket.\]
\[\Rightarrow \text{ There exists } s \text{ such that } s \in \{\varphi\} \text{ and } (s, s') \in \llbracket st \rrbracket.\]
\[\Rightarrow s' \in post(\{\varphi\}, st)\]
\[\Rightarrow s' \in \{\psi\} \quad (\text{because } post(\{\varphi\}, st) \subseteq \{\psi\})\]
Program Verification

Summer Term 2021

Lecture 12: Hoare Proof System cont’d, Ultimate Referee, Arrays

Matthias Heizmann

Wednesday 2nd June
Soundness of the Weakening Postcondition Rule

\[(\text{weakpos}) \quad \frac{\{\varphi\} \text{st}\{\psi\}}{\{\varphi\} \text{st}\{\psi'\}} \quad \text{if} \quad \psi \models \psi'
\]

**Lemma (Soundness of the Weakening Postcondition Rule)**

If the Hoare triple \(\{\varphi\} \text{st}\{\psi\}\) is valid and the side condition \(\psi \models \psi'\) is valid, then the Hoare triple \(\{\varphi\} \text{st}\{\psi'\}\) is valid.

**Proof.** See Exercise 2 on Exercise Sheet 11.
Soundness of the Conditional Rule

Lemma (Soundness of the Conditional Rule)

If the Hoare triple \( \{ \varphi \land expr \} \ st_1 \ {\psi} \) is valid and the Hoare triple \( \{ \varphi \land \neg expr \} \ st_2 \ {\psi} \) is valid, then the Hoare triple \( \{ \varphi \} \ if(expr)\{st1\}\ else\{st2\} \ {\psi} \) is valid.

Proof. See Exercise 3 on Exercise Sheet 11.
Soundness of the While Rule

Lemma (Soundness of the While Rule)

If the Hoare triple $\{\varphi \land expr\} st \{\varphi\}$ is valid, then the Hoare triple $\{\varphi\} \text{while}(expr)\{st\} \{\varphi \land \neg expr\}$ is valid.
Proof. Let \( s' \in post(\{ \varphi \}, [\text{while (expr)}\{ st \}]) \), i.e. there exists an \( s \in \{ \varphi \} \) such that

\[
(s, s') \in [\text{while (expr)}\{ st \}] = (\{ \text{expr} \} \times S_{V, \mu}) \cap [st])^* \cap (S_{V, \mu} \times \{ !\text{expr} \}).
\]

Therefore we know that \( s' \in \{ !\text{expr} \} \).

Let \( R = (\{ \text{expr} \} \times S_{V, \mu}) \cap [st]) \). It holds that \( R^* = \bigcup_{n \in \mathbb{N}_0} R^n \). Thus there exists some \( n \in \mathbb{N}_0 \) such that \( (s, s') \in R^n \).

By induction over \( n \), we show that \( s' \in \{ \varphi \} \). By the observation above, it follows that \( s' \in \{ \varphi \land \neg \text{expr} \} \). Thus we will have proven that

\[
post(\{ \varphi \}, [\text{while (expr)}\{ st \}]) \subseteq \{ \varphi \land \neg \text{expr} \}
\]

and thus the While Rule is valid.
\[ n = 0 \] We have \((s, s') \in R^0 = id = \{(s, s') \in S_{V, \mu} \times S_{V, \mu} \mid s = s'\}\).
Hence \(s' = s\), and \(s \in \{\varphi\}\) by assumption.

\[ n \rightarrow n + 1 \] Assume as induction hypothesis (IH) that for all \((\tilde{s}, \tilde{s}') \in R^n\) with \(\tilde{s} \in \{\varphi\}\), it holds that \(\tilde{s}' \in \{\varphi\}\).

In our case, \((s, s') \in R^{n+1} = R^n \circ R\). Thus by definition of composition, there exists some \(s''\) such that \((s, s'') \in R^n\) and \((s'', s') \in R\).

- From the first tuple we derive by (IH) that \(s'' \in \{\varphi\}\).
- From the second tuple and the definition of \(R\), it follows that \(s'' \in \{\text{expr}\}\) and \((s'', s') \in \llbracket st\rrbracket\).

Hence it follows that \(s' \in post(\{\varphi \land \text{expr}\}, \llbracket st\rrbracket)\). By validity of the Hoare triple \(\{\varphi \land \text{expr}\} \; st \; \{\varphi\}\), we have \(post(\{\varphi \land \text{expr}\}, \llbracket st\rrbracket) \subseteq \{\varphi\}\). Thus we conclude \(s' \in \{\varphi\}\).
Soundness of the Hoare Proof System

Reminder: Theorem (Soundness of the Hoare Proof System)

If there is a derivation whose root is labelled by $\{\varphi\}st\{\psi\}$ then the statement $st$ satisfies the precondition-postcondition pair $(\{\varphi\}, \{\psi\})$

Proof. By definition a Hoare triple, $\{\varphi\}st\{\psi\}$ is valid iff $st$ satisfies the precondition-postcondition pair $(\{\varphi\}, \{\psi\})$. We prove by induction over the height of the derivation that the root node of a derivation is always labelled by a valid Hoare triple.

Induction hypothesis (IH): For all derivations of height "$\leq n$" the root node is labelled by a valid Hoare triple.

Base case $n = 0$. The derivation consists of a single node, labelled by a Hoare triple $\{\varphi\}st\{\psi\}$. By definition of a derivation, $\{\varphi\}st\{\psi\}$ has to be an instance of some rule. The only rule of this form is the assignment axiom. From the lemma on Soundness of the Assignment Axiom we conclude that (IH) holds.
Soundness of the Hoare Proof System

Induction step $n \leadsto n + 1$. Let $\{\varphi_{m+1}\}st_{m+1}\{\psi_{m+1}\}$ be the label of the root node and $\{\varphi_1\}st_1\{\psi_1\} \ldots \{\varphi_m\}st_m\{\psi_m\}$ be the labels of the root node’s children. Each child is the root node of derivation of height $\leq n$ and from IH we conclude that it is labelled by a valid Hoare triple. By definition of a derivation,

$$
\frac{\{\varphi_1\}st_1\{\psi_1\} \ldots \{\varphi_m\}st_m\{\psi_m\}}{\{\varphi_{m+1}\}st_{m+1}\{\psi_{m+1}\}}
$$

must be an instance of some rule. The rules of this form are the composition rule, the strengthen precondition rule, the weaken postcondition rule, the conditional rule, and the while rule. For each of these rules one of the lemmas of this subsection lets us conclude that $\{\varphi_{m+1}\}st_{m+1}\{\psi_{m+1}\}$ is a valid hoare triple and hence IH also holds for $n+1$. ■
Section 8

Ultimate Referee
In this section we will partially automatize the task of checking correctness.

In this section, we will learn to

- systematically construct derivations in the Hoare proof system if suitable loop invariants are given
- use the Ultimate Referee tool check if given loop invariants are suitable to proof correctness
Guide for Finding a Derivation in the Hoare Proof System
At a first glance it looked like constructing a derivation involves a lot of guessing.

After a closer look it became clear that there is only one rule for each kind of statement and we only have to guess the loop invariant of the while rule and where to put in strepre and weakpos rules.

The following guide teaches us how we can reduce the guesswork to finding suitable loop invariants for the while rule.

Note that however finding a suitable loop invariant is usually the hardest part of the task. This guide just helps us to get the minor obstacles out of the way and helps us to face the real challenge directly.
Guide for Finding a Derivation in the Hoare Proof System

1. Guess “good” loop invariants for all loops

2. Use (weakpos) only for equivalence transformations
   equivalence transformations are sometimes needed to bring a formula syntactically in a form that is required by (condi) or (while)

3. Process sequential composition from right to left

4. Strengthen the precondition (strictly) only before loop invariants

5. Apart from that: use the (strepre) and (weakpos) rules only for equivalence transformations
Finding a derivation usually involves a lot of backtracking. We find out very late that our loop invariants were not sufficient and have to start again. I would be nice, if we could focus on the guesswork and let a computer do everything that can be done algorithmically. (See tool in next subsection.)
Ultimate Referee is a tool for checking loop invariants.

- Takes as input:
  - program where each loop is annotated by a formula (the potential loop invariants) and
  - a correctness specification (e.g., a precondition-postcondition pair)

Checks if there is some derivation in the Hoare proof system where the formulas are loop invariants of the respective while rules.

- Implemented in the Ultimate framework
- Source code available at GitHub.
- Available via a web interface.
We use the keyword `invariant` in each while loop to state our candidate invariants.

Here our candidate invariant is `invariant y == 0;`

The output of the tool tells us that our candidate invariant is too strong:

Annotation is not valid for all loop-free paths from entry of procedure main to loop head at line 7. One counterexample starts in i=1, j=2 and ends in i=1, j=2, x=1, y=2.
Ultimate Referee: Outlook

Ultimate Referee was not only build to support students who are constructing derivations in the Hoare proof system...

- Check results of other verification tools.
- Assume you verify your code with verification tool XYZ. Verification tool XYZ says that your code is correct. Do you trust verification tool XYZ?
- Let verification tool XYZ output all loop invariants and double check its result with Ultimate Referee.

Slightly different than the witness validation sigsoft/0001DDHS15; sigsoft/0001DDH16 implemented in Ultimate. The witness validator is rather lenient and tries to complete proofs that are incomplete.
Section 9

Arrays
In this section we will add support for arrays to our formal setting.

Our goals:

▶ Learn about the SMT theory of arrays.
▶ Get familiar with Boogie’s notion of arrays (arrays as maps)
▶ Add support for arrays to the Boostan language.
▶ Add support for this revised Boostan language to the Hoare proof system.
Outline of the Section on Arrays

Motivation for Adding new Features

- Arrays as Mathematical Objects
- The SMT Theory of Arrays
- Arrays in Boogie
- Arrays in Boostan
The next slides motivates the need for an SMT theory of arrays.

The diagram contrasts the approach of this lecture with the approach of the Ultimate Automizer verification tool (which we discuss later in this course).

- The verification algorithms of the Ultimate Automizer tool are not (directly) implemented for high-level programming languages. Instead, the tool translates high-level programming languages to the Boogie language. Boogie was designed such that it is closely related to SMT-LIB. Hence, the tool can delegate several sub-tasks to SMT solvers.

- In this lecture, we do not study high-level programming languages. Instead, we take basic features of high-level programming languages and add support for these features to the Boostan language. We design Boostan such that it is closely related to SMT. Hence, we can resort to SMT while defining its semantics.

Arrays are an basic feature of high-level programming languages, hence we want to have SMT support for arrays.
Arrays – Motivation

Approach of the Ultimate Automizer verification tool.

High-level imperative programming language. E.g., C.

translate → Boogie → do computations → SMT

Approach in this lecture.

High-level imperative programming language. E.g., C.

take basic features → Boostan → define semantics → SMT

We need logical formulas whose models are (also) arrays!
Outline of the Section on Arrays

Motivation for Adding new Features

Arrays as Mathematical Objects

The SMT Theory of Arrays

Arrays in Boogie

Arrays in Boostan
In school you did some math were the objects were numbers (e.g., natural numbers, reals) or shapes (triangles, circles).

Now, we would now like to do some math were the studied objects are array-like. On one hand, the objects have to be so rich that they are suitable to model arrays of computer programs. On the other hand, the objects have to be so simple that the reasoning can be implemented in tools like e.g., SMT solvers.
Problem: Arrays are modifiable.

Ideas: Consider an array as a map. Consider an array update as an operation that takes a map and returns a modified map.

**Example (that demonstrates this idea)**

- Let $f_{foo}$ be the map such that $f_{foo}(x) = 0$ for all $x$.
- $f_{foo}$ represents a zero-initialized array.
- After writing the number 23 at index 5 that array is represented by the map $f_{bar}$

\[
 f_{bar}(x) = \begin{cases} 
 23 & \text{if } x = 5 \\
 0 & \text{otherwise} 
\end{cases}
\]

We use two functions to implement this idea.

<table>
<thead>
<tr>
<th>select</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary function</td>
<td>ternary function</td>
</tr>
<tr>
<td>1st argument: a map</td>
<td>1st argument: a map</td>
</tr>
<tr>
<td>2nd argument: element of map’s domain</td>
<td>2nd argument: element of map’s domain</td>
</tr>
<tr>
<td>returns: value of map at that position</td>
<td>3rd argument: new value at that position</td>
</tr>
<tr>
<td>e.g. $select(f_{foo}, 5) = 0$</td>
<td>e.g. $store(f_{foo}, 5, 23) = f_{bar}$</td>
</tr>
</tbody>
</table>
Next we compare the theory of arrays that we are going to define with the theory of integers.

Note that the “absolute value” is a function in models of the theory of integers, but can also be an element of the interpretation domain in the theory of arrays.
The theory of arrays in comparison to the theory of integers:

<table>
<thead>
<tr>
<th>Values</th>
<th>Theory of Integers</th>
<th>Theory of Arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbers, e.g.,</td>
<td>Numbers, e.g.,</td>
<td>1-ary maps, e.g.,</td>
</tr>
<tr>
<td>▶ 23</td>
<td>▶ 23</td>
<td>▶ $f_{\text{foo}}$</td>
</tr>
<tr>
<td>▶ 42</td>
<td>▶ 42</td>
<td>▶ $f_{\text{bar}}$</td>
</tr>
<tr>
<td>▶ -17</td>
<td>▶ -17</td>
<td>▶ absolute value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ +</td>
<td></td>
<td>▶ select</td>
</tr>
<tr>
<td>▶ -</td>
<td></td>
<td>▶ store</td>
</tr>
<tr>
<td>▶ *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>▶ abs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Outline of the Section on Arrays

Motivation for Adding new Features
Arrays as Mathematical Objects
The SMT Theory of Arrays
Arrays in Boogie
Arrays in Boostan
Analogously to our introduction of various SMT theories in the section on First-Order Theories we introduce the theory of arrays.

As an exercise, we should ask ourselves: How can we define the theory of arrays formally? Which symbols and axioms are needed?
Theory of Arrays $T_{arr}$

Signature:

$\Sigma_{arr} : \{select, \ store, =\}$

Axioms:

1. the axioms of reflexivity, symmetry, and transitivity of $T_=$
2. array congruence

$$\forall a, i, j. \ i = j \rightarrow select(a, i) = select(a, j)$$

3. read-over-write 1

$$\forall a, v, i, j. \ i = j \rightarrow select(store(a, i, v), j) = v$$

4. read-over-write 2

$$\forall a, v, i, j. \ i \neq j \rightarrow select(store(a, i, v), j) = select(a, j)$$

5. extensionality

$$\forall a, b. \ (\forall i. \ select(a, i) = select(b, i)) \leftrightarrow a = b$$
The SMT-LIB definition of the theory of arrays can be found at the SMT-LIB website \(^{15}\). We will not discuss details and only look at an example (next slide).

Reminder: SMT-LIB is based on a sorted version of first-order logic. Hence, we have to specify a sort for each variable. The sort of an array whose indices are integers and whose values are Booleans is denoted by \((\text{Array Int Bool})\).

See Exercise Sheet 11 for more examples.
Arrays in SMT-LIB

Some SMT formula with symbols from the theory of arrays.

\[ a = \text{store}(b, k, v) \land \text{select}(a, i) \neq \text{select}(b, i) \land \text{select}(a, j) \neq \text{select}(b, j) \land i \neq j \]

Some SMT script for checking satisfiability of this formula.

```
1  (set-logic QF_ALIA)
2  (declare-fun i () Int)
3  (declare-fun j () Int)
4  (declare-fun k () Int)
5  (declare-fun v () Int)
6  (declare-fun a () (Array Int Int))
7  (declare-fun b () (Array Int Int))
8  (assert (= b (store a k v)))
9  (assert (not (= (select b i) (select a i))))
10 (assert (not (= (select b j) (select a j))))
11 (check-sat)
12 (get-value (k i j))
13 (assert (not (= j i)))
14 (check-sat)
```
Program Verification
Summer Term 2021
Lecture 13: Arrays cont’d

Matthias Heizmann

Monday 7th June
Outline of the Section on Arrays

Motivation for Adding new Features
Arrays as Mathematical Objects
The SMT Theory of Arrays
Arrays in Boogie
Arrays in Boostan
Arrays in Boogie are very similar to arrays in SMT-LIB. An array is a (total) map that assigns each element of the index domain and element of the value domain.

In this course we will use examples to briefly demonstrate the syntax and semantics of Boogie’s arrays, details can be found in the Boogie specification\textsuperscript{16} \texttt{leino\_this\_2016}.

See Exercise Sheet 11 and Exercise Sheet 12 for more examples.

\textsuperscript{16}\url{https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/}
Arrays in Boogie

Implementation of an insertion sort\(^\text{17}\) algorithm in Boogie:

```
procedure InsertionSort(lo : int, hi : int, a : [int]int) returns (ar : [int]int)
{
    var i, j, temp : int;
    ar := a;
    i := lo+1;
    while (i <= hi) {
        j := i;
        while (j > lo && ar[j] < ar[j-1])
        {
            temp := ar[j-1];
            ar[j-1] := ar[j];
            ar[j] := temp;
            j := j-1;
        }
        i := i+1;
    }
}
```

\(^\text{17}\)https://en.wikipedia.org/wiki/Insertion_sort
TODO say something about modeling memory via arrays
Outline of the Section on Arrays

- Motivation for Adding new Features
- Arrays as Mathematical Objects
- The SMT Theory of Arrays
- Arrays in Boogie
- Arrays in Boostan
In this subsection we will add support for arrays to the Boostan language.
Arrays in Boostan

What do we have to extend?

▶ Syntax
  ▶ expressions
  ▶ assignment statement

▶ Semantics
  ▶ expressions
  ▶ assignment statement

▶ Rules of the Hoare proof system

▶ Soundness proof for the Hoare proof system
Grammar for Boostan with Array Assignment Statement

\[ G_{\text{Boo}} = (\Sigma_{\text{Boo}}, N_{\text{Boo}}, P_{\text{Boo}}, S_{\text{Boo}}) \]

\[ \Sigma_{\text{Boo}} = \{ \text{while, if, else, }, \text{, } \} \cup \Sigma_B \]

\[ N_{\text{Boo}} = \{ X_{\text{stmt}}, X_{\text{lhs}} \} \cup N_B \]

\[ P_{\text{Boo}} = \begin{align*}
X_{\text{stmt}} & \rightarrow X_{\text{lhs}} := X_{\text{expr}} ; \\
X_{\text{stmt}} & \rightarrow X_{\text{stmt}} X_{\text{stmt}} \\
X_{\text{stmt}} & \rightarrow \text{if } (X_{\text{expr}}) \{ X_{\text{stmt}} \} \text{ else } \{ X_{\text{stmt}} \} \\
X_{\text{stmt}} & \rightarrow \text{while } (X_{\text{expr}}) \{ X_{\text{stmt}} \} \\
X_{\text{lhs}} & \rightarrow X_{\text{var}}[X_{\text{expr}}] \\
X_{\text{lhs}} & \rightarrow X_{\text{var}} \} \cup P_B 
\end{align*} \]

\[ S_{\text{Boo}} = X_{\text{stmt}} \]
Semantics of the Array Assignment Statement

Reminder (Assignment Statement)

\[ [x := \text{expr};] \] is \( \{(s_1, s_2) \in S_{V, \mu} \times S_{V, \mu} \mid [x' = \text{expr} \land \bigwedge_{v \in V, v \neq x} v' = v]_{M, \rho} \text{ is true} \}
\]

and \( \rho = s_1 \cup \text{prime}(s_2) \}

Given a program \( P = (V, \mu, T) \) we define the semantics of an array assignment statement \( [a[i] := \text{expr};] \) as the following binary relation over program states.

\[ \{(s_1, s_2) \in S_{V, \mu} \times S_{V, \mu} \mid [a' = \text{store}(a, i, \text{expr}) \land \bigwedge_{v \in V, v \neq a} v' = v]_{M, \rho} \text{ is true} \}
\]

and \( \rho = s_1 \cup \text{prime}(s_2) \}

Reminder (Assignment Axiom)

\[(\text{assig}) \frac{\{\varphi[x \mapsto \text{expr}]\} \ x:=\text{expr}; \ \{\varphi}\}}{\{\varphi[x \mapsto \text{expr}]\} \ x:=\text{expr}; \ \{\varphi}\}}\]

\[(\text{arrassig}) \frac{\{\varphi[a \mapsto \text{store(a,i,expr)}]\} \ a[i]:=\text{expr}; \ \{\varphi}\}}{\{\varphi[a \mapsto \text{store(a,i,expr)}]\} \ a[i]:=\text{expr}; \ \{\varphi}\}}\]
Soundness of the Array Assignment Axiom

Lemma (Soundness of the Array Assignment Axiom)
The Hoare triple \( \{ \varphi[a \mapsto store(a, i, expr)] \} a[i] := expr; \{ \varphi \} \) is valid.

Reminder
\[
[a[i] := expr;] \text{ is } \\
\{(s_1, s_2) \in S_{V,\mu} \times S_{V,\mu} | \ [a' = store(a, i, expr) \land \bigwedge_{v \in V, v \neq a} v' = v]_{\mathcal{M},\rho} \text{ is true } \\
\text{ and } \rho = s_1 \cup \text{prime}(s_2) \}
\]

Proof. Analogously to the proof for the proof for the assignment statement.