Program Verification

Matthias Heizmann

Summer Term 2020

Program Verification

Summer Term 2020

Lecture 1: Introduction

Matthias Heizmann

The following slides will be discussed on 13th May

Notes for students of the 2020 course

Section 1

Introduction

Outline

- ▶ Program Verification
- ▶ Motivation
- ▶ Challenges
- ▶ Course outline

Outline of the Section on Introduction

Program Verifier
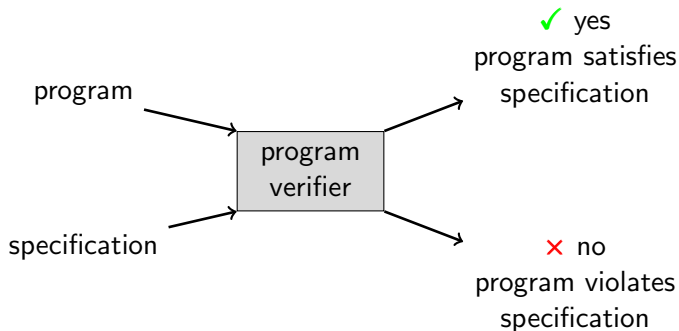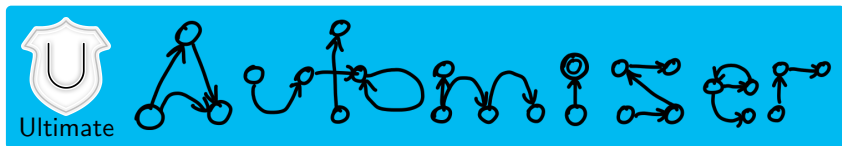
Program Verifier

Typical specifications:

- ▶ No division by zero
- ▶ Array only accessed within its bounds
- ▶ Termination
- ▶ Memory safety
- ▶ No assert statement is violated

Ultimate Automizer

`https://ultimate.informatik.uni-freiburg.de/automizer/`

▶ 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].

---

[1] SV-COMP `sv-comp.sosy-lab.org/`
[2] Group of Andreas Podelski `https://swt.informatik.uni-freiburg.de/`
[3] The Ultimate Framework `https://github.com/ultimate-pa/ultimate/`

Ultimate Automizer

Some program writte in the C language. [4] A specification written in ACSL [5] is given by assert statement in line 6.

```c
1 unsigned int foo(unsigned int x, unsigned int y) {
2     if (x < 1000 || y < 1000) {
3         return 1000;
4     }
5     unsigned int z = x + y;
6     //@ assert z >= 1000;
7     return z;
8 }
```

▶ 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[6] and assume that an unsigned int can store values from 0 to 4294967295.)

[4] https://en.wikipedia.org/wiki/C_(programming_language)
[5] https://en.wikipedia.org/wiki/ANSI/ISO_C_Specification_Language
[6] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf

Ultimate Automizer

Let's try to fix the program as follows.

```
1 unsigned long foo(unsigned int x, unsigned int y) {
2     if (x < 1000 || y < 1000) {
3         return 1000;
4     }
5     unsigned long z = x + y;
6     //@ assert z >= 1000;
7     return z;
8 }
```

▶ 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 couterexample applies again.

Ultimate Automizer

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

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 disfunctional, some bugs affect security and are exploited secretly
- ▶ Software is getting more and more complex.
- ▶ Higher complexity, more bugs.

Testing is not Always Sufficient

```
1 int foo(int x, int y) {
2    return y / (myHash(x)-23);
3 }
```

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

Motivation

▶ Find more software bugs
▶ Get mathematical proof of correctness
▶ Speed up software development

Outline of the Section on Introduction

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?

```
x := -7 / 5;
y := -7 % 5;
```

	x	y	makes sense because
C/C++	-1	-2	$(-1) \cdot 5 + (-2) = 7$
Python	-2	3	$(-2) \cdot 5 + 3 = 7$
Javascript	-1.4	-2	$(-1.4) \cdot 5 = 7$

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

```
1  int main(void) {
2    unsigned int x = 1;
3    unsigned int y = 1;
4    while(1) {
5      if (user_input()) {
6        x = 3 * x;
7        y = -2 * y + 1;
8      } else {
9        unsigned int tmp = x;
10       x = y;
11       y = tmp;
12     }
13     //@ assert y != 4;
14   }
15   return 0;
16 }
```
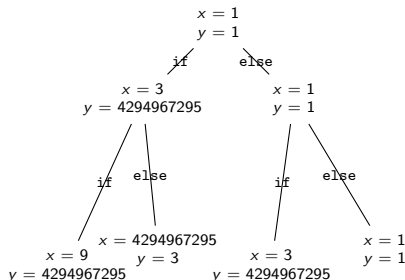

We cannot track all executions.

Simple argument for correctness:
The values of x and y are always odd.

Outline of the Section on Introduction

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 2020

Lecture 2: Propositional Logic

Matthias Heizmann

The following slides will be discussed on Monday 18th May

Notes for students of the 2020 course

Section 2

Propositional Logic

Outline

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 $(F_1 \wedge F_2)$ is a PL formula.

Abbreviations

$$
\begin{aligned}
\textbf{true} &:= \neg\textbf{false} \\
F_1 \vee F_2 &:= \neg(\neg F_1 \wedge \neg F_2) \\
F_1 \rightarrow F_2 &:= (\neg F_1 \vee F_2) \\
F_1 \leftrightarrow F_2 &:= (F_1 \rightarrow F_2) \wedge (F_2 \rightarrow F_1)
\end{aligned}
$$

Terminology

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

Notation

We may omit parentheses.

▶ Use the following order of precedence for logical connectives:
$\neg, \wedge, \vee, \rightarrow, \leftrightarrow$

▶ 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 $\rho : \mathcal{V}_{PL} \to \{\textbf{true}, \textbf{false}\}$ a *variable assignment*.

Definition

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

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

Definition

1. We call a PL formula F *satisfiable* if there is a variable assignment ρ such that $[\![F]\!]_\rho$ is **true**.
2. We call a PL formula F *valid* if for all variable assignments ρ the evaluation $[\![F]\!]_\rho$ is **true**.

Examples: Satisfiability and Validity

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

- $F_1 : P \wedge Q$
  satisfiable, not valid

- $F_2 : \neg(P \wedge Q)$
  satisfiable, not valid

- $F_3 : P \vee \neg P$
  satisfiable, valid

- $F_4 : \neg(P \vee \neg P)$
  unsatisfiable, not valid

- $F_5 : (P \rightarrow Q) \wedge (P \vee Q) \wedge \neg Q$
  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 $\llbracket \cdot \rrbracket$ 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 $F_5 : (P \rightarrow Q) \wedge (P \vee Q) \wedge \neg Q$

P	Q	$(P \rightarrow Q)$	$(P \vee Q)$	$\neg Q$	F_5
false	**false**	**true**	**false**	**true**	**false**
true	**false**	**false**	**true**	**true**	**false**
false	**true**	**true**	**true**	**false**	**false**
true	**true**	**true**	**true**	**false**	**false**

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

SAT solver	propositional logic
SMT solver	first order logic modulo theories

Example:

$$(P \rightarrow Q) \wedge (P \vee Q)$$

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
   E.g., for PL formulas F_1, F_2 we write `(and F_1 F_2)` instead of $(F_1 \wedge F_2)$

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/

Implications

Definition

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

Example

$\{A, A \rightarrow B\} \vDash A \wedge B$ $\qquad\qquad$ $\{A \rightarrow B\} \vDash \neg B \rightarrow \neg A$

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 $F_1 \wedge \ldots \wedge F_n \rightarrow F'$ is valid. (Requires algorithm for checking validity)

3. Prove that the PL formula $\neg(F_1 \wedge \ldots \wedge F_n \rightarrow F')$ 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 ¬F is not satisfiable.

Proof.

	F valid
iff	for all variable assignments ρ we have $\llbracket F \rrbracket_\rho = $ **true**
	(def of validity)
iff	for all variable assignments ρ we have $\llbracket \neg F \rrbracket_\rho = $ **false**
	(def of negation ¬)
iff	there is no variable assignment ρ such that $\llbracket \neg F \rrbracket_\rho = $ **true**
iff	¬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

$$F_1 \equiv F_2 \qquad \text{iff } \{F_1\} \vDash F_2 \text{ and } \{F_2\} \vDash F_1$$

Program Verification

Summer Term 2020

Lecture 3: Proof System $\mathcal{N}_{\mathsf{PL}}$, First-order Logic

Matthias Heizmann

The following slides will be discussed on Wednesday 20th May

Notes for students of the 2020 course

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 ⇝ 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:

$$\frac{\Gamma_1 \vDash F_1 \quad \ldots \quad \Gamma_n \vDash F_n}{\Gamma_{n+1} \vDash F_{n+1}}$$

Idea: the rule represents a step in a proof with the following meaning. If Γ_i implies F_i for $i \in \{1, \ldots n\}$ then Γ_{n+1} implies F_{n+1}.

Proof rules of $\mathcal{N}_{\text{PL}}$

$$(Ax)\frac{}{\Gamma \cup \{A\} \vDash A} \qquad (RAA)\frac{\Gamma \cup \{\neg A\} \vDash \textbf{false}}{\Gamma \vDash A}$$

Introduction rules:

$$(I_\wedge)\frac{\Gamma \vDash F_1 \qquad \Gamma \vDash F_2}{\Gamma \vDash F_1 \wedge F_2} \qquad (I_{\vee i})\frac{\Gamma \vDash F_i}{\Gamma \vDash F_1 \vee F_2}\, i \in \{1, 2\}$$

$$(I_\rightarrow)\frac{\Gamma \cup \{F_1\} \vDash F_2}{\Gamma \vDash F_1 \rightarrow F_2} \qquad (I_\neg)\frac{\Gamma \cup \{F\} \vDash \textbf{false}}{\Gamma \vDash \neg F}$$

Elimination rules:

$$(E_{\wedge i})\frac{\Gamma \vDash F_1 \wedge F_2}{\Gamma \vDash F_i}\, i \in \{1, 2\} \qquad (E_\vee)\frac{\Gamma \vDash F_1 \vee F_2 \quad \Gamma \cup \{F_1\} \vDash F_3 \quad \Gamma \cup \{F_2\} \vDash F_3}{\Gamma \vDash F_3}$$

$$(E_\rightarrow)\frac{\Gamma \vDash F_1 \qquad \Gamma \vDash F_1 \rightarrow F_2}{\Gamma \vDash F_2} \qquad (E_\neg)\frac{\Gamma \vDash F_1 \qquad \Gamma \vDash \neg F_1}{\Gamma \vDash F_2}$$

Proof system $\mathcal{N}_{\mathsf{PL}}$

Definition

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

$$\frac{\Gamma_1 \vDash F_1 \quad \ldots \quad \Gamma_n \vDash F_n}{\Gamma_{n+1} \vDash F_{n+1}}$$

must be an instance of some rule.

Example

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

$$\frac{\dfrac{\Gamma \vDash A \quad \Gamma \vDash A \to B}{\Gamma \vDash A \qquad \Gamma \vDash B}}{\Gamma \vDash A \wedge B}$$

$$(I_\wedge)\ \frac{(\text{Ax})\ \dfrac{}{\Gamma \vDash A} \qquad (I_\to)\ \dfrac{(\text{Ax})\ \dfrac{}{\Gamma \vDash A} \qquad (\text{Ax})\ \dfrac{}{\Gamma \vDash A \to B}}{\Gamma \vDash B}}{\Gamma \vDash A \wedge B}$$

▶ 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 $\dfrac{}{\Gamma \vDash F}$ is an instance of some (unary) rule.

Proof system $\mathcal{N}_{\mathsf{PL}}$

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

If a node in a derivation is labelled by $\Gamma \vDash F_{n+1}$, then the implication $\Gamma \vDash F_{n+1}$ 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 $\mathcal{N}_{\mathsf{PL}}$)

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

Proof difficult, not in the scope of this lecture.

Example: Construction of a Derivation

Let's prove the implication $\underbrace{\{A, A \to B\}}_{:=\Gamma} \vDash A \wedge B$

$(Ax)\overline{\Gamma \cup \{A\} \vDash A}$ $(RAA)\dfrac{\Gamma \cup \{\neg A\} \vDash \mathbf{false}}{\Gamma \vDash A}$

Introduction rules:

Elimination rules:

$(I_\wedge)\dfrac{\Gamma \vDash F_1 \quad \Gamma \vDash F_2}{\Gamma \vDash F_1 \wedge F_2}$ $(I_{\vee i})\dfrac{\Gamma \vDash F_i}{\Gamma \vDash F_1 \vee F_2}i \in \{1,2\}$ $(E_{\wedge i})\dfrac{\Gamma \vDash F_1 \wedge F_2}{\Gamma \vDash F_i}i \in \{1,2\}$ $(E_\vee)\dfrac{\Gamma \vDash F_1 \vee F_2 \quad \Gamma \cup \{F_1\} \vDash F_3 \quad \Gamma \cup \{F_2\} \vDash F_3}{\Gamma \vDash F_3}$

$(I_\to)\dfrac{\Gamma \cup \{F_1\} \vDash F_2}{\Gamma \vDash F_1 \to F_2}$ $(I_\neg)\dfrac{\Gamma \cup \{F\} \vDash \mathbf{false}}{\Gamma \vDash \neg F}$ $(E_\to)\dfrac{\Gamma \vDash F_1 \quad \Gamma \vDash F_1 \to F_2}{\Gamma \vDash F_2}$ $(E_\neg)\dfrac{\Gamma \vDash F_1 \quad \Gamma \vDash \neg F_1}{\Gamma \vDash F_2}$

$$(I_\wedge)\dfrac{(Ax)\dfrac{}{\Gamma \vDash A} \quad (E_\to)\dfrac{(Ax)\dfrac{}{\Gamma \vDash A} \quad (Ax)\dfrac{}{\Gamma \vDash A \to B}}{\Gamma \vDash B}}{\Gamma \vDash A \wedge B}$$

Section 3

First-Order Logic

Outline

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.

$$\forall n. \; integer(n) \wedge n > 2$$
$$\rightarrow \forall x, y, z.$$
$$integer(x) \wedge integer(y) \wedge integer(z)$$
$$\wedge x > 0 \wedge y > 0 \wedge z > 0$$
$$\rightarrow x^n + y^n \neq z^n$$

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 $i \geq 0$: $uv^i w \in L$.

$$\forall L. \ regularlanguage(L) \rightarrow$$
$$\exists n. \ integer(n) \wedge n \geq 0 \wedge$$
$$\forall z. \ z \in L \wedge |z| \geq n \rightarrow$$
$$\exists u, v, w. \ word(u) \wedge word(v) \wedge word(w) \wedge$$
$$z = uvw \wedge |v| \geq 1 \wedge |uv| \leq n \wedge$$
$$\forall i. \ integer(i) \wedge i \geq 0 \rightarrow uv^i w \in L$$

Predicates: *regularlanguage*, *integer*, *word*, $\cdot \in \cdot$, $\cdot \leq \cdot$, $\cdot \geq \cdot$, $\cdot = \cdot$
Constants: 0, 1
Functions: $|\cdot|$ (word length), concatenation, iteration

Syntax of First-order Logic

Definition

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

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

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

Definition

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

1. For each $x \in \mathcal{V}_{\mathsf{Var}}$, x is a *term*.
2. For each $c \in \mathcal{V}_{\mathsf{Const}}$, c is a *term*.
3. If $t_1, \ldots t_n$ are terms, $f \in \mathcal{V}_{\mathsf{Fun}}$, 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 φ is a formula, then $\neg\varphi$ is a formula.
4. If φ_1 and φ_2 are formulas, then $(\varphi_1 \wedge \varphi_2)$ is a formula.
5. If φ 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 $\vee, \rightarrow, \leftrightarrow$.
- ▶ 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 **true**, **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 $\mathcal{V}_{\mathsf{Const}} \cup \mathcal{V}_{\mathsf{Fun}} \cup \mathcal{V}_{\mathsf{Pred}}$.

- $\mathcal{I}$ 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 $\rho : \mathcal{V}_{\mathsf{Var}} \to \mathcal{D}$ 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 $f \lhd \{\tilde{x} \to \tilde{y}\}$ to denote the function that maps all $x \in X \setminus \{\tilde{x}\}$ to $f(x)$ and that maps $\tilde{x}$ to $\tilde{y}$.

Semantics of First-order Logic

> **Definition**
>
> The *evaluation of terms* is a mapping $[\![\cdot]\!]_{\mathcal{M},\rho}$ that takes a formula φ, a model $\mathcal{M} = (\mathcal{D}, \mathcal{I})$, and a variable assignment ρ, and returns an element of $\mathcal{D}$. It is inductively defined as follows.
>
> 1. For each $x \in \mathcal{V}_{\mathsf{Var}}$, $[\![x]\!]_{\mathcal{M},\rho}$ is $\rho(x)$.
>
> 2. For each $c \in \mathcal{V}_{\mathsf{Const}}$, $[\![c]\!]_{\mathcal{M},\rho}$ is $\mathcal{I}(c)$.
>
> 3. If $t_1, \ldots t_n$ are terms, $f \in \mathcal{V}_{\mathsf{Fun}}$, f has arity n, then $[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{M},\rho}$ is $\mathcal{I}(f)([\![t_1]\!]_{\mathcal{M},\rho}, \ldots, [\![t_n]\!]_{\mathcal{M},\rho})$.

Semantics of First-order Logic

Definition

The *evaluation of formulas* is a mapping $[\![\cdot]\!]_{\mathcal{M},\rho}$ that takes a formula φ, a model $\mathcal{M} = (\mathcal{D}, \mathcal{I})$, and a variable assignment ρ, and returns a truth value. It is inductively defined as follows.

1. $[\![\textbf{false}]\!]_{\mathcal{M},\rho}$ is **false**.

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

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

4. $[\![\varphi_1 \wedge \varphi_2]\!]_{\mathcal{M},\rho}$ is $\begin{cases} \textbf{true} & \text{if } [\![\varphi_1]\!]_{\mathcal{M},\rho} \text{ is } \textbf{true} \text{ and } [\![\varphi_2]\!]_{\mathcal{M},\rho} \text{ is } \textbf{true} \\ \textbf{false} & \text{otherwise.} \end{cases}$

5. $[\![\exists x.\varphi]\!]_{\mathcal{M},\rho}$ is $\begin{cases} \textbf{true} & \text{if there exists } v \in \mathcal{D} \\ & \text{such that } [\![\varphi]\!]_{\mathcal{M},\rho \triangleleft \{x \mapsto v\}} \text{ is } \textbf{true} \\ \textbf{false} & \text{otherwise.} \end{cases}$

Satisfiability and Validity

Definition (Satisfiability)

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

Definition (Validity)

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

Note

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

Implications

Definition

Given a (possibly infinite) set of FOL formulas Γ and a FOL formula ψ, we say that Γ *implies* ψ if for all models $\mathcal{M}$ and for all variable assignments ρ we have that
if $[\![\varphi]\!]_{\mathcal{M},\rho} = \textbf{true}$ holds for all $\varphi \in \Gamma$ then also $[\![\psi]\!]_{\mathcal{M},\rho} = \textbf{true}$ holds.
We use $\vDash$ to denote this binary implication relation and we say that the implication $\Gamma \vDash \psi$ holds if Γ implies ψ.

Program Verification

Summer Term 2020

Lecture 4: First-order Logic cont'd

Matthias Heizmann

The following slides will be discussed on Monday 25th May

Notes for students of the 2020 course

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 ψ, we define the set of *free variables* inductively as follows.

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

We call a variable that occurs in ψ but is not free *bound*.
We call a formula that does not contain free variables *closed*.

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

Notation

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

Definition (Substitution)

Given a function σ from variable symbols to terms we define the *substitution* for FOL terms t and FOL formulas ψ as follows.

$$t\sigma = \begin{cases} \sigma(x) & \text{if } t \text{ is } x \in \mathcal{V}_{\mathrm{Var}} \text{ and } x \in \mathrm{dom}(\sigma) \\ t & \text{if } t \text{ is } c \in \mathcal{V}_{\mathrm{Const}} \text{ or if } t \text{ is } x \in \mathcal{V}_{\mathrm{Var}} \text{ and } x \notin \mathrm{dom}(\sigma) \\ f(t_1\sigma, \ldots, t_n\sigma) & \text{if } t \text{ is } f(t_1, \ldots, t_n) \end{cases}$$

$$\psi\sigma = \begin{cases} \textbf{false} & \text{if } \psi \text{ is } \textbf{false} \\ p(t_1\sigma, \ldots, t_n\sigma) & \text{if } \psi \text{ is } p(t_1, \ldots, t_n) \\ \neg(\varphi\sigma) & \text{if } \psi \text{ is } \neg\varphi \\ \varphi_1\sigma \wedge \varphi_2\sigma & \text{if } \psi \text{ is } \varphi_1 \wedge \varphi_2 \\ \exists x.\varphi\sigma & \text{if } \psi \text{ is } \exists x.\varphi \text{ and } x \notin \mathrm{vars}(\sigma) \\ \exists x'.(\varphi\sigma')\sigma & \text{if } \psi \text{ is } \exists x.\varphi \text{ and } x \in \mathrm{vars}(\sigma) \end{cases}$$

where σ' is the function that maps x to x' and x' is a *fresh variable* (i.e., a variable that neither occurs in ψ nor in $\mathrm{vars}(\sigma)$).

Notation

If we do not want to specify the substitution function σ separately, we write $\varphi[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$ instead of $\varphi\sigma$ if σ 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 $\varphi[x \mapsto t]$.

Program Verification

Summer Term 2020

Lecture 5: Proof System $\mathcal{N}_{\mathsf{FOL}}$, First-order Theories

Matthias Heizmann

The following slides will be discussed on Wednesday 27th May

Notes for students of the 2020 course

▶ Please work yourself though the slides on $\mathcal{N}_{\mathsf{PL}}$.

▶ Exercise sheet 04 should help you to understand the slides.

▶ The slides on first-order theories will be presented in the interactive session (you don't have to work yourself through this slides in advance).

Analogously to $\mathcal{N}_{\mathsf{PL}}$ 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 $\mathcal{N}_{\mathsf{FOL}}$. Analogously to $\mathcal{N}_{\mathsf{PL}}$ we define the term *derivation* and use this tree as a proof.

For each rule of $\mathcal{N}_{\mathsf{PL}}$ there is an analogous rule in $\mathcal{N}_{\mathsf{FOL}}$. 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 $\mathcal{N}_{\mathsf{FOL}}$

For each rule of $\mathcal{N}_{\mathsf{PL}}$ there is an analogous rule in $\mathcal{N}_{\mathsf{FOL}}$. Additionally we have the following four rules.

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

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

(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

$$\Gamma = \{ \quad \forall x, y, z. f(x,y) \land f(y,z) \rightarrow f(x,z), \quad \forall x, y. f(x,y) \rightarrow f(y,x) \quad \}$$

Task: prove that the implication $\Gamma \vDash f(a,b) \land f(b,c) \rightarrow f(c,a)$ is valid.

$$
\cfrac{
  \cfrac{
    \cfrac{
      \cfrac{
        (Ax)\ \cfrac{}{\Gamma' \vDash f(a,b) \land f(b,c)}
        \quad
        (E_\forall)\ \cfrac{
          (E_\forall)\ \cfrac{
            (E_\forall)\ \cfrac{
              (Ax)\ \cfrac{}{\Gamma' \vDash \forall x.\ \forall y.\ \forall z.\ f(x,y) \land f(y,z) \rightarrow f(x,z)}
            }{\Gamma' \vDash \forall y.\ \forall z.\ f(a,y) \land f(y,z) \rightarrow f(a,z)}
          }{\Gamma' \vDash \forall z.\ f(a,b) \land f(b,z) \rightarrow f(a,z)}
        }{\Gamma' \vDash f(a,b) \land f(b,c) \rightarrow f(a,c)}
      }{(E_\rightarrow)\ \Gamma' \vDash f(a,c)}
      \quad
      (E_\forall)\ \cfrac{
        (E_\forall)\ \cfrac{
          (Ax)\ \cfrac{}{\Gamma' \vDash \forall x.\ \forall y.\ f(x,y) \rightarrow f(y,x)}
        }{\Gamma' \vDash \forall y.\ f(a,y) \rightarrow f(y,a)}
      }{\Gamma' \vDash f(a,c) \rightarrow f(c,a)}
    }{(E_\rightarrow)\ \Gamma' \vDash f(c,a)}
  }{(I_\rightarrow)\ \Gamma \vDash f(a,b) \land f(b,c) \rightarrow f(c,a)}
}{}
$$

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

Section 4

First-Order Theories

Outline

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

Notation

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.

First-Order Theories: Motivation

Is the following program correct?

```
1 void copyAtoBandC (int a) {
2   int b = a;
3   int c = b;
4   assert (c == a);
5 }
```

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 $\mathcal{D} = \{\clubsuit, \spadesuit\}$ and $\mathcal{I}$ maps the 2-ary predicate symbol $=$ to the binary relation $\{(\clubsuit, \spadesuit), (\spadesuit, \clubsuit)\} \subseteq \mathcal{D} \times \mathcal{D}$.

First-Order Theories: Motivation

Problem: We do not want to check if φ is valid.
We want to check if φ 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 φ.

Example

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

$$\forall x.\ x = x, \qquad \qquad \text{(reflexivity)}$$
$$\forall x, y.\ x = y \rightarrow y = x, \qquad \text{(symmetry)}$$
$$\forall x, y, z.\ x = y \wedge y = z \rightarrow x = z, \quad \text{(transitivity)}$$

and check if A_T implies φ.

Outline of the Section on First-Order Theories

First-Order Theories: Definition

Definition (First-order theory)

A *first-order theory* T consists of

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

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

Idea:

- The symbols of Σ 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 $[\![\varphi]\!]_{\mathcal{M},\rho} = \textbf{true}$ for all $\varphi \in A_T$ and for all variable assignments ρ.

Definition (T-valid)

A Σ-formula φ is *valid in theory T* (*T-valid*),
if for every T-model $\mathcal{M}$, it holds that $[\![\varphi]\!]_{\mathcal{M}} = \textbf{true}$.

Definition (T-satisfiable)

A Σ-formula φ is *satisfiable in T (T-satisfiable)*,
if there is a T-model $\mathcal{M}$ such that $[\![\varphi]\!]_{\mathcal{M}} = \textbf{true}$.

Definition (T-equivalent)

Two Σ-formulae φ_1 and φ_2 are *equivalent in T* (*T-equivalent*),
if $\varphi_1 \leftrightarrow \varphi_2$ is T-valid.

Outline of the Section on First-Order Theories

Theory of Equality: Motivation

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

$$\forall x.\ x = x, \qquad \text{(reflexivity)}$$
$$\forall x, y.\ x = y \rightarrow y = x, \qquad \text{(symmetry)}$$
$$\forall x, y, z.\ x = y \wedge y = z \rightarrow x = z, \qquad \text{(transitivity)}$$

Hint: Is the following formula implied by the axioms?
$a = b \wedge f(a) = c \rightarrow f(b) = c$

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,
   $\forall x_1, \ldots, x_n, y_1, \ldots, y_n.\ \bigwedge_i x_i = y_i \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$
                                                      (congruence)

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

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

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

A_{T_E} contains an infinite number of these axioms.

Outline of the Section on First-Order Theories

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 $\neg \exists x.\ \forall y.\ x \succ_{\mathsf{win}} y$ (no element wins against all others) valid with respect to these axioms?

Answer 1: Axioms 1-3 are not sufficient. A model in which $\succ_{\mathsf{win}}$ 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 $\neg \exists x.\ \forall y.\ x \succ_{\mathsf{win}} y$ 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 **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 Σ_{RPS}

Constant symbols: **Rock**, **Paper**, **Scissors**

Relation symbol: $\succ_{win}$

▶ Axioms $A_{T_{RPS}}$

1. **Rock** $\succ_{win}$ **Scissors**
2. **Scissors** $\succ_{win}$ **Paper**
3. **Paper** $\succ_{win}$ **Rock**
4. $\neg$**Rock** $\succ_{win}$ **Rock**
5. $\neg$**Rock** $\succ_{win}$ **Paper**
6. $\neg$**Scissors** $\succ_{win}$ **Scissors**
7. $\neg$**Scissors** $\succ_{win}$ **Rock**
8. $\neg$**Paper** $\succ_{win}$ **Paper**
9. $\neg$**Paper** $\succ_{win}$ **Rock**
10. $\forall x.\ x = $ **Rock** $\lor x = $ **Paper** $\lor x = $ **Scissors**

Are the following formulas T-valid?

▶ $\neg\exists x.\ \forall y.\ x \succ_{win} y$

▶ $\forall x.\ \exists y.\ x \succ_{win} y$

Outline of the Section on First-Order Theories

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 $\mathcal{N}_{\mathsf{FOL}}$

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

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. $\varphi[0] \wedge (\forall x. \; \varphi[x] \rightarrow \varphi[x + 1]) \rightarrow \forall x. \; \varphi[x]$ (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 Σ_{PA} as

$$x + x + x + 1 + 1 + 1 + 1 + 1 = y + y$$

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

$3x + 5 > 2y$ write as $\exists z.\ z \neq 0 \wedge 3x + 5 = 2y + z$

$3x + 5 \geq 2y$ write as $\exists z.\ 3x + 5 = 2y + z$

Examples for valid formulae:

▶ Pythagorean Theorem is T_{PA}-valid
$$\exists x, y, z.\ x \neq 0 \wedge y \neq 0 \wedge z \neq 0 \wedge xx + yy = zz$$

▶ Fermat's Last Theorem is T_{PA}-valid (Andrew Wiles, 1994)
$$\forall n.\ n > 2 \rightarrow \neg \exists x, y, z.\ x \neq 0 \wedge y \neq 0 \wedge z \neq 0 \wedge x^n + y^n = z^n$$

Expressiveness of Peano Arithmetic (2)

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

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

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

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 $f : \mathbb{N}^n \to \mathbb{N}$ there is a Σ_{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 Σ_{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: $\Sigma_\mathbb{N} : \{0, 1, +, =\}$ no multiplication!

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

1. $\forall x. \ \neg(x + 1 = 0)$ (zero)
2. $\forall x, y. \ x + 1 = y + 1 \to x = y$ (successor)
3. $\varphi[0] \wedge (\forall x. \ \varphi[x] \to \varphi[x+1]) \to \forall x. \ \varphi[x]$ (induction)
4. $\forall x. \ x + 0 = x$ (plus zero)
5. $\forall x, y. \ x + (y + 1) = (x + y) + 1$ (plus successor)

3 is an axiom schema.

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

Theory of Integers $T_{\mathbb{Z}}$

Signature:

$\Sigma_{\mathbb{Z}} : \{\ldots, -2, -1, 0, 1, 2, \ldots, -3\cdot, -2\cdot, 2\cdot, 3\cdot, \ldots, +, -, =, >\}$

where

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

Relation between $T_{\mathbb{Z}}$ and $T_{\mathbb{N}}$

$T_{\mathbb{Z}}$ and $T_{\mathbb{N}}$ have the same expressiveness:

- For every $\Sigma_{\mathbb{Z}}$-formula there is an equisatisfiable $\Sigma_{\mathbb{N}}$-formula.
- For every $\Sigma_{\mathbb{N}}$-formula there is an equisatisfiable $\Sigma_{\mathbb{Z}}$-formula.

$\Sigma_{\mathbb{Z}}$-formula φ and $\Sigma_{\mathbb{N}}$-formula G are *equisatisfiable* iff:

$$\varphi \text{ is } T_{\mathbb{Z}}\text{-satisfiable} \quad \text{iff} \quad G \text{ is } T_{\mathbb{N}}\text{-satisfiable}$$

Example: $\Sigma_{\mathbb{N}}$-formula to $\Sigma_{\mathbb{Z}}$-formula.

Example: The $\Sigma_{\mathbb{N}}$-formula

$$\forall x. \ \exists y. \ x = y + 1$$

is equisatisfiable to the $\Sigma_{\mathbb{Z}}$-formula:

$$\forall x. \ x > -1 \rightarrow \exists y. \ y > -1 \wedge x = y + 1.$$

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

$$F_1 : \quad \begin{array}{l} \forall w_p, w_n, x_p, x_n. \; \exists y_p, y_n, z_p, z_n. \\ \quad (x_p - x_n) + 2(y_p - y_n) - (z_p - z_n) - 7 > -3(w_p - w_n) + 4 \end{array}$$

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

$$F_2 : \quad \begin{array}{l} \forall w_p, w_n, x_p, x_n. \; \exists y_p, y_n, z_p, z_n. \\ \quad x_p + 2y_p + z_n + 3w_p > x_n + 2y_n + z_p + 7 + 3w_n + 4 \end{array}$$

Eliminate $>$ and numbers:

$$F_3 : \quad \begin{array}{l} \forall w_p, w_n, x_p, x_n. \; \exists y_p, y_n, z_p, z_n. \; \exists u. \\ \quad \neg(u = 0) \wedge x_p + y_p + y_p + z_n + w_p + w_p + w_p \\ \qquad = x_n + y_n + y_n + z_p + w_n + w_n + w_n + u \\ \qquad + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \end{array}$$

which is a $\Sigma_{\mathbb{N}}$-formula equisatisfiable to F_0.

Reducing $T_{\mathbb{Z}}$ to $T_{\mathbb{N}}$.

To decide $T_{\mathbb{Z}}$-validity for a $\Sigma_{\mathbb{Z}}$-formula φ:

▶ transform $\neg\varphi$ to an equisatisfiable $\Sigma_{\mathbb{N}}$-formula $\neg\psi$,

▶ decide $T_{\mathbb{N}}$-validity of ψ.

Program Verification

Summer Term 2020

Lecture 6: First-order Theories, SMT-LIB, Boogie, Syntax of Boostan

Matthias Heizmann

The following slides will be discussed on Discussed on Wednesday 3rd June

Notes for students of the 2020 course

- ▶ Please work yourself through the remaining slides on first-order theories and through the slides on SMT-LIB.
- ▶ Exercise sheet 05 should help you to understand the slides.
- ▶ You may omit the slides on the theory of arrays. We will skip these slides now and have a look at them only a few lectures later.
- ▶ You do not have to understand all details of the section of first order theories. The learning objectives and the exercises given you some guidance which aspects are important.
- ▶ Please work yourself also through the definition in the subsection on context-free grammars. We will discuss this subsection of the next section already on Wednesday 3rd June. Exercise 04 of sheet 05 should help you to understand context-free grammars.

Outline of the Section on First-Order Theories

Rationals and Reals

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

▶ Theory of Reals $T_{\mathbb{R}}$ (with multiplication)

$$x \cdot x = 2 \quad \Rightarrow \quad x = \pm\sqrt{2}$$

▶ Theory of Rationals $T_{\mathbb{Q}}$ (no multiplication)

$$\underbrace{2x}_{x+x} = 7 \quad \Rightarrow \quad x = \frac{2}{7}$$

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 $T_\mathbb{R}$

Signature: $\Sigma_\mathbb{R}$: $\{0, 1, +, -, \cdot, =, \geq\}$ with multiplication.

Axioms of $T_\mathbb{R}$: 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. $\forall x, y, z. \ (x \cdot y) \cdot z = x \cdot (y \cdot z)$ ($\cdot$ associativity)
6. $\forall x, y. \ x \cdot y = y \cdot x$ ($\cdot$ commutativity)
7. $\forall x. \ x \cdot 1 = x$ ($\cdot$ identity)
8. $\forall x. \ x \neq 0 \rightarrow \exists y. \ x \cdot y = 1$ ($\cdot$ inverse)
9. $\forall x, y, z. \ x \cdot (y + z) = x \cdot y + x \cdot z$ (distributivity)
10. $0 \neq 1$ (separate identies)
11. $\forall x, y. \ x \geq y \wedge y \geq x \rightarrow x = y$ (antisymmetry)
12. $\forall x, y, z. \ x \geq y \wedge y \geq z \rightarrow x \geq z$ (transitivity)
13. $\forall x, y. \ x \geq y \vee y \geq x$ (totality)
14. $\forall x, y, z. \ x \geq y \rightarrow x + z \geq y + z$ (+ ordered)
15. $\forall x, y. \ x \geq 0 \wedge y \geq 0 \rightarrow x \cdot y \geq 0$ ($\cdot$ ordered)
16. $\forall x. \ \exists y. \ x = y \cdot y \vee x = -y \cdot y$ (square root)
17. for each odd integer n,
   $\forall x_0, \ldots, x_{n-1}. \ \exists y. \ y^n + x_{n-1} y^{n-1} \cdots + x_1 y + x_0 = 0$ (at least one root)

Decidability of $T_\mathbb{R}$

$T_\mathbb{R}$ is decidable (Tarski, 1930)
High time complexity: $O(2^{2^{kn}})$

Theory of Rationals $T_{\mathbb{Q}}$

Signature: $\Sigma_{\mathbb{Q}} : \{0, 1, +, -, =, \geq\}$ no multiplication!
Axioms of $T_{\mathbb{Q}}$: axioms of T_E,

1. $\forall x, y, z.\ (x + y) + z = x + (y + z)$ $\hfill$ (+ associativity)
2. $\forall x, y.\ x + y = y + x$ $\hfill$ (+ commutativity)
3. $\forall x.\ x + 0 = x$ $\hfill$ (+ identity)
4. $\forall x.\ x + (-x) = 0$ $\hfill$ (+ inverse)
5. $1 \geq 0 \wedge 1 \neq 0$ $\hfill$ (one)
6. $\forall x, y.\ x \geq y \wedge y \geq x \rightarrow x = y$ $\hfill$ (antisymmetry)
7. $\forall x, y, z.\ x \geq y \wedge y \geq z \rightarrow x \geq z$ $\hfill$ (transitivity)
8. $\forall x, y.\ x \geq y \vee y \geq x$ $\hfill$ (totality)
9. $\forall x, y, z.\ x \geq y \rightarrow x + z \geq y + z$ $\hfill$ (+ ordered)
10. For every positive integer n:
    $\forall x.\ \exists y.\ x = \underbrace{y + \cdots + y}_{n}$ $\hfill$ (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

$$x + x + x + y + y + y + y \geq \underbrace{1 + 1 + \cdots + 1}_{24}$$

$T_{\mathbb{Q}}$ is decidable.
Efficient algorithm for quantifier free fragment.

Outline of the Section on First-Order Theories

Theory of Arrays T_A

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

- $a[i]$ binary function –
  read array a at index i ("read(a,i)")
- $a\langle i \lhd v\rangle$ 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. $\forall a, i, j.\ i = j \rightarrow a[i] = a[j]$ (array congruence)
3. $\forall a, v, i, j.\ i = j \rightarrow a\langle i \lhd v\rangle[j] = v$ (read-over-write 1)
4. $\forall a, v, i, j.\ i \neq j \rightarrow a\langle i \lhd v\rangle[j] = a[j]$ (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 $T_A^=$ (with extensionality)

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

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

Example:

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

is $T_A^=$-valid.

$T_A^=$ is undecidable.
Quantifier-free fragment of $T_A^=$ is decidable.

Outline of the Section on First-Order Theories

Combination of Theories

How do we show that

$$1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

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

$$\Sigma_1 \cap \Sigma_2 \quad = \quad \{=\}$$

The *combined theory* $T_1 \cup T_2$ has

▶ signature $\Sigma_1 \cup \Sigma_2$

▶ axioms $A_1 \cup A_2$

Nelson & Oppen

$$\boxed{\text{qff} = \text{quantifier-free fragment}}$$

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
  $bvadd_8(11111101, 000000100) = 00000001$
  $bvand_8(11111101, 000000100) = 00000100$
  $bvshl_8(11111101, 000000001) = 11111010$
- ▶ Predicates: comparisons
  $bvult_8(11111101, 000000100)$ is **false**
  $bvslt_8(11111101, 000000100)$ is **true**
  Meaning of bit-vector as number only given by operator.

Signature Σ

- ▶ Constant symbols: $0, 1, 01, 10, 11, 001, \ldots$
- ▶ Function symbols: $bvadd_1, bvadd_2, bvadd_3 \ldots, bvmul_1 \ldots$
- ▶ Predicate symbols: $bvult_1, bvult_2, bvult_3 \ldots, bvslt_1 \ldots$

Axioms A_T
Many

Theory of Bit-vectors

```
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($u1$)
)

Theory of Floats

Does the following loop terminate?

```
1    for(double d = 0; d != 0.3; d += 0.1) {
2    }
```

Outline of the Section on First-Order Theories

First-Order Theories

	Theory	Decidable	QFF Dec.
T_E	Equality	–	√
T_{PA}	Peano Arithmetic	–	–
$T_{\mathbb{N}}$	Presburger Arithmetic	√	√
$T_{\mathbb{Z}}$	Linear Integer Arithmetic	√	√
$T_{\mathbb{R}}$	Real Arithmetic	√	√
$T_{\mathbb{Q}}$	Linear Rationals	√	√
T_A	Arrays	–	√
$T_A^=$	Arrays with Extensionality	–	√
T_{BV}	Bitvectors	√	√
T_{Float}	FloatingPoint	√	√

Section 5

SMT-LIB

Outline

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., $=, +, -, \cdot, mod, \leq, >$ for the remaining course.

SMT-LIB

Quote from `http://smtlib.org/` (2019-05-12)

> 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)
```

SMT-LIB: Theories

Theories defined by SMT-LIB standard:

- ▶ Integer
  `-,+,-,*,div,mod,abs,<=,<,>=,>`
- ▶ Reals
  `-,+,-,*,/,<=,<,>=,>`
- ▶ 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, ...`

http://smtlib.cs.uiowa.edu/theories.shtml

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*).
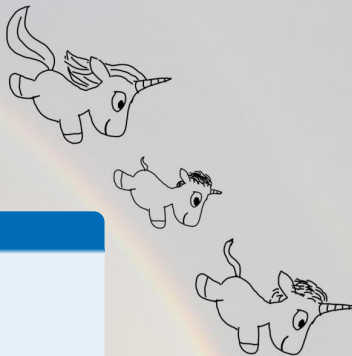
Program Verification
Lecture 6: SMT-LIB, Boogie, Boo, Context-free Grammars

Matthias Heizmann

Wednesday 15th May

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: **Q**uantifier-**F**ree **L**inear **I**nteger **A**rithmetic

▶ QF_AX: **Q**uantifier-**F**ree formulas over **A**rrays with e**X**tensionality

▶ UFLRA: **L**inear **R**eal **A**rithmetic with **U**ninterpreted sort and **F**unction 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 $\models$ between sets of formulas Γ and formulas φ.

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: $\Gamma \models_T \psi$ if and only if for every T-model $\mathcal{M}$ and every assignment ρ we have that if $\llbracket \varphi \rrbracket_{\mathcal{M}, \rho} = \textbf{true}$ for all $\varphi \in \Gamma$, then $\llbracket \psi \rrbracket_{\mathcal{M}, \rho} = \textbf{true}$ 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

Term or formula	SMT-LIB term
x	x
c	c
$f(t_1 \ldots t_n)$	(f t1 ... tn)
false	false
$\neg F$	(not F)
$F \wedge G$	(and F G)
$\exists x.\ F$	(exists ((x Sort)) (F))

SMT Solvers

SMT solvers are tools that execute SMT scripts.

- Z3[7] [**tacas/MouraB08**]
  Often used in this course because there is a Z3 web interface
- SMTInterpol[8] [**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 or the list of SMT solvers at Wikipedia.

You can submit SMT scripts to the SMT-LIB benchmark repository and the annual SMT competition evaluates how SMT solver perform on these benchmarks.

---

[7]Z3 `https://github.com/Z3Prover/z3`
[8]SMTInterpol `https://ultimate.informatik.uni-freiburg.de/smtinterpol/`

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 (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).

SMT-LIB: Commands

Important commands to communicate with the solver:

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

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

▶ Assert formulas:
  (assert (> x 0))

▶ Check satisfiability:
  (check-sat)

▶ Get models:
  (get-model)

Program Verification

Summer Term 2020

Lecture 7: Boogie, Boostan

Matthias Heizmann

The following slides will be discussed on Discussed on Monday 8th June

Notes for students of the 2020 course

- **TODO**

Section 6

Boogie and Boostan

Outline

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

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

Boogie Tools: Boogaloo

Boogaloo is an interpreter for Boogie developed by Nadia Polikarpova.

▶ Available via web interface[9]

▶ 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[10]

---

[9] http://comcom.csail.mit.edu/comcom/#Boogaloo

[10] https://bitbucket.org/nadiapolikarpova/boogaloo/wiki/UserManual

Boogie Tools: Boogaloo (Example)

Running the following program through Boogaloo with option -o 3
produces the output below, listing arguments, output, and return value.

```
1  procedure Square(a : int) returns (square: int) {
2      square := a * a;
3      if (square == 0) {
4          assume {: print "a is zero" } true;
5      } else {
6          assume {: print "a = ", a } true;
7      }
8  }
```

```
1  Execution 0: Square(0) passed
2  a is zero
3  Outs: square -> 0
4
5  Execution 1: Square(-1) passed
6  a = -1
7  Outs: square -> 1
8
9  Execution 2: Square(1) passed
10 a = 1
11 Outs: square -> 1
```

Boogie Tools: Boogaloo (Example)

Running the following program through Boogaloo with options `-o 4 -n 1 -c=0` produces the output below.

```
 1  procedure ZeroInit(a : [int]int, lo : int, hi : int) returns (b :
        [int]int)
 2  {
 3      var i : int;
 4      b := a;
 5      i := lo;
 6      while (i <= hi) {
 7          b[i] := 0;
 8          i := i+1;
 9      }
10  }
```

```
 1  Execution 0: ZeroInit([], 0, -1) passed
 2  Outs: b -> []
 3
 4  Execution 1: ZeroInit([0 -> 0], 0, 0) passed
 5  Outs: b -> [0 -> 0]
 6
 7  Execution 2: ZeroInit([0 -> 0, 1 -> 0], 0, 1) passed
 8  Outs: b -> [0 -> 0, 1 -> 0]
 9
10  Execution 3: ZeroInit([0 -> 0, 1 -> 0, 2 -> 0], 0, 2) passed
11  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.

Boostan

The specification of Boogie[11] [**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.

---

[11]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
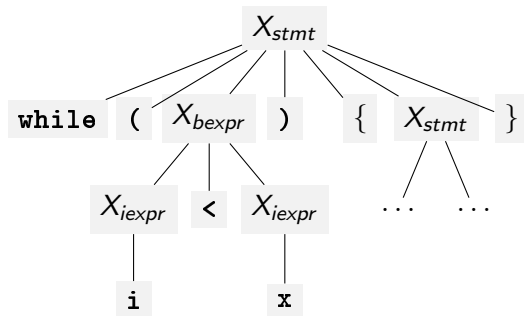
- The syntax of a programming language is typically defined via a context-free grammar or via a closely related concept.
- We will define the syntax of Boostan via a context-free grammar and use a notation that is typically used in lectures on theoretical computer science.
- In order to make you (again) 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 $\mathcal{G} = (\Sigma, N, P, S)$ such that

- ▶ Σ 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*.

Example

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

$$
\begin{aligned}
P = \{ \ S \ &\rightarrow aSbS, \\
S \ &\rightarrow bSaS, \\
S \ &\rightarrow \varepsilon \}.
\end{aligned}
$$

Definition

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

- a node $v \in V$ may only have children $v_1, \ldots, v_n \in V$ if
  $\lambda(v) \to \lambda(v_1) \ldots \lambda(v_n)$ is a rule in P and
- all leafs are labelled by terminal symbols or by ε.

Example

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

$$P = \{ \begin{array}{rcl} S & \to & aSbS, \\ S & \to & bSaS, \\ S & \to & \varepsilon \}. \end{array}$$

Example
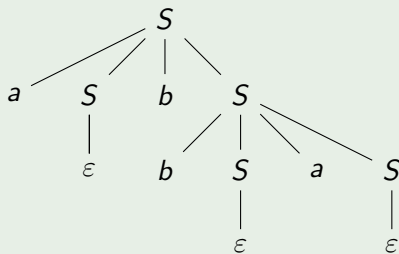
Definition

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

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

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 $\mathcal{G}$, denoted $L(\mathcal{G})$.

Example

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

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

Exercise: Construct a context-free grammar
$\mathcal{G}_{\mathsf{TInt}} = (\Sigma_{\mathsf{TInt}}, N_{\mathsf{TInt}}, P_{\mathsf{TInt}}, S_{\mathsf{TInt}})$ that generates the language of all FOL terms for the vocabulary $(\mathcal{V}_{\mathsf{Var}}, \mathcal{V}_{\mathsf{Const}}, \mathcal{V}_{\mathsf{Fun}}, \mathcal{V}_{\mathsf{Pred}})$ such that

▶ $\mathcal{V}_{\mathsf{Const}}$ is the set of all non-empty words over the alphabet 0-9.
▶ $\mathcal{V}_{\mathsf{Var}}$ is the set of all non-empty words over the alphabet a-zA-Z0-9 that are not constant symbols.
▶ $\mathcal{V}_{\mathsf{Fun}}$ is the set that contains
  ▶ the unary minus symbol - and
  ▶ the binary symbols +,-,*,div,mod,abs.

Outline of the Section on Boogie and Boostan

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. ᴛᴏᴅᴏ 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.

$$\mathcal{G}_{\mathsf{num}} = (\Sigma_{\mathsf{num}}, N_{\mathsf{num}}, P_{\mathsf{num}}, S_{\mathsf{num}})$$

$\Sigma_{\mathsf{num}} = \{0, \ldots, 9\}$
$N_{\mathsf{num}} = \{X_{num}, X_{num'}\}$
$P_{\mathsf{num}} = \{X_{num} \quad \rightarrow \quad 0X_{num'} | \ldots | 9X_{num'}$
$\qquad\qquad X_{num'} \quad \rightarrow \quad 0X_{num'} | \ldots | 9X_{num'} | \varepsilon\}$
$S_{\mathsf{num}} = X_{num}$

Grammar for Variables

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

$$\mathcal{G}_{\text{var}} = (\Sigma_{\text{var}}, N_{\text{var}}, P_{\text{var}}, S_{\text{var}})$$

$\Sigma_{\text{var}} = \Sigma_{\text{num}} \cup \{\mathtt{a}, \dots, \mathtt{z}, \mathtt{A}, \dots \mathtt{Z}\}$
$N_{\text{var}} = \{X_{var}, X_{var'}\}$
$P_{\text{var}} = \{X_{var} \quad \rightarrow \quad \mathtt{a}X_{var'} | \dots | \mathtt{z}X_{var'} | \mathtt{A}X_{var'} | \dots | \mathtt{Z}X_{var'}$
$\qquad\qquad X_{var'} \quad \rightarrow \quad \mathtt{a}X_{var'} | \dots | \mathtt{z}X_{var'} | \mathtt{A}X_{var'} | \dots | \mathtt{Z}X_{var'} | \mathtt{0}X_{var'} | \dots | \mathtt{9}X_{var'} | \varepsilon\}$
$S_{\text{var}} = X_{var}$

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 `div` and we would like to use the symbol % instead of `mod`.

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

$\Sigma_I = \{-, +, *, /, \%, (,)\} \cup \Sigma_{var} \cup \Sigma_{num}$
$N_I = \{X_{iexpr}\} \cup N_{var} \cup N_{num}$
$P_I = \{X_{iexpr} \quad \rightarrow \quad (X_{iexpr})$
$\qquad X_{iexpr} \quad \rightarrow \quad -X_{iexpr}$
$\qquad X_{iexpr} \quad \rightarrow \quad X_{iexpr}+X_{iexpr}|X_{iexpr}-X_{iexpr}|X_{iexpr}*X_{iexpr}$
$\qquad X_{iexpr} \quad \rightarrow \quad X_{iexpr}/X_{iexpr}|X_{iexpr}\%X_{iexpr}$
$\qquad X_{iexpr} \quad \rightarrow \quad X_{var}$
$\qquad X_{iexpr} \quad \rightarrow \quad X_{num}\} \cup P_{var} \cup P_{num}$
$S_I = X_{iexpr}$

Example

See Exercise 3 on Exercise Sheet 06 for a derivation tree of $\mathcal{G}_\mathsf{I}$

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. $\neg$) and we would like to use the symbol && instead of `and` (resp. $\wedge$) and we would like to use the symbol || instead of `or` (resp. $\vee$) and we would like to use the symbol ==> instead of => (resp. $\rightarrow$).

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

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

$N_B = \{X_{bexpr}\} \cup N_I$

$$\begin{aligned}
P_B = \{X_{bexpr} &\rightarrow (X_{bexpr}) \\
X_{bexpr} &\rightarrow !X_{bexpr} \\
X_{bexpr} &\rightarrow X_{bexpr}\&\&X_{bexpr}|X_{bexpr}||X_{bexpr}|X_{bexpr}==>X_{bexpr} \\
X_{bexpr} &\rightarrow |X_{iexpr}<X_{iexpr}|X_{iexpr}>X_{iexpr}|X_{iexpr}<=X_{iexpr}|X_{iexpr}>=X_{iexpr} \\
X_{bexpr} &\rightarrow X_{bexpr}==X_{bexpr}|X_{iexpr}==X_{iexpr} \\
X_{bexpr} &\rightarrow X_{var} \\
X_{bexpr} &\rightarrow \textbf{true}|\textbf{false}\} \cup P_I
\end{aligned}$$

$S_B = X_{bexpr}$

Grammar for Boostan

See Exercise Sheet 06.

Grammar for Boostan

Requirements: We would like to have a while statement, an if-then-else statement and an assignment statement. The concatenation of statements should be a statement. (By now) we neither need procedures nor statements for declaring variables.

$$\mathcal{G}_{\text{Boo}} = (\Sigma_{\text{Boo}}, N_{\text{Boo}}, P_{\text{Boo}}, S_{\text{Boo}})$$

$\Sigma_{\text{Boo}} = \{\texttt{while}, \texttt{if}, \texttt{else}, \{, \}\} \cup \Sigma_{\text{B}}$

$N_{\text{Boo}} = \{X_{\text{stmt}}, X_{\text{lhs}}\} \cup N_{\text{B}}$

$P_{\text{Boo}} = \{X_{stmt} \rightarrow X_{lhs}\texttt{:=}X_{expr}\texttt{;}$

$\qquad\quad X_{stmt} \rightarrow X_{stmt}X_{stmt}$

$\qquad\quad X_{stmt} \rightarrow \texttt{if } (X_{expr})\{X_{stmt}\} \texttt{ else } \{X_{stmt}\}$

$\qquad\quad X_{stmt} \rightarrow \texttt{while } (X_{expr})\{X_{stmt}\}$

$\qquad\quad\ X_{lhs} \rightarrow X_{var}\} \cup P_{\text{B}}$

$S_{\text{Boo}} = X_{stmt}$

Terminology

We call

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

Definition

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

- V is a set of (program) variables,

- μ is a map that assigns each variable either $\mathbb{Z}$ or $\{\textbf{true}, \textbf{false}\}$

- $\mathcal{T}$ is a derivation tree for the start symbol S_{Boo} in the Boostan grammar

such that the translation of each expression/type to an SMT term/sort is well-sorted wrt. the map μ.

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

Example

$P_{\text{ab}} = (V_{\text{ab}}, \mu_{\text{ab}}, \mathcal{T}_{\text{ab}})$ where

- $V_{\text{ab}} = \{a, b\}$,

- $\mu(a) = \mathbb{Z}$, $\mu(b) = \mathbb{Z}$, and

- $\mathcal{T}_{\text{ab}}$ is the derivation tree for the text on the right.

```
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

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[12] for several hours in order to give definite answers.

---

[12] E.g., ISO/IEC 9899:2011 informally called C11

Program Semantics: Motivation

Puzzle 1:

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

What is the value of y? 5? 6?

Puzzle 2:

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

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

What is the value of y? 10? 11? 12?

Program Semantics: Motivation

Puzzle 3:

```
1 int x;
2 ...
3   int y = 23;
4   x = 5;
5   if (x++ >= 5 && x++ >= 6) {
6     y = 42;
7   }
```

What is the value of y? 23? 42?

Puzzle 4:

```
1 int x;
2 ...
3   int y = 23;
4   x = 5;
5   if (x++ >= 6 && x++ >= 6) {
6     y = 42;
7   }
```

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

Program Semantics: Motivation

Puzzle 5:

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

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

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

What is the value of y? 40? 60?

Program Verification

Summer Term 2020

Lecture 7: Semantics of Boostan

Matthias Heizmann

The following slides will be discussed on Discussed on Monday 8th June

Notes for students of the 2020 course

- ▶ This time it is not required to read some slides in advance.
- ▶ However, the exercises on the transitive closure are very helpful for understanding the lecture content.

Outline of the Section on Boogie and Boostan

There are various ways to define the semantics of a programming language[13]. We will define the semantics of Boostan via relations. This definition of semantics is sometimes called *relational semantics*.

---

[13] see https://en.wikipedia.org/wiki/Semantics_(computer_science)

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

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 }
```

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 $P = (V, \mu, \mathcal{T})$, 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 φ denotes that set of all program states s such that for $s = \rho$ the evaluation $[\![\varphi]\!]_{\mathcal{M},\rho}$ is **true**.

▶ We will introduce the notation for the set of states denoted by a formula later.

Example

▶ The formula $a = 23 \wedge b = 42$ denotes the singleton set $\{\{a \mapsto 23, b \mapsto 42\}\} \subseteq S_{V_{ab}, \mu_{ab}}$

▶ 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 $[\![expr]\!]$ 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: $[\![\texttt{2 * (x \% 16)+42}]\!]$ is $2 \cdot (x \bmod 16) + 42$.

Convention

Since Boostan expressions and SMT formulas are so closely related, we may omit the double brackets and will often write $expr$ instead of $[\![expr]\!]$.

Semantics of the Assignment Statement

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

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

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 $\{(s_1, s_2) \mid [\![a' = a + 1 \wedge b' = b]\!]_{\mathcal{M},\rho} \text{ and } \rho = s_1 \cup \text{prime}(s_2)\}$

Semantics of the Assignment Statement

Example (continued)

$[\![a := a + 1]\!]$ is $\{(s_1, s_2) \mid [\![a' = a + 1 \land b' = b]\!]_{\mathcal{M}, \rho}$ and $\rho = s_1 \cup \mathrm{prime}(s_2)\}$

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 $\rho = s_1 \cup \mathrm{prime}(s_2) = \{a \mapsto 5, b \mapsto 1, a' \mapsto 6, b' \mapsto 1$ the evaluation $[\![a' = a + 1 \land b' = b]\!]$ is **true**.

Alternatively, we could write this relation as follows.
$\{(s_1, s_2) \mid s_2(a) = s_1(a) + 1$ and $s_2(b) = s_1(b)\}$.

Reminder: Relational Composition

Reminder: Relational Composition

The *relational composition* of two binary relations R_1, R_2 over a set X is defined as follows.
$R_1 \circ R_2 := \{(x, z) \mid \text{ there exists } y \in X \text{ s.t. } (x, y) \in R_1 \text{ and } (y, z) \in R_2\}$

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 07

Reminder: (Convention)

We defined the formula/term $[\![expr]\!]$ 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 φ 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 φ.

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

Warning

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

▶ the set that contains the formula φ (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
$$[\![\texttt{if}(expr)\{st1\}\texttt{else}\{st2\}]\!]$$
as
$$(\{expr\} \times S_{V,\mu}) \cap [\![st1]\!]$$
$$\cup (\{!expr\} \times S_{V,\mu}) \cap [\![st2]\!]$$

Example

$$[\![\texttt{if (b>=0)\{b:=b-1\} else \{b:=b+1\}}]\!]$$

$$\underbrace{(\{\texttt{b>=0}\} \times S_{V,\mu})}_{\{(s,s')|s(b) \geq 0\}} \cap \underbrace{[\![\texttt{b:=b-1}]\!]}_{\{(s,s')|^{s'(b)\,=\,s(b)\,-\,1}_{\text{and }s'(a)\,=\,s(a)}\}} \cup \underbrace{(\{\texttt{!b>=0}\} \times S_{V,\mu})}_{\{(s,s')|s(b) < 0\}} \cap \underbrace{[\![\texttt{b:=b+1}]\!]}_{\{(s,s')|^{s'(b)\,=\,s(b)\,+\,1}_{\text{and }s'(a)\,=\,s(a)}\}}$$

$$\{(s,s') \mid \text{and } {s(b) \geq 0 \wedge s'(b) = s(b) - 1 \text{ or } s(b) < 0 \wedge s'(b) = s(b) + 1 \atop s'(a) = s(a)} \}$$

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

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 $[\![\texttt{while }(expr)\{st\}]\!]$ as

$$((\{expr\} \times S_{V,\mu}) \cap [\![st]\!])^* \cap (S_{V,\mu} \times \{!expr\})$$

Example

$$[\![\texttt{while (x>=0)\{x:=x-1;y:=y+1;\}}]\!]$$

Let us use R to denote $\underbrace{(\{x >= 0\} \times S_{V,\mu}) \cap [\![x := x - 1; y := y + 1]\!]}_{\{(s,s')|s(x)\geq 0 \wedge s'(x)=s(x)-1 \wedge s'(y)=s(y)+1\}}$

$R^0 = id$

$R^1 = \{(s, s') \mid s(x) \geq 0 \wedge s'(x) = s(x) - 1 \wedge s'(y) = s(y) + 1\}$

$R^2 = \{(s, s') \mid s(x) \geq 1 \wedge s'(x) = s(x) - 2 \wedge s'(y) = s(y) + 2\}$

$\vdots$

$R^* = \{(s, s') \mid s = s' \vee (s(x) > s'(x) \geq -1 \wedge s'(y) - s(y) = s(x) - s'(x))\}$

$R^* \cap (S_{V_\mu} \times \{!\texttt{x>=0}\}) = \{(s, s') \mid (s = s' \wedge s'(x) < 0)$
$\vee (s(x) > s'(x) = -1 \wedge s'(y) - s(y) = s(x) + 1)\}$

Reminder

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

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 }
```

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.

$$\llbracket \mathtt{b := b-1} \rrbracket = \{\, (s,s') \mid \llbracket b' = b - 1 \wedge a' = a \rrbracket_{\mathcal{M},\rho} = \mathbf{true} \text{ and } \rho = s \cup \mathrm{prime}(s')\,\}$$
$$= \{\, (s,s') \mid s'(b) = s(b) - 1 \text{ and } s'(a) = s(a)\,\}$$
$$\llbracket \mathtt{b := b+1} \rrbracket = \{\, (s,s') \mid s'(b) = s(b) + 1 \text{ and } s'(a) = s(a)\,\}$$
$$\llbracket \mathtt{a := a+1} \rrbracket = \{\, (s',s'') \mid s''(a) = s'(a) + 1 \text{ and } s''(b) = s'(b)\,\}$$
$$\llbracket \mathit{if/else} \rrbracket = \{\mathtt{b >= 0}\} \times S_{V,\mu} \cap \llbracket \mathtt{b := b-1} \rrbracket \cup \{\mathtt{!b >= 0}\} \times S_{V,\mu} \cap \llbracket \mathtt{b := b+1} \rrbracket$$
$$= \{\, (s,s') \mid s'(a) = s(a) \text{ and } \big((s(b) \geq 0 \text{ and } s'(b) = s(b) - 1)$$
$$\text{or } (s(b) < 0 \text{ and } s'(b) = s(b) + 1) \big)\,\}$$
$$\llbracket \mathit{loop\ body} \rrbracket = \{\, (s,s'') \mid \text{ex. } s' \text{ s.t. } (s,s') \in \llbracket \mathit{if/else} \rrbracket, (s',s'') \in \llbracket \mathtt{a := a+1} \rrbracket \,\}$$
$$= \{\, (s,s'') \mid s''(a) = s(a) + 1 \text{ and } \big((s(b) \geq 0 \text{ and } s''(b) = s(b) - 1)$$
$$\text{or } (s(b) < 0 \text{ and } s''(b) = s(b) + 1) \big)\,\}$$
$$\llbracket P_{ab} \rrbracket = \big((\{\mathtt{!(b==0)}\} \times S_{V,\mu}) \cap \llbracket \mathit{loop\ body} \rrbracket \big)^{*} \cap (S_{V,\mu} \times \{\mathtt{!!(b==0)}\})$$
$$= \{\, (s,s') \mid s(b) \neq 0 \text{ and } s'(a) = s(a) + 1 \text{ and } |s'(b)| = |s(b)| - 1\,\}^{*}$$
$$\cap (S_{V,\mu} \times \{\mathtt{!!(b==0)}\})$$
$$= \{\, (s,s') \mid s'(a) + |s'(b)| = s(a) + |s(b)| \text{ and } |s'(b)| \leq |s(b)|\,\}$$
$$\cap (S_{V,\mu} \times \{\mathtt{!!(b==0)}\})$$
$$= \{\, (s,s') \mid s'(a) = s(a) + |s(b)| \text{ and } s'(b) = 0\,\}$$

Example

See Exercise 1 on Exercise Sheet 07 for another example where we compute the relation of a program.

Program Verification

Summer Term 2020

Lecture 8: Semantics of Boostan

Matthias Heizmann

The following slides will be discussed on Wednesday 10th June

Notes for students of the 2020 course

► The section on precondition-postcondition pairs was briefly presented in the last lecture. If thre are questions, we will discuss these in more detail. Otherwise we will continue with the section on the Hoare proof system which builds upon precondition-postcondition pairs.

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.

Precondition-Postcondition Pairs

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

Definition

We say that program P *satisfies the precondition-postcondition pair* $(\{\varphi_{\mathsf{pre}}\}, \{\varphi_{\mathsf{post}}\})$ if the inclusion $post(\{\varphi_{\mathsf{pre}}\}, [\![st]\!]) \subseteq \{\varphi_{\mathsf{post}}\}$ holds.

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 $(\{a \cdot b \geq 0\}, \{a \geq 0\})$?

Post Image

Definition

Post Image Given a binary relation R over the set X and a subset of $Y \subseteq X$, the *postimage of Y under R*, denoted $post(Y, R)$, 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

$$post(Y, R) = \{y \in \mathbb{Z} \mid y \geq 6\}$$

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 $post(\{a \geq 0\}, \llbracket st \rrbracket) \overset{?}{\subseteq} \{a \geq 0\}$!

$\llbracket st \rrbracket = \{ (s, s') \mid s'(a) = s(a) + |s(b)| \text{ and } s'(b) = 0 \}$

Example

See Exercise 2 on Exercise Sheet 08 for more examples.

Section 7

Hoare Proof System

Outline

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 $\mathcal{N}_{\mathsf{PL}}$ proof system to derive valid implications of the form $\Gamma \vDash F$.

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 $\mathcal{N}_{\mathsf{PL}}$),
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 notvalid

Proof Systems of this Course

$\mathcal{N}_{\text{PL}}$	$\mathcal{N}_{\text{FOL}}$	Hoare proof system
proof system for deriving	proof system for deriving	proof system for deriving
valid PL implications	valid FOL implications	valid Hoare triples
$\Gamma \vDash F$	$\Gamma \vDash \varphi$	$\{\varphi\}st\{\psi\}$

Outline of the Section on Hoare Proof System

Rules of the Hoare Proof System – Overview

Assignment axiom

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

Composition rule

$$(compo) \frac{\{\varphi_1\}st_1\{\varphi_2\} \quad \{\varphi_2\}st_2\{\varphi_3\}}{\{\varphi_1\}st_1 st_2\{\varphi_3\}}$$

Strengthen precondition rule

$$(strepre) \frac{\{\varphi\}st\{\psi\}}{\{\varphi'\}st\{\psi\}} \text{ if } \varphi' \vDash \varphi$$

Weaken postcondition rule

$$(weakpos) \frac{\{\varphi\}st\{\psi\}}{\{\varphi\}st\{\psi'\}} \text{ if } \psi \vDash \psi'$$

Conditional rule

$$(condi) \frac{\{\varphi \wedge expr\} \, st_1 \, \{\psi\} \quad \{\varphi \wedge \neg expr\} \, st_2 \, \{\psi\}}{\{\varphi\} \, \texttt{if(}expr\texttt{)}\{st1\}\texttt{else}\{st2\} \, \{\psi\}}$$

While rule

$$(while) \frac{\{\varphi \wedge expr\} \, st \, \{\varphi\}}{\{\varphi\} \, \texttt{while(}expr\texttt{)}\{st\} \, \{\varphi \wedge \neg expr\}}$$

Hore Proof System – Derivation

Definition

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 $\{\varphi_1\}st_1\{\psi_1\}$ ... $\{\varphi_n\}st_n\{\psi_n\}$, 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.

Program Verification

Summer Term 2020

Lecture 9: Hoare Proof System

Matthias Heizmann

The following slides will be discussed on Monday 15th June

Notes for students of the 2020 course
- None.

On Exercise Sheet 08 we learned how to use the following four rules.

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

We next discuss the remaining two rules.

Conditional Rule

$$(condi) \frac{\{\varphi \wedge expr\} \; st_1 \; \{\psi\} \quad \{\varphi \wedge \neg expr\} \; st_2 \; \{\psi\}}{\{\varphi\} \; \texttt{if(}expr\texttt{)}\{st1\}\texttt{else}\{st2\} \; \{\psi\}}$$

Example

$$\frac{\{ {}^{x=y}_{\wedge y \geq 0} \}\texttt{y:=y-1;}\{ {}^{(x \geq 0 \rightarrow y=x-1)}_{\wedge(x<0 \rightarrow y=x+1)} \} \quad \{ {}^{x=y}_{\wedge \neg(y \geq 0)} \}\texttt{y:=y+1;}\{ {}^{(x \geq 0 \rightarrow y=x-1)}_{\wedge(x<0 \rightarrow y=x+1)} \}}{\{y = x\}\texttt{if(y>=0)}\{\texttt{y:=y-1;}\}\texttt{else}\{\texttt{y:=y+1;}\}\{ {}^{(x \geq 0 \rightarrow y=x-1)}_{\wedge(x<0 \rightarrow y=x+1)} \}}$$

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

$$(while)\frac{\{\varphi \wedge expr\} \ st \ \{\varphi\}}{\{\varphi\} \ \texttt{while}(expr)\{st\} \ \{\varphi \wedge \neg expr\}}$$

We call the formula φ an *inductive loop invariant*.

Example

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

$$\frac{\dfrac{\{\substack{z=x+y \\ \wedge x \geq 0} \wedge x > 0\} \ \texttt{x:=x-1;y:=y+1;} \ \{\substack{z=x+y \\ \wedge x \geq 0}\}}{\{\substack{z=x+y \\ \wedge x \geq 0}\} \ \texttt{while(x>0)\{x:=x-1;y:=y+1;\}} \ \{\substack{z=x+y \\ \wedge x \geq 0} \wedge \neg(x > 0)\}} (while)}{\{\substack{z=x+y \\ \wedge x \geq 0}\} \ \texttt{while(x>0)\{x:=x-1;y:=y+1;\}} \ \{z = y\}} (weakpos)$$

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 $x = 0$).

Hoare Proof System – Example

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

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

$$
\cfrac{
  \cfrac{
    \cfrac{}{\left\{\substack{b\leq 0 \rightarrow \\ a-(b-1)\geq 52}\right\}\texttt{b:=b-1;}\left\{\substack{b\leq 0 \rightarrow \\ a-b\geq 52}\right\}} \ (assig)
  }{\{\varphi_I \wedge b\geq 0\}\texttt{b:=b-1;}\left\{\substack{b\leq 0 \rightarrow \\ a-b\geq 52}\right\}} \ (strepre)
  \qquad
  \cfrac{
    \cfrac{}{\left\{\substack{b\leq 0 \rightarrow \\ a-(b+1)\geq 52}\right\}\texttt{b:=b+1;}\left\{\substack{b\leq 0 \rightarrow \\ a-b\geq 52}\right\}} \ (assig)
  }{\{\varphi_I \wedge \neg b\geq 0\}\texttt{b:=b+1;}\left\{\substack{b\leq 0 \rightarrow \\ a-b\geq 52}\right\}} \ (strepre)
}{\cfrac{\cfrac{\{\varphi_I\}\texttt{if (b>=0) \{b:=b-1;\} else \{b:=b+1;\}}\left\{\substack{b\leq 0 \rightarrow \\ a-b\geq 52}\right\} \qquad (*)}{\{\varphi_I\}\texttt{if (b>=0) \{b:=b-1;\} else \{b:=b+1;\} a:=a+1;}\{\varphi_I\}}\ (compo)}{\cfrac{\cfrac{\{(\varphi_I)\wedge\neg(b=0)\}\texttt{if (b>=0) \{b:=b-1;\} else \{b:=b+1;\} a:=a+1;}\{\varphi_I\}}{\{\varphi_I\}\texttt{while(!(b==0))\{if (b>=0) \{b:=b-1;\} else \{b:=b+1;\} a:=a+1;\}}\{(\varphi_I)\wedge\neg\neg(b=0)\}}\ (while)}{\cfrac{\{a\geq 42\wedge b\leq -23\}\texttt{while(!(b==0))\{if (b>=0) \{b:=b-1;\} else \{b:=b+1;\} a:=a+1;\}}\{(\varphi_I)\wedge\neg\neg(b=0)\}}{\{a\geq 42\wedge b\leq -23\}\texttt{while(!(b==0))\{if (b>=0) \{b:=b-1;\} else \{b:=b+1;\} a:=a+1;\}}\{a\geq 53\}}\ (weakpos)}\ (strepre)}}
$$

(condi) *(strepre)*

where (*) is the following subtree

$$
\cfrac{
  \cfrac{}{\left\{\substack{b\leq 0 \rightarrow \\ (a+1)-b\geq 53}\right\}\texttt{a:=a+1;}\{\varphi_I\}} \ (assig)
}{\left\{\substack{b\leq 0 \rightarrow \\ a-b\geq 52}\right\}\texttt{a:=a+1;}\{\varphi_I\}} \ (strepre)
$$

Hoare Proof System – Example

Example

See Exercise 4 on Exercise Sheet 08 or Exercise 1 on Exercise Sheet 09 for more examples of derivations.

Outline of the Section on Hoare Proof System

Introduction
Rules of the Hoare Proof System
Soundness of the Hoare Proof System

This last subsection ist dedicated to the proof of the theorem that states that every derived Hoare triple is indeed valid.

We follow the typical approch 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 $\{\varphi\} st \{\psi\}$ then the statement st satisfies the precondition-postcondition pair $(\{\varphi\}, \{\psi\})$

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 $\{\varphi_{n+1}\} st_{n+1} \{\psi_{n+1}\}$ has children that are labelled by Hoare triples $\{\varphi_1\} st_1 \{\psi_1\}$ ... $\{\varphi_n\} st_n \{\psi_n\}$, 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.

Definition (Sound Rule)

We call a rule of the form $\dfrac{\{\varphi_1\} st_1 \{\psi_1\} \quad \ldots \quad \{\varphi_n\} st_n \{\psi_n\}}{\{\varphi_{n+1}\} st_{n+1} \{\psi_{n+1}\}}$ *sound* if the following holds. If for all $i \in \{1, \ldots n\}$ the Hoare triple $\{\varphi_i\} st_i \{\psi_i\}$ is valid, then the Hoare triple $\{\varphi_{n+1}\} st_{n+1} \{\psi_{n+1}\}$ is also valid.

Soundness of the Assignment Axiom

Lemma (Soundness of the Assignment Axiom)

The Hoare triple $\{\varphi[x \mapsto expr]\}$ `x:=expr;` $\{\varphi\}$ is valid.

Reminder

$[\![\texttt{x:=expr}]\!]$ is $\{(s_1, s_2) \in S_{V,\mu} \times S_{V,\mu} \mid [\![x' = [\![expr]\!] \wedge \bigwedge\limits_{v \in V, v \neq x} v' = v]\!]_{\mathcal{M},\rho}$ is **true**

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

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

$\Rightarrow$ There exists s such that $s \in \{\varphi[x \mapsto expr]\}$ and $(s, s') \in [\![\texttt{x:=expr;}]\!]$

$\Rightarrow$ There exists s such that for $\rho = s \cup \text{prime}(s')$
$[\![\varphi[x \mapsto expr] \wedge x' = [\![expr]\!] \wedge \bigwedge\limits_{v \in V, v \neq x} v' = v]\!]_{\mathcal{M},\rho}$ is **true** and
$\rho = s \cup \text{prime}(s')$

$\Rightarrow$ for $\rho = s'$ the evaluation $[\![\varphi]\!]_{\mathcal{M},\rho}$ is **true**

$\Rightarrow$ $s' \in \{\varphi\}$

Program Verification

Summer Term 2020

Lecture 10: Hoare Proof System

Matthias Heizmann

The following slides will be discussed on Wednesday 17th June

Notes for students of the 2020 course

► In the section on SMT theories, we skipped the subsection on the theory of arrays. Please have a look at this subsection. We will discuss this theory in this lecture and there will be exercises about this theory on the next exercise sheet.

► In this lecture we stopped before the soundness proof for the while rule. But additionally, we discussed the section on the ULTIMATE ELIMINATOR tool.

Soundness of the Composition Rule

$$(compo)\frac{\{\varphi_1\}st_1\{\varphi_2\} \quad \{\varphi_2\}st_2\{\varphi_3\}}{\{\varphi_1\}st_1st_2\{\varphi_3\}}$$

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 $\{\varphi_1\}st_1st_2\{\varphi_3\}$ is valid.

Proof. See Exercise 2 on Exercise Sheet 10.

Soundness of the Strengthen Precondition Rule

$$(strepre)\frac{\{\varphi\}st\{\psi\}}{\{\varphi'\}st\{\psi\}} \text{ if } \varphi' \vDash \varphi$$

Lemma (Soundness of the Strengthen Precondition Rule)

If the Hoare triple $\{\varphi\}st\{\psi\}$ is valid and the side condition $\varphi' \vDash \varphi$ is valid, then the Hoare triple $\{\varphi'\}st\{\psi\}$ is valid.

Proof. Let $s' \in post(\{\varphi'\}, [\![st]\!])$

$\Rightarrow$ There exists s such that $s \in \{\varphi'\}$ and $(s, s') \in [\![st]\!]$.

$\Rightarrow$ There exists s such that $s \in \{\varphi\}$ and $(s, s') \in [\![st]\!]$.

$\Rightarrow$ $s' \in post(\{\varphi\}, st)$

$\Rightarrow$ $s' \in \{\psi\}$ (because $post(\{\varphi\}, st) \subseteq \{\psi\}$)

Soundness of the Weakening Postcondition Rule

$$(weakpos) \frac{\{\varphi\} st \{\psi\}}{\{\varphi\} st \{\psi'\}} \text{ if } \psi \vDash \psi'$$

Lemma (Soundness of the Weakening Postcondition Rule)

If the Hoare triple $\{\varphi\} st \{\psi\}$ is valid and the side condition $\psi \vDash \psi'$ is valid, then the Hoare triple $\{\varphi\} st \{\psi'\}$ is valid.

Proof. See Exercise 1 on Exercise Sheet 10.

Soundness of the Conditional Rule

$$(condi)\frac{\{\varphi \wedge expr\}\ st_1\ \{\psi\} \quad \{\varphi \wedge \neg expr\}\ st_2\ \{\psi\}}{\{\varphi\}\ \texttt{if(}expr\texttt{)\{}st1\texttt{\}else\{}st2\texttt{\}}\ \{\psi\}}$$

Lemma (Soundness of the Conditional Rule)

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

Proof. See Exercise 3 on Exercise Sheet 10.

Program Verification

Summer Term 2020

Lecture 11: Hoare Proof System cont'd

Matthias Heizmann

The following slides will be discussed on Monday 19th June

Notes for students of the 2020 course

► The section on the ULTIMATE ELIMINATOR tool. was already discussed in the last lecture.

Soundness of the While Rule

$$(while) \frac{\{\varphi \wedge expr\} \; st \; \{\varphi\}}{\{\varphi\} \; \texttt{while(}expr\texttt{)}\{st\} \; \{\varphi \wedge \neg expr\}}$$

Lemma (Soundness of the While Rule)

If the Hoare triple $\{\varphi \wedge expr\} \; st \; \{\varphi\}$ is valid, then the Hoare triple $\{\varphi\} \; \texttt{while(}expr\texttt{)}\{st\} \; \{\varphi \wedge \neg expr\}$ is valid.

Proof. Let $s' \in post(\{\varphi\}, [\![\texttt{while } (expr)\{st\}]\!])$, i.e. there exists an $s \in \{\varphi\}$ such that

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

Therefore we know that $s' \in \{!expr\}$.

Let $R = \big((\{expr\} \times S_{V,\mu}) \cap [\![st]\!]\big)$. 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 $s' \in \{\varphi \wedge \neg expr\}$. Thus we will have proven that

$$post(\{\varphi\}, [\![\texttt{while } (expr)\{st\}]\!]) \subseteq \{\varphi \wedge \neg expr\}$$

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.

$n \to n+1$ 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 $s'' \in \{expr\}$ and $(s'', s') \in [\![st]\!]$.

   Hence it follows that $s' \in post(\{\varphi \wedge expr\}, [\![st]\!])$. By validity of the Hoare triple $\{\varphi \wedge expr\}$ st $\{\varphi\}$, we have $post(\{\varphi \wedge expr\}, [\![st]\!]) \subseteq \{\varphi\}$. 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, $\dfrac{}{\{\varphi\}st\{\psi\}}$ 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 $\{\varphi_1\}st_1\{\psi_1\}$ ... $\{\varphi_m\}st_m\{\psi_m\}$ 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,

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

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

Outline

In this section we will partially automate the task of checking correctness.

In this section, we will learn to

- systematically construct derivations in the Hoare proof system if suiteable loop invariants are given
- use the Ultimate Referee tool check if given loop invariants are suiteable to proof correctness

Outline of the Section on Ultimate Referee

Guide for Finding a Derivation in the Hoare Proof System

Ultimate Referee

► 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 challange directly.

Guide for Finding a Derivation in the Hoare Proof System

$(assig) \dfrac{}{\{\varphi[x \mapsto \texttt{expr}]\}\texttt{x := expr;}\{\varphi\}}$

$(compo) \dfrac{\{\varphi_1\}st_1\{\varphi_2\} \quad \{\varphi_2\}st_2\{\varphi_3\}}{\{\varphi_1\}st_1 st_2\{\varphi_3\}}$

$(strepre) \dfrac{\{\varphi\}st\{\psi\}}{\{\varphi'\}st\{\psi\}}$ if $\varphi' \models \varphi$

$(weakpos) \dfrac{\{\varphi\}st\{\psi\}}{\{\varphi\}st\{\psi'\}}$ if $\psi \models \psi'$

$(condi) \dfrac{\{\varphi \wedge expr\}\ st_1\ \{\psi\} \quad \{\varphi \wedge \neg expr\}\ st_2\ \{\psi\}}{\{\varphi\}\ \texttt{if(}expr\texttt{)}\{st1\}\texttt{else}\{st2\}\ \{\psi\}}$

$(while) \dfrac{\{\varphi \wedge expr\}\ st\ \{\varphi\}}{\{\varphi\}\ \texttt{while(}expr\texttt{)}\{st\}\ \{\varphi \wedge \neg expr\}}$

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 coputer to everything that can be done algorithmically. (See tool in next subsection.)

Outline of the Section on Ultimate Referee

Guide for Finding a Derivation in the Hoare Proof System

Ultimate Referee

Ultimate Referee

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.

Ultimate Referee and Boogie

```
1 procedure main(i,j : int)
      returns (x,y : int)
2 requires true;
3 ensures (i == j) ==> (y == 0);
4 {
5   x := i;
6   y := j;
7   while (x != 0)
8     invariant y==0;
9   {
10    x := x - 1;
11    y := y - 1;
12  }
13 }
```

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.

Program Verification

Summer Term 2020

Lecture 12: Arrays

Matthias Heizmann

The following slides will be discussed on Wednesday 21st June

Notes for students of the 2020 course
- None.

Section 9

Arrays

Outline

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

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.
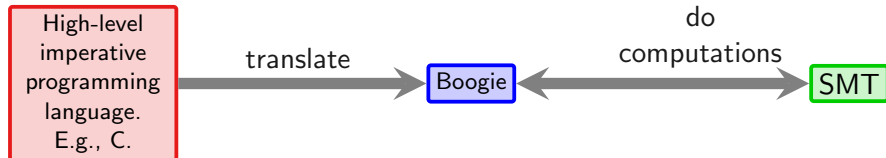
Approach in this lecture.
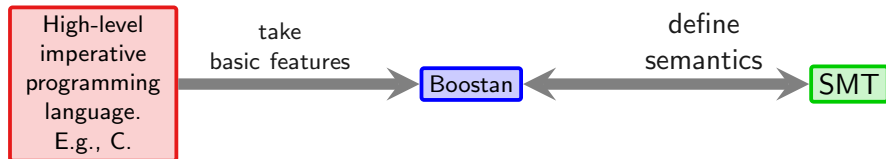
We need logical formulas whose models are (also) arrays!

Outline of the Section on Arrays

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

We use two functions to implement this idea.

select	store
▶ binary function	▶ ternary function
▶ 1st argument: a map	▶ 1st argument: a map
▶ 2nd argument: element of map's domain	▶ 2nd argument: element of map's domain
▶ returns: value of map at that position	▶ 3rd argument: new value at that position
▶ e.g. $select(f_{foo}, 5) = 0$	▶ returns: updated map
▶ e.g. $select(f_{bar}, 5) = 23$	▶ e.g. $store(f_{foo}, 5, 23) = f_{bar}$

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.

Theory of Arrays in Comparison to the Theory of Integers

	Theory of Integers	Theory of Arrays
Values	Numbers, e.g, ▶ 23 ▶ 42 ▶ -17	1-ary maps, e.g., ▶ f_{foo} ▶ f_{bar} ▶ absolute value $\lvert \cdot \rvert$
Functions	▶ + ▶ - ▶ * ▶ abs	▶ select ▶ store

Outline of the Section on Arrays

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

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 [14] 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 (Array Int Bool).

See Exercise Sheet 10 and Exercise Sheet 11 for more examples.

---

[14] http://smtlib.cs.uiowa.edu/theories-ArraysEx.shtml

Arrays in SMT-LIB

Some SMT formula with symbols from the theory of arrays.

$a = store(b, k, v) \land select(a, i) \neq select(b, i) \land select(a, j) \neq select(b, j) \land i \neq j$

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)
```

Outline of the Section on Arrays

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[15] [**leino˙this˙2016**].

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

---

[15] https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

Arrays in Boogie

Implementation of an insertion sort[16] algorithm in Boogie:

```
1  procedure InsertionSort(lo : int, hi : int, a : [int]int) returns
       (ar : [int]int)
2  {
3      var i, j, temp : int;
4      ar := a;
5      i := lo+1;
6      while (i <= hi) {
7          j := i;
8          while (j > lo && ar[j] < ar[j-1])
9          {
10             temp := ar[j-1];
11             ar[j-1] := ar[j];
12             ar[j] := temp;
13             j := j-1;
14         }
15         i := i+1;
16     }
17 }
```

---

[16]https://en.wikipedia.org/wiki/Insertion_sort

TODO say something about modeling memory via arrays

Outline of the Section on Arrays

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

$$\mathcal{G}_{\mathsf{Boo}} = (\Sigma_{\mathsf{Boo}}, N_{\mathsf{Boo}}, P_{\mathsf{Boo}}, S_{\mathsf{Boo}})$$

$\Sigma_{\mathsf{Boo}} = \{\mathtt{while}, \mathtt{if}, \mathtt{else}, \{, \}\} \cup \Sigma_{\mathsf{B}}$

$N_{\mathsf{Boo}} = \{X_{\mathsf{stmt}}, X_{\mathsf{lhs}}\} \cup N_{\mathsf{B}}$

$P_{\mathsf{Boo}} = \{X_{stmt} \quad \rightarrow \quad X_{lhs}\mathtt{:=}X_{expr}\mathtt{;}$

$\qquad\quad X_{stmt} \quad \rightarrow \quad X_{stmt}X_{stmt}$

$\qquad\quad X_{stmt} \quad \rightarrow \quad \mathtt{if} \ (X_{expr})\{X_{stmt}\} \ \mathtt{else} \ \{X_{stmt}\}$

$\qquad\quad X_{stmt} \quad \rightarrow \quad \mathtt{while} \ (X_{expr})\{X_{stmt}\}$

$\qquad\quad {\color{purple}X_{lhs}} \quad {\color{purple}\rightarrow} \quad {\color{purple}X_{var}[X_{expr}]}$

$\qquad\quad X_{lhs} \quad \rightarrow \quad X_{var}\} \cup P_{\mathsf{B}}$

$S_{\mathsf{Boo}} = X_{stmt}$

Semantics of the Array Assignment Statement

> **Reminder (Assignment Statement)**
>
> $[\![\texttt{x := expr;}]\!]$ is $\{(s_1, s_2) \in S_{V,\mu} \times S_{V,\mu} \mid [\![x' = expr \land \bigwedge_{v \in V, v \neq x} v' = v]\!]_{\mathcal{M}, \rho}$ is **true**
>
> and $\rho = s_1 \cup \text{prime}(s_2)\}$

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

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

An Array Assignment Axiom for the Hoare Proof System

Reminder (Assignment Axiom)

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

$$(arrassig) \frac{}{\{\varphi[a \mapsto \texttt{store(a,i,expr)}]\} \ \texttt{a[i]:=expr;} \ \{\varphi\}}$$

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

$[\![\text{a[i]:=expr;}]\!]$ is
$\{(s_1, s_2) \in S_{V,\mu} \times S_{V,\mu} \mid \; [\![a' = store(a, i, expr) \wedge \bigwedge_{v \in V, v \neq a} v' = v]\!]_{\mathcal{M}, \rho}$ is **true**

$\qquad\qquad$ and $\rho = s_1 \cup prime(s_2)\}$

Proof. Analogously to the proof for the assignment statement.

Section 10

Boogie and Boostan – Part 2

Outline

In this section we will discuss nondeterminism and assumptions

Our goals:

▶ Learn that we can model input and other kinds of nondeterminism via Boogie's havoc statement.

▶ Add support for the havoc statement to Boostan.

▶ Learn that we can model additional assumptions via Boogie's assume statement.

▶ Add support for the assume statement to Boostan.

Reminder: Many verification tools do not apply their algorithms directly to the input language. Instead, they translate the input program to a program in a language with a sparse syntax (e.g., Boogie) and apply the verification algorithms on the translation output. Hence, language like Boogie must allow one to model all features of high level languages.

We take that into account and wa add features to Boostan such that we are in principle able to model high-level languages.

Nondeterminism and Havoc statement

Nondeterminism

Typical feature of computer programs: input

- ▶ user input
- ▶ network input
- ▶ input from other hardware

How can we model input in a general/abstract way?

- ▶ In some sense we already do ...
- ▶ Variables are not initialized, may have any value at the beginning
- ▶ Can we just use one auxiliary variable per user input?
- ▶ No. Input may occur inside a loop.

Nondeterminism in Boogie

тодо havoc statement in Boogie
example for copy&paste,
reference to chapter in Boogie specification

Nondeterminsm in Boostan

What do we have to extend?

- ▶ Syntax
- ▶ Semantics
- ▶ Rules of the Hoare proof system
- ▶ Soundness proof for the Hoare proof system

Grammar for Boostan with Havoc Statement

$$\mathcal{G}_{\mathsf{Boo}} = (\Sigma_{\mathsf{Boo}}, N_{\mathsf{Boo}}, P_{\mathsf{Boo}}, S_{\mathsf{Boo}})$$

$\Sigma_{\mathsf{Boo}} = \{\mathtt{while}, \mathtt{if}, \mathtt{else}, \{, \}, \mathtt{havoc}\} \cup \Sigma_{\mathsf{B}}$

$N_{\mathsf{Boo}} = \{X_{\mathsf{stmt}}, X_{\mathsf{lhs}}\} \cup N_{\mathsf{B}}$

$$
\begin{aligned}
P_{\mathsf{Boo}} = \{X_{stmt} &\rightarrow X_{lhs} \mathtt{:=} X_{expr}\mathtt{;} \\
X_{stmt} &\rightarrow \mathtt{havoc}\ X_{var}\mathtt{;} \\
X_{stmt} &\rightarrow X_{stmt} X_{stmt} \\
X_{stmt} &\rightarrow \mathtt{if}\ (X_{expr})\{X_{stmt}\}\ \mathtt{else}\ \{X_{stmt}\} \\
X_{stmt} &\rightarrow \mathtt{while}\ (X_{expr})\{X_{stmt}\} \\
X_{lhs} &\rightarrow X_{var}[X_{expr}] \\
X_{lhs} &\rightarrow X_{var}\} \cup P_{\mathsf{B}}
\end{aligned}
$$

$S_{\mathsf{Boo}} = X_{Boo}$

Semantics of the Havoc Statement

Reminder (Assignment Statement)

$[\![\texttt{x := expr;}]\!]$ is $\{(s_1, s_2) \in S_{V,\mu} \times S_{V,\mu} \mid \ [\![x' = expr \wedge \bigwedge_{v \in V, v \neq x} v' = v]\!]_{\mathcal{M}, \rho}$ is **true**

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

Given a program $P = (V, \mu, \mathcal{T})$ we define the semantics of a havoc statement $[\![\texttt{havoc x;}]\!]$ as the following binary relation over program states.

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

Program Verification

Summer Term 2020

Lecture 13: Havoc, Assume, CFGs

Matthias Heizmann

The following slides will be discussed on Monday 29th June

Notes for students of the 2020 course
- None.

A Havoc Axiom for the Hoare Proof System

Reminder (Assignment Axiom)

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

$$(havoc)\frac{}{\{\forall x. \ \varphi\} \ \texttt{havoc x;} \ \{\varphi\}}$$

Soundness of the Havoc Axiom

Lemma (Soundness of the Havoc Axiom)

The Hoare triple $\{\forall x.\ \varphi\}$ `havoc x;` $\{\varphi\}$ is valid.

Reminder

$[\![$`havoc x;`$]\!]$ is $\{(s_1, s_2) \in S_{V,\mu} \times S_{V,\mu} \mid [\![\bigwedge_{v \in V, v \neq x} v' = v]\!]_{\mathcal{M},\rho}$ is **true** and $\rho = s_1 \cup \text{prime}(s_2)\}$

Proof. Let $s' \in post(\{\forall x.\varphi\}, [\![$`havoc x;`$]\!])$

$\Rightarrow$ There exists s such that $s \in \{\forall x.\varphi\}$ and $(s, s') \in [\![$`havoc x;`$]\!]$

$\Rightarrow$ There exists s such that for $\rho = s \cup \text{prime}(s')$
$[\![\forall x.\varphi \wedge \bigwedge_{v \in V, v \neq x} v' = v]\!]_{\mathcal{M},\rho}$ is **true** and $\rho = s \cup \text{prime}(s')$

$\Rightarrow$ for $\rho = s'$ the evalution $[\![\varphi]\!]_{\mathcal{M},\rho}$ is **true**

$\Rightarrow$ $s' \in \{\varphi\}$

Assumptions

How can we restrict input to certain values?

Not a feature of programming languages.

Assumptions in Boogie

TODO assume statement in Boogie
example for copy&paste,
reference to chapter in Boogie specification

Nondeterminism in Boostan

What do we have to extend?

- ▶ Syntax
- ▶ Semantics
- ▶ Rules of the Hoare proof system
- ▶ Soundness proof for the Hoare proof system

Grammar for Boostan with Assume Statement

$$\mathcal{G}_{\text{Boo}} = (\Sigma_{\text{Boo}}, N_{\text{Boo}}, P_{\text{Boo}}, S_{\text{Boo}})$$

$\Sigma_{\text{Boo}} = \{\texttt{while}, \texttt{if}, \texttt{else}, \{, \}, \texttt{havoc}, \texttt{assume}\} \cup \Sigma_{\text{B}}$

$N_{\text{Boo}} = \{X_{\text{stmt}}, X_{\text{lhs}}\} \cup N_{\text{B}}$

$\begin{aligned}
P_{\text{Boo}} = \{ X_{stmt} &\rightarrow X_{lhs}\texttt{:=}X_{expr}\texttt{;} \\
X_{stmt} &\rightarrow \texttt{havoc } X_{var}\texttt{;} \\
X_{stmt} &\rightarrow \texttt{assume } X_{expr}\texttt{;} \\
X_{stmt} &\rightarrow X_{stmt}X_{stmt} \\
X_{stmt} &\rightarrow \texttt{if } (X_{expr})\{X_{stmt}\} \texttt{ else } \{X_{stmt}\} \\
X_{stmt} &\rightarrow \texttt{while } (X_{expr})\{X_{stmt}\} \\
X_{lhs} &\rightarrow X_{var}[X_{expr}] \\
X_{lhs} &\rightarrow X_{var}\} \cup P_{\text{B}}
\end{aligned}$

$S_{\text{Boo}} = X_{Boo}$

Semantics of the Assume Statement

Given a program $P = (V, \mu, \mathcal{T})$ we define the semantics of an assume statement $[\![\texttt{assume expr;}]\!]$ as the following binary relation over program states.

$$\{(s_1, s_2) \in S_{V,\mu} \times S_{V,\mu} \mid s_1 = s_2 \text{ and } s_2 \in \{expr\}\}$$

Alternatively

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

An Assume Axiom for the Hoare Proof System

$$(assu) \frac{}{\{\varphi\} \; \texttt{assume expr;} \; \{\varphi \land expr\}}$$

Soundness of the Assume Axiom

Lemma (Soundness of the Assume Axiom)

The Hoare triple $\{\varphi\}$ `assume expr;` $\{\varphi \wedge expr\}$ is valid.

Reminder

$[\![$`assume expr;`$]\!]$ is $\{(s_1, s_2) \in S_{V,\mu} \times S_{V,\mu} \mid s_1 = s_2 \text{ and } s_1 \in \{expr\}\}$

Section 11

Control-flow graphs

Outline

In this section we will see a new formalism for computer programs, namely *control-flow graphs*. Control-flow graphs[17] are a well-established concept in coputer science for which several different but very similar notions exists.

Goals of this section are

▶ fix our notation of a control-flow graph and the corresponding terminology

▶ give a characterization of program correctness (in the sense of safety, precondition-postcondition pairs)

▶ see (again) an example of a complex object is defined by (structural) induction over a context-free grammar

▶ see (again) an example of a complex proof that is given by (structural) induction over a context-free grammar

---

[17]https://en.wikipedia.org/wiki/Control-flow_graph

Motivation
Formal Definition
Program Executions
Proof of the Error Execution Theorem

From the Guide for Finding a Derivation in the Hoare Proof System and the UltimateEliminator tool we learned that it is (at least conceptually) rather easy to give a proof once we found suitable loop invariants.

In the remaining weeks we will see techniques for finding loop invariants but in order to do so we need a new formalism for programs: the control-flow graph.

Relational Semantics vs. Semantics based on control-flow graphs

Relational semantics
- ▶ conceptually simple
- ▶ suitable for defining the Hoare proof system
- ▶ intractable for algorithms (e.g., because we cannot compute the reflexive transitive closure)

Semantics based on control-flow graphs
- ▶ more suitable for verifiation algorithms that do not need loop invariants
- ▶ uses relational semantics for "simple statements"

On the next slide, we see an example of a control-flow graph and we probably already have an idea what a control-flow graph should be.
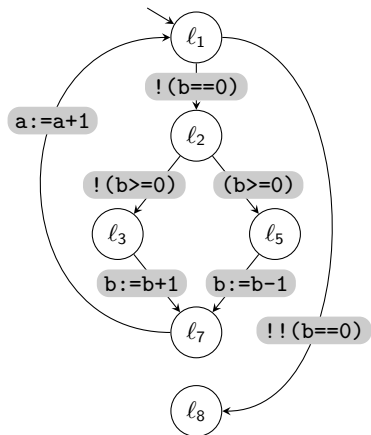
Question: how can we introduce the notion of a control-flow graph formally?

Example: Control-flow Graph

Code of program P_{ab}

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

Control-flow graph of P_{ab}

Outline of the Section on Control-flow graphs

Motivation
Formal Definition
Program Executions
Proof of the Error Execution Theorem

Definition (Control-Flow Graph)

A *control-flow graph* is a tuple $G = (Loc, \Delta, \ell_{\text{init}}, \ell_{\text{ex}})$ where

- *Loc* is a finite set whose elements we call *locations*,
- Δ is a ternary relation that consists of triples (ℓ, st, ℓ') where ℓ and ℓ' are locations and *st* is either
  - an assignment statement,
  - an array assignment statement,
  - a havoc statement, or
  - an assume statement.
- ℓ_{init} is a location that we call the *initial location*
- ℓ_{ex} is a location that we call the *exit location*

Definition (Control-Flow Graph for a Program)

Given a program $P = (V, \mu, st)$ we say that a control-flow graph G is a *control-flow graph for P* if G is a control-flow graph for *st* which we define inductively on the next slides.

Control-Flow Graph for Simple Statements
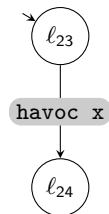
Definition:

Let *st* be

- an assignment statement,
- an array assignment statement,
- a havoc statement, or
- an assume statement,

then $G = (Loc, \Delta, \ell_{\text{init}}, \ell_{\text{ex}})$ such that

- $Loc = \{\ell_{\text{init}}, \ell_{\text{ex}}\}$,
- $\Delta = \{(\ell_{\text{init}}, st, \ell_{\text{ex}})\}$, and
- $\ell_{\text{init}} \neq \ell_{\text{ex}}$

is a control-flow graph for *st*.

Example:

Notational Conventions

In order to improve legibility of a control-flow graph, we typically

- ▶ put a gray box around statements,
- ▶ omit the "assume" prefix of assume statements (i.e., write `b>=0` instead of `assume b>=0`), and
- ▶ omit the semicolon at the end of (array-)assume statements and havoc statements.

Control-Flow Graphs are Unique up to Locations

We do not define "the" control-flow graph for a statement, we only define when a graph is "a" control-flow graph of a given statement. We do so because we do not want to fix a naming scheme for the locations. Using the terminology of graph theory we can say that all control-flow graphs for a given statement are isomorphic to each other.

Control-Flow Graph for a Sequential Composition

Let $G^1 = (Loc^1, \Delta^1, \ell^1_{\text{init}}, \ell^1_{\text{ex}})$ be a control-flow graph for st_1,
let $G^2 = (Loc^2, \Delta^2, \ell^2_{\text{init}}, \ell^2_{\text{ex}})$ be a control-flow graph for st_2
such that Loc^1 and Loc^2 are disjoint.

Let $G^3 = (Loc^3, \Delta^3, \ell^3_{\text{init}}, \ell^3_{\text{ex}})$ be the modification of G^2 where we replaced every
occurrence of ℓ^2_{init} by ℓ^1_{ex}, i.e.,

▶ $Loc^3 = Loc^2 \setminus \{\ell^2_{\text{init}}\} \cup \{\ell^1_{\text{ex}}\}$

▶ $\Delta^3 = \quad \{(\ell^1_{\text{ex}}, st, \ell') \mid (\ell^2_{\text{init}}, st, \ell') \in \Delta^2\}$
$\cup \{(\ell, st, \ell^1_{\text{ex}}) \mid (\ell, st, \ell^2_{\text{init}}) \in \Delta^2\}$
$\cup \{(\ell, st, \ell') \mid (\ell, st, \ell') \in \Delta^2 \text{ s.t. } \ell \neq \ell^2_{\text{init}} \text{ and } \ell' \neq \ell^2_{\text{init}}\}$

▶ $\ell^3_{\text{init}} = \ell^1_{\text{ex}}$

▶ $\ell^3_{\text{ex}} = \ell^2_{\text{ex}}$

Then $G = (Loc^1 \cup Loc^3, \Delta^1 \cup \Delta^3, \ell^1_{\text{init}}, \ell^3_{\text{ex}})$ is a control-flow graph for the
sequential composition $st_1 st_2$.

Control-Flow Graph for a Conditional Statement

Let $G^1 = (Loc^1, \Delta^1, \ell_{init}^1, \ell_{ex}^1)$ be a control-flow graph for st_1,
let $G^2 = (Loc^2, \Delta^2, \ell_{init}^2, \ell_{ex}^2)$ be a control-flow graph for st_2
such that Loc^1 and Loc^2 are disjoint.

The definition of a control-flow graph for the conditional statement

$$\texttt{if(expr)}\{st_1\} \texttt{ else } \{st_2\}$$

is the task of Exercise 2 on Exercise Sheet 13.

Program Verification

Summer Term 2020

Lecture 14: CFGs cont'd

Matthias Heizmann

The following slides will be discussed on Wednesday 1st July

Notes for students of the 2020 course
- None.

Control-Flow Graph for a While Statement

Let $G = (Loc, \Delta, \ell_{\text{init}}, \ell_{\text{ex}})$ be a control-flow graph for st.
Let ℓ_{ex}^w be a location that does not occur in Loc.

Then $G^w = (Loc^w, \Delta^w, \ell_{\text{init}}^w, \ell_{\text{ex}}^w)$ such that

- $Loc^w = Loc \cup \{\ell_{\text{ex}}^w\}$
- $\Delta^w = \Delta \cup \{(\ell_{\text{ex}}, \boxed{\texttt{assume expr}}, \ell_{\text{init}})\} \cup \{(\ell_{\text{ex}}, \boxed{\texttt{assume !expr}}, \ell_{\text{ex}}^w)\}$
- $\ell_{\text{init}}^w = \ell_{\text{ex}}$
- $\ell_{\text{ex}}^w = \ell_{\text{ex}}^w$

is a control-flow graph for the statement

```
while (expr) { st }.
```

Example: Control-flow Graph

Code of program P_{ab}

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

Control-flow graph of P_{ab}
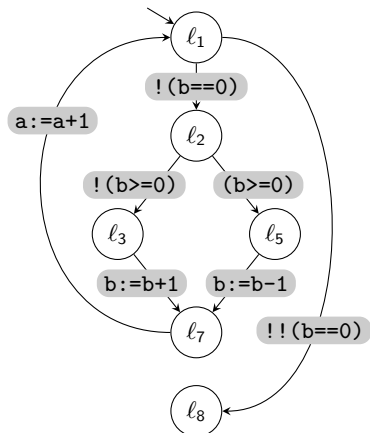
Outline of the Section on Control-flow graphs

Motivation
Formal Definition
Program Executions
Proof of the Error Execution Theorem

Control-Flow and Data

The control-flow graph is only an alternative syntactic representation of a program. In order to get also an alternative view on the program's behavior we will make several new definitions in this subsection.

The graph structure of the control-flow graph captures only one aspect of a program, namely it defines the way in which the programmer arranged the statements in the code. This graph structure allows us the specify where the program currently is but this formalism does not (directly) allow us to talk about the data that is stored in the program's variables.

Our definition of a program state is focussed solely on the program's data but it is not sufficient to specify the situation in which a program currently is, because the state does not provide information about the next statements that can be executed.

We will next give several definitions that combine control-flow apects and data aspects of a program and allow us to talk about program correctness in our control-flow graph-based formalism.

Program Configuration and Execution

Let $P = (V, \mu, st)$ be a program and $G = (Loc, \Delta, \ell_{\text{init}}, \ell_{\text{ex}})$ be a control-flow graph for P.

Definition (Program Configuration)

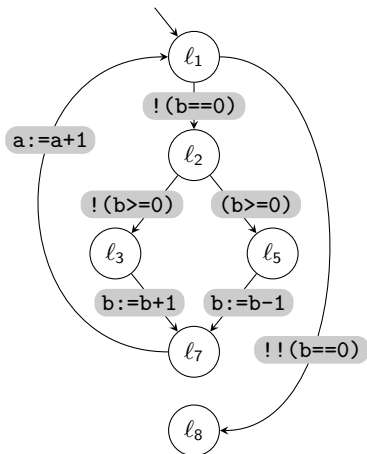We call a pair (ℓ, s) a *program configuration* of P if $\ell \in Loc$ is a location and s is a state of P.

Definition (Execution)

We call a sequence of program configurations $(\ell_0, s_0), \ldots, (\ell_n, s_n)$ an *execution* of P if there exists a sequence of statements $st_1 \ldots st_n$ such that for each $i \in \{0, \ldots n - 1\}$

▶ $(\ell_i, st_{i+1}, \ell_{i+1}) \in \Delta$ and

▶ $(s_i, s_{i+1}) \in [\![st_{i+1}]\!]$

Example: Execution

Control-flow graph of P_{ab}

An execution of program P_{ab}

$$(\ell_5, \{a \mapsto 42, b \mapsto 23\})$$
$$(\ell_7, \{a \mapsto 42, b \mapsto 22\})$$
$$(\ell_1, \{a \mapsto 43, b \mapsto 22\})$$
$$(\ell_2, \{a \mapsto 43, b \mapsto 22\})$$
$$(\ell_5, \{a \mapsto 43, b \mapsto 22\})$$
$$(\ell_7, \{a \mapsto 43, b \mapsto 21\})$$

This is a typical (boring) example of an execution. Executions do not have to start at the initial location. Executions do not have to end at the exit location.

Let $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$ be a precondition-postcondition pair for P.

Definition

We call the program configuration (ℓ, s)

▶ *initial*, if $\ell = \ell_{\mathsf{init}}$ and $s \in \{\varphi_{\mathsf{pre}}\}$

▶ an *error configuration* if $\ell = \ell_{\mathsf{ex}}$ and $s \notin \{\varphi_{\mathsf{post}}\}$

Note that later in this course we will introduce assert statements and then we will extend the definition of an error configuration.

Theorem (PppSatAndExec)

The program $P = (V, \mu, st)$ satisfies the precondition-postcondition pair $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$ iff there exists no execution $(\ell_0, s_0), \ldots, (\ell_n, s_n)$ such that (ℓ_0, s_0) is an initial configuration and (ℓ_n, s_n) is an error configuration.

We will discuss the proof of this theorem later. First we will see how we can use this theorem.

Does the following program satisfy the given precondition-postcondition pair?

```
1  while (x % 1337 != 0) {
2    if (y % 37 != 0) {
3      x = (3 * x) % (256 * 256);
4      y = (- 2 * y + 1) % (256 * 256);
5    } else {
6      tmp = x;
7      x = y;
8      y = tmp;
9    }
10 }
```

φ_{pre} : $\quad x = 1 \wedge y = 1$
φ_{post} : $\quad y \leq 31337$

If a program is correct, we can give a derivation in the Hore proof system. Let's assume that we tried for hours to find a derivation but failed and now have the impression that the program does not satisfy the precondition-postcondition pair. So far we only had one way to formally show that $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$ is not satisfied: compute the binary relation over states for this program to check if every pair satisfies the precondition-postcondition pair. Unfortunately, the relation of this program is very complex and we (resp. at least the lecturer) do not have an idea how to compute it efficiently. Thanks to Theorem PppSatAndExec we now have an alternative within our formal setting: we can give an execution.

Since there is only one single initial configuration we can run the program and in case it terminates we can check the value of the variable y.

```c
#include <stdio.h>

int main(void) {
  unsigned short x = 1;
  unsigned short y = 1;
  while(x % 1337 != 0) {
    if (y % 37 !=0 ) {
      x = (3 * x);
      y = (- 2 * y + 1);
    } else {
      unsigned short tmp = x;
      x = y;
      y = tmp;
    }
    printf("value of x is %d\n", x);
    printf("value of y is %d\n", y);
  }
  return 0;
}
```

In order to do so we implemented the program in C.

The output has 8616 lines, it starts with the lines

```
value of x is 3
value of y is 65535
```

and ends with the lines

```
value of x is 33425
value of y is 43691
```

We assume optimistically that this C program really mimics the Boostan program from the preceding slide and conclude that the Boo program does not satisfy the given precondition-postcondition pair.
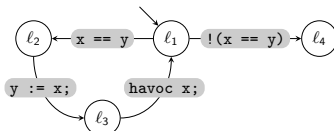
Example

Consider the program P_{xor} with $V = \{x, y\}$, $\mu(x) = \mu(y) = \{\textbf{true}, \textbf{false}\}$.

Q: Does the following program satisfy the given precondition-postcondition pair?

```
1  while (x == y) {
2    y := x;
3    havoc x;
4  }
```

$\varphi_{pre}:$ x
$\varphi_{post}:$ $x \rightarrow \neg y$

A: Yes. Loop invariant **true** is sufficient. If we leave the loop then x and y are disjoint.

Q: Does Theorem PppSatAndExec also allow us to give an alternative proof that uses executions?

A: Not directly. The program has infinitely many executions that start in the initial location and end in the exit location. We cannot check all of them.

The definitions on the next slide will however help us to approach the problem.

Definition (Reachable Program Configuration)

We call a configuration (ℓ, s) *reachable* if there exists a program execution $(\ell_0, s_0), \ldots, (\ell_n, s_n)$ such that (ℓ_0, s_0) is an initial configuration and $(\ell_n, s_n) = (\ell, s)$.

Theorem

A program satisfies a given precondition-postcondition pair iff the set of reachable configurations does not contain an error configuration.

Proof. Follows directly from Theorem PppSatAndExec and the definition above.

Can we compute the set of reachable program configurations?

Theorem

The set of reachable configurations RC is the smallest set such that
- ▶ *each initial configuration is an element of RC*
- ▶ *if $(\ell, s) \in RC$, $(\ell, st, \ell') \in \Delta$ and $(s, s') \in [\![st]\!]$ then $(\ell', s') \in RC$.*

Proof. Nontrivial. Later in this course.

This algorithm hints a simple (possibly nonterminating) algorithm for the construction of the set of reachable configurations. For the construction of this set the following graph can be helpful.

Definition

The *reachability graph* is a pair (RC, T) such that $\big((\ell, s), st, (\ell', s')\big) \in T$ iff $(\ell, st, \ell') \in \Delta$ and $(s, s') \in [\![st]\!]$

Exercise: Construct the reachability graph for the Program P_{xor}. (Exercise Sheet 14)

Outline of the Section on Control-flow graphs

In this subsection, we prove the Theorem PppSatAndExec.

We start by stating the following lemma. The theorem follows directly from this lemma, the definition of an error configuration and the definition of satisfiability of precondition-postcondition pairs..

Lemma (RelAndExec)

Let $G = (Loc, \Delta, \ell_{\text{init}}, \ell_{\text{ex}})$ be a control-flow graph for st, then there exists a program execution $(\ell_0, s_0), \ldots, (\ell_n, s_n)$ with $\ell_0 = \ell_{\text{init}}$ and $\ell_n = \ell_{\text{ex}}$, iff $(s_0, s_n) \in [\![st]\!]$.

Proof. By induction over the height of st's derivation tree. Using the five lemmas from the remaining subsection, the proof can be carried out analogously to the soundness proof for the Hoare proof system.

Lemma (RelAndExec.1)

Let st be an assignment statement, an array assignment statement, a havoc statement, or an assume statement, and let $G = (Loc, \Delta, \ell_{\text{init}}, \ell_{\text{ex}})$ be a control-flow graph for st. There exists a program execution $(\ell_0, s_0), \ldots, (\ell_n, s_n)$ with $\ell_0 = \ell_{\text{init}}$ and $\ell_n = \ell_{\text{ex}}$, iff $(s_0, s_n) \in [\![st]\!]$.

Proof.
Since st is an assignment statement, an array assignment statement, a havoc statement, or an assume statement, the control-flow graph has the form
$G = (\{\ell_{\text{init}}, \ell_{\text{ex}}\}, \{(\ell_{\text{init}}, st, \ell_{\text{ex}})\}, \ell_{\text{init}}, \ell_{\text{ex}})$.

"$\Rightarrow$" By definition of a program execution we have $(\ell_i, st_{i+1}, \ell_{i+1}) \in \Delta$ for each $i \in \{0, \ldots n-1\}$. Since Δ contains only one element and $\ell_{\text{init}} \neq \ell_{\text{ex}}$, the execution is a sequence of length 2 and has the form $(\ell_{\text{init}}, s_0), (\ell_{\text{ex}}, s_1)$. By definition of a program execution there has to be some statement such st_1 that $(\ell_{\text{init}}, st_1, \ell_{\text{ex}}) \in \Delta$ and $(s_0, s_1) \in [\![st_1]\!]$. Since there is only one statement in the control-flow graph, st_1 is st.

"$\Leftarrow$" Let (s, s') be a pair of states for which the assumption $(s, s') \in [\![st]\!]$ holds. Since $(\ell_{\text{init}}, st, \ell_{\text{ex}}) \in \Delta$, the sequence $(\ell_{\text{init}}, s), (\ell_{\text{ex}}, s')$ is an execution.

Lemma (RelAndExec.2)

Let $G = (Loc, \Delta, \ell_{\text{init}}, \ell_{\text{ex}})$ be a control-flow graph for the sequential composition $st_1 st_2$. There exists a program execution $(\ell_0, s_0), \ldots, (\ell_n, s_n)$ with $\ell_0 = \ell_{\text{init}}$ and $\ell_n = \ell_{\text{ex}}$, iff $(s_0, s_n) \in [\![st_1 st_2]\!]$.

Proof. Exercise 1 on Exercise Sheet 14.

Lemma (RelAndExec.3)

Let $G = (Loc, \Delta, \ell_{\text{init}}, \ell_{\text{ex}})$ be a control-flow graph for the conditional statment $if(expr)\{st_1\}\,else\{st_1\}$. There exists a program execution $(\ell_0, s_0), \ldots, (\ell_n, s_n)$ with $\ell_0 = \ell_{\text{init}}$ and $\ell_n = \ell_{\text{ex}}$, iff $(s_0, s_n) \in [\![if(expr)\{st_1\}\,else\{st_1\}]\!]$.

Proof. Analogously to the other proofs in this subsection. No carried out in the lecture.

Program Verification

Summer Term 2020

Lecture 15: CFGs cont'd, Predicate Transformers

Matthias Heizmann

The following slides will be discussed on Monday 6th July

Notes for students of the 2020 course
- None.

Lemma (RelAndExec.4)

Let $G = (Loc, \Delta, \ell_{\text{init}}, \ell_{\text{ex}})$ be a control-flow graph for the statement $while(expr)\{st\}$. There exists a program execution $(\ell_0, s_0), \ldots, (\ell_n, s_n)$ with $\ell_0 = \ell_{\text{init}}$ and $\ell_n = \ell_{\text{ex}}$, iff $(s_0, s_n) \in [\![while(expr)\{st\}]\!]$.

Proof.

Let $G' = (Loc^{st}, \Delta^{st}, \ell_{\text{init}}^{st}, \ell_{\text{ex}}^{st})$ be the control flow graph for st from which G is built. Then $\ell_{\text{init}} = \ell_{\text{ex}}^{st}$. Furthermore, let $R = (\{expr\} \times S_{V,\mu}) \cap [\![st]\!]$.

" $\Longleftarrow$ " Let $(s_0, s_n) \in [\![\texttt{while(expr)}\{st\}]\!] = R^* \cap (S_{V,\mu} \times \{\texttt{!expr}\})$. Then $s_n \in \{\texttt{!expr}\}$, and $(s_0, s_n) \in R^k$ for some $k \in \mathbb{N}_0$. We perform induction over k:

▶ For $k = 0$, it follows that $s_0 = s_n \in \{\texttt{!expr}\}$. By the definition of a CFG for while-statements, G has an edge $(\ell_{\text{init}}, \boxed{\texttt{assume !expr;}}, \ell_{\text{ex}})$. Hence the sequence $(\ell_{\text{init}}, s_0), (\ell_{\text{ex}}, s_n)$ is a program execution.

▶ For $k + 1$, we observe that there exists some s' such that $(s_0, s') \in R$ and $(s', s_n) \in R^k$. By induction hypothesis, there is an execution $(\ell_m, s_m), \ldots, (\ell_n, s_n)$ with $s_m = s'$, $\ell_m = \ell_{\text{init}}$ and $\ell_n = \ell_{\text{ex}}$.

From $(s_0, s') \in R$ we conclude $s_0 \in \{\texttt{expr}\}$ and $(s_0, s') \in [\![st]\!]$. By structural induction over the program, it follows that there is an execution $(\ell_1, s_1), \ldots, (\ell_m, s_m)$ with $\ell_1 = \ell_{\text{init}}^{st}$, $s_1 = s_0$, $\ell_m = \ell_{\text{ex}}^{st}$ and $s_m = s'$. Furthermore, $(\ell_{\text{init}}, \boxed{\texttt{assume expr}}, \ell_1) \in \Delta$ and $(s_0, s_1) \in [\![\texttt{assume expr;}]\!]$. Hence we combine the executions as $(\ell_0, s_0), (\ell_1, s_1), \ldots, (\ell_m, s_m), \ldots, (\ell_n, s_n)$.

" $\Rightarrow$ " Let $(\ell_0, s_0), \ldots, (\ell_n, s_n)$ be such a program execution. We perform the induction over k, the number of occurrences of $\ell_{ex}^{st} = \ell_{init}$ among the ℓ_i.

▶ For $k = 1$ ($k = 0$ is not possible), we must have $n = 1$, as the only incoming transition of $\ell_n = \ell_{ex}$ is $(\ell_{init},$ `assume !expr` $, \ell_{ex})$. Hence we must have $s_0 = s_n \in \{!\mathtt{expr}\}$ and thus $(s_0, s_n) \in [\![\mathtt{while\ (expr)}\ \{st\}]\!]$.

▶ For $k + 1$, let m be the second occurrence of ℓ_{init}, i.e., $\ell_m = \ell_{init}$ with $m > 0$ and $\ell_j \neq \ell_{init}$ for all $0 < j \leq m$. Then $(\ell_m, s_m), \ldots, (\ell_n, s_n)$ is an execution and by induction $(s_m, s_n) \in [\![\mathtt{while(expr)}\{st\}]\!]$, i.e., $(s_m, s_n) \in R^*$ and $s_n \in \{!\mathtt{expr}\}$.

Then $\ell_1 = \ell_{init}^{st}$, as that is the only other outgoing edge from $\ell_{init} = \ell_0$, and hence $(s_0, s_1) \in [\![\mathtt{assume\ expr;}]\!]$ and $s_0 = s_1 \in \{\mathtt{expr}\}$. The sequence $(\ell_1, s_1), \ldots, (\ell_m, s_m)$ is then an execution of st (in G') with $\ell_m = \ell_{ex}^{st}$, and by structural induction it follows that $(s_1, s_m) \in [\![st]\!]$. Thereby $(s_0, s_m) \in R$.

By sequential composition with (s_m, s_n) we conclude $(s_0, s_n) \in R^*$. Finally, it follows that $(s_0, s_n) \in [\![\mathtt{while(expr)}\{st\}]\!]$.

∎

Section 12

Predicate Transformers

Outline

In this section we will learn about program transformers[18] which will be our main means for analyzing the effect of statements in our control-flow graph-based view on programs..

Goals of this section are

▶ learn to execute a loop-free program symbolically (i.e., on all inputs in parallel)

▶ deepen our understanding about the connection between formulas and sets of states

▶ learn to simplify formulas by eliminating quantifiers

Outline of the Section on Predicate Transformers

Motivation

Example

Consider the program P_{xor} with $V = \{x, y, z\}$, $\mu(x) = \mu(y) = \mu(z) = \mathbb{Z}$.

Q: Does the following program satisfy the given precondition-postcondition pair?

```
1 z := y + x;
2 assume x <= -25;
3 x := z * z - 2;
4 assume x < 0;
5 havoc x;
6 assume (z*(y\%23)<-20);
```
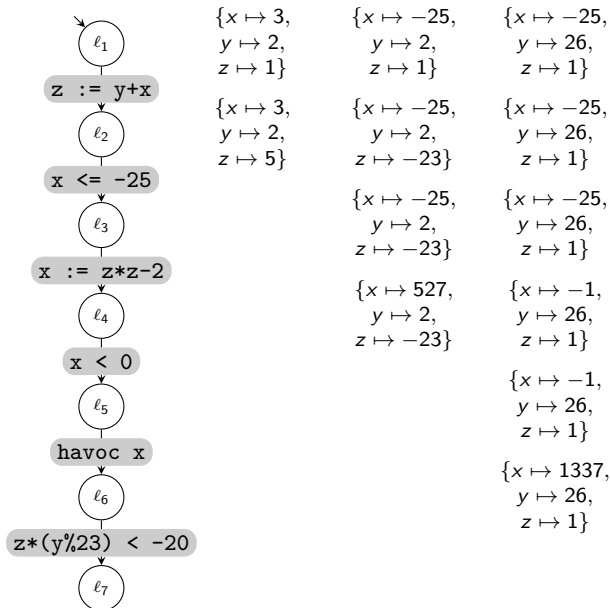
$\varphi_{\text{pre}}: \quad y \geq 1$
$\varphi_{\text{post}}: \quad y \geq 45$

From Theorem PppSatAndExec we know that we can disprove correctness by finding an execution that starts in an initial configuration and ends in an error configuration. On the next slide we will try to find such an execution.

Unlike a program from the last section this program has more than one initial states.

▶ We do not really know where we should start, apply an naive approach where we pick some state and run the program.

▶ The execution "gets stuck" at the first assume statement. We wonder if only this execution does not reach the error configuration or if all executions cannot reach an error configuration. We realize that we could have passed the first assume statement if we would have started in a different state. We pick a different initial configuration and restart do construct an execution.

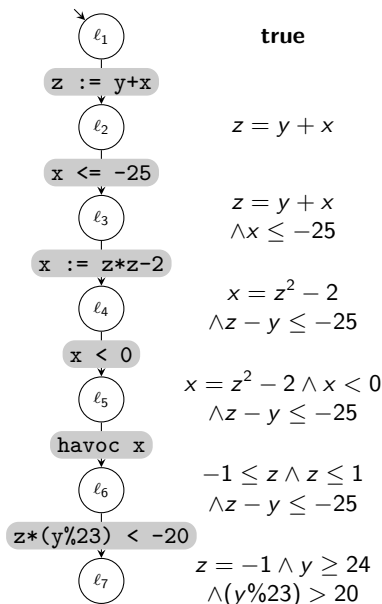▶ The execution "gets stuck" at the second assume statement. We wonder if only this execution does not reach the error configuration or if all executions cannot reach an error configuration. We realize that we could have passed the second assume statement if we would have started in a different state. We pick a different initial configuration and restart do construct an execution.

▶ ...

Example

$$\{x \mapsto 3, \\ y \mapsto 2, \\ z \mapsto 1\}$$

$$\{x \mapsto -25, \\ y \mapsto 2, \\ z \mapsto 1\}$$

$$\{x \mapsto -25, \\ y \mapsto 26, \\ z \mapsto 1\}$$

$$\{x \mapsto 3, \\ y \mapsto 2, \\ z \mapsto 5\}$$

$$\{x \mapsto -25, \\ y \mapsto 2, \\ z \mapsto -23\}$$

$$\{x \mapsto -25, \\ y \mapsto 26, \\ z \mapsto 1\}$$

$$\{x \mapsto -25, \\ y \mapsto 2, \\ z \mapsto -23\}$$

$$\{x \mapsto -25, \\ y \mapsto 26, \\ z \mapsto 1\}$$

$$\{x \mapsto 527, \\ y \mapsto 2, \\ z \mapsto -23\}$$

$$\{x \mapsto -1, \\ y \mapsto 26, \\ z \mapsto 1\}$$

$$\{x \mapsto -1, \\ y \mapsto 26, \\ z \mapsto 1\}$$

$$\{x \mapsto 1337, \\ y \mapsto 26, \\ z \mapsto 1\}$$

In order to find a suitable execution you probably did (maybe implicitly) compute the set of all states that are reachable after each of the statements. The next slide shows formulas whose satisfying variable assignments are exactly the reachable sets of states.

In the next subsection we define the *strongest post predicate transfomer* which allows us to compute these formulas.

Example

ℓ_1 **true**

`z := y+x`

ℓ_2 $z = y + x$

`x <= -25`

ℓ_3 $z = y + x$
$\wedge x \leq -25$

`x := z*z-2`

ℓ_4 $x = z^2 - 2$
$\wedge z - y \leq -25$

`x < 0`

ℓ_5 $x = z^2 - 2 \wedge x < 0$
$\wedge z - y \leq -25$

`havoc x`

ℓ_6 $-1 \leq z \wedge z \leq 1$
$\wedge z - y \leq -25$

`z*(y%23) < -20`

ℓ_7 $z = -1 \wedge y \geq 24$
$\wedge (y\%23) > 20$

Outline of the Section on Predicate Transformers

Strongest Post

First, we state informally the properties that our definition of the strongest post operator sp should have. Then we discuss how we could give a formal definition.

Idea:
Given a set of states S and a statement st, the strongest postcondition $sp(S, st)$ is the set of states for which the following holds. If there is a state $s \in S$

- in which we can execute st,
- in which st terminates, and
- s' is a successor after executing st

then $s' \in sp(S, st)$.

Post Image

Reminder (Post Image)

Given a binary relation R over the set X and a subset of $Y \subseteq X$, the *postimage of Y under R*, denoted $post(Y, R)$, is the set
$\{x \in X \mid$ exists $y \in Y$ 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

$$post(Y, R) = \{y \in \mathbb{Z} \mid y \geq 6\}$$

Definition (Strongest Postcondition)

Given a set of states S and a statement st the *strongest postcondition* is the post image of S under the relation $[\![st]\!]$, i.e.

$$\mathrm{sp}(S, st) = post(S, [\![st]\!]).$$

Example

see Exercise Sheet 15

In one of the next subsections we will see some special cases in which the resulting state of the strongest postcondition can be represented by a formula if the input was represented by a formula.

Outline of the Section on Predicate Transformers

The goal of this short subsection/excursus is to deepen/recap our understanding of the connection between formulas and sets of states.

Reminder (Implication)

Given a (possibly infinite) set of FOL formulas Γ and a PL formula ψ, we say that Γ *implies* ψ if for all models $\mathcal{M}$ and for all variable assignments ρ the following holds.

$$\text{If } [\![\varphi]\!]_{\mathcal{M},\rho} \text{ is \textbf{true} for all } \varphi \in \Gamma \text{ then also } [\![\psi]\!]_{\mathcal{M},\rho} \text{ is \textbf{true}}$$

We use $\vDash$ to denote this binary implication relation and we say that the implication $\Gamma \vDash \psi$ holds if Γ implies ψ. Furthermore, we say that φ implies ψ, written $\varphi \vDash \psi$, if the implication $\{\varphi\} \vDash \psi$ holds.

Reminder: Since the end of our introduction to logics we consider only models $\mathcal{M}$ in which the axioms of all SMT-LIB theories are valid.

Theorem (Duality of Implication and Subset)

$$\varphi \text{ implies } \psi \qquad \textit{iff} \qquad \{\rho \mid [\![\psi]\!]_{\mathcal{M},\rho} \text{ is true}\} \subseteq \{\rho \mid [\![\varphi]\!]_{\mathcal{M},\rho} \text{ is true}\}$$

Reminder (Sets of Program States)

Given a program $P = (V, \mu, \mathcal{T})$ we defined $\{\varphi\} := \{s \in S_{V,\mu} \mid [\![\varphi]\!]_{\mathcal{M},\rho} \text{ is \textbf{true} for } \rho = s\}$

Corollary (Duality of Implication and Subset)
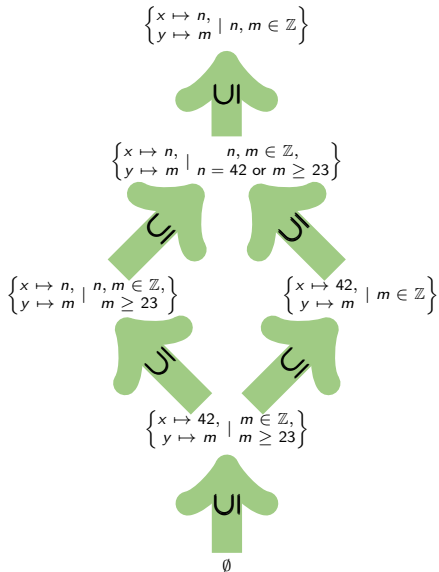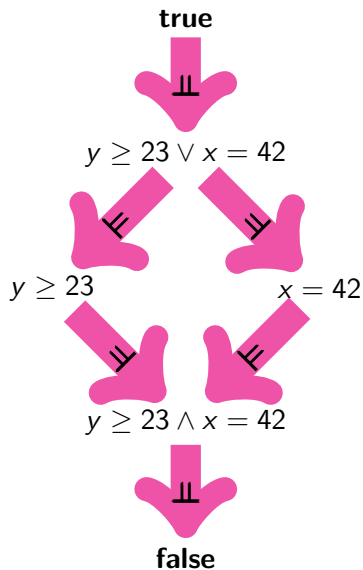
$$\varphi \text{ implies } \psi \qquad \text{iff} \qquad \{\psi\} \subseteq \{\varphi\}$$

true

$y \geq 23 \lor x = 42$

$y \geq 23$ $x = 42$

$y \geq 23 \land x = 42$

false

$$\left\{ \begin{array}{l} x \mapsto n, \\ y \mapsto m \end{array} \;\middle|\; n, m \in \mathbb{Z} \right\}$$

$$\left\{ \begin{array}{l} x \mapsto n, \\ y \mapsto m \end{array} \;\middle|\; \begin{array}{l} n, m \in \mathbb{Z}, \\ n = 42 \text{ or } m \geq 23 \end{array} \right\}$$

$$\left\{ \begin{array}{l} x \mapsto n, \\ y \mapsto m \end{array} \;\middle|\; \begin{array}{l} n, m \in \mathbb{Z}, \\ m \geq 23 \end{array} \right\}$$ $$\left\{ \begin{array}{l} x \mapsto 42, \\ y \mapsto m \end{array} \;\middle|\; m \in \mathbb{Z} \right\}$$

$$\left\{ \begin{array}{l} x \mapsto 42, \\ y \mapsto m \end{array} \;\middle|\; \begin{array}{l} m \in \mathbb{Z}, \\ m \geq 23 \end{array} \right\}$$

$\emptyset$

Outline of the Section on Predicate Transformers

Quantified formulas are notoriously difficult to solve. Later in this section we have to deal with quantified formulas. In this subsection we will learn about *quantifier elimination* wich is the task of finding an equivalent quanfitier-free formula for a given formula.

Quantifier Elimination

Theorem (Destructive Equality Resolution 1)

The formula $\exists x.\varphi \wedge x = t$ and the formula $\varphi[x \mapsto t]$ are equivalent.

Proof. Not given in this course. Follows directly from the axioms of equality and the semantics of existential quantification and conjunction.

Problem: Formula does not have required form.

Solution: Do equivalence transformation which solves equality for subject $\hat{x}$.

Example

	$\exists \hat{x}.(\hat{x}\%2) = 0 \wedge x = \hat{x} + 1$
equivalent to	$\exists \hat{x}.(\hat{x}\%2) = 0 \wedge \hat{x} = x - 1$
equivalent to	$(x - 1)\%2 = 0$

Program Verification

Summer Term 2020

Lecture 16: Predicate Transformers

Matthias Heizmann

The following slides will be discussed on Wednesday 8th July

Notes for students of the 2020 course
- None.

Problem: Since $\hat{x}$ is an integer we cannot simply divide by 2.

Solution: We can divide by 2 if we add the conjunct $(x\%2) = 0$.

Example

Let $x, \hat{x}$ be variable symbols whose sort is *Int*.

$$\exists \hat{x}.\, select(a, \hat{x}) = 23 \wedge x = 2 \cdot \hat{x}$$

equivalent to $\quad \exists \hat{x}.\, select(a, \hat{x}) = 23 \wedge \hat{x} = x \operatorname{div} 2 \wedge (x\%2) = 0$

equivalent to $\quad select(a, x \operatorname{div} 2) = 23 \wedge (x\%2) = 0$

Problem: Since y could be 0, we cannot simply divide by y.

Solution: Case distinction. (Does eliminate quantifier but reduces its scope.)

Example

Let $x, \hat{x}, y$ be variable symbols whose sort is *Real*.

$$\exists \hat{x}.\, select(a, \hat{x}) = 23 \wedge x = y \cdot \hat{x}$$

equivalent to $\quad \exists \hat{x}. \quad select(a, \hat{x}) = 23 \wedge x = y \cdot \hat{x} \wedge y \neq 0$

$\qquad \vee \quad select(a, \hat{x}) = 23 \wedge x = y \cdot \hat{x} \wedge y = 0$

equivalent to $\qquad select(a, x/y) = 23 \wedge y \neq 0$

$\qquad \vee \quad (\exists \hat{x}.select(a, \hat{x}) = 23) \wedge x = 0 \wedge y = 0$

Quantifier Elimination

Theorem (Destructive Equality Resolution 2)

The formula $\forall x.\varphi \vee x \neq t$ *and the formula* $\varphi[x \mapsto t]$ *are equivalent.*

Proof. Negate und and use the destructive equality resolution theorem for existential quantification. Discussed only very briefly in the lecture.

Outline of the Section on Predicate Transformers

In practice we represent sets of states by formulas and we would like to let a machine compute the strongest post operator.
The definition of the strongest post operator does not directly tell us how we can implement the operator.

In this subsection we will see characterizations of the strongest post operator that will ease an implementation of the operator. For these characterizations, we consider each kind of statement individually and we always consider the special case where the set of states is given by a formula.

Theorem (Strongest Post of the Assignment Statement)

$sp(\{\varphi\},$ `x := expr` $)$ $\quad$ is $\quad \{\exists \hat{x}.\varphi[x \mapsto \hat{x}] \wedge x = expr[x \mapsto \hat{x}]\}$

Reminder (Semantics of the Assignment Statement)

$[\![x := expr;]\!]$ is $\{(s_1, s_2) \in S_{V,\mu} \times S_{V,\mu} \mid [\![x' = expr \wedge \bigwedge\limits_{v \in V, v \neq x} v' = v]\!]_{\mathcal{M},\rho}$ is **true**

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

Proof.

$sp(\{\varphi\}, \text{x := expr;})$

$= \{s_2 \mid \text{exists } s_1 \in \{\varphi\} \text{ and } (s_1, s_2) \in [\![x := expr;]\!]\}$

$= \{s_2 \mid \text{exists } s_1 \in S_{V,\mu} \text{ and } [\![\varphi \wedge x' = expr \wedge \bigwedge\limits_{v \in V, v \neq x} v' = v]\!]_{\mathcal{M},\rho} \text{ is } \textbf{true}$

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

$= \{s_2 \mid \ [\![\exists \hat{v}_1 \ldots \exists \hat{v}_n.\varphi[v_1 \mapsto \hat{v}_1, \ldots, v_n \mapsto \hat{v}_n] \wedge x = expr[v_1 \mapsto \hat{v}_1, \ldots, v_n \mapsto \hat{v}_n]$

$\qquad \wedge \bigwedge\limits_{v \in V, v \neq x} v = \hat{v}]\!]_{\mathcal{M},\rho} \text{ is } \textbf{true} \text{ and } \rho = s_2\}$

$= \{s_2 \mid \ [\![\exists \hat{x}.\varphi[x \mapsto \hat{x}] \wedge x = expr[x \mapsto \hat{x}]]\!]_{\mathcal{M},\rho} \text{ is } \textbf{true} \text{ and } \rho = s_2\}$

$= \{\exists \hat{x}.\varphi[x \mapsto \hat{x}] \wedge x = expr[x \mapsto \hat{x}]\}$

Theorem (Strongest Post of the Havoc Statement)

$$sp(\{\varphi\}, \texttt{havoc x;}) \quad is \quad \{\exists x.\varphi\}$$

Reminder (Semantics of the Havoc Statement)

$\llbracket \texttt{havoc x;} \rrbracket$ is $\{(s_1, s_2) \in S_{V,\mu} \times S_{V,\mu} \mid \llbracket x' = expr \wedge \bigwedge\limits_{v \in V, v \neq x} v' = v \rrbracket_{\mathcal{M}, \rho}$ is **true**

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

Proof. Very similar to the proof for the Assignment Statement.

Theorem (Strongest Post of the Assume Statement)

$$sp(\{\varphi\}, \texttt{assume expr;}) \quad is \quad \{\varphi \wedge expr\}$$

Reminder (Semantics of the Assume Statement)

$\llbracket\texttt{assume expr;}\rrbracket$ is $\{(s_1, s_2) \in S_{V,\mu} \times S_{V,\mu} \mid s_1 = s_2 \text{ and } s_2 \in \{expr\}\}$

Proof.

$sp(\{\varphi\}, \texttt{assume expr;})$

$= \{s_2 \mid \text{exists } s_1 \in \{\varphi\} \text{ and } (s_1, s_2) \in \llbracket\texttt{assume expr;}\rrbracket\}$

$= \{s_2 \mid \text{exists } s_1 \in \{\varphi\} \text{ and } s_1 = s_2 \text{ and } s_2 \in \{expr\}\}$

$= \{\varphi \wedge expr\}$

Theorem (Strongest Post of the Sequential Composition)

If st is an sequential composition of the form $st_1 st_2$, then $sp(S, st)$ is $sp(sp(S, st_1), st_2)$.

Reminder (Semantics of the Sequential Composition)

$$[\![st_1 st_2]\!] \text{ is } [\![st_1]\!] \circ [\![st_2]\!] \qquad \text{"the relational composition"}$$

Reminder (Post Image)

$$post(Y, R) = \{x \in X \mid \text{exists } y \in Y \text{ such that } (y, x) \in R\}$$

Proof.
$sp(S, st_1 st_2)$
$= \{s_3 \in S_{V,\mu} \mid \text{exists } s_1 \in \{\varphi\} \text{ such that } (s_1, s_3) \in [\![st_1 st_2]\!]\}$
$= \{s_3 \in S_{V,\mu} \mid \text{exists } s_1 \in \{\varphi\}, \text{exists } s_2 \in S_{V,\mu} \text{ such that } (s_1, s_2) \in [\![st_1]\!]$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{and } (s_2, s_3) \in [\![st_2]\!]\}$
$= \{s_3 \in S_{V,\mu} \mid \text{exists } s_2 \in sp(\{\varphi\}, st_1) \text{ such that } (s_2, s_3) \in [\![st_2]\!]\}$
$= sp(sp(S, st_1), st_2)$

Theorem (Strongest Post of the Conditional Statement)

See Exercise 1 of Exercise Sheet 16.

Strongest Post of the While Statement

In general, we cannot express the strongest post of the while statement as a formula. However, we give a characterization that motivates that the strongest post of a while statement can be very complex in some cases and that is can also be computed in some special cases.

We define the *k-th iterative application application* of the strongest post operator $\underbrace{sp(\ldots sp(}_{k \text{ times}} S, st) \ldots)$ formally as follows.

Notation

$$sp^k(S, st) = \begin{cases} S & \text{if } k = 0 \\ sp(sp^{k-1}(S, st), st) & \text{if } k > 0 \end{cases}$$

Theorem

If st is a while statement of the form `while(expr){st}` *then* $sp(S, st)$ *is*

$$\bigcup_{k \in \mathbb{N}} sp(sp^k(S, \boxed{\texttt{assume expr; }} st), \boxed{\texttt{assume !expr; }})$$

Strongest Post of the While Statement

There are however several examples in which the strongest post of a while statement can be expressed by a formula.

Example

$sp(\{i = 0\}, \texttt{while(b)}\{\texttt{i := i+1; havoc b;}\})$ is $\{\neg b \wedge i \geq 0\}$

Example

$sp(\{i = 0 \wedge i \geq n\}, \texttt{while(i<n)}\{\texttt{a[i] := 0; i := i+1;}\})$ is

$$\{i = n \wedge (\forall k.(0 \leq k \wedge k < n) \rightarrow select(a, k) = 0)\}$$

ᴛᴏᴅᴏ Say something about the research area of loop acceleration.

Outline of the Section on Predicate Transformers

Analogously to the strongest post predicate transformer *sp*, we defined the weakest precondition predicate transformer *wp* in Exercise 4 of Exercise Sheet 16

Program Verification

Summer Term 2020

Lecture 17: Bounded Model Checking

Matthias Heizmann

The following slides will be discussed on Monday 13th July

Notes for students of the 2020 course

- None.

Section 13

Bounded Model Checking

Outline

In this section we will learn the bounded model checking [**ac/BiereCCSZ03**] verification technique. We give a rather non-standard introduction to the topic because we use want to use definition that we need later in this course.

Goals of this section are

▶ set formal basis for the later sections on abstractions

▶ learn that positive tests results can be deceptive even if we have a high test coverage

▶ see an algorithm that can find bugs is erroneous programs

Outline of the Section on Bounded Model Checking

Abstract Reachability Graph

Precise Abstract Reachability Graph

Algorithms for Constructing Graphs

In this subsection we will only see two definitions. This definitions will be the basis for the following subsection that the basis for the sections on abstractions. There will be no examples in this subsection but we will see several examples in the remaining course. TODO add links to examples

Let $G = (Loc, \Delta, \ell_{\text{init}}, \ell_{\text{ex}})$ be a control-flow graph for a program $P = (V, \mu, st)$.

Definition

An *abstract (program) configuration* is a pair $(\ell, \{\varphi\})$ where ℓ is a location and φ is a formula over the program's variables.

Definition

An *abstract reachability graph* is a pair (AC, T) such that AC is a set of abstract configurations such that

1. for each abstract configuration $(\ell, \{\varphi\})$ for which $\varphi \neq$ **false** and there exists $(\ell, st, \ell') \in \Delta$, there is a an abstract configuration $(\ell', \{\varphi'\})$ such that $sp(\{\varphi\}, st) \subseteq \{\varphi'\}$ and $((\ell, \{\varphi\}), st, (\ell', \{\varphi'\})) \in T$

2. $(\ell_{\text{init}}, \{\varphi_{\text{pre}}\}) \in AC$, and

3. for each abstract configuration $(\ell, \{\varphi\})$ there is a path from $(\ell_{\text{init}}, \{\varphi_{\text{pre}}\})$ to $(\ell, \{\varphi\})$.

We will come back to the (general) abstract reachability graph later and next consider a special case first.

Outline of the Section on Bounded Model Checking

Definition

A *precise abstract reachability graph* is an abstract reachability graph (AC, T) such that for each $(\ell, \{\varphi\}), st, (\ell', \{\varphi'\}))$ the equality $sp(\{\varphi\}, st) = \{\varphi'\}$ holds.

▶ This definition is very similar to the definition of the abstract reachability graph but the inclusion in the first bullet point is always an equality.

▶ A precise abstract reachability graph for a given control-flow graph is unique up to the formulas that represent the set of state at each location.

We will next consider an implementation of the greatest common divisor (GCD) (see Exercise 1 on Exercise Sheet 06) and wonder if the implementation is correct.

We have seen in Exercise 2 of Exercise Sheet 04 that formulas that express the GCD are rather complex and hence we take as the postcondition only one property of the GCD. Rationale: if this property is violated, our implementation is bad and if the property is satisfied, we can strengthen the property.

In order to check correctness, we proceed as follows. We first run some tests. For each test, we pick an initial program configuration, construct an execution that starts with that initial configuration and check if the execution ends in an error configuration.
In order to improve legibility, we depict the tests as tables and omit the values of the variables a_{in} and b_{in}.

Example: a (faulty?) Implementation of the GCD

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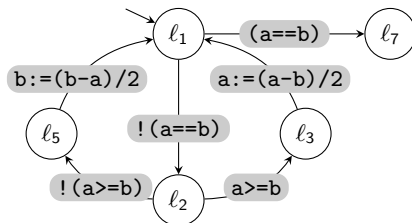

$$\varphi_{\text{pre}} : \quad a_{\text{in}} = a \wedge b_{\text{in}} = b \wedge a_{\text{in}} > 0 \wedge b_{\text{in}} > 0$$
$$\varphi_{\text{post}} : \quad a_{\text{in}}\%a == 0 \wedge b_{\text{in}}\%b == 0$$

	a	b
ℓ_1	9	15
ℓ_2	9	15
ℓ_5	9	15
ℓ_1	9	3
ℓ_2	9	3
ℓ_3	9	3
ℓ_1	3	3
ℓ_7	3	3

	a	b
ℓ_1	40	24
ℓ_2	40	24
ℓ_5	40	24
ℓ_1	8	24
ℓ_2	8	24
ℓ_3	8	24
ℓ_1	8	8
ℓ_7	8	8

	a	b
ℓ_1	11	5
ℓ_2	11	5
ℓ_3	11	5
ℓ_1	3	5
ℓ_2	3	5
ℓ_5	3	5
ℓ_1	3	1
ℓ_2	3	1
ℓ_3	3	1
ℓ_1	1	1
ℓ_7	1	1

None of the three tests showed a violation of the postcondition. Furthermore these test look like they cover most of the program's behavior.

▶ Every statement is covered by some test.

▶ We have a test that takes the if-branch first, and we have a test that takes the else-branch first.

▶ We have a test for the corner case that both inputs are prime numbers.

▶ We have a test for the corner case that the result consists of several similar prime factors.

We might be tempted to believe that the program is correct.

However, if we start to build the precise abstract reachability graph, we see after a few iterations that there is an abstract program configuration whose set of states is not a subset of the postcondition.

Control-flow graph of our
GCD implementation
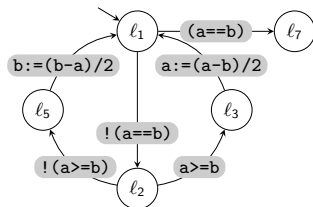
Prefix of
precise abstract reachability graph
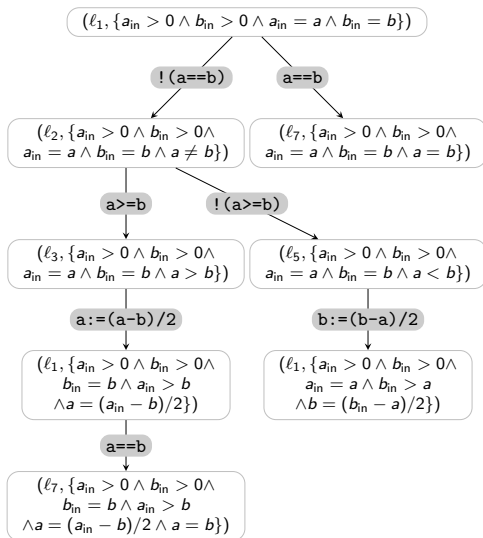
We saw that there is an abstract program configuration whose set of states is not a subset of the postcondition. The next lemma and the next theorem explain the consequences.

Let P be a program and let G be a control-flow graph for P.

Lemma

If the precise abstract reachability graph for G contains an abstract error configuration, then the set of reachable configurations contains an error configuration.

Proof. TODO

Theorem

Let $(\varphi_{\text{pre}}, \varphi_{\text{post}})$ be a precondition-postcondition pair. If the precise abstract reachability graph for G contains an abstract error configuration, then the program does not satisfy $(\varphi_{\text{pre}}, \varphi_{\text{post}})$.

Proof. TODO

Outline of the Section on Bounded Model Checking

Abstract Reachability Graph
Precise Abstract Reachability Graph
Algorithms for Constructing Graphs

In this subsection we will see

▶ an algorithm for the construction of the reachability graph (RC, T) and

▶ an algorithm for the construction of the precise abstract reachability graph (AC, T).

TODO Explain the algorithms: BFS traversal of the graph while it is build, perhaps no suprises if you are familiar with BFS traveral of graphs.

1: **procedure** $\text{CONSTRUCTRC}((Loc, \Delta, \ell_{\text{init}}, \ell_{\text{ex}}) : \text{CFG}, \varphi_{\text{pre}} : \text{Precondition})$
      returns (RC, T)
2:    $RC \leftarrow \emptyset,\ T \leftarrow \emptyset,\ \text{worklist} \leftarrow \emptyset$
3:    **for all** $s \in \{\varphi_{\text{pre}}\}$ **do**
4:        $RC \leftarrow RC \cup \{(\ell_{\text{init}}, s)\}$
5:        $\text{worklist} \leftarrow \text{worklist} \cup \{(\ell_{\text{init}}, s)\}$
6:    **end for**
7:    **while** $\text{worklist} \neq \emptyset$ **do**
8:        $(\ell, s) \leftarrow \text{REMOVEFIRST}(\text{worklist})$
9:        **for all** ℓ', st with $(\ell, st, \ell') \in \Delta$ **do**
10:          **for all** s' with $(s, s') \in [\![st]\!]$ **do**
11:              $T \leftarrow T \cup \{((\ell, s), st, (\ell', s'))\}$
12:              **if** $(\ell', s') \notin RC$ **then**
13:                  $RC \leftarrow RC \cup \{(\ell', s')\}$
14:                  $\text{worklist} \leftarrow \text{worklist} \cup \{(\ell', s')\}$
15:              **end if**
16:          **end for**
17:      **end for**
18:    **end while**
19: **end procedure**

```
 1: procedure ConstructAC((Loc, Δ, ℓ_init, ℓ_ex) : CFG, φ_pre : Precondition)
              returns (AC, T)
 2:     T ← ∅
 3:     AC ← {(ℓ_init, {φ_pre})}
 4:     worklist ← {(ℓ_init, {φ_pre})}

 5:     while worklist ≠ ∅ do
 6:         (ℓ, S) ← RemoveFirst(worklist)
 7:         for all ℓ', st with (ℓ, st, ℓ') ∈ Δ do
 8:             S' ← sp(S, st)
 9:             T ← T ∪ {((ℓ, S), st, (ℓ', S'))}
10:             if (ℓ', S') ∉ AC then
11:                 AC ← AC ∪ {(ℓ', S')}
12:                 if S' ≠ {false} then
13:                     worklist ← worklist ∪ {(ℓ', S')}
14:                 end if
15:             end if
16:         end for
17:     end while
18: end procedure
```

Algorithm for Bounded Model Checking

Apply the algorithm CONSTRUCTAC with the following modifications.

- ▶ Check if (ℓ', S) is an error configuration while adding it to the graph.
- ▶ If (ℓ', S) is an error configuration, then discontinue to build the graph and tell the user that his program is not safe.
- ▶ Construct a path $(\ell_0, S_0), \ldots, (\ell_0, S_0)$ such that (ℓ_0, S_0) is $(\ell_{\text{init}}, \{\varphi_{\text{pre}}\})$ and (ℓ_n, S_n) is the error configuration (ℓ', S).
- ▶ Let $st_1, \ldots s_n$ be a sequence of statements such that $(\ell_{i-1}, st_i, \ell_i) \in \Delta$ for each $i \in \{1, \ldots, n\}$. Return $st_1, \ldots s_n$ to the user as a counterexample to safety of his program.

Guarantees correctness up to a *bound*:

After we explored all nodes whose distance to the initial node is smaller than k, we can guarantee the program cannot reach an error by executing less than k statements.

Program Verification

Summer Term 2020

Lecture 18: Assert, Abstractions

Matthias Heizmann

The following slides will be discussed on Wednesday 15th July

Notes for students of the 2020 course
- ▶ None.

Section 14

Correctness Specification via Assert Statement

Outline

In Boogie, Java, Python and many other programming languages there is an *assert statement* [19] . The assert statement consists of a Boolean expression that is evaluated when the program executes the statement. If the expression is evaluated to *true* the execution continues regularly. If the expression is evaluated to *false* the program is considered erroneous and e.g., an exception is thrown.

TODO example
We will extend Boostan by an assert statement. We will not define the relational semantics for this statement and use it only in contexts where we work with control-flow graphs. In Exercise 3 of Exercise Sheet 18 we will see that we can translate every specification given by assert statements into a specification given by a precondition-postcondition pair if we also allow a minor modification of the program.

---

[19]Wikipedia: Assertion

Grammar for Boostan with Assert Statement

$$\mathcal{G}_{\text{Boo}} = (\Sigma_{\text{Boo}}, N_{\text{Boo}}, P_{\text{Boo}}, S_{\text{Boo}})$$

$\Sigma_{\text{Boo}} = \{\texttt{while}, \texttt{if}, \texttt{else}, \{,\}, \texttt{havoc}, \texttt{assume}, \texttt{assert}\} \cup \Sigma_{\text{B}}$

$N_{\text{Boo}} = \{X_{\text{stmt}}, X_{\text{lhs}}\} \cup N_{\text{B}}$

$P_{\text{Boo}} = \{X_{stmt} \;\; \rightarrow \;\; X_{lhs}\texttt{:=}X_{expr}\texttt{;}$

$\qquad\qquad X_{stmt} \;\; \rightarrow \;\; \texttt{havoc } X_{var}\texttt{;}$

$\qquad\qquad X_{stmt} \;\; \rightarrow \;\; \texttt{assume } X_{expr}\texttt{;}$

$\qquad\qquad X_{stmt} \;\; \rightarrow \;\; \texttt{assert } X_{expr}\texttt{;}$

$\qquad\qquad X_{stmt} \;\; \rightarrow \;\; X_{stmt}X_{stmt}$

$\qquad\qquad X_{stmt} \;\; \rightarrow \;\; \texttt{if } (X_{expr})\{X_{stmt}\} \texttt{ else } \{X_{stmt}\}$

$\qquad\qquad X_{stmt} \;\; \rightarrow \;\; \texttt{while } (X_{expr})\{X_{stmt}\}$

$\qquad\qquad X_{lhs} \;\; \rightarrow \;\; X_{var}[X_{expr}]$

$\qquad\qquad X_{lhs} \;\; \rightarrow \;\; X_{var}\} \cup P_{\text{B}}$

$S_{\text{Boo}} = X_{\text{Boo}}$

Reminder(Control-Flow Graph)

A *control-flow graph* is a tuple $G = (Loc, \Delta, \ell_{init}, \ell_{ex})$ where

- Loc is a finite set whose elements we call *locations*,

- Δ is a ternary relation that consists of triples (ℓ, st, ℓ') where ℓ and ℓ' are locations and st is either

  - an assignment statement,
  - an array assignment statement,
  - a havoc statement, or
  - an assume statement.

- ℓ_{init} is a location that we call the *initial location*
- ℓ_{ex} is a location that we call the *exit location*

Definition (Control-flow graph with error locations)

A *control-flow graph with error locations* is a tuple
$G = (Loc, \Delta, \ell_{init}, \ell_{ex}, Loc_{err})$ where

- $G = (Loc, \Delta, \ell_{init}, \ell_{ex})$ is a control-flow graph and

- $Loc_{err} \subseteq Loc$ is a subset of locations that we call *error locations*

Definition (Control-Flow Graph With Error Locations for a Program)

Given a program $P = (V, \mu, st)$ we define the *control-flow graph with error locations for P* analogously to the control-flow graph for P. We always take the union of error locations of "sub control-flow graphs" and define the control-flow graph for an assert statement below.

Definition:

Let st be an assert statement of the form

```
assert expr;
```

then $G = (Loc, \Delta, \ell_{init}, \ell_{ex}, Loc_{err})$ such that

▶ $Loc = \{\ell_1, \ell_2, \ell_3\}$,

▶ $\Delta = \{(\ell_{init}, \text{assume !expr;}, \ell_{err}),$
$(\ell_{init}, \text{assume expr;}, \ell_{ex})\}$,

▶ $\ell_{init} = \ell_1$,

▶ $\ell_{ex} = \ell_3$,

▶ $Loc_{err} = \{\ell_2\}$,

▶ $\ell_{init} \neq \ell_{err}, \ell_{init} \neq \ell_{ex}$, and $\ell_{ex} \neq \ell_{err}$.

is a control-flow graph for st.

Example:
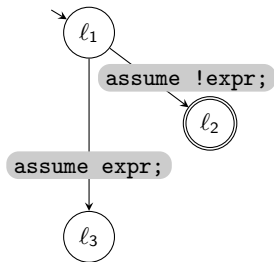
We defined the notion of an error configuration for programs with precondition-postcondition pairs. We will next extend this definition to programs with assert statements.

Definition (Error Configuration)

We call a program configuration (ℓ, s) an *error configuration* if $\ell \in Loc_{err}$.

Please note that it is bad practise to extend an existing definition

Section 15

Abstractions – Part 1

Outline

In this section we will learn a kind of correctness proof that is based on the control-flow graph.

In this section we see

▶ that bounded model checking is typically not sufficient to prove correctness of a program

▶ how abstractions help to prove correctness

(This section is directly build upon the preceding section on bounded model checking.)

Consider the program on the following slide.

- ▶ The program has infinitely many configurations. Infinitely many configurations are reachable, we cannot draw a reachability graph.
- ▶ The precise abstract reachability graph has infinitely many configurations too. Our bounded model checking algorithm will fail.
- ▶ Observation: The variable x is initially zero. We only increment x. Hence x can never become -1 and the program satisfies its precondition-postcondition pair.

Consider the graph on the following slide.

- ▶ Each node represents a configuration of the program. Different rows represent different evaluations of the variables. Different columns represent different locations. The initial configurations are highlighted blue, the error configurations are highlighted red.
- ▶ This graph is not a reachability graph because we show also non-reachable configurations.

Idea of our abstraction:

Partition configurations into finitely many equivalence classes (here: light blue boxes).
Build a graph whose nodes are the equivalence classes. Connect two equivalence classes C_1, C_2 by an edge if there are configurations $c_1 \in C_1$ and $c_2 \in C_2$ that are connected by an edge. Call an equivalence class *initial* if it contains an initial configuration. Call an equivalence class *error* if it contains an error configuration.

If there is no path from an initial equivalence class to an error equivalence class, there is also no path from an initial configuration to an error configuration and the analyzed program is correct.

Unfortunately, the contrary direction does not hold. If there is a path from an initial equivalence class to an error equivalence class, the program might be incorrect but it might also be the case the we have chosen a partition that is not helpful.

$$\varphi_{\mathsf{pre}} : x = 0$$
$$\varphi_{\mathsf{post}} : x \neq -1$$

$\{b \mapsto \mathbf{true}, x \mapsto 2\}$
$\{b \mapsto \mathbf{false}, x \mapsto 2\}$

$\{b \mapsto \mathbf{true}, x \mapsto 1\}$
$\{b \mapsto \mathbf{false}, x \mapsto 1\}$

$\{b \mapsto \mathbf{true}, x \mapsto 0\}$
$\{b \mapsto \mathbf{false}, x \mapsto 0\}$

$\{b \mapsto \mathbf{true}, x \mapsto -1\}$
$\{b \mapsto \mathbf{false}, x \mapsto -1\}$

$\{b \mapsto \mathbf{true}, x \mapsto -2\}$
$\{b \mapsto \mathbf{false}, x \mapsto -2\}$
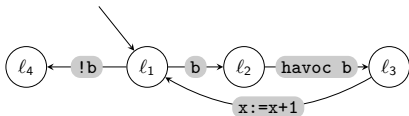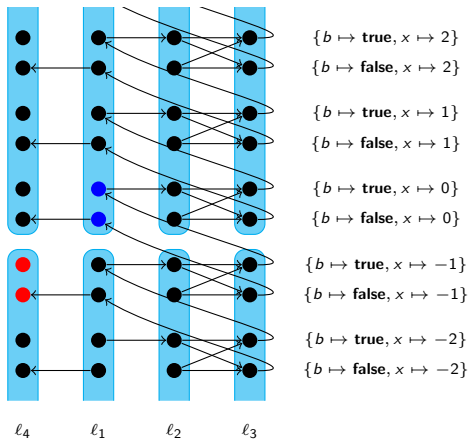
Next, we will rephrase our ideas for an abstraction in our formal setting.

We will use abstract configurations as equivalence classes (i.e., like in the example above we will never have different locations in one equivalence class).

We note that these lemma/theorem are slightly more general than the idea mentioned above. The abstract abstract reachability graph does not require that its abstract configurations form a partition of the program's configurations.

Let P be a program and let G be a control-flow graph for P.

Lemma

If some abstract reachability graph for G does not contain an abstract error configuration, then the set of reachable configurations does not contain an error configuration.

Note that there is an existential quantification: it is sufficient to find *some* abstract reachability graph that does not contain an abstract error configuration.

Theorem

Let $(\varphi_{\text{pre}}, \varphi_{\text{post}})$ be a precondition-postcondition pair. If some abstract reachability graph for G does not contain an abstract error configuration, then P satisfies $(\varphi_{\text{pre}}, \varphi_{\text{post}})$.

As a consequence we make the following definition. We will prove the theorem later and consider next some examples.

Definition

We call an abstract reachability graph for G a *safety proof* if is does not contain an abstract error configuration.

We will discuss these lemma/theorem using the following program.

TODO Explain the following program
- motivated by ...
- correct because ...

Example

Program code and control-flow graph of the program P_{goanna}

```
1 assume p != 0;
2 while (n >= 0) {
3     assert p != 0;
4     if (n == 0) {
5         p := 0;
6     }
7     n := n - 1;
8 }
```
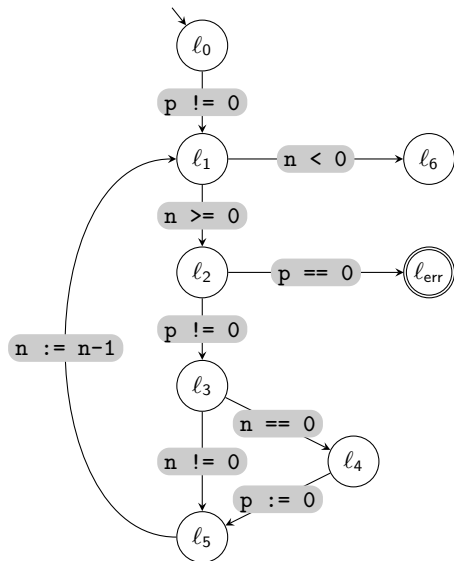
Example

Some abstract reachability graph that is suitable to show that the assert statement of P_{goanna} is always valid.
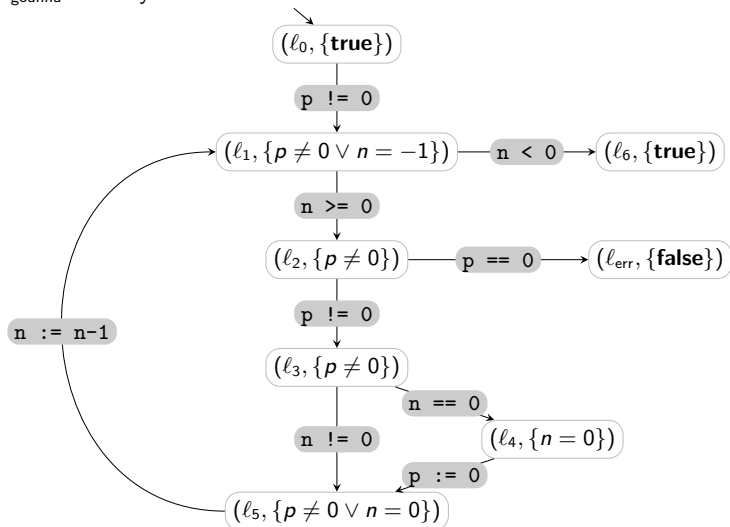
▶ The abstract reachability graph of the preceding slide is sufficient to prove safety of the program because all (resp. the only) abstract configuration whose location is ℓ_{err} has an empty set of states.

▶ The graph structure of this abstract reachability graph coincides with the graph structure of the control-flow graph. The reason for that is that the lecturer tried to find a small abstract reachability graph. We will later see examples for abstract reachability graphs of the same programs that are larger.

▶ Note that we have not yet seen an algorithm for inferring an abstract reachability graph. What we learned in the lecture allow us only the check that the above graph is indeed an abstract reachability graph for P_{goanna} and to check that this abstract reachability graph is a safety proof for P_{goanna}.

From a technical point of few that lecture has not made any progress since the presentation of the Hoare proof system.

The Hoare proof system allowed us to give a correctness proof in case we guessed "good" loop invariants.

The notion of an abstract reachability graph allows us to give a correctness proof in case we guessed "good" abstract configurations.

So, for the task of finding correctness proofs, the last sections on graph-based representations gave us only a new formalism but the task seems to be as hard as before.

In the next slides we present a approach that reduces the guesswork to the task of finding a set of formulas B. We will see an algorithm that uses the set of formulas B to construct an abstract reachability graph. If the set of formulas B was "good" the abstract reachability graph will be a safety proof.

Definition (Abstract Strongest Post)

Given a set of formulas B we define the *abstract strongest post* operator as follows.

$$sp_B^{\#}(\{\psi\}, st) = \{\bigwedge\{\varphi \in B \mid sp(\{\psi\}, st) \subseteq \{\varphi\}\}\}$$

todo Explain also all of the remaining slides

Definition

We call an abstract reachability graph (AC, T) *precise for B* if for each $(\ell, \{\varphi\}), st, (\ell', \{\varphi'\}))$ the equality $sp_B^{\#}(\{\varphi\}, st) = \{\varphi'\}$ holds.
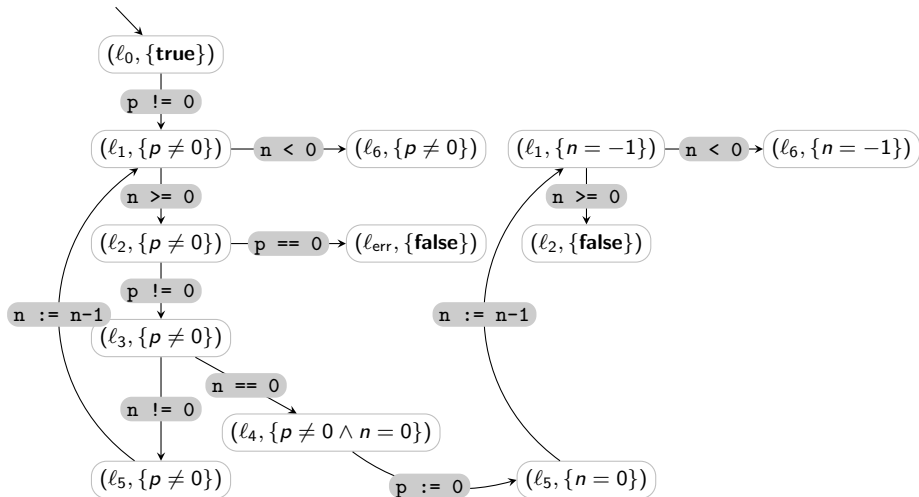
1: **procedure** $\text{CONSTRUCTACB}((Loc, \Delta, \ell_{\text{init}}, \ell_{\text{ex}}) : \text{CFG}, \varphi_{\text{pre}}, B : \text{formulas})$
       returns (AC, T)
2:     $T \leftarrow \emptyset$
3:     $AC \leftarrow \{(\ell_{\text{init}}, \{\varphi_{\text{pre}}\})\}$
4:     worklist $\leftarrow \{(\ell_{\text{init}}, \{\varphi_{\text{pre}}\})\}$

5:     **while** worklist $\neq \emptyset$ **do**
6:         $(\ell, S) \leftarrow \text{REMOVEFIRST}(\text{worklist})$
7:         **for all** ℓ', st with $(\ell, st, \ell') \in \Delta$ **do**
8:             $S' \leftarrow sp_B^{\#}(S, st)$
9:             $T \leftarrow T \cup \{((\ell, S), st, (\ell', S'))\}$
10:            **if** $(\ell', S') \notin AC$ **then**
11:                $AC \leftarrow AC \cup \{(\ell', S')\}$
12:                **if** $S' \neq \{\text{false}\}$ **then**
13:                    worklist $\leftarrow$ worklist $\cup \{(\ell', S')\}$
14:                **end if**
15:            **end if**
16:        **end for**
17:    **end while**
18: **end procedure**

Abstract reachability graph that is precise for
$B = \{p \neq 0, n = 0, n = -1, \textbf{true}, \textbf{false}\}$:

Abstract reachability graph that is precise for
$B = \{p \neq 0, n \geq 0, n = -1, \textbf{true}, \textbf{false}\}$:
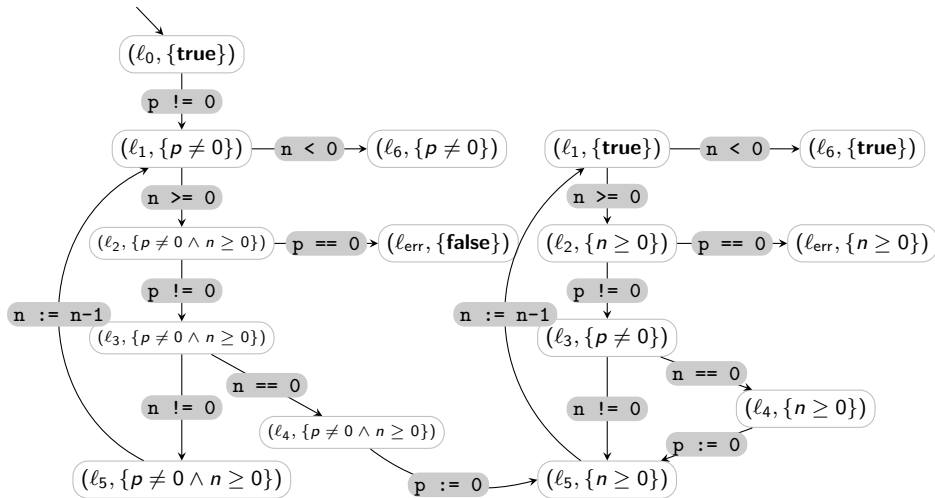
Program Verification

Summer Term 2020

Lecture 19: Infeasibility Proofs

Matthias Heizmann

The following slides will be discussed on Monday 20th July

Notes for students of the 2020 course

- None.

Section 16

Infeasibility Proofs

Outline

Motivation

Program Verification
↓
Finding Loop Invariants
↓
Finding "good" formulas for construction of an abstract reachability graph
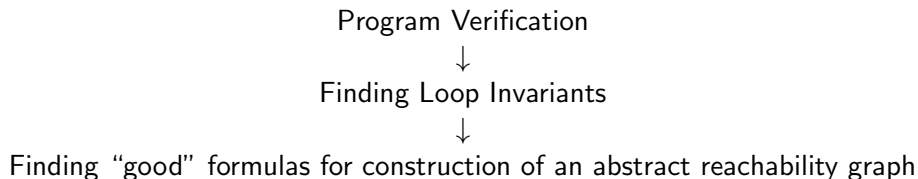
How can we get a set of formulas *B*?

Naive approach: Take all Boolean expressions that occur in the program.

Problem: Insufficient in many cases.

Workaround: Take also variations.

Problem: In the worst case the size of an abstract reachability graph (that is precise for B) grows exponentially in the size of B.

We use the next slides to discuss an idea for obtaining useful formulas.

If we want to know whether a sequence of statements has an execution or not, we can compute *sp* for the sequence and check if the resulting formula is logically equivalent to false. This does not yet help us for constructing a "good" abstract reachability graph because we need a formula after every statement.

Hence we devise a new kind of proof for the non-existence of an execution. In this new kind of proof, we have a formula after each statement and the *i*-formula denotes a superset of the states that are reachable after executing the first *i* statements.

We can obtain such a proof by applying *sp* iteratively (see middle column). However there is also a simpler proof (see right column).

Idea: Consider Proofs for sequences of statements

sequence of statements that leads from initial location to error location	proof that there is no execution	simplified proof
	φ_0 **true**	φ_0 **true**
st_1 `p != 0`		
	φ_1 $p \neq 0$	φ_1 $p \neq 0$
st_2 `n >= 0`		
	φ_2 $p \neq 0 \land n \geq 0$	φ_2 $p \neq 0$
st_3 `p == 0`		
	φ_3 **false**	φ_3 **false**

Next: a formalism for generating "simple proofs"

Definition (Trace, Feasibility)

We call a sequence of statements a *trace*. We call a trace π *feasible* if there is some execution for π.

Definition (Inductive sequence of sets of states)

Given a sequence of statements $\pi = st_1, \ldots st_n$, we call a sequence of sets of states $\{\varphi_0\}, \ldots, \{\varphi_n\}$ inductive for π if $sp(\{\varphi_i\}, st_{i+1}) \subseteq \{\varphi_{i+1}\}$ for all $i \in \{0, \ldots n-1\}$

Theorem

If there exists a sequence of sets of states $\{\varphi_0\}, \ldots, \{\varphi_n\}$ that is inductive for π such that φ_0 is true *and φ_n is* false*, then π is infeasible.*

Definition (Proof of infeasibility)

We call a sequence of sets of states $\{\varphi_0\}, \ldots, \{\varphi_n\}$ a *proof of infeasibility* if the sequence is inductive for π, φ_0 is **true**, and φ_n is **false**.

Definition (Abstraction of a statement)

We define the *abstraction of a statement* abstract(st) as follows.

$$\text{abstract}(st) = \begin{cases} \texttt{assume true} & \text{if } st \text{ is of the form} \quad \texttt{assume } \psi \\ \texttt{havoc x} & \text{if } st \text{ is of the form} \quad \texttt{x:=e} \\ \texttt{havoc x} & \text{if } st \text{ is of the form} \quad \texttt{havoc x} \end{cases}$$

Definition (Abstraction of a trace)

We call a trace $\pi^\# = st_1^\#, \ldots st_n^\#$ an *abstraction of a trace* $\pi = st_1, \ldots st_n$ if each $st_i^\#$ is either the statement st_i or the abstraction abstract(st_i).

Theorem

If $\pi^\#$ is an abstraction of π and $\{\varphi_0\}, \ldots, \{\varphi_n\}$ is a proof of infeasibility for $\pi^\#$, then $\{\varphi_0\}, \ldots, \{\varphi_n\}$ is a proof of infeasibility for π.

On the next two slides we see two examples for the construction of a "simplified proof".

The first column shows a trace. The second column shows the (unnecessarily large) infeasibility proof that is obtained by an iterative application of *sp*. The third column shows an abstraction of the trace in which we highlighted the abstracted statements in orange. The last column shows the proof that is obtained by applying *sp* to the abstraction of the trace.

trace π	sp for π	abstract trace $\pi^\#$	sp for $\pi^\#$

φ_0 **true**

st_1 `p != 0`   `p != 0`

φ_1 $p \neq 0$

st_2 `n >= 0`   `true`

φ_2 $p \neq 0 \land n \geq 0$

st_3 `p == 0`   `p == 0`

φ_3 **false**

φ_0 **true**

φ_1 $p \neq 0$

φ_2 $p \neq 0$

φ_3 **false**

trace π	sp for π	abstract trace $\pi^{\#}$	sp for $\pi^{\#}$
	φ_0 **true**		φ_0 **true**
st_1 `p != 0`		`true`	
	φ_1 $p \neq 0$		φ_1 **true**
st_2 `n >= 0`		`true`	
	φ_2 $\begin{array}{l} p \neq 0 \\ \wedge \quad n \geq 0 \end{array}$		φ_2 **true**
st_3 `p != 0`		`true`	
	φ_3 $\begin{array}{l} p \neq 0 \\ \wedge \quad n \geq 0 \end{array}$		φ_3 **true**
st_4 `n == 0`		`n == 0`	
	φ_4 $\begin{array}{l} p \neq 0 \\ \wedge \quad n = 0 \end{array}$		φ_4 $n = 0$
st_5 `p := 0`		`havoc p`	
	φ_5 $\begin{array}{l} p = 0 \\ \wedge \quad n = 0 \end{array}$		φ_5 $n = 0$
st_6 `n := n-1`		`n := n-1`	
	φ_6 $\begin{array}{l} p = 0 \\ \wedge \quad n = -1 \end{array}$		φ_6 $n = -1$
st_7 `n >= 0`		`n >= 0`	
	φ_7 **false**		φ_7 **false**
st_8 `p == 0`		`true`	
	φ_8 **false**		φ_8 **false**

Question: How can we construct the abstract trace $\pi^{\#}$?

Naive Approach: Iteratively abstract statements and check if abstract trace is still infeasible.

Advanced Approaches: (not discussed in this course) Encode trace as logical formula such that the formula is satisfiable iff the trace is feasible (SSA form). Use then either unsatisfiable cores or Craig interpolation.

In the worst case, the "advanced approaches" are not better than the "naive approach".

Good Infeasibility Proofs

trace π	abstract trace $\pi_2^\#$	sp for $\pi_2^\#$	abstract trace $\pi_1^\#$	sp for $\pi_1^\#$
		φ_0 **true**		φ_0 **true**
st_1 `a[0]:=x*x`	`havoc a[0]`		`a[0] := x*x`	
		φ_1 **true**		φ_1 $a[0] = x^2$
st_2 `n:=1000`	`n := 1000`		`havoc n`	
		φ_2 $n = 1000$		φ_2 $a[0] = x^2$
st_3 `!(n>=0)`	`!(n>=0)`		`true`	
		φ_3 **false**		φ_3 $a[0] = x^2$
st_4 `a[k]==-1`	`true`		`a[k]==-1`	
		φ_4 **false**		φ_4 $\begin{aligned} a[0] &= x^2 \\ \wedge\ \ a[k] &= -1 \end{aligned}$
st_5 `k==0`	`true`		`k==0`	
		φ_5 **false**		φ_5 **false**

Good Infeasibility Proofs

| | $\pi_2^\#$ | sp for $\pi_2^\#$ | $\pi_1^\#$ | sp for $\pi_1^\#$ |

```
1 a[0] = x * x;
2 n := 1000;
3 while (n >= 0) {
4   n := n - 1;
5 }
6 if (a[k] == -1) {
7   assert k != 0;
8 }
```

φ_0 **true**

`havoc a[0]`

φ_1 **true**

`n := 1000`

φ_2 $n = 1000$

`!(n>=0)`

φ_3 **false**

`true`

φ_4 **false**

`true`

φ_5 **false**

φ_0 **true**

`a[0] := x*x`

φ_1 $a[0] = x^2$

`havoc n`

φ_2 $a[0] = x^2$

`true`

φ_3 $a[0] = x^2$

`a[k]==-1`

φ_4 $\begin{array}{l} a[0] = x^2 \\ \wedge \quad a[k] = -1 \end{array}$

`k==0`

φ_5 **false**

Program Verification

Summer Term 2020

Lecture 20: CEGAR, Trace Abstraction

Matthias Heizmann

The following slides will be discussed on Wednesday 22th July

Notes for students of the 2020 course
- None.

Section 17

CEGAR

Outline

In this section we will see an approach that can be used to develop a verification algorithm. The approach is called CEGAR which stands for CounterExample-Guided Abstraction Refinement.

The approach is motivated by the following two facts.

▶ We do not know how to construct a set of formulas B that is suitable for a program P.
▶ We do know how we can construct a set of formulas that is suitable for a sequence of statements.

The idea is that we start with an empty set of formulas B and that we iteratively enlarge this set by formulas that we obtain from the analysis of traces.

Definition

Given an abstract reachability graph (AC, T), we call a sequence of statements $st_1, \ldots, st_n$ an *error trace in (AC, T)* if there exists a sequence of abstract configurations $(\ell_0, \{\varphi_0\}), \ldots, (\ell_n, \{\varphi_n\})$ such that

- $(\ell_0, \{\varphi_0\})$ is the initial abstract configuration,
- $((\ell_i, \{\varphi_i\}), st_{i+1}, (\ell_{i+1}, \{\varphi_{i+1}\})) \in T$ for $i \in \{0, \ldots, n-1\}$, and
- $(\ell_n, \{\varphi_n\})$ is an abstract error configuration.

Intuitively an error trace is a sequence of labelings along a path from the initial abstract configuration to an abstract error configuration.

Let us prove that P_{goanna} is correct.

```
1 assume p != 0;
2 while (n >= 0) {
3   assert p != 0;
4   if (n == 0) {
5     p := 0;
6   }
7   n := n - 1;
8 }
```
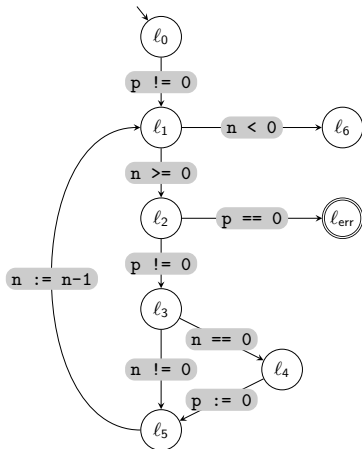

Start with $B = \emptyset$, construct abstract reachability graph that is precise for B.

Abstract reachablity graph for $B = \{p \neq 0, \textbf{false}\}$:
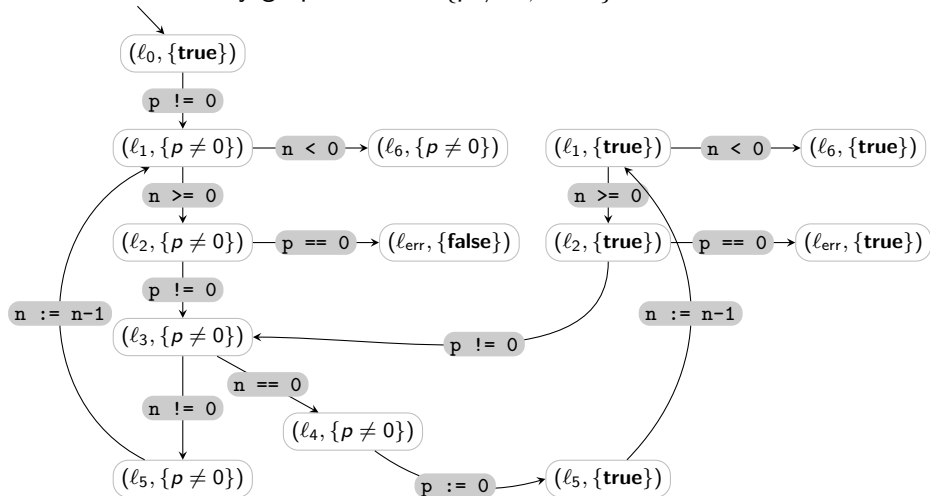
Abstract reachability graph that is precise for
$B = \{p \neq 0, n = 0, n = -1, \textbf{true}, \textbf{false}\}$:

The CEGAR Approach (Pseudocode)

Step 1: Set B to the empty set.

Step 2: Construct an abstract reachbility graph ARG that is precise for B.

Step 3: Check if ARG is safe.
If yes, report that P satisfies its specification and return.
If no, construct an error trace π of ARG.

Step 4: Check is π is feasible.
If yes, report that P does not satisfy its specification, construct an execution for π, and return.
If no, construct an infeasibility proof $\{\varphi_0\}, \ldots, \{\varphi_n\}$ for π, add the set of formulas $\{\varphi_0, \ldots, \varphi_n\}$ to B, and continue with Step 2.

The CEGAR Approach (Diagram)
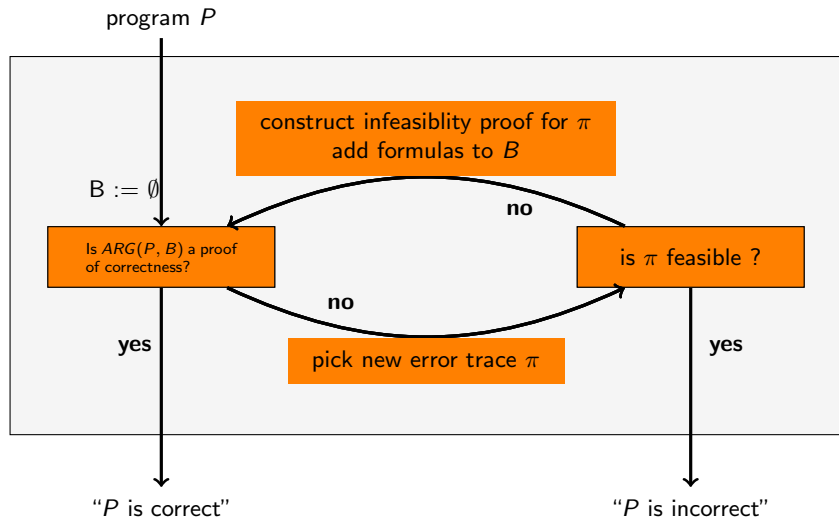
where $ARG(P, B)$ is an abstract reachability graph of P that is precise for B.

Lemma

Let π be a trace. If $\{\varphi_0\}, \ldots, \{\varphi_n\}$ is an infeasibility proof for π and $B \supseteq \{\varphi_0, \ldots \varphi_n\}$ then π is not an error trace in an abstract reachability graph that is precise for B.

Proof. Let (AC, T) be an abstract reachability graph that is precise for B. We prove by induction over the length k of prefixes of the trace $\pi = st_1, \ldots, st_n$ the following. If there is a path $(\ell_1, \{\psi_1\}), \ldots, (\ell_k, \{\psi_k\})$ such that $((\ell_i, \{\psi_i\}), st_i, (\ell_{i+1}, \{\psi_{i+1}\})) \in T$ for $i \in \{1, \ldots, k\}$ then $\{\psi_k\} \subseteq \{\varphi_k\}$.
We conclude that there is no path for π or that for the last element of the path $(\ell_n, \{\psi_n\})$ the formula ψ_n is equivalent to **false** and hence $(\ell_n, \{\psi_n\})$ is not an abstract error configuration.

Theorem (Progress property)

If an algorithm follows the CEGAR approach and π is the error trace that is analyzed in iteration i then π will not be an error trace of the abstract reachability graph in further iterations.

The theorem in the preceding slide is stated rather informally ..

Shortcomings of predicate abstraction

We need a "good" set of formulas B.

My opinion:
- yet, no good "solution" known
- many promising approaches that mitigate the problem
- for every program that was considered, someone found an algorithm that works for this program

Shortcomings of predicate abstraction

The computation of $sp_B^\#$ is costly.

Computed in every iteration, for every abstract configuration, one SMT solver call per element of B. Especially costly for "expensive" SMT theories or theory combinations, e.g., floats, bitvectors, and arrays.

Reminder (Abstract Strongest Post)

$$sp_B^\#(\{\psi\}, st) = \{\bigwedge\{\varphi \in B \mid sp(\{\psi\}, st) \subseteq \{\varphi\}\}\}$$

Optimizations:

- ▶ Use different sets B for different locations.
- ▶ Do not use general SMT formulas and an SMT solver, but certain classes of formulas ("domains", e.g., intervals, octagon, polyhedra) and specialized algorithms for construction of $sp_B^\#$.
- ▶ Do not construct ARG explicitly. Construct a tree that represents the breadth-first search for new counterexamples. Label nodes with formulas. Reuse tree in next iteration.
- ▶ Use the partial order on formulas induced by implication. If there are two nodes $(\ell, \{\varphi_1\})$ and $(\ell, \{\varphi_2\})$ such that $\varphi_2 \vDash \varphi_1$, we can ignore $(\ell, \{\varphi_2\})$. We say that $(\ell, \{\varphi_2\})$ is already "covered" by $(\ell, \{\varphi_1\})$.

Section 18

Trace Abstraction

Outline

Definition (Floyd-Hoare annotation)

A Floyd-Hoare annotation is a mapping that assigns each location ℓ_i a formula φ_i such that there is an edge

$$\boxed{\varphi_i}\;(\ell_i)\!\!-\!\!\boxed{\texttt{stmt}}\!\!\rightarrow\!(\ell_j)\;\boxed{\varphi_j}$$
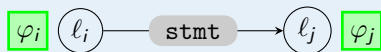
only if the Hoare triple

$$\{\boxed{\varphi_i}\}\;\texttt{stmt}\;\{\boxed{\varphi_j}\}$$
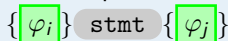
is valid.

Theorem

Given a program P, if there is a Floyd-Hoare annotation such that

- ▶ every initial location is labeled with true and
- ▶ every error location is labeled with false

then P is safe.

Example:
Floyd-Hoare annotation for P_{goanna}

Idea
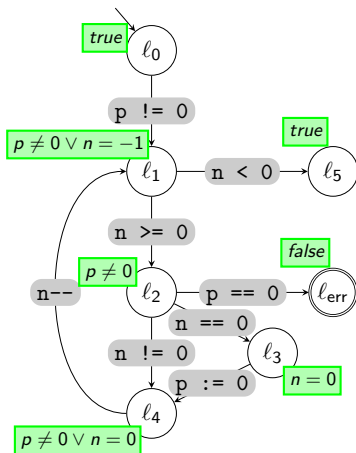
While analyzing a program P, consider automata whose alphabet Σ is the set of all statements that occur in P's control-flow graph.

Define a Floyd-Hoare annotation for such an automaton analogously to the definition of a Floyd-Hoare annotation for a control-flow graph.

Definition

We call an automaton $\mathcal{A} = (Q, \Sigma, \Delta, Q_{\text{init}}, F)$ a *Floyd-Hoare automaton* if there exists a Floyd-Hoare annotation $\beta : Q \to Fmrl(V)$ such that

- $\beta(q) = $ **true** for all $q \in Q_{\text{init}}$ and
- $\beta(q) = $ **false** for all $q \in F$.

Theorem

Every trace that is accepted by a Floyd-Hoare automaton is infeasible.

Let $\mathcal{A}_P$ be the automaton whose graph structure is similar to the control-flow graph.

Theorem

If there are Floyd-Hoare automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ such that the inclusion

$$\mathcal{L}(\mathcal{A}_P) \subseteq \mathcal{L}(\mathcal{A}_1) \cup \ldots \cup \mathcal{L}(\mathcal{A}_n)$$

holds then the program P is safe.

We've omitted the proofs of the previous two theorems in the lecture. However, they are not difficult:

▶ Every trace that is accepted by a Floyd-Hoare automaton is infeasible. Proof: Let τ be a trace that is accepted by a Floyd-Hoare automaton $\mathcal{A}$ with annotation β. Then there exists an accepting run $q_0 \ldots q_n$ for τ. By the definition of Floyd-Hoare automata, $\beta(q_0) \ldots \beta(q_n)$ is an infeasibility proof for τ.

▶ The second theorem follows directly: Every error trace in P is accepted by one of the Floyd-Hoare automata $\mathcal{A}_i$. Thus it is infeasible, and thus no error configuration can be reached.

New View on Programs

"A program defines a language over the alphabet of statements."

▶ Set of statements: alphabet of formal language
e.g., $\Sigma = \{$ `p != 0` , `n >= 0` , `n == 0` , `p := 0` , `n != 0` ,
`p == 0` , `n--` , `n < 0` , $\}$

▶ Control flow graph: automaton over the alphabet of statements
▶ Error location: accepting state of this automaton

▶ Error trace of program: word accepted by this automaton

Note that in this formalism, infeasible traces (i.e., traces for which there exists no execution of the program P) may still be accepted by the automaton $\mathcal{A}_P$. The finite automaton cannot distinguish between feasible and infeasible traces.

In fact, the verification task consists precisely of showing that *all* the traces accepted by $\mathcal{A}_P$ are infeasible.

```
1 assume p != 0;
2 while (n >= 0) {
3   assert p != 0;
4   if (n == 0) {
5     p := 0;
6   }
7   n := n - 1;
8 }
```

Source Code
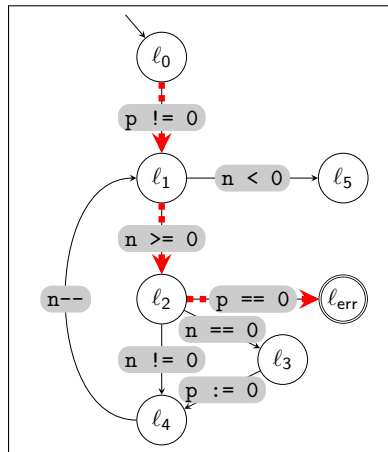
Control Flow Graph
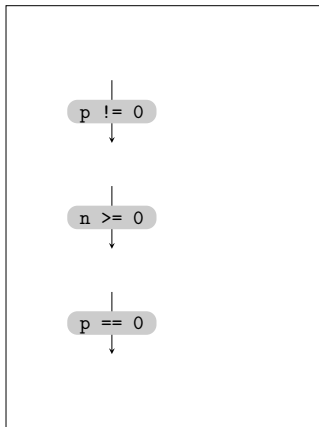
1. take trace π_1

1. take trace π_1
2. consider trace as automaton $\mathcal{A}_1$
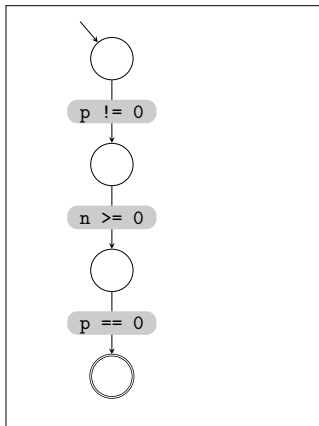
1. take trace π_1
2. consider trace as automaton $\mathcal{A}_1$
3. analyze correctness of $\mathcal{A}_1$, compute Floyd-Hoare annotation
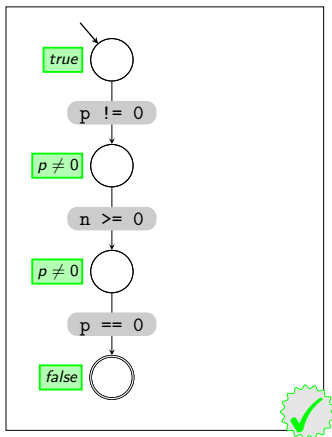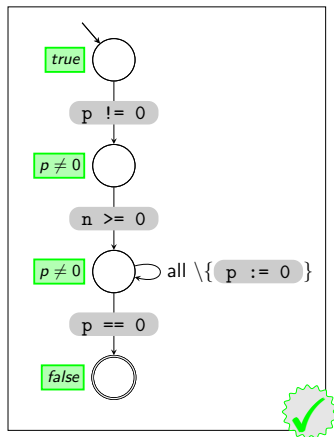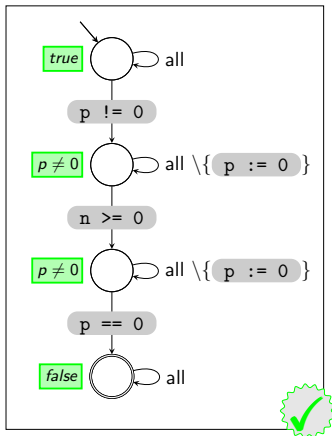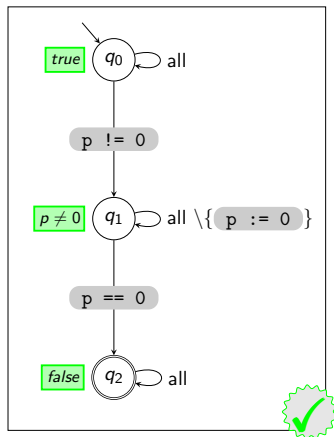
Trace Abstraction: Example

1. take trace π_1
2. consider trace as automaton $\mathcal{A}_1$
3. analyze correctness of $\mathcal{A}_1$, compute Floyd-Hoare annotation
4. generalize automaton $\mathcal{A}_1$
   - ▶ add transitions
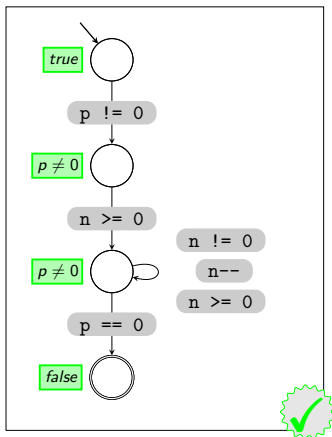
$\{\,\boxed{p \neq 0}\,\}$ $\boxed{\text{n--}}$ $\{\,\boxed{p \neq 0}\,\}$ is valid Hoare triple

$\{\,\boxed{p \neq 0}\,\}$ $\boxed{\text{n != 0}}$ $\{\,\boxed{p \neq 0}\,\}$ is valid Hoare triple

$\{\,\boxed{p \neq 0}\,\}$ $\boxed{\text{n >= 0}}$ $\{\,\boxed{p \neq 0}\,\}$ is valid Hoare triple

Trace Abstraction: Example

1. take trace π_1
2. consider trace as automaton $\mathcal{A}_1$
3. analyze correctness of $\mathcal{A}_1$, compute Floyd-Hoare annotation
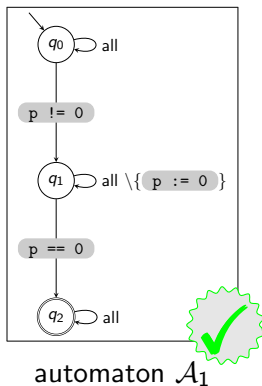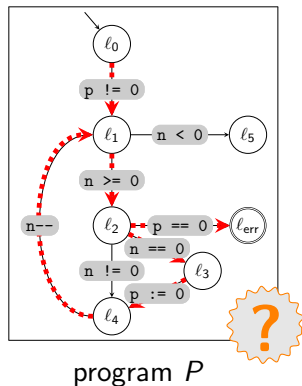4. generalize automaton $\mathcal{A}_1$
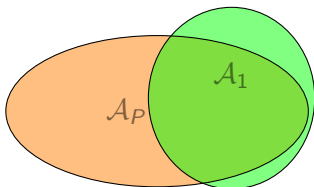   - add transitions

Trace Abstraction: Example

1. take trace π_1
2. consider trace as automaton $\mathcal{A}_1$
3. analyze correctness of $\mathcal{A}_1$, compute Floyd-Hoare annotation
4. generalize automaton $\mathcal{A}_1$
   - add transitions

Trace Abstraction: Example

1. take trace π_1
2. consider trace as automaton $\mathcal{A}_1$
3. analyze correctness of $\mathcal{A}_1$, compute Floyd-Hoare annotation
4. generalize automaton $\mathcal{A}_1$
   - ▶ add transitions
   - ▶ merge states with same annotation
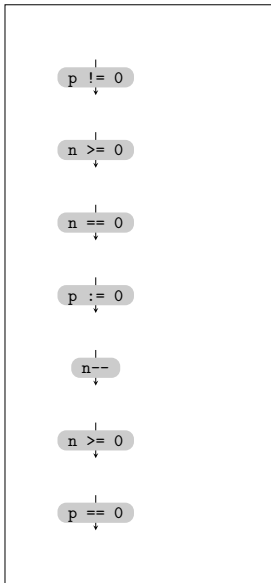
Trace Abstraction: Example

program P automaton $\mathcal{A}_1$

Consider only traces in set theoretic difference $\mathcal{L}(\mathcal{A}_P) \setminus \mathcal{L}(\mathcal{A}_1)$.

1. take trace π_2
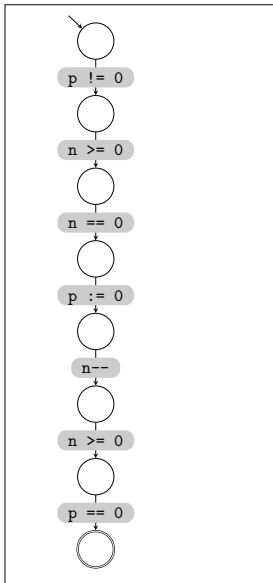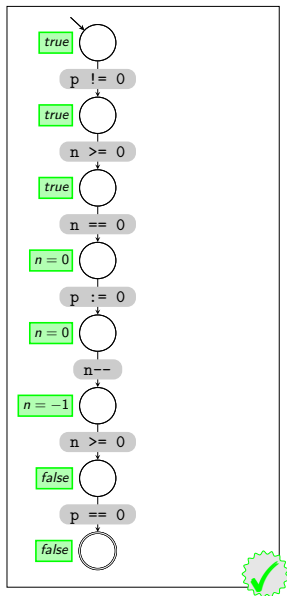
1. take trace π_2
2. consider trace as automaton $\mathcal{A}_2$

Trace Abstraction: Example

1. take trace π_2
2. consider trace as automaton $\mathcal{A}_2$
3. analyze correctness of $\mathcal{A}_2$, compute annotation
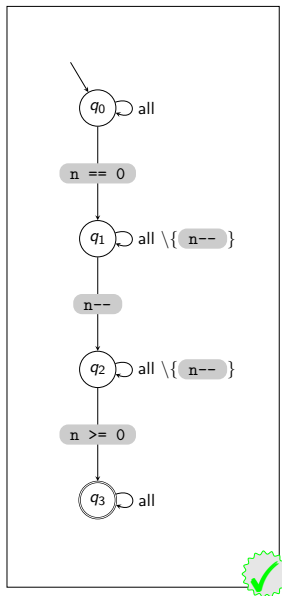
Trace Abstraction: Example

1. take trace π_2
2. consider trace as automaton $\mathcal{A}_2$
3. analyze correctness of $\mathcal{A}_2$, compute annotation
4. generalize automaton $\mathcal{A}_2$
   - add transitions
   - merge states with same annotation

Trace Abstraction: Example
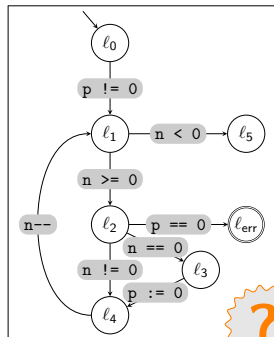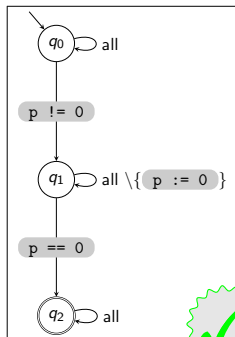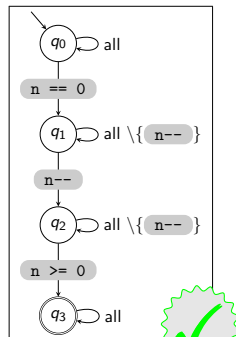
program P automaton $\mathcal{A}_1$ automaton $\mathcal{A}_2$

$$\mathcal{L}(\mathcal{A}_P) \quad \subseteq \quad \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$$

Trace Abstraction: Verification Algorithm
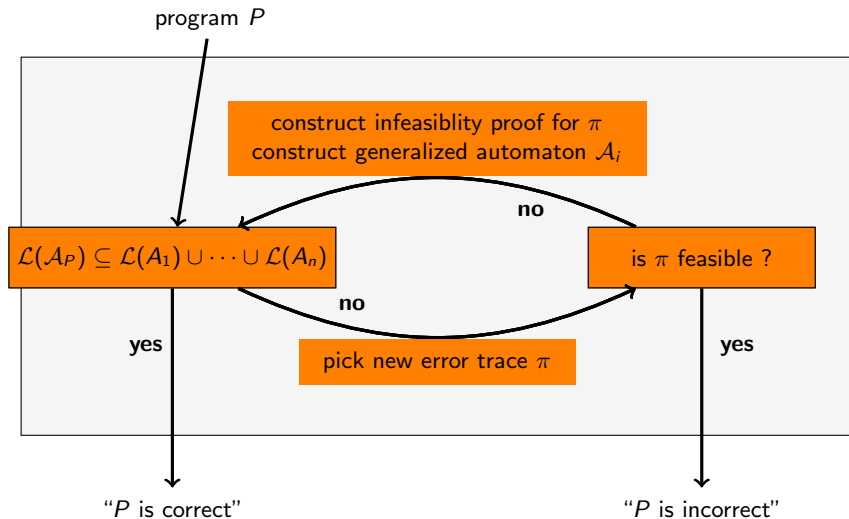
Program Verification

Summer Term 2020

Lecture 30: Termination

Matthias Heizmann

The following slides will be discussed on Monday 27th July

Notes for students of the 2020 course
- None.

Section 19

Termination Analysis

Outline

How should we define "termination" of a computer program?

We will next discuss four properties of programs.

1. Can the program reach the exit location?
   Is there some input for which the program reaches the exit location?
2. Can the program stop?
   Is there some input for which the program stops?
3. Does the program always reach the exit location?
   Does the program reach the exit location for all inputs?
4. Does the program always stop?
   Does the program stop for all inputs?

Results of the discussion:

- ▶ The properties are not stated precisely enough to give definite answers.
- ▶ On Exercise Sheet 22 we define four properties of the Boo language and use the terminology from the definition of Boo's semantics.
- ▶ The first two properties and the last two properties are fundamentally different: we can state the first two using techniques that we saw in this course. (E.g., if we want to check the first property we could put an assert false at the end of the program.)
- ▶ Differences between "stopping" and "reaching the error location". In C: program crashing. In Boogie or Boo: assume statements.
- ▶ If we consider Boo programs without assume statements there is no difference between Property 1 and Property 3 (resp. Property 2 and Property 4).
- ▶ Property 4 is the property that we want to call "termination". We will give the formal definition for Boo on the next slides.

Infinite Executions

Let $P = (V, \mu, st)$ be a program and $G = (Loc, \Delta, \ell_{\text{init}}, \ell_{\text{ex}})$ be a control-flow graph for P.

Definition (Infinite Execution)

We call a sequence of program configurations $(\ell_0, s_0), \ldots$ an *infinite execution* of P if there exists an infinite sequence of statements $st_1 \ldots$ such that for each $i \in \mathbb{N}$

- $(\ell_i, st_{i+1}, \ell_{i+1}) \in \Delta$ and
- $(s_i, s_{i+1}) \in [\![st_{i+1}]\!]$

Definition

We call P *terminating* if P does not have an infinite execution that starts in an initial configuration.

For the forthcoming definition of a *ranking function* we need the notion of a *well-founded* relation which was introduced in Exercise Sheet 21.

Definition

Let X be a set. We call a binary relation $R \subseteq X \times X$ *well-founded* if there is no infinite sequence $x_1, x_2, \ldots$ such that $(x_i, x_{i+1}) \in R$ for all $i \in \mathbb{N}$.

Our main means for proving termination will be *ranking functions*. We will first give a formal definition without further motivation and discuss its applications afterwards.

Informally, a ranking function for a loop is a function whose value is bounded from below but decreasing in every iteration. (Hence, we can conclude by reductio ad absurdum that only a finite number of loop iterations is possible).

On Wikipedia ranking functions are called Loop variants. In the research community on termination analysis, the term ranking function is however used more often.

Definition (Ranking Function)

Given a program $P = (V, \mu, st)$, a while loop `while(expr){st}` and a set W together with a well-founded relation $R \subseteq W \times W$, we call a function $f : S_{V,\mu} \to W$ a ranking function if for each pair of states $(s, s') \in [\![\texttt{assume expr; st}]\!]$ the relation $(f(s), f(s')) \in R$ holds.

Example:

```
1 while (x + y < 100) {
2     x := x + 1;
3 }
```

If we choose (W, R) as $(\mathbb{N}, >)$ then

$$f(s) = 100 - s(x) - s(y)$$

is a ranking function for this program.

Notation:

In order to improve legibility, people usually write

$$f(x, y) = 100 - x - y$$

instead of $f(s) = 100 - s(x) - s(y)$. In this course we will also use both notations.

In Exercise 3 ofExercise Sheet 22 the task was to find ranking functions for programs.

In fact, if we require that a ranking function is a total function we typically cannot use $\mathbb{N}$ as the range of the function.
We discuss the problem of a ranking function's range a couple of slides later.

Question: Is every loop that has a ranking function terminating?

Answer: No. There might be a nontermating loop inside a the loop that has a ranking function.

```
1 while (x < 100) {
2   x := x + 1;
3   while (y < 100) {
4     y := y -1;
5   }
6 }
```

Theorem

Let P be a program. If every while loop of P has a ranking function then P is terminating.

Proof. (Not given in the lecture)
(Informally) Assume there is an infinite execution $(\ell_0, s_0), (\ell_1, s_1), \ldots$ that starts in an initial configuration. Let $\ell_0', \ell_1', \ldots$ be the subsequence of all locations that are loop heads (definition of loop head was introduced on Exercise Sheet 23). Because of the structure of control flow graphs the sequence $\ell_0', \ell_1', \ldots$ is an infinite subsequence (a formal proof would need more details here). Because there a only finitely many different locations in a control-flow graph, at least one loop head occurs infinitely often. Let $\hat{\ell}$ be a loop head that occurs infinitely often in the sequence. Between each two visits of $\hat{\ell}$ the ranking function of the corresponding loop is decreasing which is a contradiction to well-foundedness.

In the remaining section on termination, we will discuss the following questions.

► Are ranking functions into $(\mathbb{N}, >)$ always convenient?
► How can we check if a function f is a ranking function?
► What if a ranking function is only decreasing for reachable states?
► How can we compute ranking functions?
► How can we build an algorithm for checking termination?
► How can we find nonterminating executions?
► What is more difficult? Safety or termination?

Question: Are ranking functions into $(\mathbb{N}, >)$ always convenient?

Problem: Negative return value of function after the last loop iteration.

```
1 while (x>=0) {
2   x := x - 1;
3 }
```

For $(\mathbb{N}, >)$ the function
$f(x) = x$ is not a ranking
function because after the
last loop iteration the
function returns -1.
$f(x) = x + 1$ is a ranking
function

```
1 while (x>=0) {
2   assume y >= 1;
3   x := x - y;
4 }
```

For $(\mathbb{N}, >)$ the function the
function $f(x, y) = x$ is not a
ranking function.
$f(x, y) = x + y$ is a ranking
function

```
1 while (x>=0) {
2   havoc y;
3   assume y >= 1;
4   x := x - y;
5 }
```

For $(\mathbb{N}, >)$ there is no
ranking function.

Solution: Do not use $(\mathbb{N}, >)$ but $(\mathbb{Z}, >_{\mathbb{N}})$ whose relation we define as
follows.

$$x >_{\mathbb{N}} y \quad \text{iff} \quad x > y \text{ and } x \in \mathbb{N}$$

This relation also solves another problem: If we require that the function f is defined for
all states $s \in S_{V,\mu}$ (i.e., f is a total function) then the functions that we saw so far were
in fact not well-defined.

Question: How can we check if a function f is a ranking function?

We present a solution for the schematic example on the right where we assume

1. we have one loop in the loop body,
2. the program's variables are x1,…,xn,
3. that we can express the function as an expression fexpr(x1,...,xn) over the program's variables, and
4. the range of f is $\mathbb{Z}$ and we consider the well-founded ordering $>_{\mathbb{N}}$.

```
1 while (expr1) {
2   // outer loop body part 1
3   while (expr2) {
4     // inner loop body
5   }
6   //outer loop body part 2
7 }
```

```
1  oldf := fexpr(x1,..,xn);
2  while (expr1) {
3    // outer loop body part 1
4    while (expr2) {
5      // inner loop body
6    }
7    // outer loop body part 2
8    assert fexpr(x1,...,xn)<oldf && oldf>=0;
9    oldf := fexpr(x1,...,xn);
10 }
```

We introduce a new variable oldf whose values are integers and transform the program above to the program on the left. The function f is a ranking function for the outer while loop iff the program on the left is safe.

On the preceding slide we made four assumptions.

► The generalization where we drop the first two assumptions is obviously staightforward.

► The third assumption is a vital restriction since not every function is computable (no proof given in lecture).

► Whenever there is a computable ranking function, there is also a (computable) *lexicographic ranking function* (see Exercise 3 on Exercise Sheet 22) where each lexicographic component is $\mathbb{Z}, >_{\mathbb{N}}$. If the function f is given in that form we can drop the fourth assumption, introduce an additional variable for each lexicographic component and modify lines 1,8, and 9 accordingly.

Question: What if a ranking function is only decreasing for reachable states? The discussion of this question was mainly done on Exercise Sheet 23.

Consider the following program which is obviously terminating.

```
1 assume(y >= 1);
2 while (x >= 0) {
3     x := x - y;
4 }
```
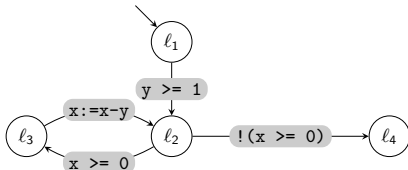

Q: Is $f(x, y) = x$ a ranking function for this loop?

A: No. The value of x is increasing if y is negative.

Q: Is there a ranking function that allows us to prove termination of this program?

A: Our first definition of a ranking function refers only to a loop. In order to prove termination of the program above, we also have to take the reachable states into account.

Definition (Loop Entry)

Given a while loop `while(expr){st}` and a control-flow graph $G = (Loc, \Delta, \ell_{\text{init}}, \ell_{\text{ex}})$ for this while loop, we call ℓ_{init} the *entry location* of the while loop.

If we would do this section with more formal rigor, we would redo the definition of a control flow graph and add to the tuple $(Loc, \Delta, \ell_{\text{init}}, \ell_{\text{ex}})$ a partial function that maps entry locations to the respective while loops.

Definition (Ranking Function)

Given a program $P = (V, \mu, st)$, a Floyd-Hoare annotation β for P, a while loop while(expr){st} whose loop head is the location ℓ, and a set W together with a well-founded relation $R \subseteq W \times W$, we call a function $f : S_{V,\mu} \to W$ a ranking function if for each pairs of states where $s \in \{\beta(\ell)\}$ and $(s, s') \in [\![\text{assume expr; st}]\!]$ the relation $(f(s), f(s')) \in R$ holds.

Theorem

Let P be a program and β be a Floyd-Hoare annotation for P. If every while loop of P has a ranking function for β then P is terminating.

Proof. (Not given in the lecture)
Analogously to the proof for the theorem on termination that does not yet have a name. Additionally we have to argue that the Floyd-Hoare annotation denotes a superset of the reachable states at each location.

Question: How can we compute ranking functions?

▶ For general programs: very difficult.

Although there is some research that follows this direction [**popl/CousotC12**, **esop/UrbanM14**, **tacas/UrbanGK16**]. See e.g., the FUNCTION tool of Caterina Urban.

▶ For infinite traces: sometimes doable

For several termination analyses TODO cite some it is sufficient to compute ranking functions for *ultimately periodic* traces. An ultimately periodic trace is an infinite trace in which some (finite) sequence of statements is repeated infinitely often. E.g., the trace where `x>0` `y>0` `y:=y-1` is repeted infinitely often is an ultimately periodic trace of the Program P_2 from Exercise 3 on Exercise Sheet 22. This ultimately periodic trace is then considered as a program that consists of a single while loop. For programs of this form several approaches are available [**tacas/ColonS01**, **vmcai/PodelskiR04**, **cav/BradleyMS05**, **tacas/CookKRW10**, **popl/Ben-AmramG13**, **atva/HeizmannHLP13**, **tacas/LeikeH14**, **cav/Ben-AmramG17**]. We outline the basic idea of these approaches on the next slides.

- ▶ "Synthesize" ranking functions analogously to the synthesis of invariants.
- ▶ Compute one transition formula τ_{loop} for all statements on the path from the loop entry to the loop entry.
- ▶ Do not set up constraints that encode the existence of a general ranking function. Set up constraints that encode the existence of a ranking function that has a certain form.
- ▶ If the variables of the program are x and y and $r_x, r_y, r_0 \in \mathbb{Z}$ then we call $r_x \cdot x + r_y \cdot y + r_0$ a *linear function*.

  Constraints for the existence of a linear ranking function.

  There exist r_x, r_y, r_0 such that
  - ▶ $\forall x, y, x', y'. \ \tau_{\text{loop}}(x, x') \rightarrow (r_x \cdot x + r_y \cdot y + r_0) - (r_x \cdot x' + r_y \cdot y' + r_0) \geq 1$
  - ▶ $\forall x, y, x', y'. \ \tau_{\text{loop}}(x, x') \rightarrow r_x \cdot x + r_y \cdot y + r_0 \geq 0$

  The first line states that the function $f(x, y) = r_x \cdot x + r_y \cdot y + r_0$ is decreasing in every interation.
  The second line states that the function $f(x, y) = r_x \cdot x + r_y \cdot y + r_0$ is bounded from below.

- ▶ Use Farkas' Lemma to simplify the constraints.
- ▶ Apply an SMT solver to the resulting contraints. E.g., in the schematic example above we use the satisfying assignments to r_x, r_y, r_0 to build our linear ranking function.

Usually, the part of an ultimately periodic trace that is infinitely often repeated is preceded by a sequence of statements. E.g., if you consider the (there is only one) infinite trace that starts at the initial location of the first program on Exercise Sheet 23 the infinite repetition of `x>=0` `x:=x-y` is preceded by `y>=1`.

We translate these infinte traces into a while loop that is preceded by a sequence of statements and call these programs *lasso programs* because their control flow graphs have the shape of a lasso.
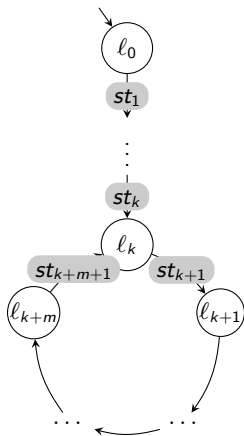
In lasso programs the loop sometimes does not have a ranking function but there is a ranking function for the combination of the loop and a given Floyd-Hoare annotation.

See discussion on ranking functions for reachable states.

For these program we synthesize a ranking function together with a loop invariant [cav/BradleyMS05, atva/HeizmannHLP13].

The constraints for the special case where we are searching for a ranking function of the form $\vec{r}^\mathsf{T} \cdot \vec{v} + r_0$ and an invariant of the form $\vec{s}^\mathsf{T} \cdot \vec{v} + s_0$ are given below.

In order to shorten the presentation, we use $\vec{r}^\mathsf{T}$ to denote the coefficients of the ranking function and $\vec{s}^\mathsf{T}$ to denote the coefficients of the invariant. We use τ_stem to denote a transition formula of the sequential composition of all statements before the loop.

There exist $\vec{r}, r_0, \vec{s}, s_0$ such that

▶ $\forall \vec{v}\vec{v}'.\ \tau_\mathsf{stem}(\vec{v}, \vec{v}') \rightarrow \vec{s}^\mathsf{T} \cdot \vec{v}' + s_0 \geq 0$

▶ $\forall \vec{v}\vec{v}'.\ \vec{s}^\mathsf{T} \cdot \vec{v} + s_0 \geq 0 \wedge \tau_\mathsf{loop}(\vec{v}, \vec{v}') \rightarrow \vec{s}^\mathsf{T} \cdot \vec{v}' + s_0 \geq 0$

▶ $\forall \vec{v}\vec{v}'.\ \vec{s}^\mathsf{T} \cdot \vec{v} + s_0 \geq 0 \wedge \tau_\mathsf{loop}(\vec{v}, \vec{v}') \rightarrow \vec{r}^\mathsf{T} \cdot \vec{v} - \vec{r}^\mathsf{T} \cdot \vec{v}' \geq 1$

▶ $\forall \vec{v}\vec{v}'.\ \vec{s}^\mathsf{T} \cdot \vec{v} + s_0 \geq 0 \wedge \tau_\mathsf{loop}(\vec{v}, \vec{v}') \rightarrow \vec{r}^\mathsf{T} \cdot \vec{v} + r_0 \geq 0$

Synthesis of ranking functions available in Ultimate: LASSORANKER

▶ Supports synthesis of ranking functions together with invariants [atva/HeizmannHLP13] and various kinds of ranking functions [tacas/LeikeH14]. E.g., linear ranking functions, nested ranking functions, multiphase ranking functions, lexicographic ranking functions or piecewise ranking funtions. Implements an approach based on trace abstraction [cav/HeizmannHP14, pldi/ChenHLLTTZ18] that uses Büchi automata.

▶ Frontend currently supports the languages Boogie and C.

▶ Available via web interface.

Question: How can we build an algorithm for checking termination?

Basic idea of the approach of the Terminator tool [**pldi/CookPR06**].

Iteratively collect ranking functions until termination of all loops is shown.

1. Start with the empty set of ranking functions.
2. Pick an ultimately periodic trace for which termination is not yet shown (if termination is not yet proven).
3. Compute a ranking function for this trace and add it to our collection (if the trace does not have in infinite execution).
4. Check if the collection of ranking functions is sufficient to prove termination and continue with the second step.

A strength of Terminator's approach is that it does not need one (possibly complicated) ranking function for each loop but that it can use a combination of several ranking functions to prove termination of a single loop. The theoretical basis for this are *disjunctively well-founded transition invariants* [**lics/PodelskiR04**]. In this lecture we will only demonstrate the basic idea on one example.

Let us prove that the program whose code is depicted on the right is terminating.

The `if (*)` means that the computer which runs the program can nondeterministically pick one of the two branches. This is a syntax of Boogie that we did not introduce in Boostan.

We will need three iterations and two ranking functions.

```
1  while (x>0 && y>0) {
2    if (*) {
3      x := x-1;
4      havoc y;
5    } else {
6      y := y-1;
7    }
8  }
```

Initially, our set of ranking functions is emtpy. We construct the first program depicted on the next slide and pass it to a tool that checks safety (resp. that every assert statement is valid). The safety checker tells us that the assert is reachable via the if-branch and we conclude that there is an ultimately periodic infinite trace that repeats the if-branch. We pass this trace to a tool that infers ranking functions and obtain $f1(x, y) = x$.

In the second iteration we construct the second program of the next slide in order to check whether $f1$ is sufficient to prove termination. This second safety check looks similar to the check that we discussed a few slides ago but the (re-)initialization of the `oldf1` variable is done nondeterministically. The safety checker tells us that the assert can be violated by an execution that takes the else branch. We conclude that there is an ultimately periodic trace that repeats the else-branch whose termination cannot be shown by the ranking function $f1$. We pass this trace to a tool that infers ranking functions and obtain $f2(x, y) = y$.

In the third iteration we construct the third program of the next slide in order to check whether the combination of `f1` and `f2` is sufficient to prove termination. The safety checker tells us that the assert statement is valid and we conclude termination of the original program.

We note that the expression of the assert statement is a disjunction; we do not require that both ranking functions are decreasing, we only require that at least one of ranking function is decreasing. For concluding termination the nondeterministic assignments to `oldf1` and `oldf1` are vital. The safety proof does not only show that in each iteration the function `f1` or the function `f2` is decreasing, the safety proof shows that between every two (not necessarily consecutive) visits of the loop head the function `f1` or the function `f2` is decreasing.

```
1 while (x>0 && y>0) {
2   if (*) {
3     x := x-1;
4     havoc y;
5   } else {
6     y := y-1;
7   }
8   assert false;
9 }
```

```
1 if (*) {
2   oldf1 := f1(x,y);
3 }
4 while (x>0 && y>0) {
5   if (*) {
6     x := x-1;
7     havoc y;
8   } else {
9     y := y-1;
10   }
11   assert oldf1 > f1(x,y) &&
       oldf1 >= 0;
12   if (*) {
13     oldf1 := f1(x,y);
14   }
15 }
```

```
1 if (*) {
2   oldf1 := f1(x,y);
3   oldf2 := f2(x,y);
4 }
5 while (x>0 && y>0) {
6   if (*) {
7     x := x-1;
8     havoc y;
9   } else {
10     y := y-1;
11   }
12   assert (oldf1 > f1(x,y) &&
       oldf1 >= 0)
13     || (oldf2 > f2(x,y) &&
       oldf2 >= 0);
14   if (*) {
15     oldf1 := f1(x,y);
16     oldf2 := f2(x,y);
17   }
18 }
```

$$f1(x,y) = x$$
$$f2(x,y) = y$$

Development of Termination has be discontinued, successor is the T2 tool.
There are several other tools and approaches for termination analysis. ᴛᴏᴅᴏ cite some

Termination analysis available in Ultimate: Büchi Automizer

▶ Implements an approach based on trace abstraction [**cav/HeizmannHP14**, **pldi/ChenHLLTTZ18**] that uses Büchi automata.

▶ Frontend currently supports the languages Boogie and C.

▶ Won the termination category at the Competition on Software Verification (SV-COMP) several times.

▶ Available via web interface. (Because of a bug, one has to use the command line version to see the ranking functions)

Question: How can we prove nontermination?

In the lecture, we discussed the poster on the *Geometric Nontermination Arguments* approach [**tacas/LeikeH18**] very briefly.

Not relevant for exam. Mainly shown to attract students that like linear algebra.

Question: What is more difficult? Safety or termination?

For both kinds of properties it is undecidable whether the property holds for a given program.

There exists a small program for which yet no one could prove or disprove termination.

Given a starting value a_0, let us consider the infinite series of integers $a_0, a_1, \ldots$ such that

$$a_{i+1} = \begin{cases} a_i/2 & \text{if } a_i \text{ is even} \\ 3 \cdot a_i + 1 & \text{if } a_i \text{ is odd} \end{cases}$$

Collatz conjecture: For any starting value, the sequence will finally reach 1.

The Collatz conjecture is correct if and only if the program on the right is terminating.
Although many people tried, yet no one could prove or disprove the conjecture.
To the best of my knowledge, no small safety problem with the same level of difficulty is known.

```
1  while ( x != 1) {
2    if (x%2==0) {
3      x := x/2;
4    } else {
5      x := 3x+1
6    }
7  }
```

References I