Automata-based Software Verification with the Ultimate Program Analysis Framework

Matthias Heizmann

joint work with Dominik Klumpp, Tanja Schindler, Andreas Podelski and the developers of the Ultimate Program Analysis Framework

VTSA Summer School 2025

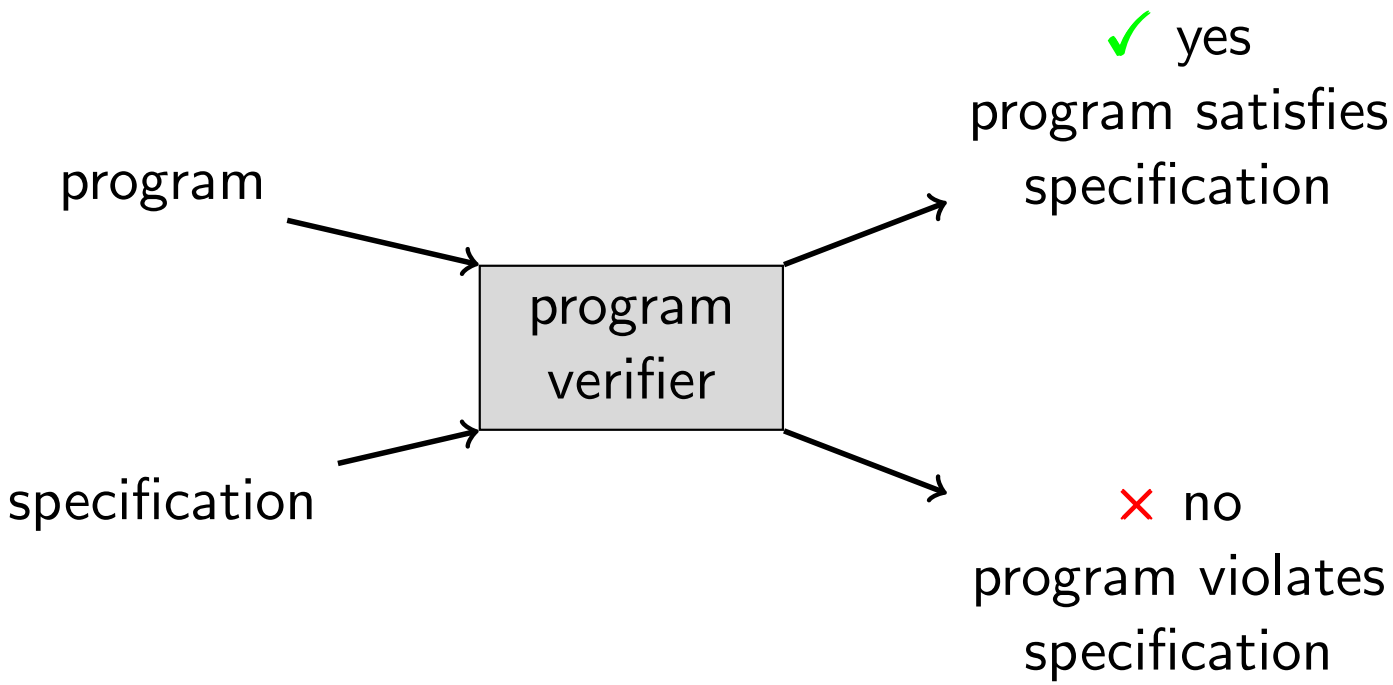
Section 1

Introduction

Outline

Outline of the Section on Introduction

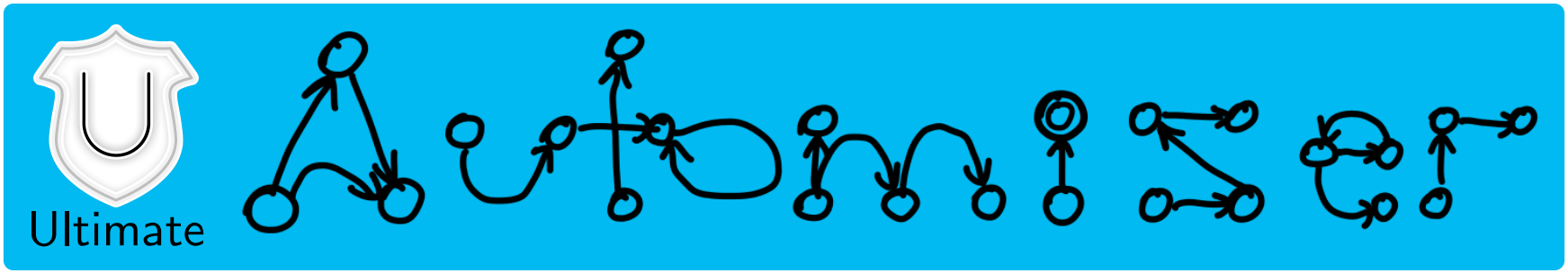
Program Verifier



Typical specifications:

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

Ultimate Automizer



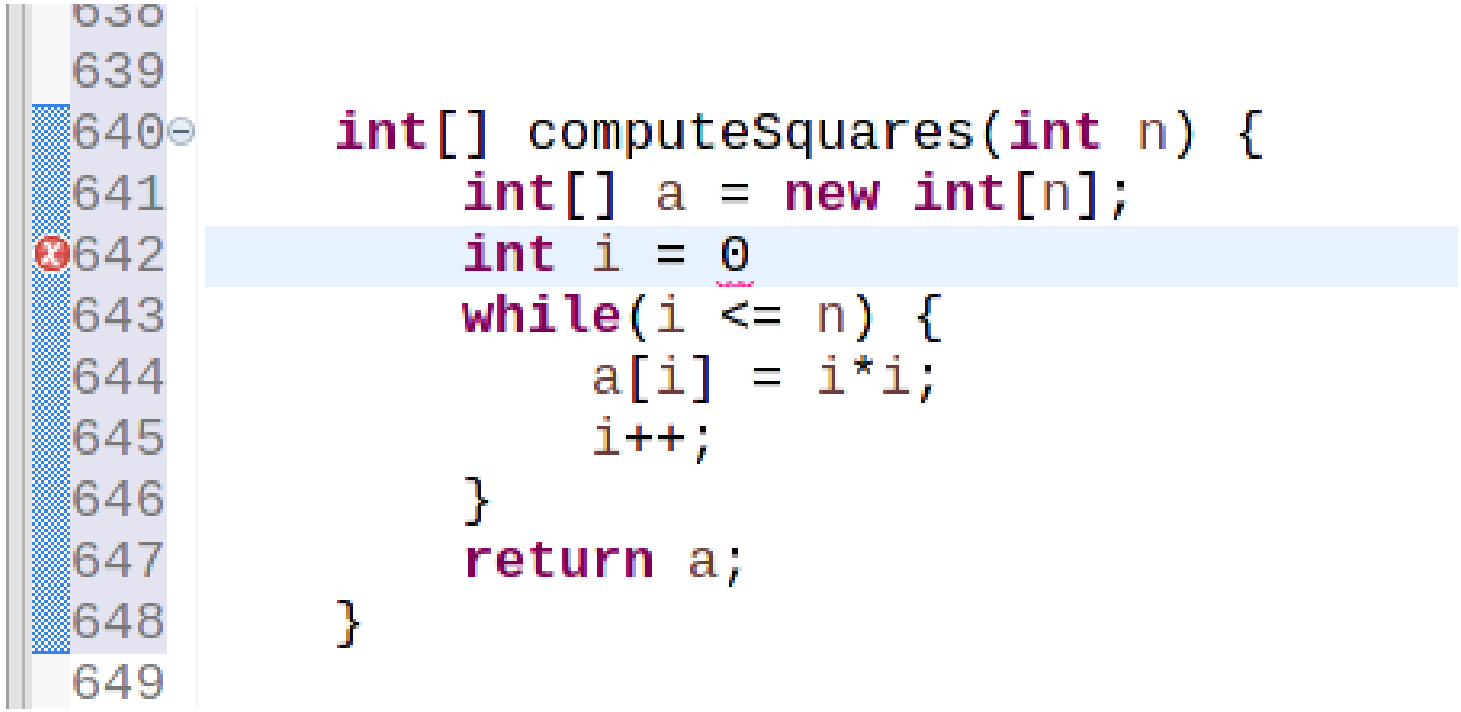
`https://www.ultimate-pa.org/?ui=tool&tool=automizer`

- According to the international competition on software verification SV-COMP [1] one of the best verification tools.
- Research tool mainly developed by the software engineering 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/`



Computers are very good in detecting synax errors. (Here, Eclipse complains about a missing semicolon). Is would be great if tools could also underline bugs that we have seen before.

- Program Verification
- Motivation
- Challenges
- Course outline

Ultimate Automizer

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

```
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.

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

Ultimate Automizer confirms that the program satisfies its specification.

Outline of the Section on Introduction

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;
```

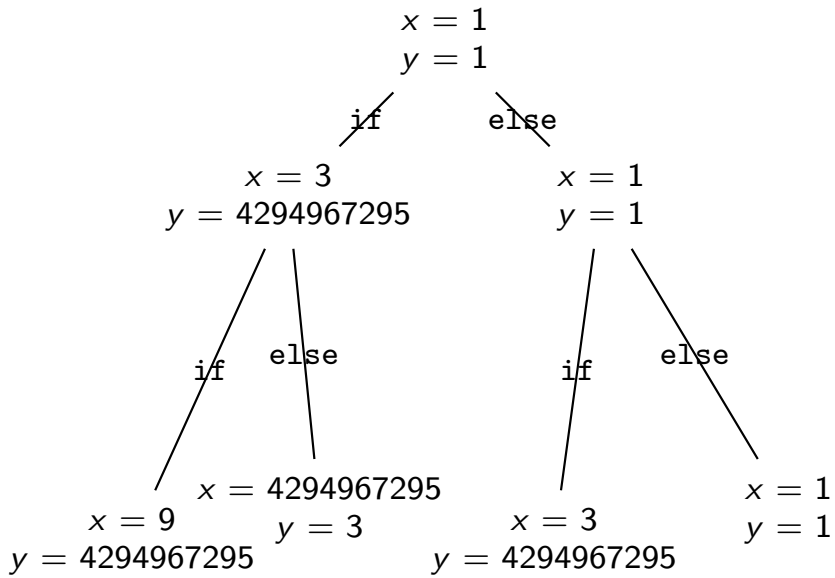
	x	y	makes sense because
C/C++	-1	-2	$(-1) \cdot 5 + (-2) = 7$
Python	-2	3	$(-2) \cdot 5 + 3 = 7$
Javascript	-1.4	-2	$(-1.4) \cdot 5 = 7$

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.

Section 2

SMT-LIB

Outline

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`

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

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.

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] [7]
  Often used in this course because there is a Z3 web interface
- SMTInterpol[8] [3]
  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)
  ```

Tasks

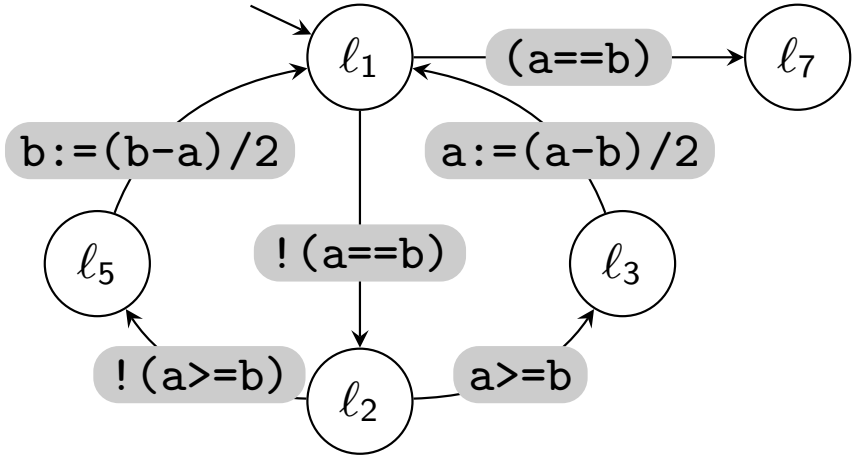
Encode the following statements as SMT-LIB formulas.

1. Integer y is an odd number
2. There exists a positive integer x such that y is $x + x$

Use the webinterface of Z3 to check syntax and to test the semantics.

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

Demo

Use the webinterface of Z3 to encode whether certain paths satisfy the precondition-postcondition pair.

Section 3

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

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. 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.

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)

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://github.com/nadia-polikarpova/boogaloo/wiki/User-Manual`

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] [6] 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.

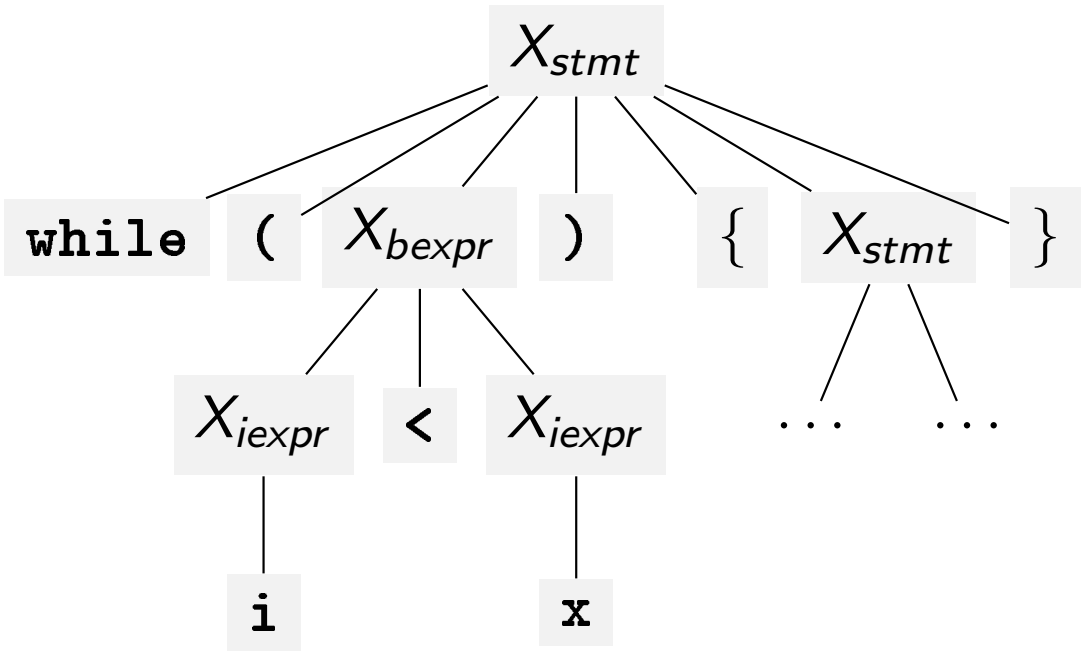
[11] `https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/`

Outline of the Section on Boogie and Boostan

Motivation: How can we Formalize Programs?

Sequence of characters vs. tree

```
while (i<x) {
  x := x + i;
  i := i + 1;
}
```



- The syntax of a programming language is typically defined via a context-free grammar or via a closely related concept.
- We will define the syntax of Boostan via a context-free grammar and use a notation that is typically used in lectures on theoretical computer science.
- In order to make you (again) familiar with context-free grammars and in order to fix a notation for this course we give a formal definition on the next slides.

Definition

A *context-free grammar* is a 4-tupel $\mathcal{G} = (\Sigma, N, P, S)$ 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*

and $\Sigma \cap N \neq \emptyset$.

Example

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

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

Definition

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

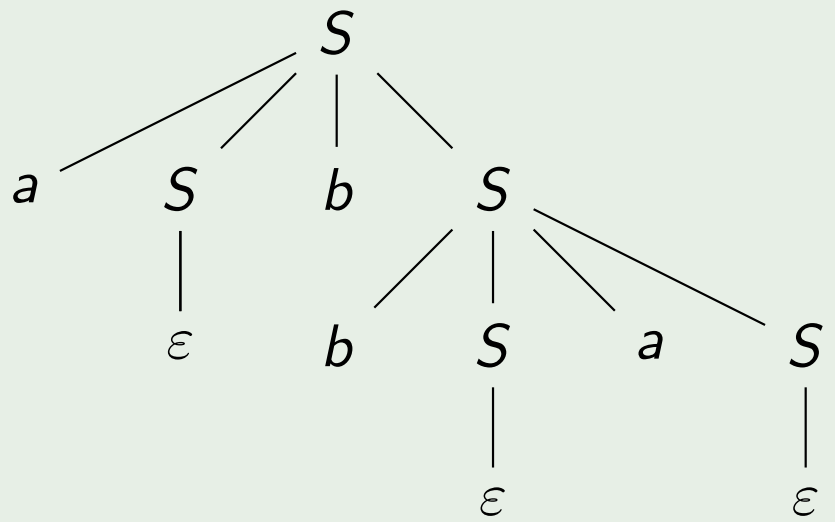
- a node $v \in V$ may only have children $v_1, \ldots, v_n \in V$ if $\lambda(v) \to \lambda(v_1) \ldots \lambda(v_n)$ is a rule in P and
- all leafs are labelled by terminal symbols or by ε.

Example

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

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

Example



Definition

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

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

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 $\mathrm{dw}(v) = w$.
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. TODO add link

Grammar for Numbers

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

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

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

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

$$\begin{aligned}
\Sigma_{\text{num}} &= \{\mathtt{0}, \ldots, \mathtt{9}\} \\
N_{\text{num}} &= \{X_{num}, X_{num'}\} \\
P_{\text{num}} &= \{X_{num} \rightarrow \mathtt{0}X_{num'} \mid \ldots \mid \mathtt{9}X_{num'} \\
&\qquad X_{num'} \rightarrow \mathtt{0}X_{num'} \mid \ldots \mid \mathtt{9}X_{num'} \mid \varepsilon\} \\
S_{\text{num}} &= X_{num}
\end{aligned}$$

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.

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

$$\Sigma_{\text{var}} = \Sigma_{\text{num}} \cup \{\mathtt{a}, \ldots, \mathtt{z}, \mathtt{A}, \ldots \mathtt{Z}\}$$

$$N_{\text{var}} = \{X_{var}, X_{var'}\}$$

$$\begin{aligned} P_{\text{var}} = \{ X_{var} &\rightarrow \mathtt{a}X_{var'} | \ldots | \mathtt{z}X_{var'} | \mathtt{A}X_{var'} | \ldots | \mathtt{Z}X_{var'} \\ X_{var'} &\rightarrow \mathtt{a}X_{var'} | \ldots | \mathtt{z}X_{var'} | \mathtt{A}X_{var'} | \ldots | \mathtt{Z}X_{var'} | \mathtt{0}X_{var'} | \ldots | \mathtt{9}X_{var'} | \varepsilon \} \end{aligned}$$

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

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`.

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

$$\Sigma_{\mathsf{I}} = \{\texttt{-}, \texttt{+}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{(}, \texttt{)}\} \cup \Sigma_{\mathsf{var}} \cup \Sigma_{\mathsf{num}}$$

$$N_{\mathsf{I}} = \{X_{iexpr}\} \cup N_{\mathsf{var}} \cup N_{\mathsf{num}}$$

$$\begin{aligned} P_{\mathsf{I}} = \{ X_{iexpr} &\to \texttt{(} X_{iexpr} \texttt{)} \\ X_{iexpr} &\to \texttt{-} X_{iexpr} \\ X_{iexpr} &\to X_{iexpr} \texttt{+} X_{iexpr} \mid X_{iexpr} \texttt{-} X_{iexpr} \mid X_{iexpr} \texttt{*} X_{iexpr} \\ X_{iexpr} &\to X_{iexpr} \texttt{/} X_{iexpr} \mid X_{iexpr} \texttt{\%} X_{iexpr} \\ X_{iexpr} &\to X_{var} \\ X_{iexpr} &\to X_{num} \} \cup P_{\mathsf{var}} \cup P_{\mathsf{num}} \end{aligned}$$

$$S_{\mathsf{I}} = X_{iexpr}$$

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. $\rightarrow$).

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

$$\Sigma_{\mathsf{B}} = \{\mathbf{!}, \mathbf{\&\&}, \mathbf{||}, \mathbf{==>}, \mathbf{<}, \mathbf{>}, \mathbf{<=}, \mathbf{>=}, \mathbf{true}, \mathbf{false}\} \cup \Sigma_{\mathsf{I}}$$

$$N_{\mathsf{B}} = \{X_{bexpr}\} \cup N_{\mathsf{I}}$$

$$\begin{aligned}
P_{\mathsf{B}} = \{ X_{bexpr} &\rightarrow (X_{bexpr}) \\
X_{bexpr} &\rightarrow \mathbf{!}\, X_{bexpr} \\
X_{bexpr} &\rightarrow X_{bexpr} \mathbf{\&\&} X_{bexpr} \mid X_{bexpr} \mathbf{||} X_{bexpr} \mid X_{bexpr} \mathbf{==>} X_{bexpr} \\
X_{bexpr} &\rightarrow X_{iexpr} \mathbf{<} X_{iexpr} \mid X_{iexpr} \mathbf{>} X_{iexpr} \mid X_{iexpr} \mathbf{<=} X_{iexpr} \mid X_{iexpr} \mathbf{>=} X_{iexpr} \\
X_{bexpr} &\rightarrow X_{bexpr} \mathbf{==} X_{bexpr} \mid X_{iexpr} \mathbf{==} X_{iexpr} \\
X_{bexpr} &\rightarrow X_{var} \\
X_{bexpr} &\rightarrow \mathbf{true} \mid \mathbf{false}\} \cup P_{\mathsf{I}}
\end{aligned}$$

$$S_{\mathsf{B}} = X_{bexpr}$$

Grammar for Boostan

$\mathcal{G}_{\text{Boo}} = (\Sigma_{\text{Boo}}, N_{\text{Boo}}, P_{\text{Boo}}, S_{\text{Boo}})$ with $N_{\text{Boo}} = N_{\text{B}} \cup \{X_{stmt}, X_{lhs}, X_{expr}\}$, $S_{\text{Boo}} = X_{stmt}$ and

$$
\begin{aligned}
P_{\text{Boo}} = \{ X_{stmt} &\rightarrow X_{lhs}\texttt{:=}X_{expr}\texttt{;} \\
X_{stmt} &\rightarrow X_{stmt}X_{stmt} \\
X_{stmt} &\rightarrow \texttt{if (}X_{bexpr}\texttt{)\{}X_{stmt}\texttt{\} else \{}X_{stmt}\texttt{\}} \\
X_{stmt} &\rightarrow \texttt{while (}X_{bexpr}\texttt{)\{}X_{stmt}\texttt{\}} \\
X_{lhs} &\rightarrow X_{var} \\
X_{expr} &\rightarrow X_{bexpr} | X_{iexpr} \}
\end{aligned}
$$

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?

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 $S_{V_{\text{ab}},\mu_{\text{ab}}}$

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 $\{\{a \mapsto 23, b \mapsto 42\}\} \subseteq S_{V_{\text{ab}},\mu_{\text{ab}}}$
- 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: $[\![$`2 * (x % 16)+42`$]\!]$ is $2 \cdot (x \; mod \; 16) + 42$.

Convention

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

Semantics of the Assignment Statement

Given a program $P = (V, \mu, \mathcal{T})$ we define the semantics of an assignment statement $[\![\texttt{x := expr}]\!]$ 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., $\text{prime}(\{a \mapsto 23, b \mapsto 42\})$ is $\{a' \mapsto 23, b' \mapsto 42\}$.

Example

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

Semantics of the Assignment Statement

Example (continued)

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

E.g., the pair of states (s_1, s_2) where $s_1 = \{a \mapsto 5, b \mapsto 1\}$ and $s_2 = \{a \mapsto 6, b \mapsto 1\}$ is an element of this relation, because for $\rho = s_1 \cup \text{prime}(s_2) = \{a \mapsto 5, b \mapsto 1, a' \mapsto 6, b' \mapsto 1$ the evalution $[\![a' = a + 1 \wedge b' = b]\!]$ 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.

$R_1 \circ R_2 := \{(x, z) \mid$ there exists $y \in X$ s.t. $(x, y) \in R_1$ 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 $[\![st1]\!] \circ [\![st2]\!]$

Reminder: (Convention)

We defined the formula/term $[\![\texttt{expr}]\!]$ 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

$$[\![\texttt{if(}expr\texttt{)\{}st1\texttt{\}else\{}st2\texttt{\}}]\!] \quad \text{as} \quad \begin{array}{c}(\{expr\} \times S_{V,\mu}) \cap [\![st1]\!] \\ \cup\, (\{!expr\} \times S_{V,\mu}) \cap [\![st2]\!]\end{array}$$

Example

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

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

$$\{(s,s') \mid \begin{array}{l}(s(b) \geq 0 \text{ and } s'(b) = s(b)-1) \text{ or } (s(b) < 0 \text{ and } s'(b) = s(b)+1) \\ \text{and} \quad s'(a) = s(a)\end{array} \}$$

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 $[\![\texttt{while (}expr\texttt{)\{}st\texttt{\}}]\!]$ 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 $\underbrace{(\{x >= 0\} \times S_{V,\mu}) \cap [\![x := x - 1; y := y + 1]\!]}_{\{(s,s') \mid s(x) \geq 0 \wedge s'(x) = s(x) - 1 \wedge s'(y) = s(y) + 1\}}$

$$\begin{aligned}
R^0 &= id \\
R^1 &= \{(s,s') \mid s(x) \geq 0 \text{ and } s'(x) = s(x) - 1 \text{ and } s'(y) = s(y) + 1\} \\
R^2 &= \{(s,s') \mid s(x) \geq 1 \text{ and } s'(x) = s(x) - 2 \text{ and } s'(y) = s(y) + 2\} \\
&\vdots \\
R^* &= \{(s,s') \mid s = s' \text{ or } \big(s(x) > s'(x) \geq -1 \text{ and } s'(y) - s(y) = s(x) - s'(x)\big)\} \\
R^* \cap (S_{V\mu} \times \{\texttt{!x>=0}\}) &= \{(s,s') \mid \big(s = s' \text{ and } s'(x) < 0\big) \\
&\qquad \text{or } \big(s(x) > s'(x) = -1 \text{ and } s'(y) - s(y) = s(x) + 1\big)\}
\end{aligned}$$

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.

$$
\begin{aligned}
[\![\texttt{b := b-1}]\!] &= \{\,(s,s') \mid [\![b' = b-1 \wedge a' = a]\!]_{\mathcal{M},\rho} = \textbf{true} \text{ and } \rho = s \cup \mathrm{prime}(s')\,\} \\
&= \{\,(s,s') \mid s'(b) = s(b) - 1 \text{ and } s'(a) = s(a)\,\} \\
[\![\texttt{b := b+1}]\!] &= \{\,(s,s') \mid s'(b) = s(b) + 1 \text{ and } s'(a) = s(a)\,\} \\
[\![\texttt{a := a+1}]\!] &= \{\,(s',s'') \mid s''(a) = s'(a) + 1 \text{ and } s''(b) = s'(b)\,\} \\
[\![\mathit{if/else}]\!] &= \{\texttt{b >= 0}\} \times S_{V,\mu} \cap [\![\texttt{b := b-1}]\!] \cup \{\texttt{!b >= 0}\} \times S_{V,\mu} \cap [\![\texttt{b := b+1}]\!] \\
&= \{\,(s,s') \mid s'(a) = s(a) \text{ and } \big((s(b) \geq 0 \text{ and } s'(b) = s(b) - 1) \\
&\qquad\qquad \text{or } (s(b) < 0 \text{ and } s'(b) = s(b) + 1)\big)\,\} \\
[\![\mathit{loop\ body}]\!] &= \{\,(s,s'') \mid \text{ex. } s' \text{ s.t. } (s,s') \in [\![\mathit{if/else}]\!], (s',s'') \in [\![\texttt{a := a+1}]\!]\,\} \\
&= \{\,(s,s'') \mid s''(a) = s(a) + 1 \text{ and } \big((s(b) \geq 0 \text{ and } s''(b) = s(b) - 1) \\
&\qquad\qquad \text{or } (s(b) < 0 \text{ and } s''(b) = s(b) + 1)\big)\,\} \\
[\![P_{ab}]\!] &= \big((\{\texttt{!(b == 0)}\} \times S_{V,\mu}) \cap [\![\mathit{loop\ body}]\!]\big)^* \cap (S_{V,\mu} \times \{\texttt{!!(b == 0)}\}) \\
&= \{\,(s,s') \mid s(b) \neq 0 \text{ and } s'(a) = s(a) + 1 \text{ and } |s'(b)| = |s(b)| - 1\,\}^* \\
&\qquad \cap (S_{V,\mu} \times \{\texttt{!!(b == 0)}\}) \\
&= \{\,(s,s') \mid s'(a) + |s'(b)| = s(a) + |s(b)| \text{ and } |s'(b)| \leq |s(b)|\,\} \\
&\qquad \cap (S_{V,\mu} \times \{\texttt{!!(b == 0)}\}) \\
&= \{\,(s,s') \mid s'(a) = s(a) + |s(b)| \text{ and } s'(b) = 0\,\}
\end{aligned}
$$

Outline of the Section on Boogie and Boostan

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 $(\{\varphi_{\text{pre}}\}, \{\varphi_{\text{post}}\})$ that we call precondition-postcondition pair, we want to define the following formally. Whenever we run st in some state where 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* $(\{\varphi_{\text{pre}}\}, \{\varphi_{\text{post}}\})$ if the inclusion $post(\{\varphi_{\text{pre}}\}, [\![st]\!]) \subseteq \{\varphi_{\text{post}}\}$ 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 $post(\{a \geq 0\}, [\![st]\!]) \stackrel{?}{\subseteq} \{a \geq 0\}$!

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

Section 4

Hoare Proof System

Outline

Outline of the Section on Hoare Proof System

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

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

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

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

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

While rule

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

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 $\{\varphi_1\}st_1\{\psi_1\} \quad \dots \quad \{\varphi_n\}st_n\{\psi_n\}$, then

$$\frac{\{\varphi_1\}st_1\{\psi_1\} \quad \dots \quad \{\varphi_n\}st_n\{\psi_n\}}{\{\varphi_{n+1}\}st_{n+1}\{\psi_{n+1}\}}$$

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. Not part of this presentation.

Conditional Rule

$$(\mathit{condi})\frac{\{\varphi \wedge \mathit{expr}\}\ \mathit{st}_1\ \{\psi\} \quad \{\varphi \wedge \neg \mathit{expr}\}\ \mathit{st}_2\ \{\psi\}}{\{\varphi\}\ \texttt{if(}\mathit{expr}\texttt{)\{}\mathit{st1}\texttt{\}else\{}\mathit{st2}\texttt{\}}\ \{\psi\}}$$

Example

$$\frac{\left\{\begin{smallmatrix} x{=}y \\ \wedge y{\geq}0 \end{smallmatrix}\right\}\texttt{y:=y-1;}\left\{\begin{smallmatrix} (x{\geq}0{\rightarrow}y{=}x{-}1) \\ \wedge(x{<}0{\rightarrow}y{=}x{+}1) \end{smallmatrix}\right\} \qquad \left\{\begin{smallmatrix} x{=}y \\ \wedge\neg(y{\geq}0) \end{smallmatrix}\right\}\texttt{y:=y+1;}\left\{\begin{smallmatrix} (x{\geq}0{\rightarrow}y{=}x{-}1) \\ \wedge(x{<}0{\rightarrow}y{=}x{+}1) \end{smallmatrix}\right\}}{\{y = x\}\texttt{if(y>=0)\{y:=y-1;\}else\{y:=y+1;\}}\left\{\begin{smallmatrix} (x{\geq}0{\rightarrow}y{=}x{-}1) \\ \wedge(x{<}0{\rightarrow}y{=}x{+}1) \end{smallmatrix}\right\}}$$

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

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

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

Example

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

$$\dfrac{\dfrac{\left\{\begin{smallmatrix} z=x+y \\ \wedge x\geq 0 \end{smallmatrix}\right\} \wedge x > 0\} \texttt{x:=x-1;y:=y+1;} \left\{\begin{smallmatrix} z=x+y \\ \wedge x\geq 0 \end{smallmatrix}\right\}}{\left\{\begin{smallmatrix} z=x+y \\ \wedge x\geq 0 \end{smallmatrix}\right\} \texttt{while(x>0)}\{\texttt{x:=x-1;y:=y+1;}\} \left\{\begin{smallmatrix} z=x+y \\ \wedge x\geq 0 \end{smallmatrix}\right\} \wedge \neg(x > 0)\}}(\mathit{while})}{\left\{\begin{smallmatrix} z=x+y \\ \wedge x\geq 0 \end{smallmatrix}\right\} \texttt{while(x>0)}\{\texttt{x:=x-1;y:=y+1;}\} \{z = y\}}(\mathit{weakpos})$$

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 $b \leq 0 \rightarrow a - b \geq 53$.

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

where (*) is the following subtree

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

Section 5

Ultimate Referee

Outline

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

In this section, we will learn to

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

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 challenge directly.

Guide for Finding a Derivation in the Hoare Proof System

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

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

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

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

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

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

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 computer do everything that can be done algorithmically. (See tool in next subsection.)

Outline of the Section on Ultimate Referee

Ultimate Referee

Ultimate Referee is a tool for checking loop invariants.

- Takes as input:
  - program where each loop is annoted 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 [2, 1] implemented in Ultimate. The witness validator is rather lenient and tries to complete proofs that are incomplete.

Section 6

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

The next slides motivates the need for an SMT theory of arrays.

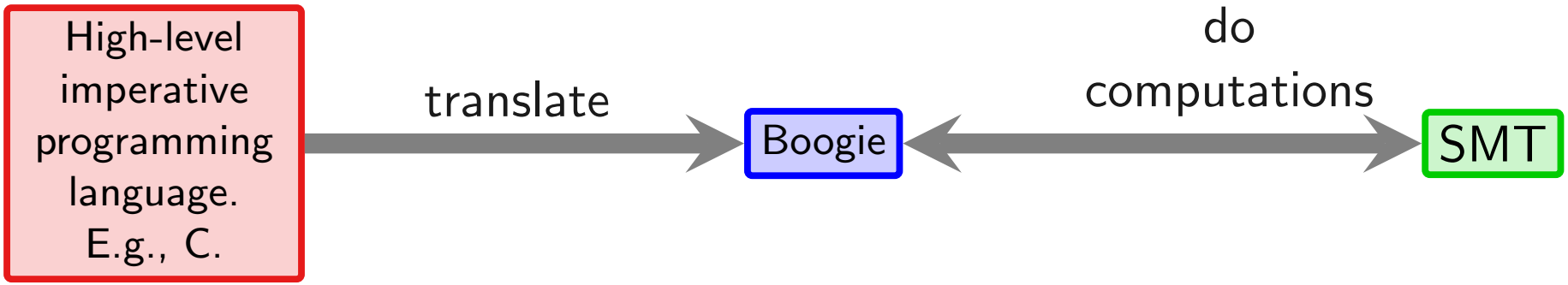
The diagram contrasts the approach of this lecture with the approach of the Ultimate Automizer verification tool (which we discuss later in this course).

- 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.

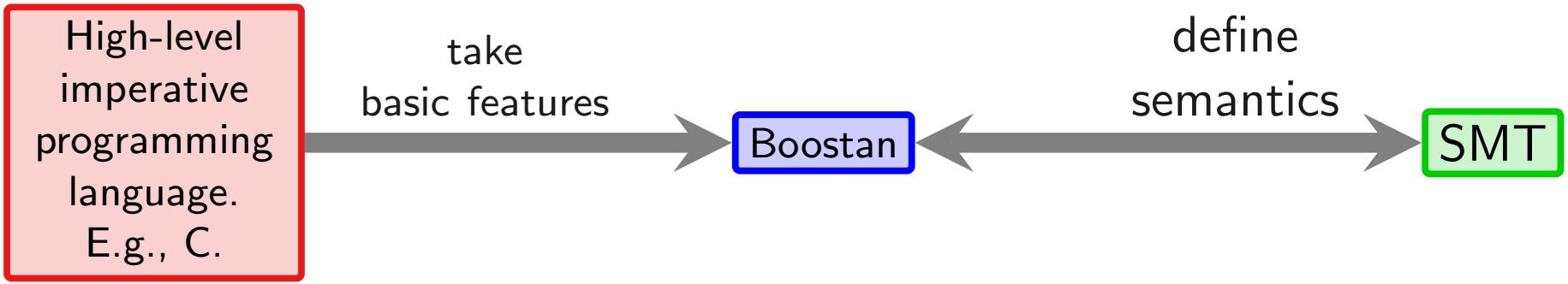
Arrays are an basic feature of high-level programming languages, hence we want to have SMT support for arrays.

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

- binary function
- 1st argument: a map
- 2nd argument: element of map's domain
- returns: value of map at that position
- e.g. $\mathit{select}(f_{foo}, 5) = 0$
- e.g. $\mathit{select}(f_{bar}, 5) = 23$

store

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

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:

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

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 12 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.

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

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] [6].

See 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

Excursus: Modeling Memory via Arrays

Feature of many high-level languages: Pointers / References

Simplest way to model in Boogie: global array `mem : [int]int`

C	Boogie	SMT
pointer dereference `*ptr`	array access `mem[ptr]`	$select(mem, ptr)$
pointer assignment `*ptr = expr;`	array assignment `mem[ptr] := expr;`	$mem' = store(mem, ptr, expr)$
reference assignment ptr2 = ptr;	assignment `ptr2 := ptr;`	$ptr2' = ptr$

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

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

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

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

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

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

Semantics of the Array Assignment Statement

Reminder (Assignment Statement)

$[\![$ `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 an array assignment statement $[\![$ `a[i] := expr;` $]\!]$ as the following binary relation over program states.

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

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 $\{\varphi[a \mapsto \mathit{store}(a, i, \mathit{expr})]\}$ `a[i]:=expr;` $\{\varphi\}$ is valid.

Reminder

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

Proof. Analogously to the proof for the assignment statement.

Section 7

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.

Outline of the Section on Boogie and Boostan – Part 2

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.

Modelling Input in Programs

C program:

```
1 unsigned char x = 0;
2 while (x < '1' || x > '9') {
3     println("Please input a number from 1 to 9.")
4     x = readchar();
5 }
6 // work with input x
```

Boogie program:

```
1 var x : int;
2 x := 0;
3 while (x < 49 || x > 57) {
4     // println("Please input a number from 1 to 9.")
5     havoc x;
6     assume 0 <= x && x <= 255;
7 }
8 // work with input x
```

Section 9.2 of the Boogie specification[17] explains the assume statement.

[17] K. Rustan M. Leino. "This is Boogie 2". 2008.

Nondeterminism in Boogie

Modelling input in Boogie: In each iteration, an arbitrary value is assigned to the variable `x`.

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

Question: Does this program satisfy the precondition-postcondition pair $(\{y = 1\}, \{y \leq 81\})$?

$\Rightarrow$ Let's ask Ultimate Automizer!

Section 9.4 of the Boogie specification[18] explains the havoc statement.

[18] K. Rustan M. Leino. "This is Boogie 2". 2008.

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 Havoc Statement

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

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

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

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

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

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.

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

Assumptions

How can we restrict input to certain values?

Not a feature of programming languages.

Modelling Input in Programs

C program:

```
1 unsigned char x = 0;
2 while (x < '1' || x > '9') {
3     println("Please input a number from 1 to 9.")
4     x = readchar();
5 }
6 // work with input x
```

Boogie program:

```
1 var x : int;
2 x := 0;
3 while (x < 49 || x > 57) {
4     // println("Please input a number from 1 to 9.")
5     havoc x;
6     assume 0 <= x && x <= 255;
7 }
8 // work with input x
```

Section 9.2 of the Boogie specification[17] explains the assume statement.

[17] K. Rustan M. Leino. "This is Boogie 2". 2008.

Assumptions 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

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

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

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

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

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

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

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

Section 8

Control-flow graphs

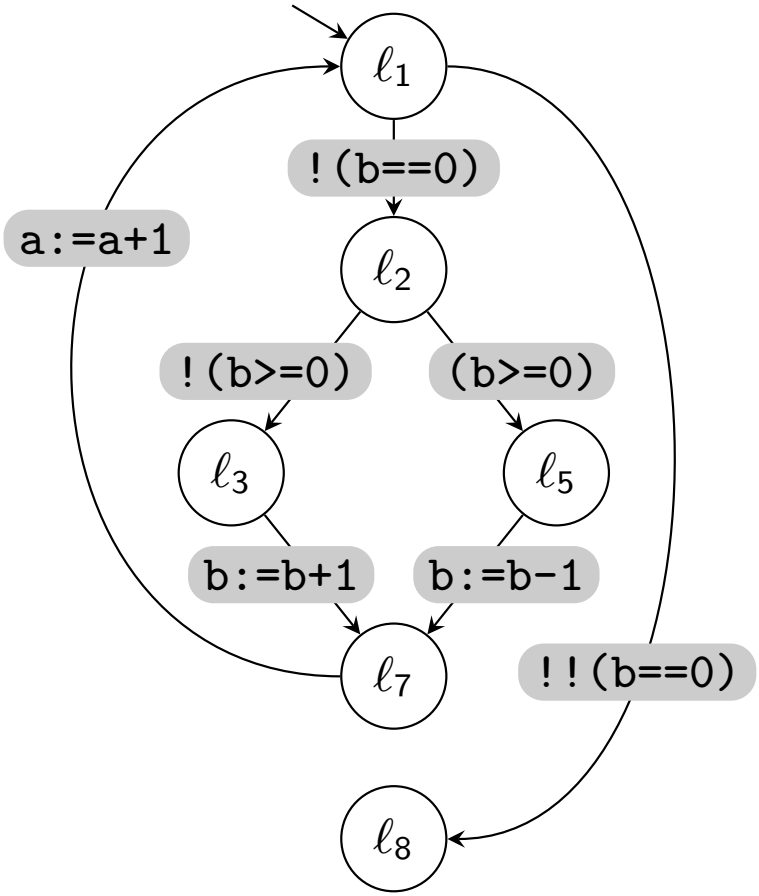
Outline

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}



Section 9

Predicate Transformers

Outline

Outline of the Section on Predicate Transformers

Example

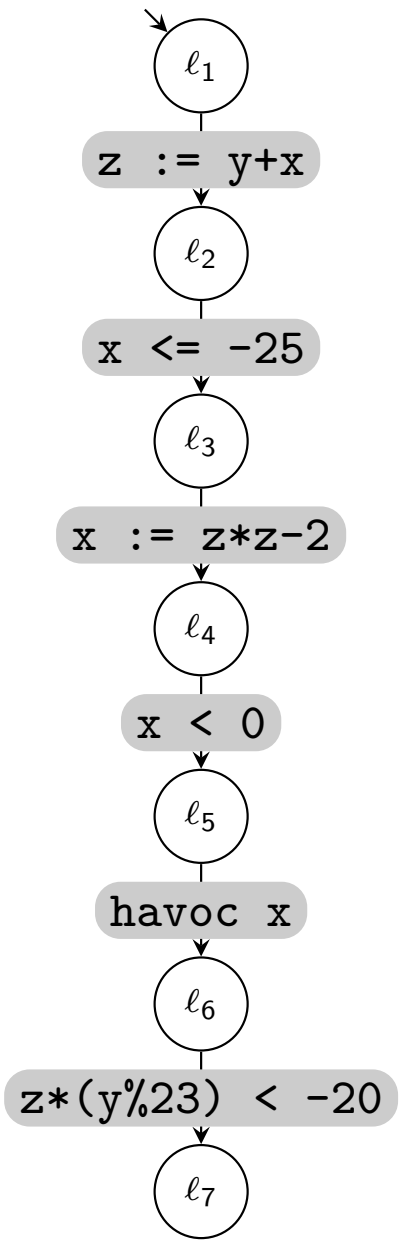
Consider the program P_{xor} with $V = \{x, y, z\}$, $\mu(x) = \mu(y) = \mu(z) = \mathbb{Z}$.

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);
```

$$\varphi_{\text{pre}}: \quad y \geq 1$$
$$\varphi_{\text{post}}: \quad y \geq 45$$

Example



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

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

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

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

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

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

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

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

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

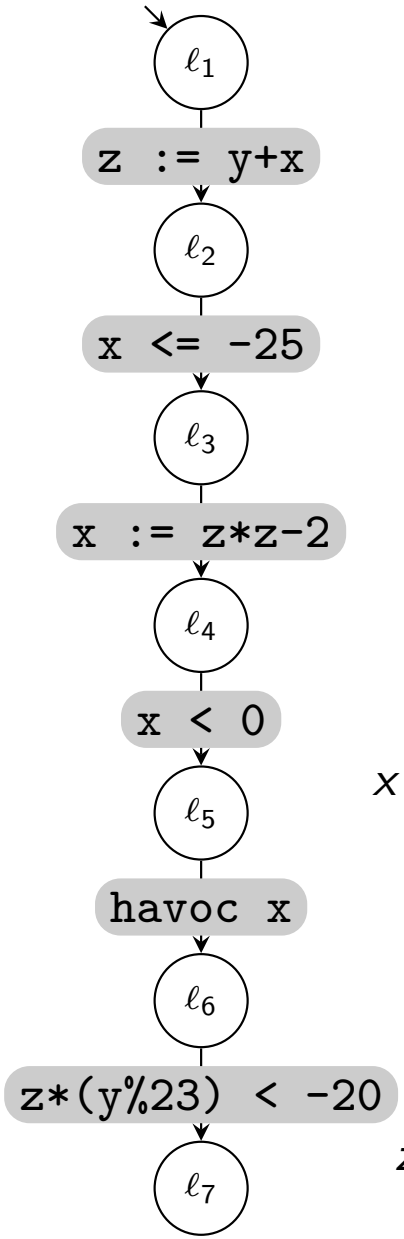
$\{x \mapsto -1, y \mapsto 26, z \mapsto 1\}$

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

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

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



true

$z = y + x$

$z = y + x$
$\wedge x \leq -25$

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

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

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

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

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

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

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

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.

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

Theorem (Strongest Post of the Assignment Statement)

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

Theorem (Strongest Post of the Havoc Statement)

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

Theorem (Strongest Post of the Assume Statement)

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

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)

If the variable x does not occur in the term t then 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 $(x - 1)\%2 = 0$

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*.

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

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

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

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*.

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

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

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

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

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

Quantifier Elimination

Theorem (Destructive Equality Resolution 2)

If the variable x does not occur in the term t then 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.

Tool Demo

Use Ultimate Eliminator for quantifier elimination.

Section 10

Correctness Specification via Assert Statement

Outline

Grammar for Boostan with Assert Statement

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

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

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

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

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

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

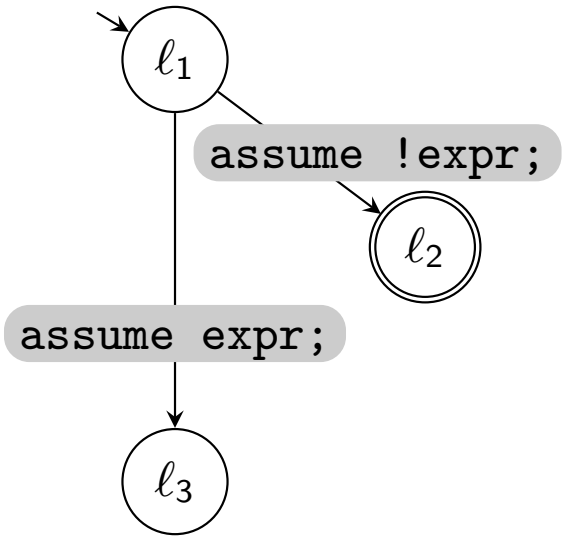
`assert expr;`

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

- $Loc = \{\ell_1, \ell_2, \ell_3\}$,
- $\Delta = \{(\ell_{\text{init}}, \texttt{assume !expr;}, \ell_{\text{err}}), (\ell_{\text{init}}, \texttt{assume expr;}, \ell_{\text{ex}})\}$,
- $\ell_{\text{init}} = \ell_1$,
- $\ell_{\text{ex}} = \ell_3$,
- $Loc_{\text{err}} = \{\ell_2\}$,
- $\ell_{\text{init}} \neq \ell_{\text{err}}$, $\ell_{\text{init}} \neq \ell_{\text{ex}}$, and $\ell_{\text{ex}} \neq \ell_{\text{err}}$.

is a control-flow graph for st.

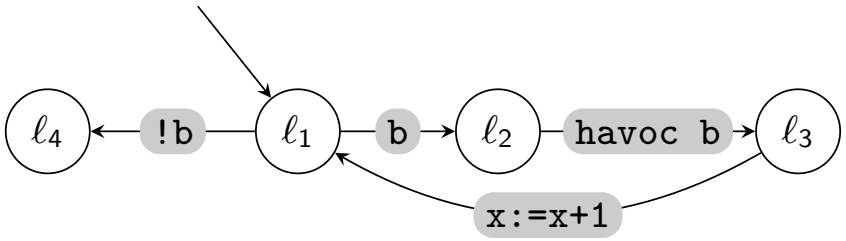
Example:



Section 11

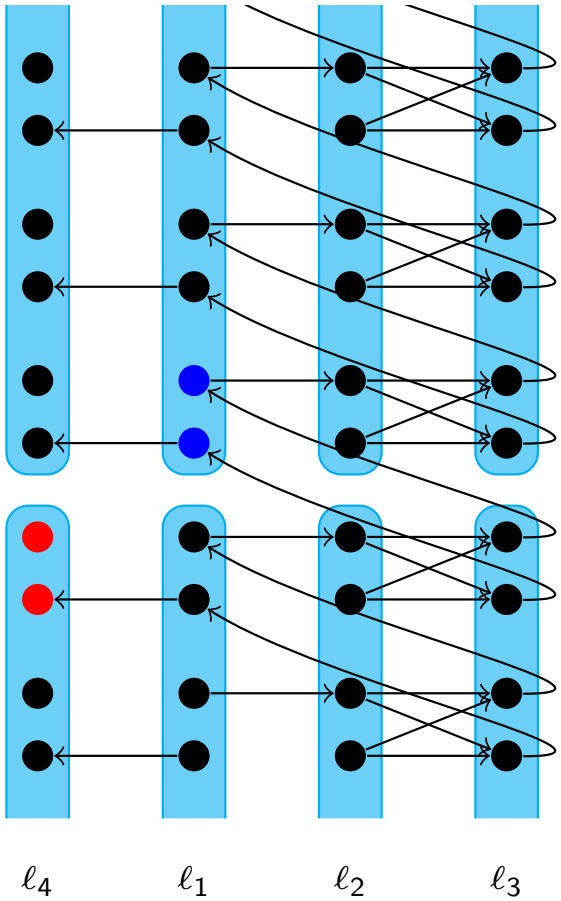
Abstractions – Part 1

Outline



$\varphi_{\text{pre}} : x = 0$

$\varphi_{\text{post}} : x \neq -1$



$\{b \mapsto \textbf{true}, x \mapsto 2\}$

$\{b \mapsto \textbf{false}, x \mapsto 2\}$

$\{b \mapsto \textbf{true}, x \mapsto 1\}$

$\{b \mapsto \textbf{false}, x \mapsto 1\}$

$\{b \mapsto \textbf{true}, x \mapsto 0\}$

$\{b \mapsto \textbf{false}, x \mapsto 0\}$

$\{b \mapsto \textbf{true}, x \mapsto -1\}$

$\{b \mapsto \textbf{false}, x \mapsto -1\}$

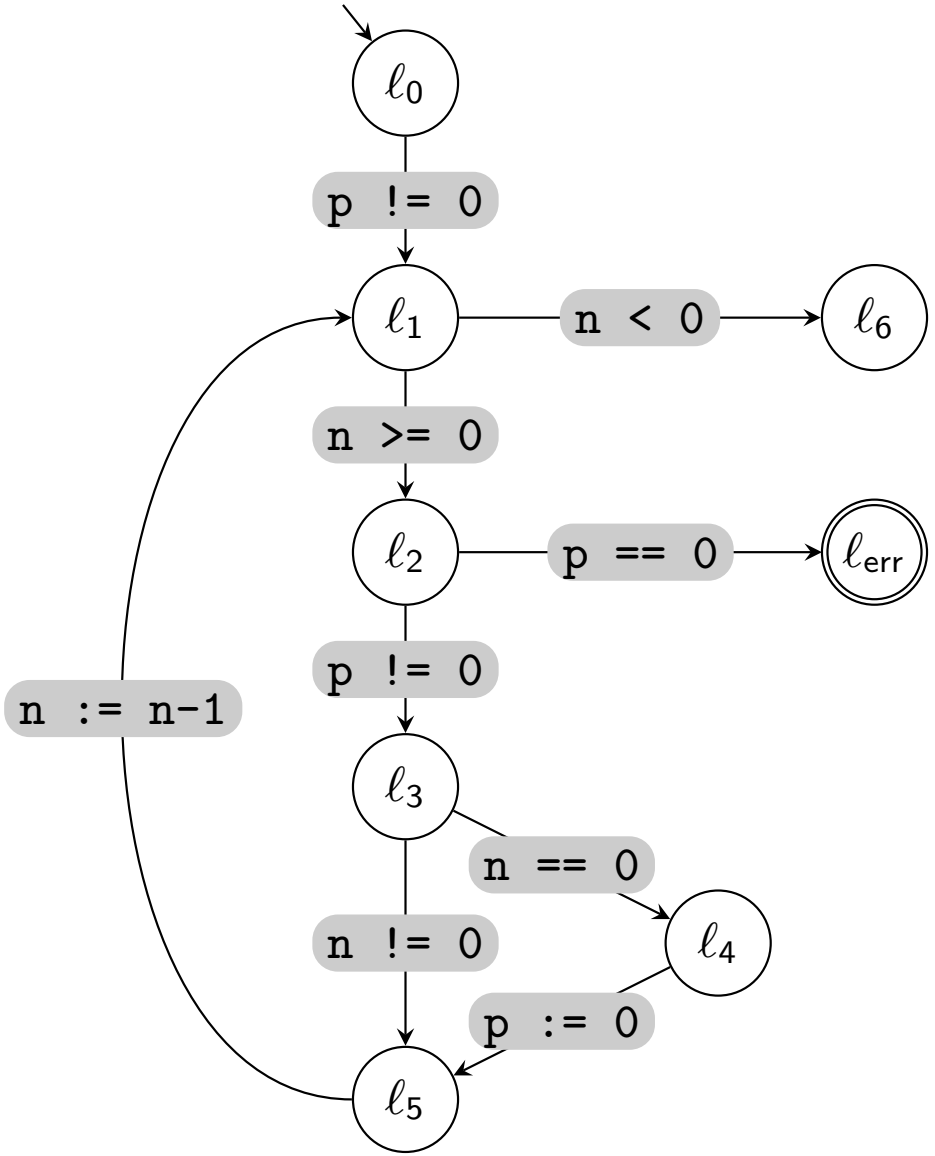
$\{b \mapsto \textbf{true}, x \mapsto -2\}$

$\{b \mapsto \textbf{false}, x \mapsto -2\}$

Example

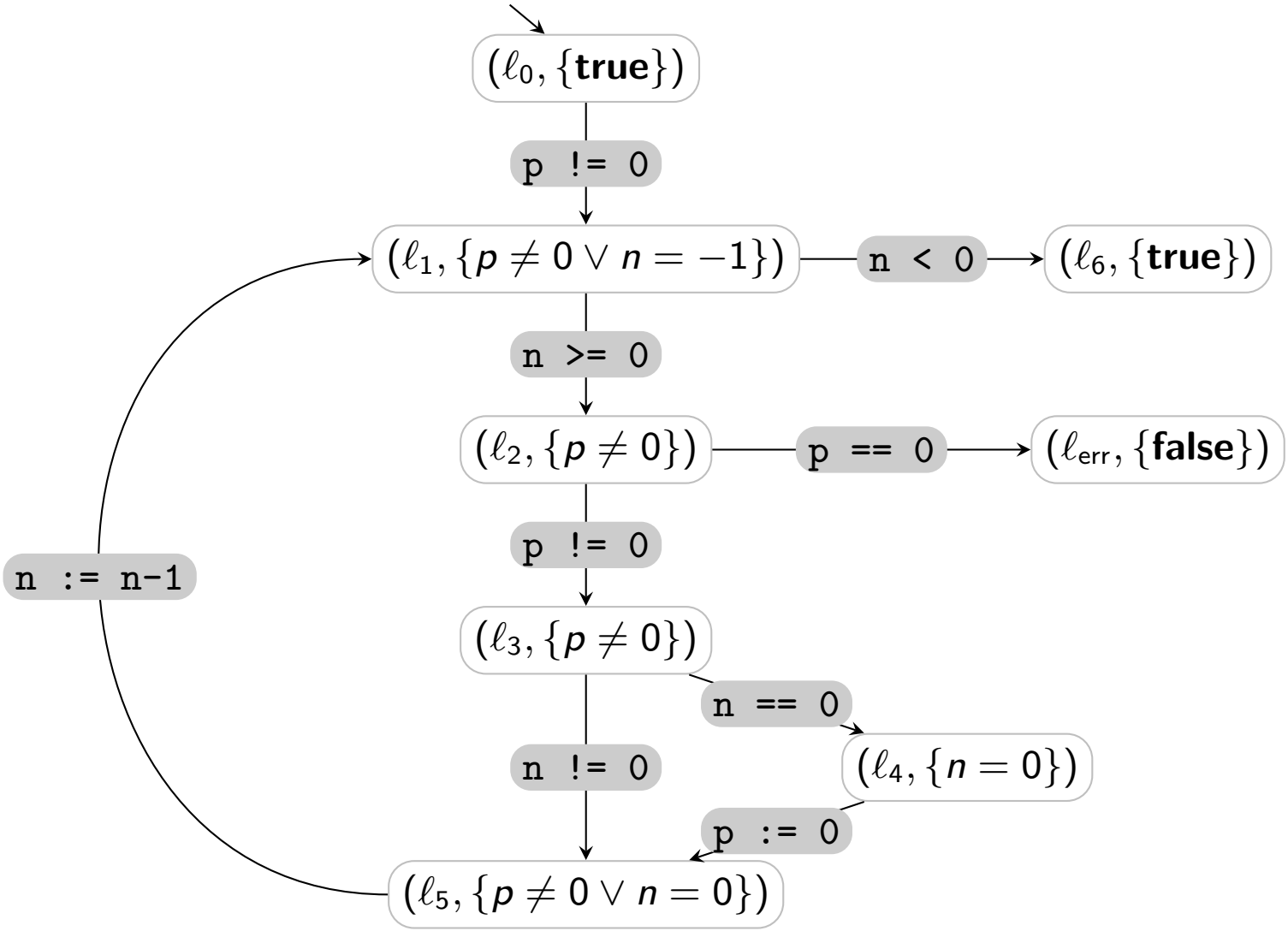
Program code and control-flow graph of the program P_{goanna}

```
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.



Definition (Abstract Strongest Post)

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

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

If the set B is empty, the abstract strongest post operator is always $\{\bigwedge\{\}\}$, i.e., the set of states for which the formula $\bigwedge\{\}$ holds. This formula is called the "empty conjunction" and one usually uses the convention that the empty conjunction is **true**. We follow this convention.[19]

[19] The convention to define the empty conjunction as **true** is a rather random choice and cannot be concluded from other definitions. This choice is however sometimes convenient because then e.g., the formula $\varphi \wedge \bigwedge B$ and the formula $\bigwedge(B \cup \{\varphi\})$ are equivalent. Analogously, the empty disjunction is **false**. In general, the convention is that the result for an empty set of operands is the neutral element. E.g., for real numbers, the empty sum is 0 and the empty product is 1.

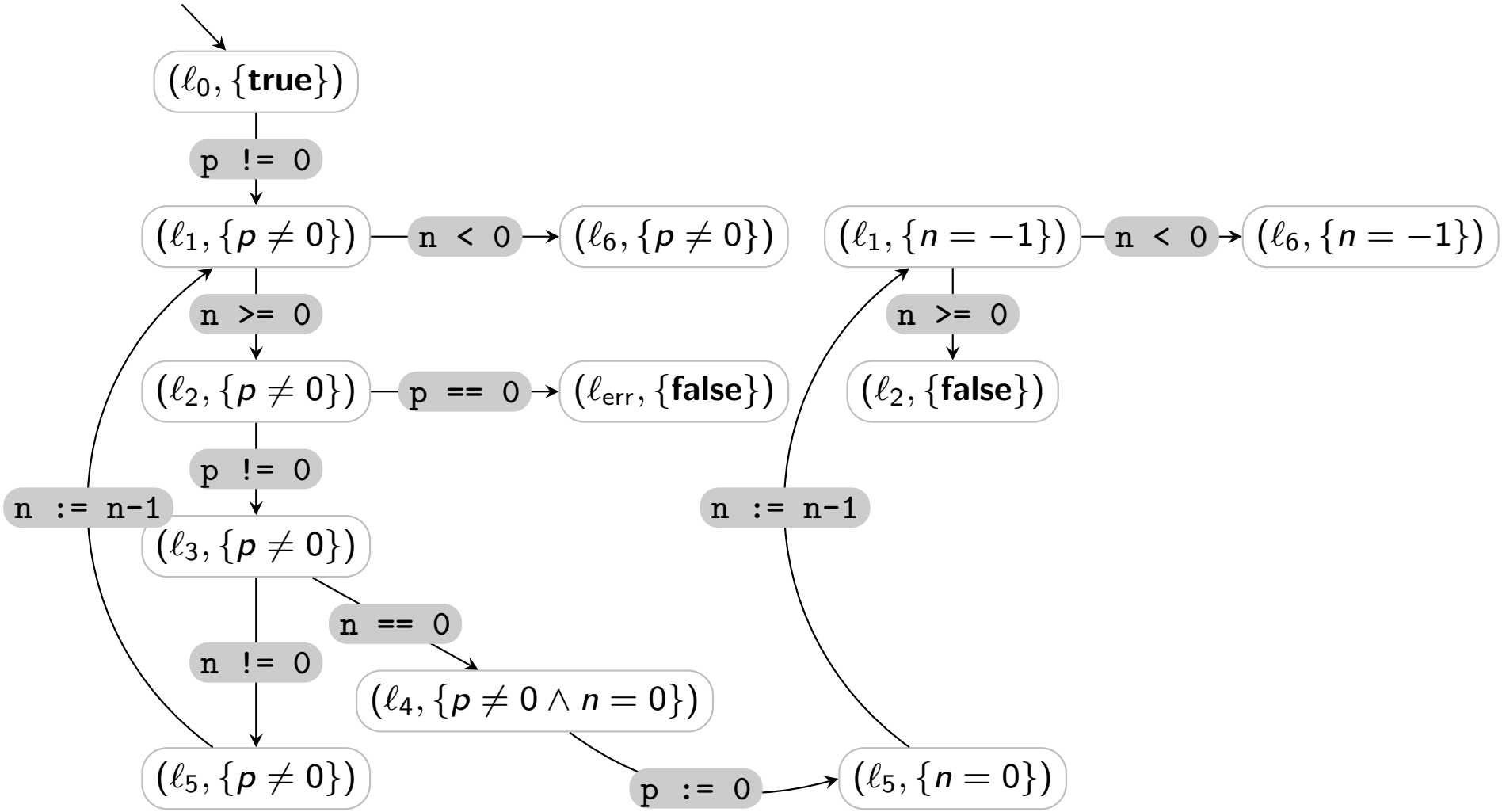
Definition

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.

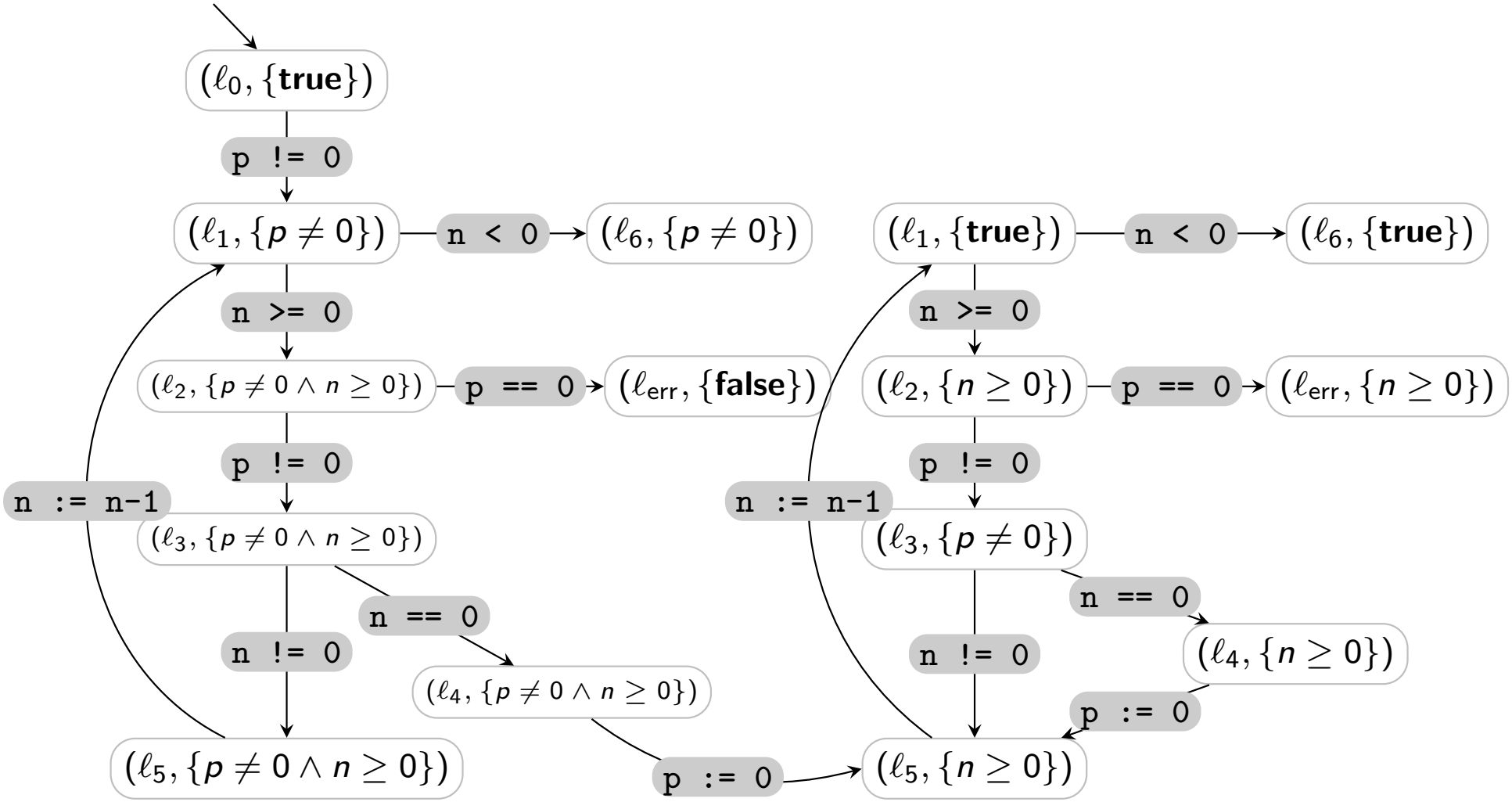
1: **procedure** ConstructACB($(Loc, \Delta, \ell_{\text{init}}, \ell_{\text{ex}})$: CFG, φ_{pre}, B : formulas)
returns (AC, T)
2: $T \leftarrow \emptyset$
3: $AC \leftarrow \{(\ell_{\text{init}}, \{\varphi_{\text{pre}}\})\}$
4: worklist $\leftarrow \{(\ell_{\text{init}}, \{\varphi_{\text{pre}}\})\}$

5: **while** worklist $\neq \emptyset$ **do**
6: $(\ell, S) \leftarrow$RemoveFirst(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** $S' \neq \{\textbf{false}\}$ **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 $B = \{p \neq 0, n \geq 0, n = -1, \textbf{true}, \textbf{false}\}$:



Section 12

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.

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 $p \neq 0 \wedge n \geq 0$	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.

$$\text{abstract}(st) = \begin{cases} \texttt{assume true} & \text{if } st \text{ is of the form } \texttt{assume } \psi \\ \texttt{havoc x} & \text{if } st \text{ is of the form } \texttt{x:=e} \\ \texttt{havoc x} & \text{if } st \text{ is of the form } \texttt{havoc x} \end{cases}$$

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.

trace π	sp for π	abstract trace $\pi^{\#}$	sp for $\pi^{\#}$
	φ_0 **true**		φ_0 **true**
st_1 `p != 0`		`p != 0`	
	φ_1 $p \neq 0$		φ_1 $p \neq 0$
st_2 `n >= 0`		`true`	
	φ_2 $p \neq 0 \wedge n \geq 0$		φ_2 $p \neq 0$
st_3 `p == 0`		`p == 0`	
	φ_3 **false**		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 $p \neq 0 \wedge n \geq 0$		φ_2 **true**
st_3 `p != 0`		`true`	
	φ_3 $p \neq 0 \wedge n \geq 0$		φ_3 **true**
st_4 `n == 0`		`n == 0`	
	φ_4 $p \neq 0 \wedge n = 0$		φ_4 $n = 0$
st_5 `p := 0`		`havoc p`	
	φ_5 $p = 0 \wedge n = 0$		φ_5 $n = 0$
st_6 `n := n-1`		`n := n-1`	
	φ_6 $p = 0 \wedge n = -1$		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**

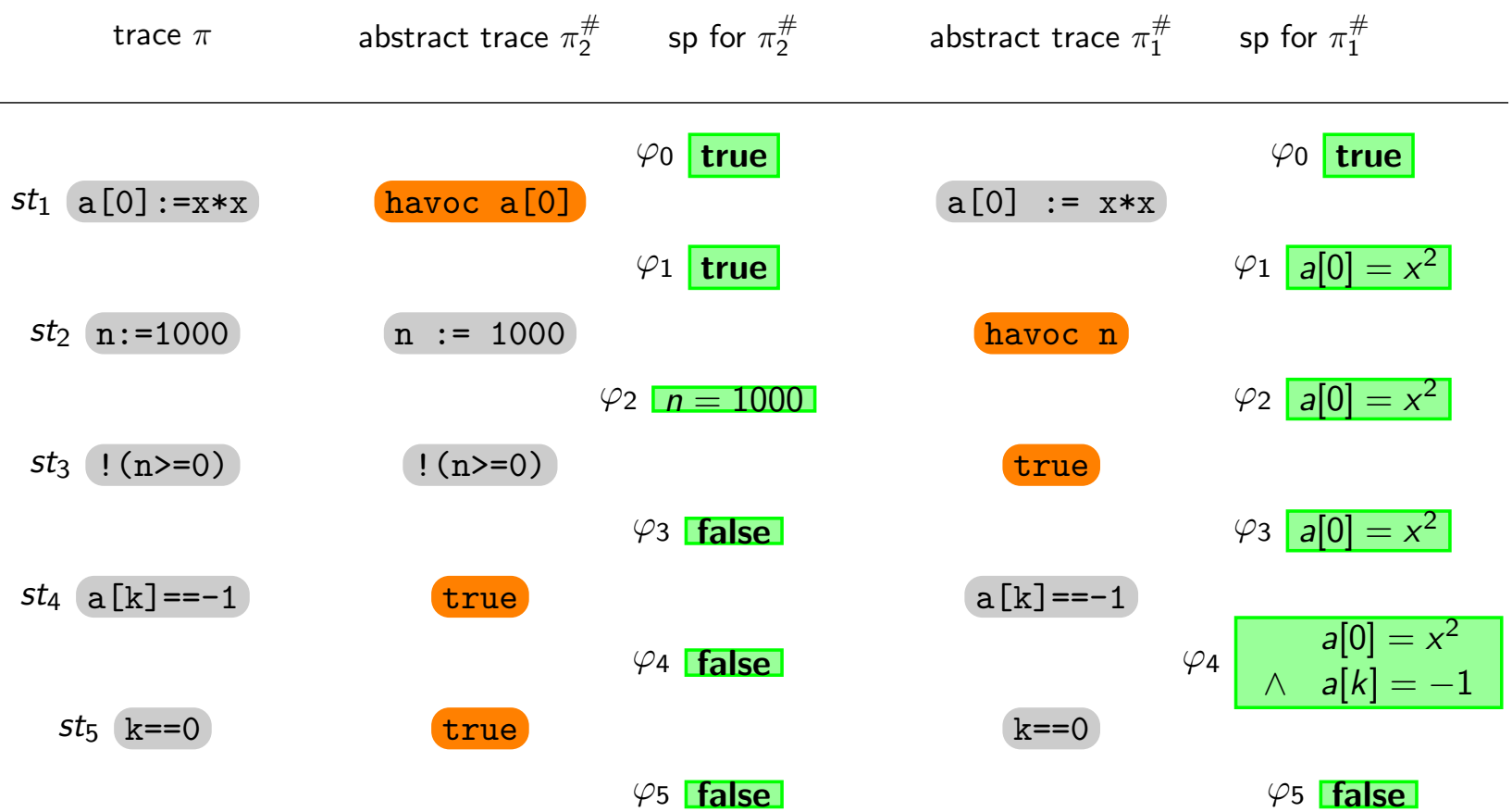
Question: How can we construct the abstract trace $\pi^{\#}$?

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.

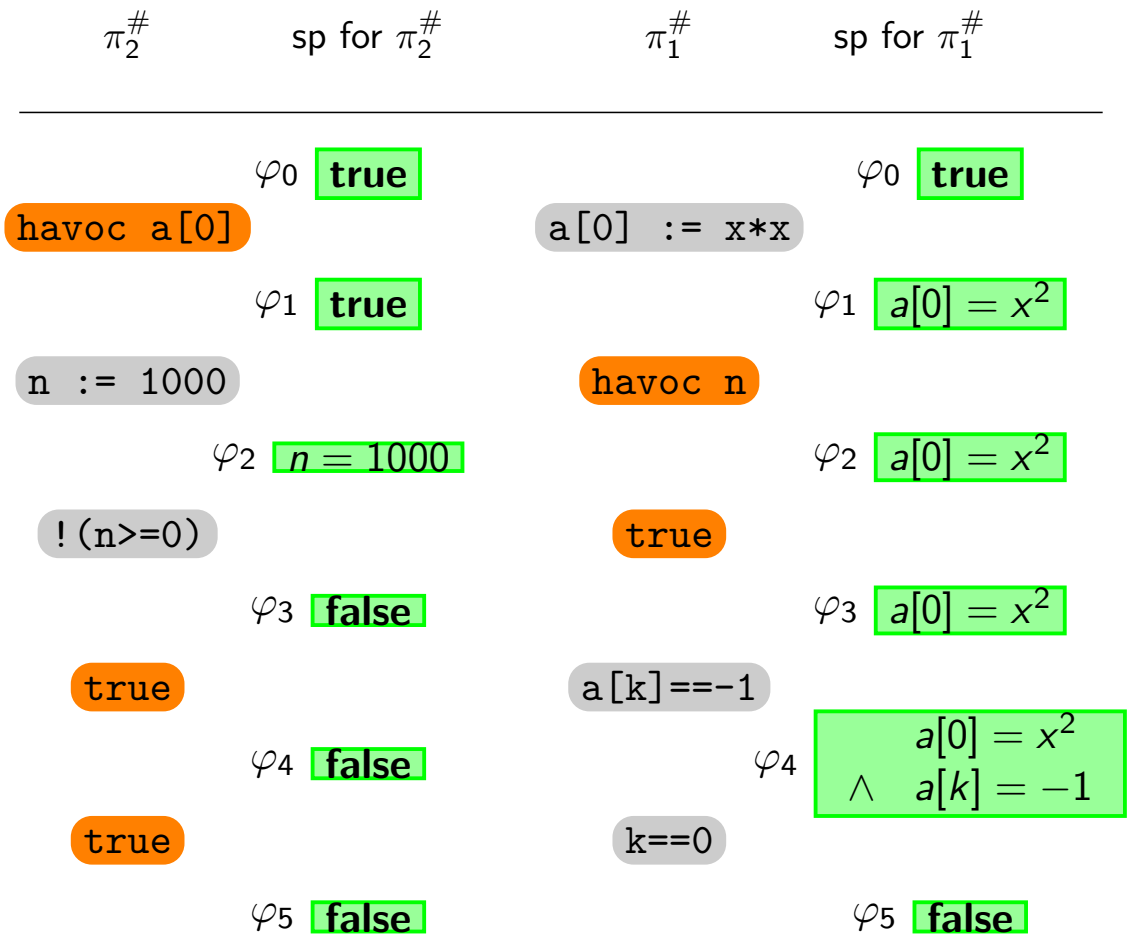
In the worst case, the "advanced approaches" are not better than the "naive approach".

Good Infeasibility Proofs



Good Infeasibility Proofs

```
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 }
```



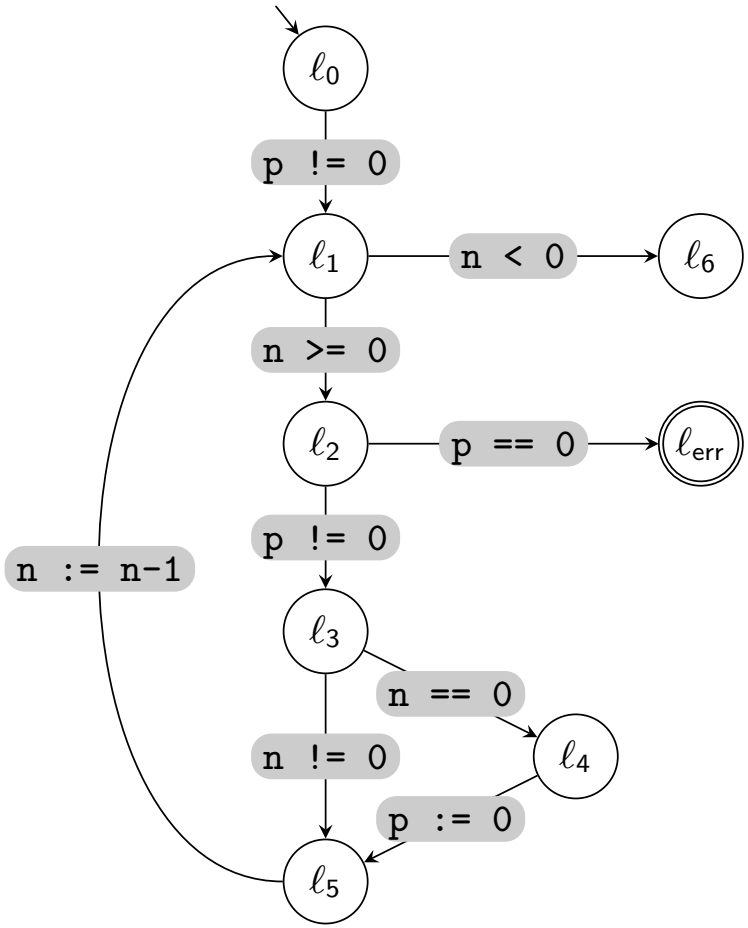
Section 13

CEGAR

Outline

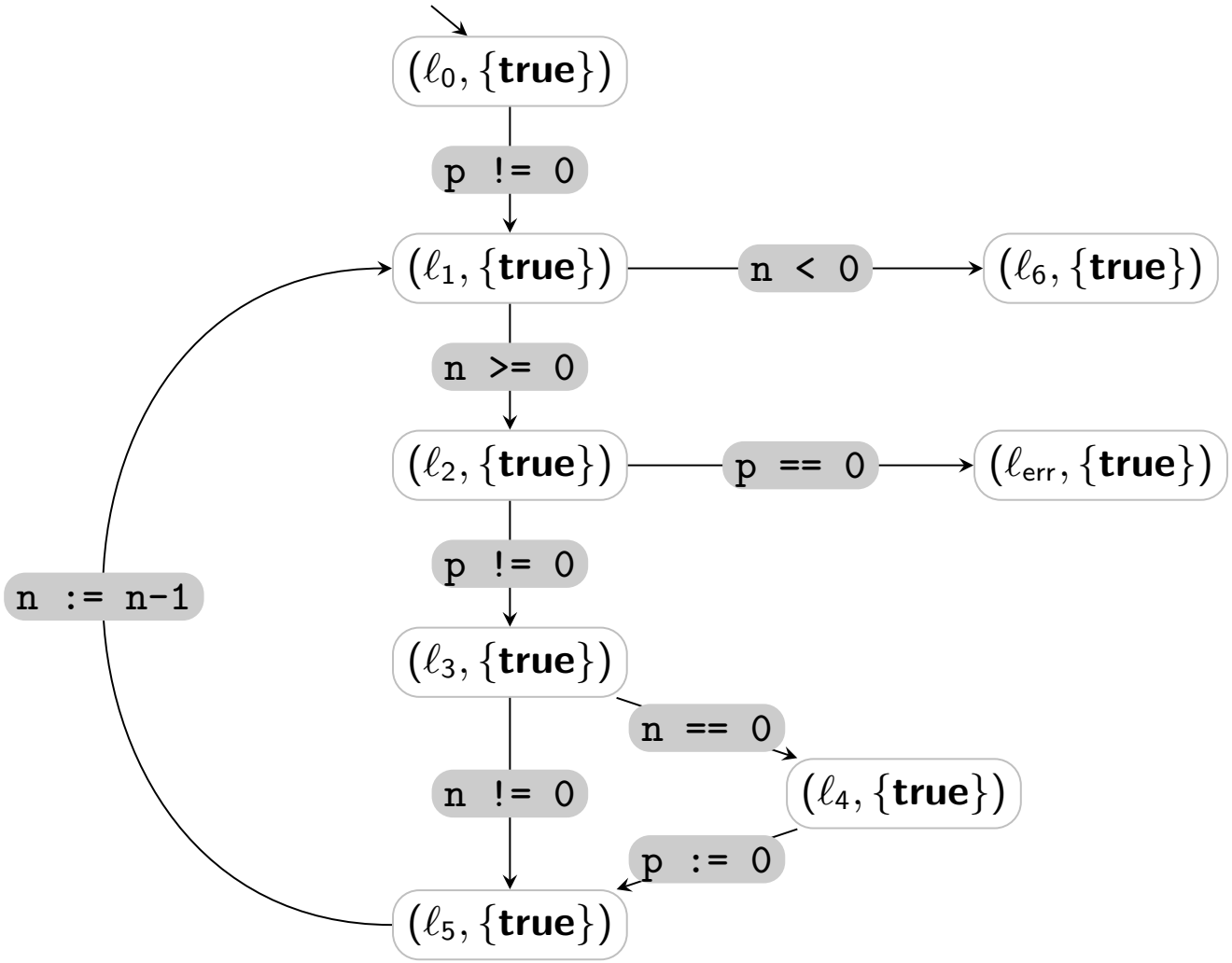
Let us prove that P_{goanna} is correct.

```
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 }
```

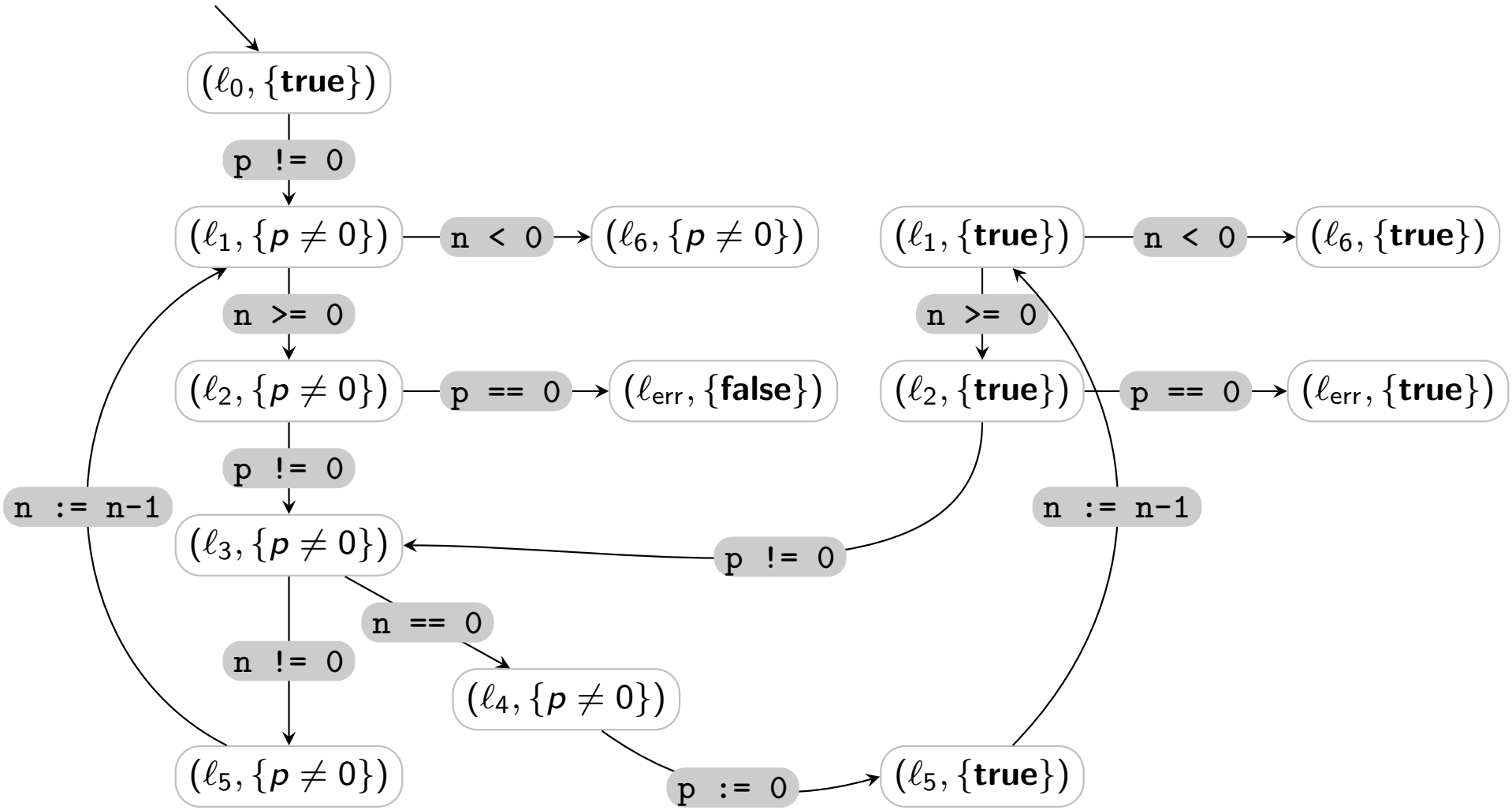


Start with $B = \emptyset$, construct abstract reachability graph that is precise for B.

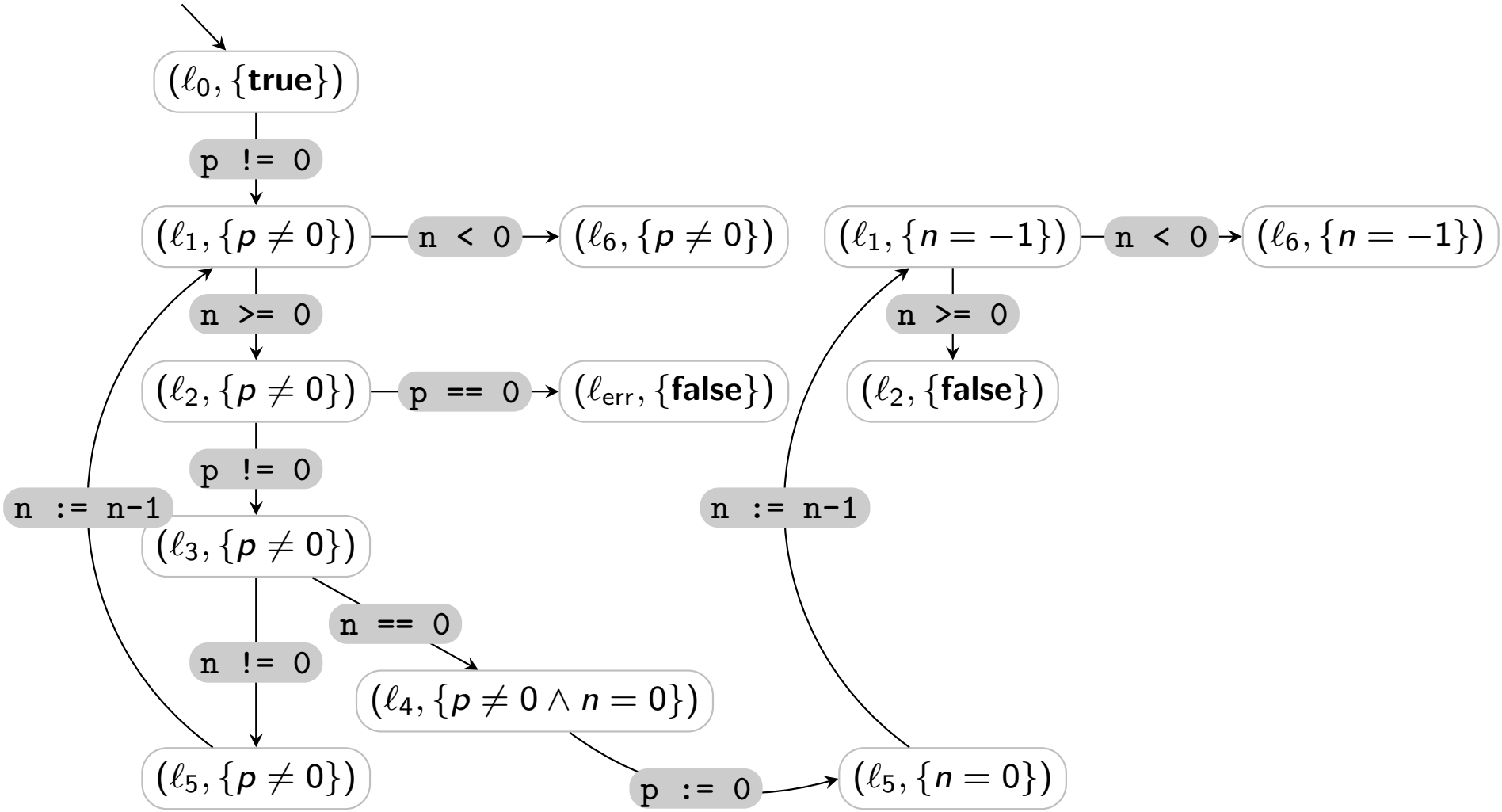
Abstract reachablity graph for $B = \emptyset$:



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)

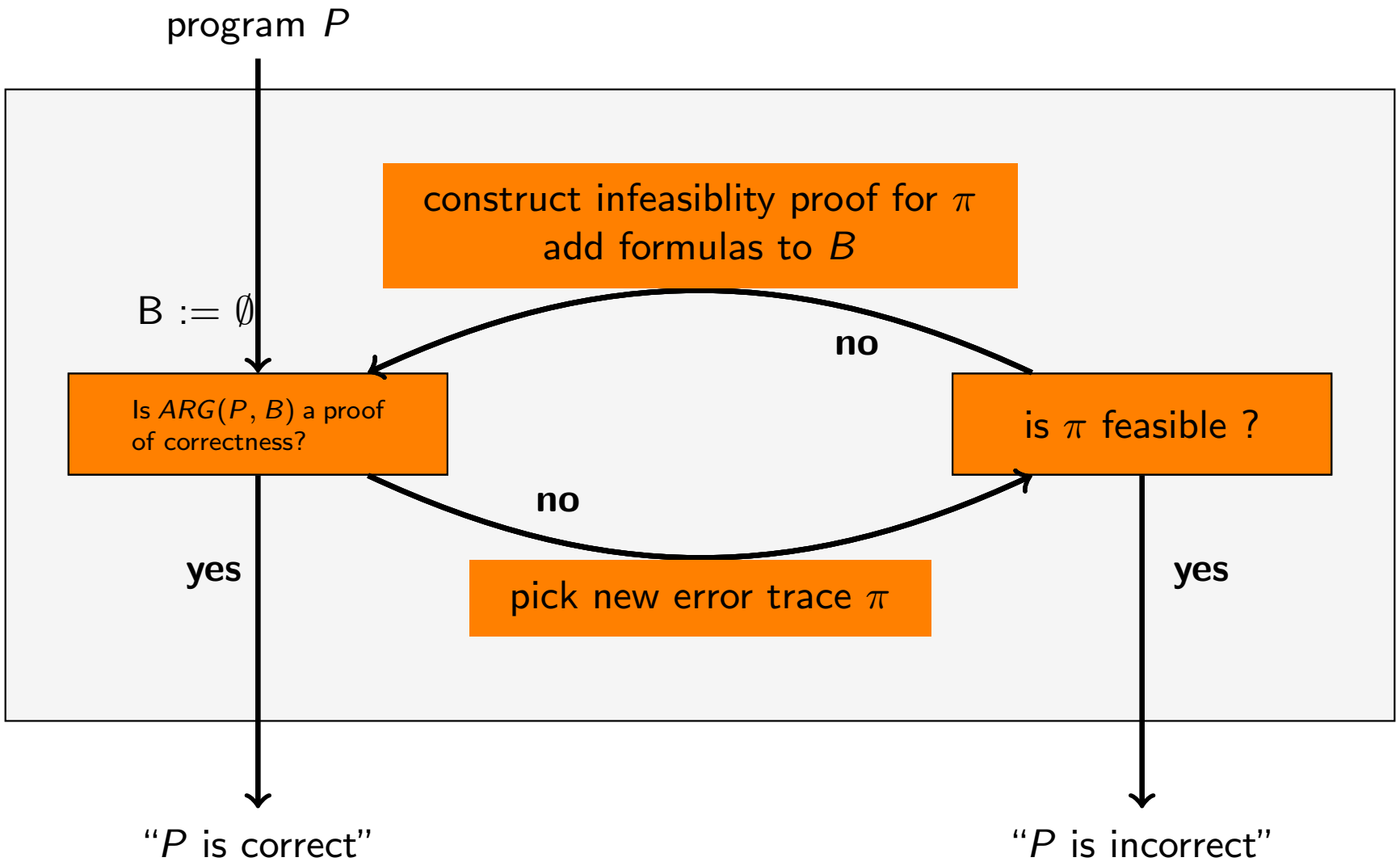
Step 1: Set B to the empty set.

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.

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 $\varphi_2 \models \varphi_1$, we can ignore $(\ell, \{\varphi_2\})$. We say that $(\ell, \{\varphi_2\})$ is already "covered" by $(\ell, \{\varphi_1\})$.

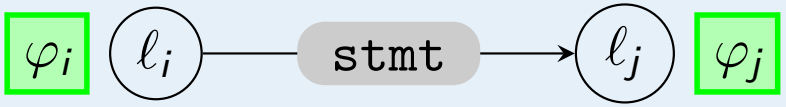
Section 14

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



only if the Hoare triple

$$\{\varphi_i\}\ \texttt{stmt}\ \{\varphi_j\}$$

is valid.

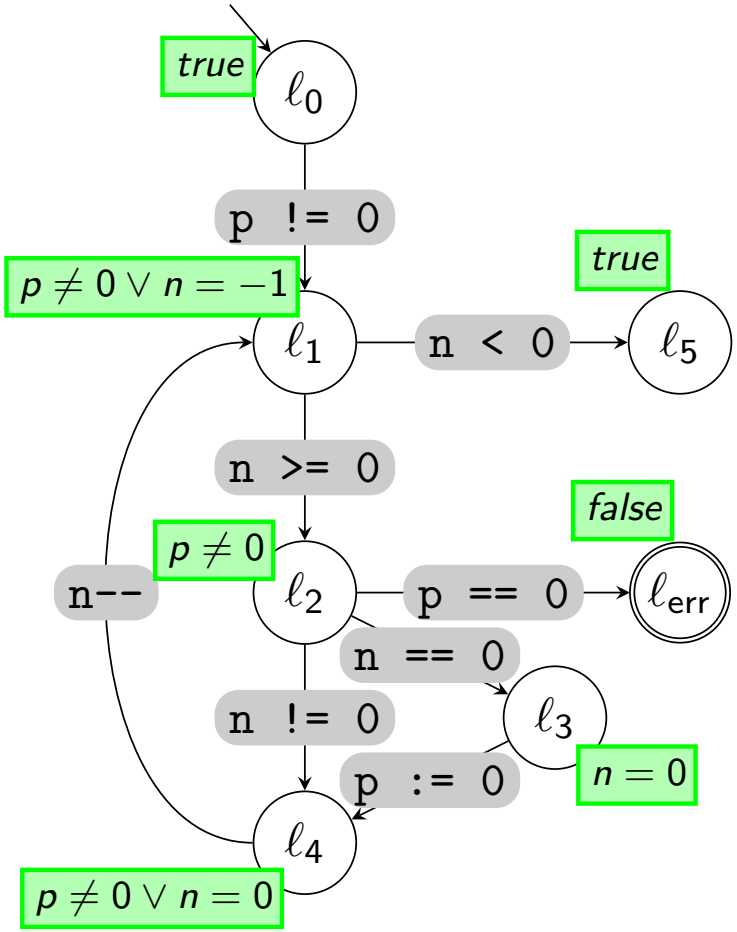
Theorem

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 $\beta : Q \to \mathit{Fmrl}(V)$ such that

- $\beta(q) = \textbf{true}$ for all $q \in Q_{\text{init}}$ and
- $\beta(q) = \textbf{false}$ 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 $q_0 \dots q_n$ for τ. By the definition of Floyd-Hoare automata, $\beta(q_0) \dots \beta(q_n)$ 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

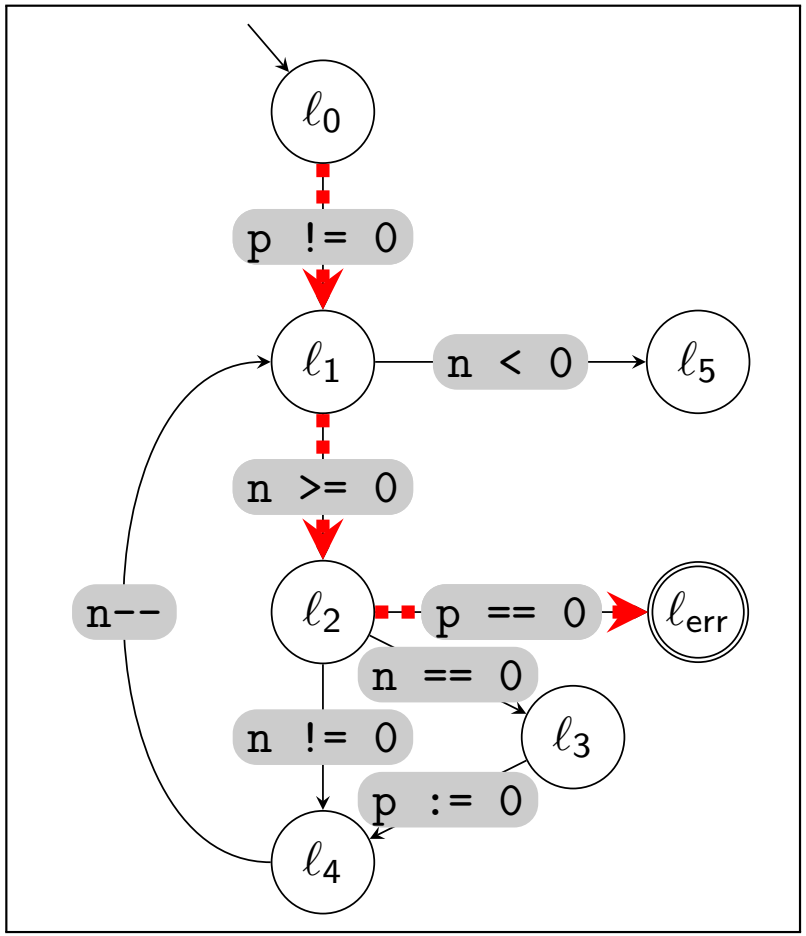
Note that in this formalism, infeasible traces (i.e., traces for which there exists no execution of the program P) may still be accepted by the automaton $\mathcal{A}_P$. The finite automaton cannot distinguish between feasible and infeasible traces.

In fact, the verification task consists precisely of showing that *all* the traces accepted by $\mathcal{A}_P$ are infeasible.

Trace Abstraction: Example

```
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 }
```

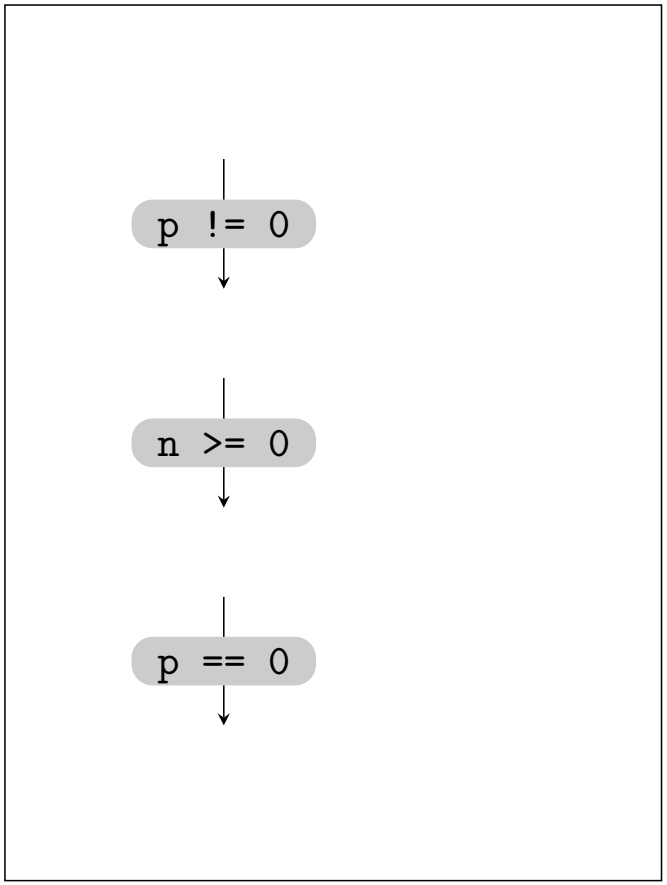
Source Code



Control Flow Graph

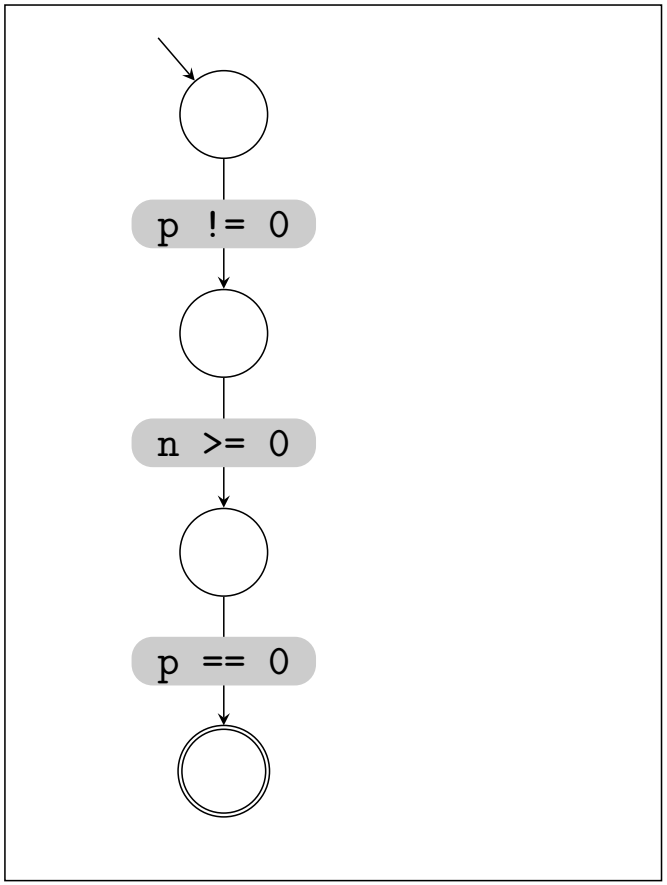
Trace Abstraction: Example

1. take trace π_1



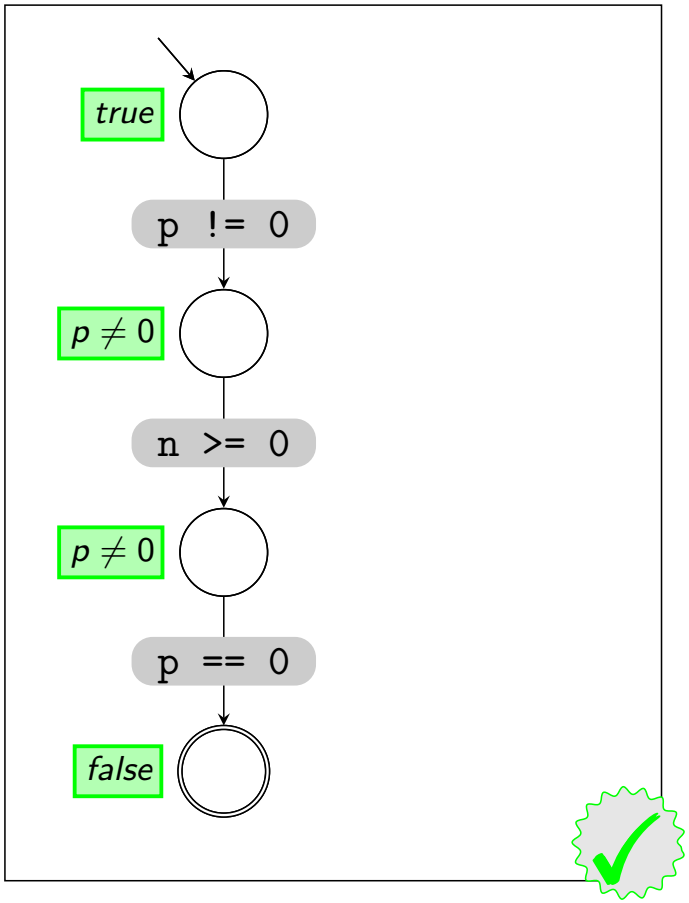
Trace Abstraction: Example

1. take trace π_1
2. consider trace as automaton $\mathcal{A}_1$



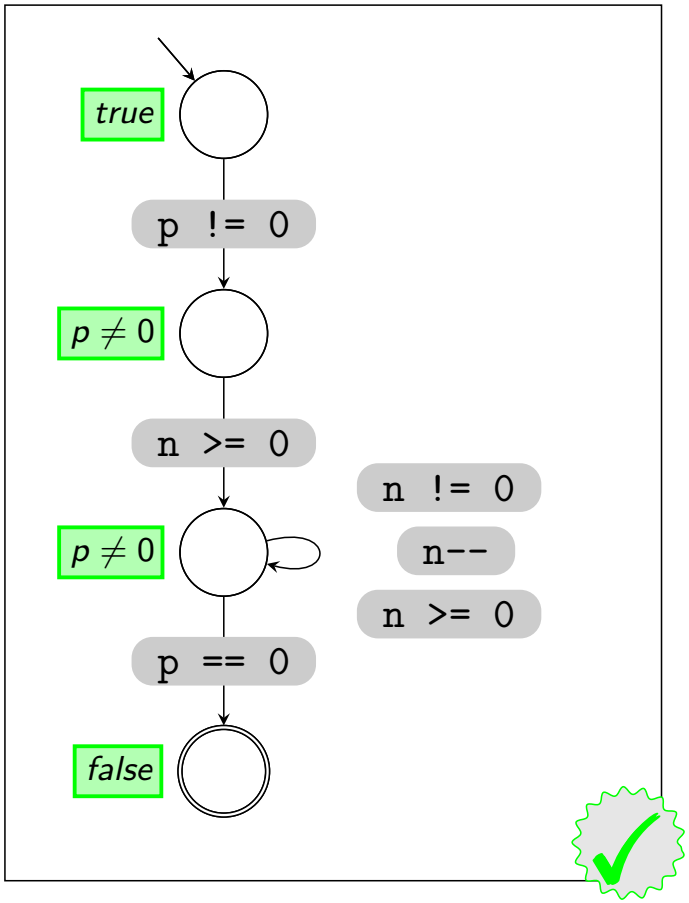
Trace Abstraction: Example

1. take trace π_1
2. consider trace as automaton $\mathcal{A}_1$
3. analyze correctness of $\mathcal{A}_1$, compute Floyd-Hoare annotation



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



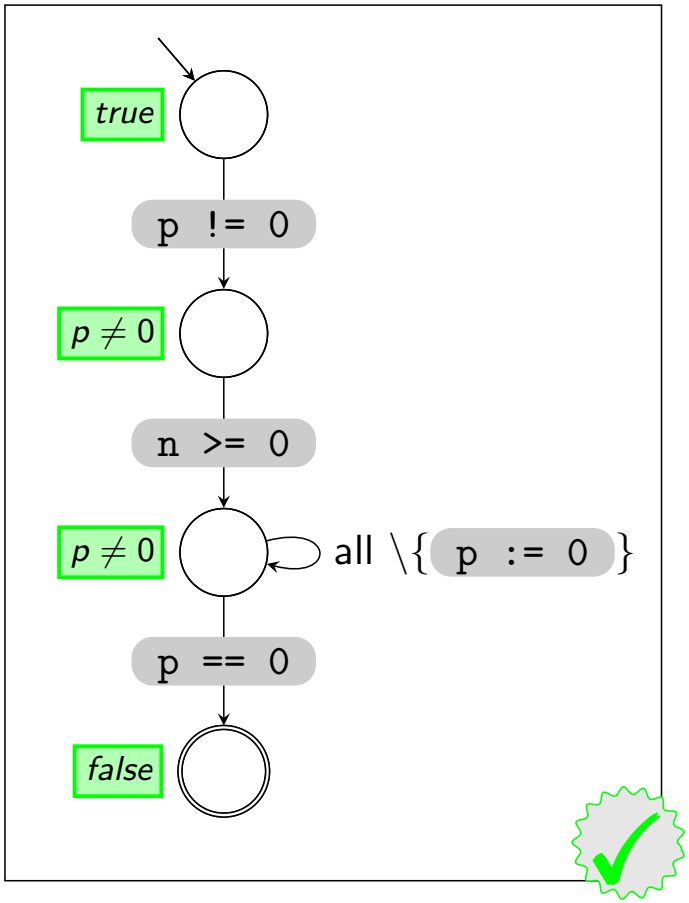
$\{p \neq 0\}$ `n--` $\{p \neq 0\}$ is valid Hoare triple

$\{p \neq 0\}$ `n != 0` $\{p \neq 0\}$ is valid Hoare triple

$\{p \neq 0\}$ `n >= 0` $\{p \neq 0\}$ is valid Hoare triple

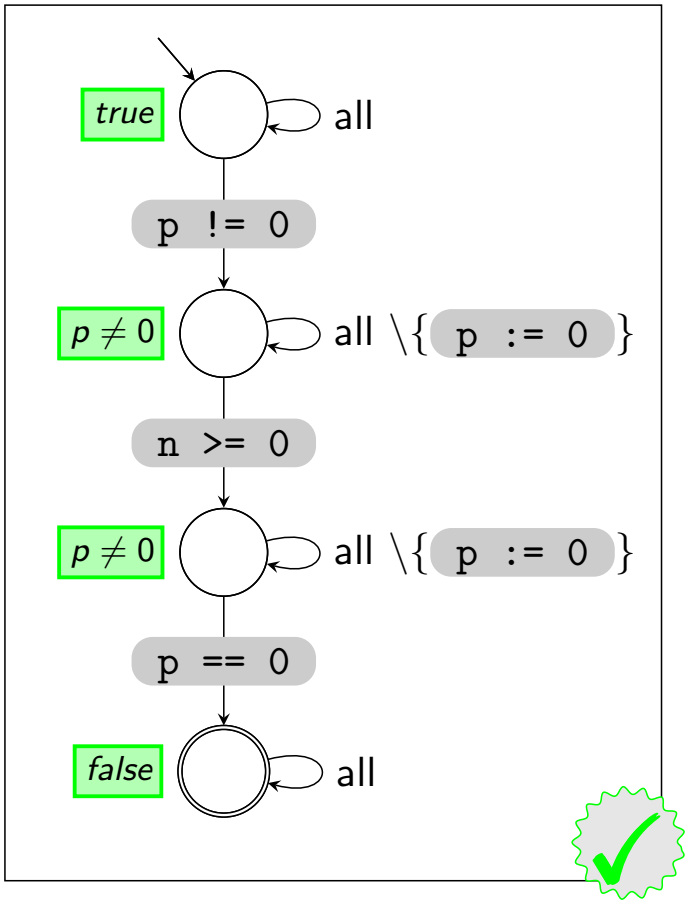
Trace Abstraction: Example

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



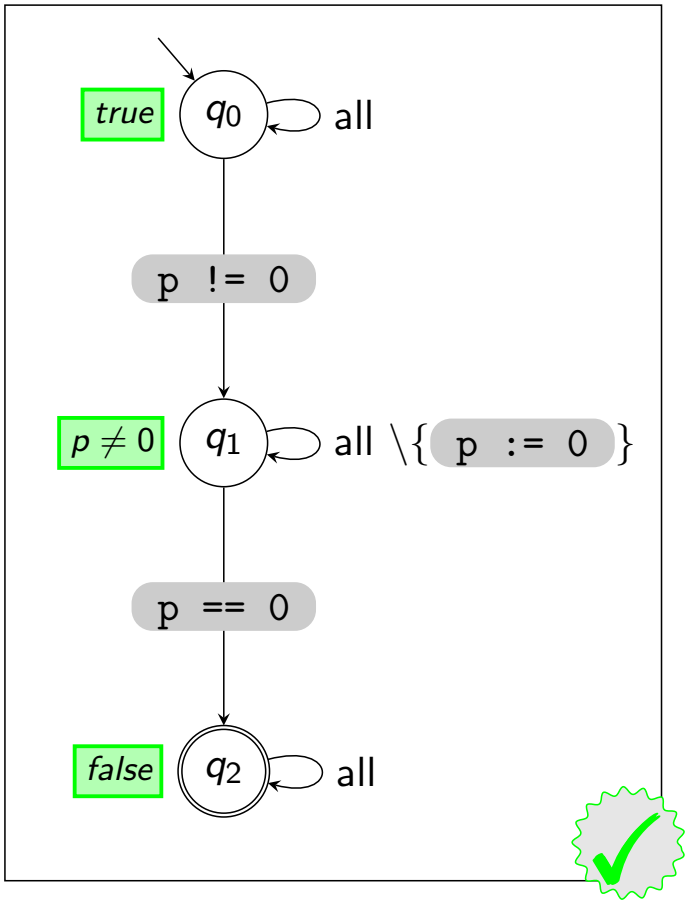
Trace Abstraction: Example

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

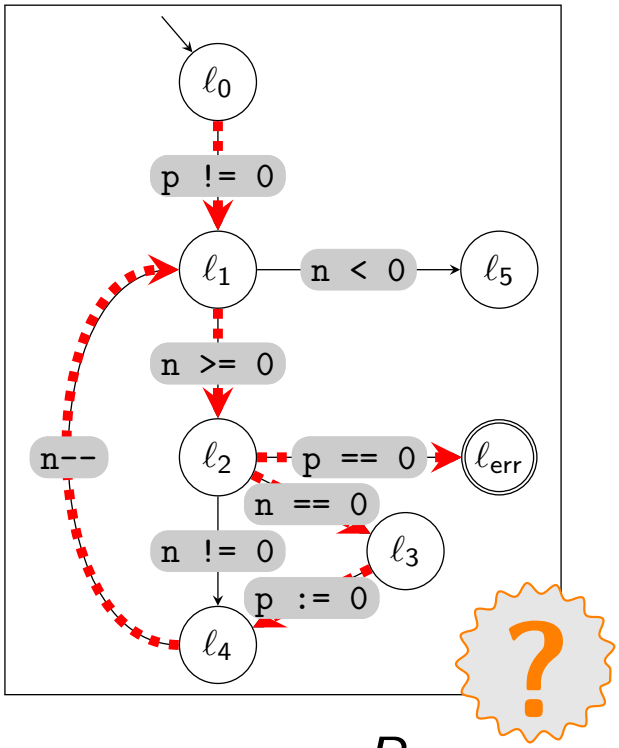


Trace Abstraction: Example

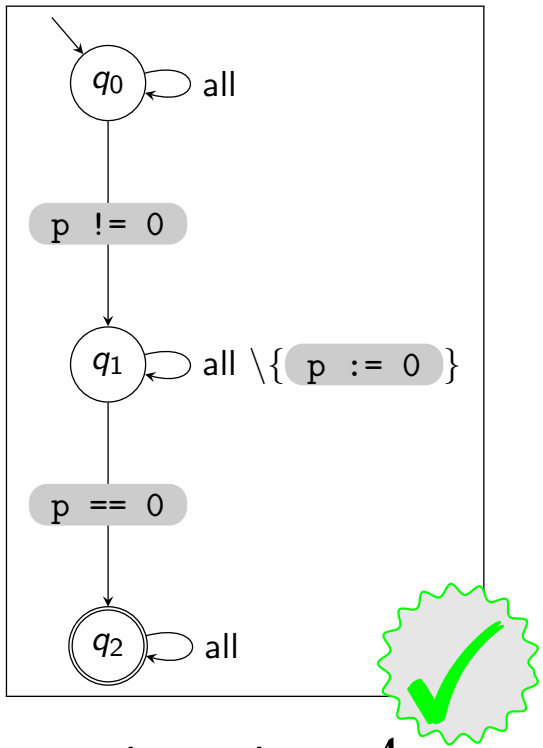
1. take trace π_1
2. consider trace as automaton $\mathcal{A}_1$
3. analyze correctness of $\mathcal{A}_1$, compute Floyd-Hoare annotation
4. generalize automaton $\mathcal{A}_1$
   - add transitions
   - merge states with same annotation



Trace Abstraction: Example

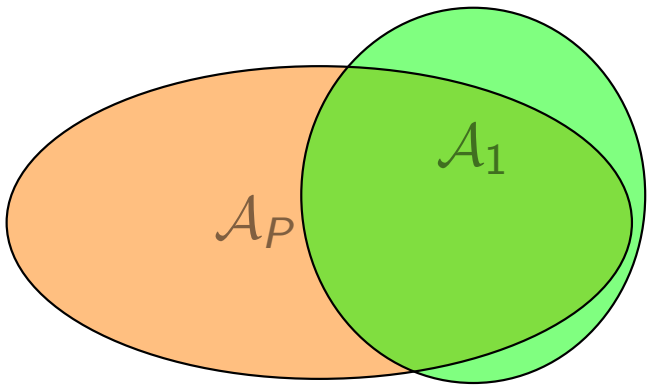


program P



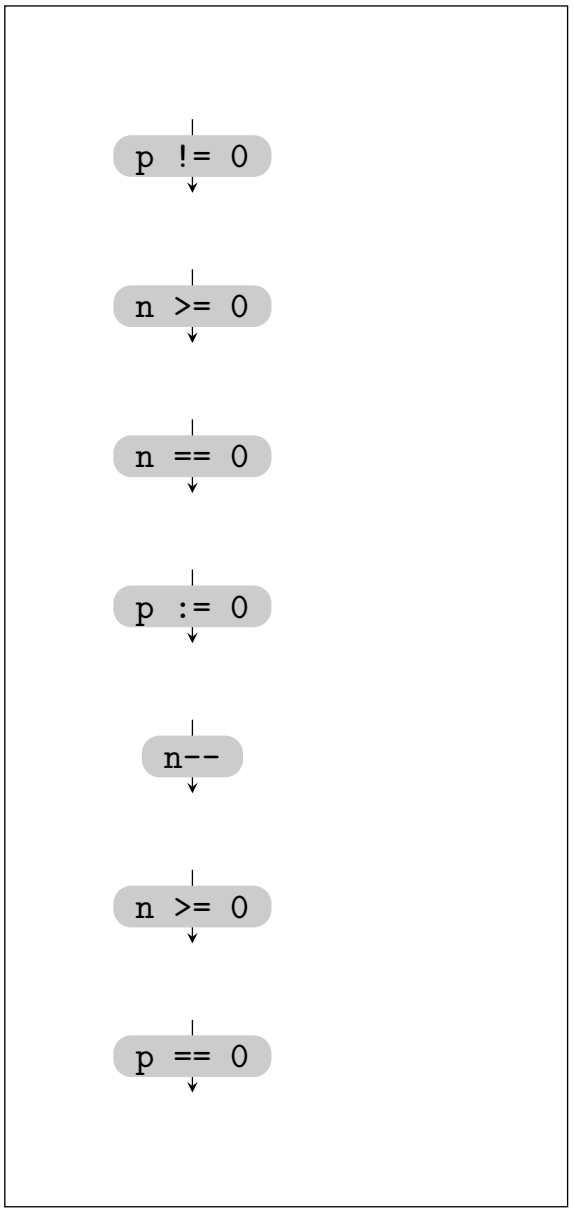
automaton $\mathcal{A}_1$

Consider only traces in set theoretic difference $\mathcal{L}(\mathcal{A}_P) \setminus \mathcal{L}(\mathcal{A}_1)$.



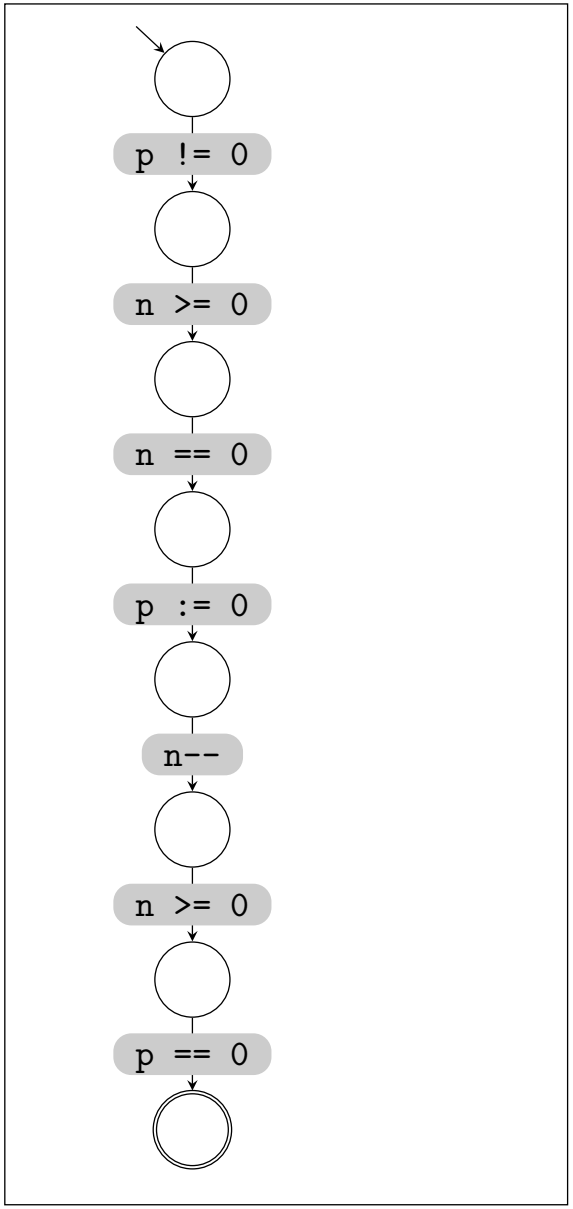
Trace Abstraction: Example

1. take trace π_2



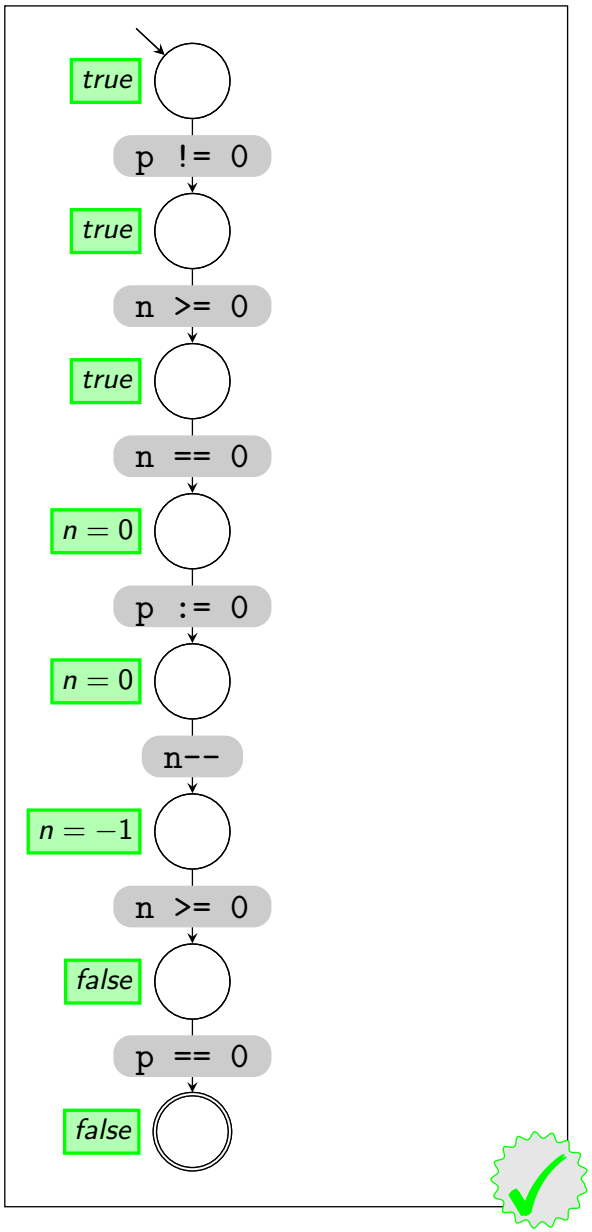
Trace Abstraction: Example

1. take trace π_2
2. consider trace as automaton $\mathcal{A}_2$



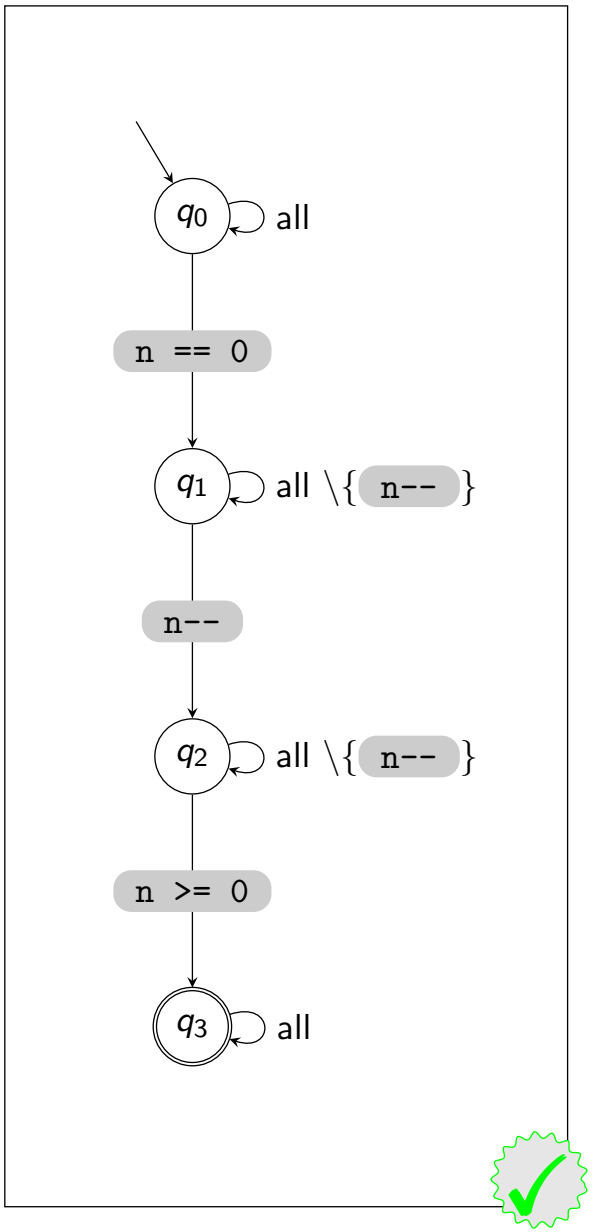
Trace Abstraction: Example

1. take trace π_2
2. consider trace as automaton $\mathcal{A}_2$
3. analyze correctness of $\mathcal{A}_2$, compute annotation

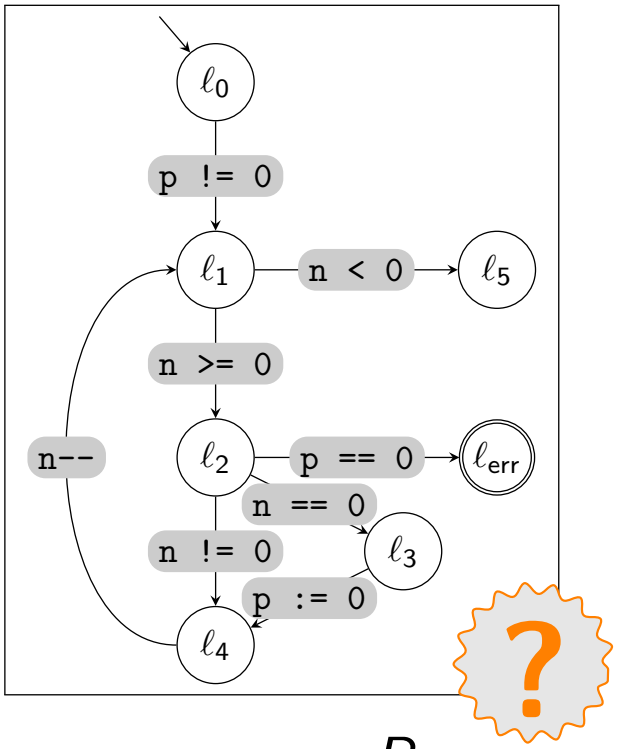


Trace Abstraction: Example

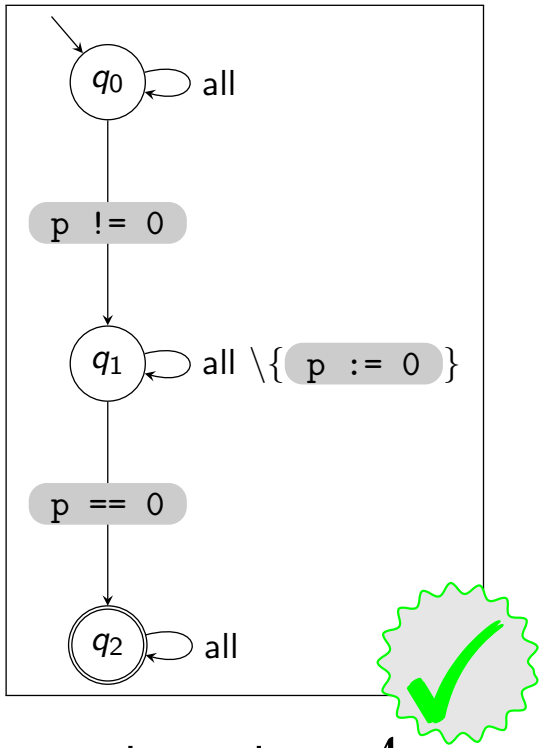
1. take trace π_2
2. consider trace as automaton $\mathcal{A}_2$
3. analyze correctness of $\mathcal{A}_2$, compute annotation
4. generalize automaton $\mathcal{A}_2$
   - add transitions
   - merge states with same annotation



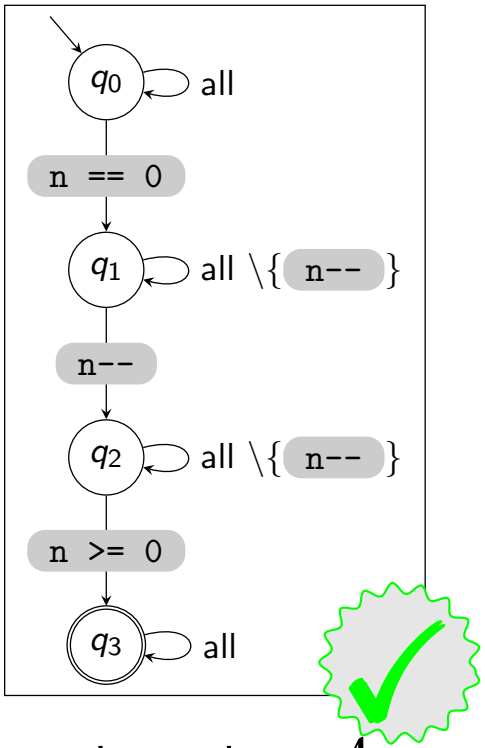
Trace Abstraction: Example



program P



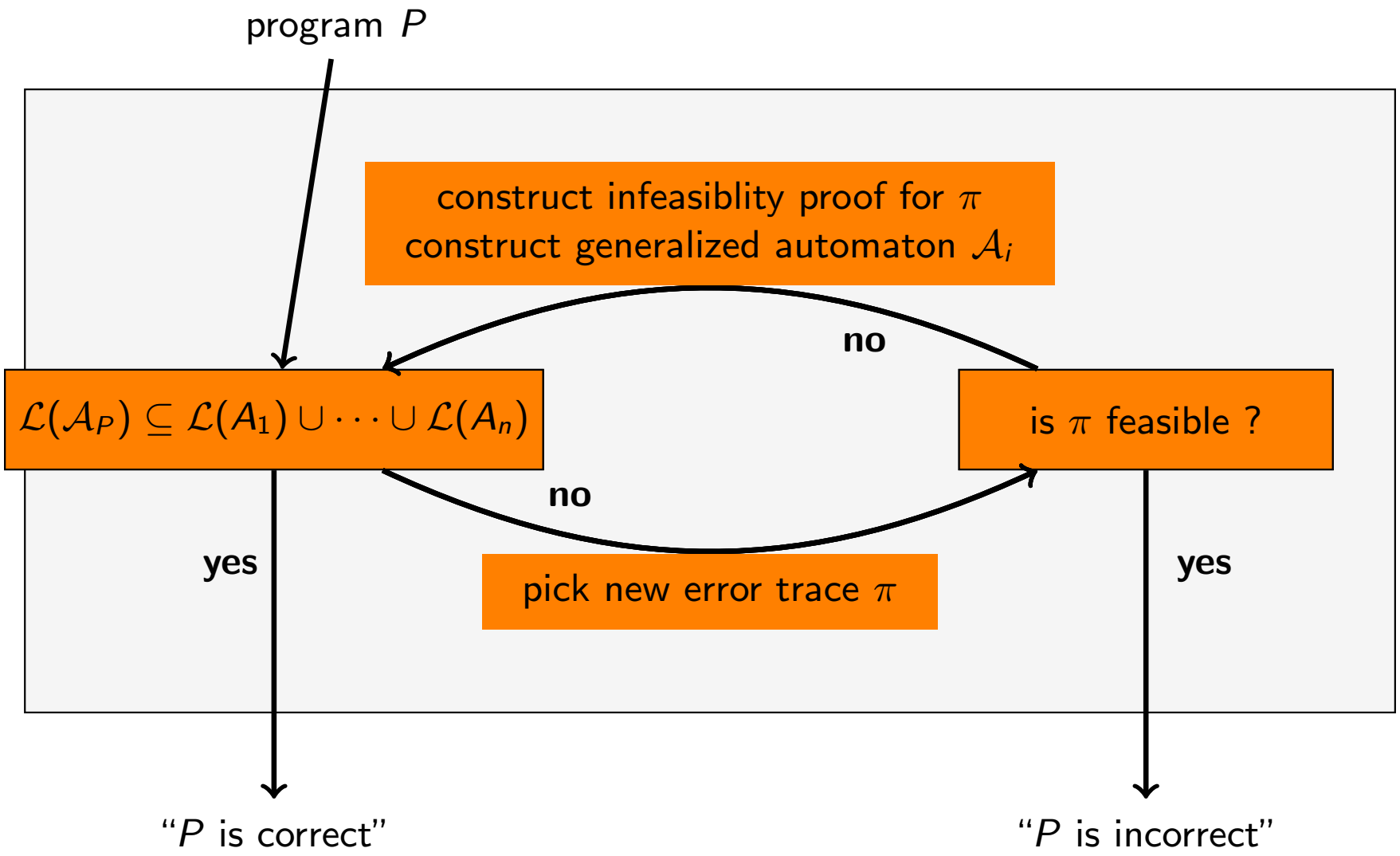
automaton $\mathcal{A}_1$



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



Tool Demo

Ultimate Automizer

Section 15

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.
- 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 Boostan: assume statements.
- If we consider Boostan 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 Boostan 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.

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 2 ofExercise Sheet 23 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.

Outline of the Section on Termination Analysis

Synthesis of ranking functions

Termination analysis via Büchi automata

Tool Demo

Lasso Ranker

Outline of the Section on Termination Analysis

Synthesis of ranking functions

Termination analysis via Büchi automata

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 }
```

Checking Termination of Programs

- classical approach

  compose termination arguments

- our approach

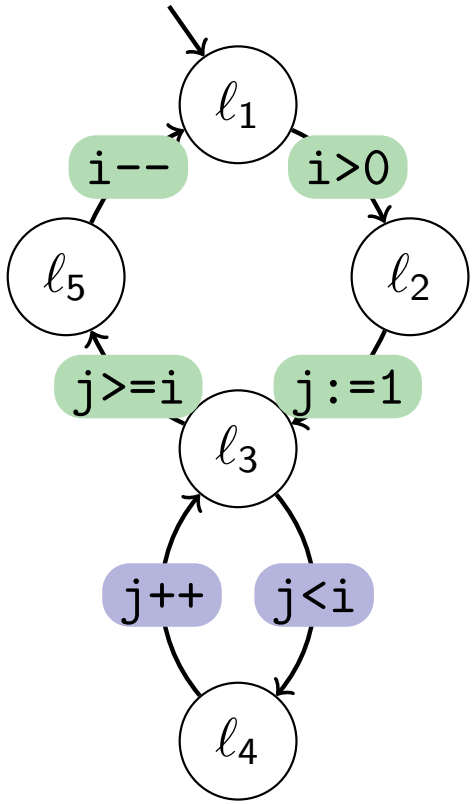
  decompose program into modules

Question:

- What kind of module is suitable for termination proof?

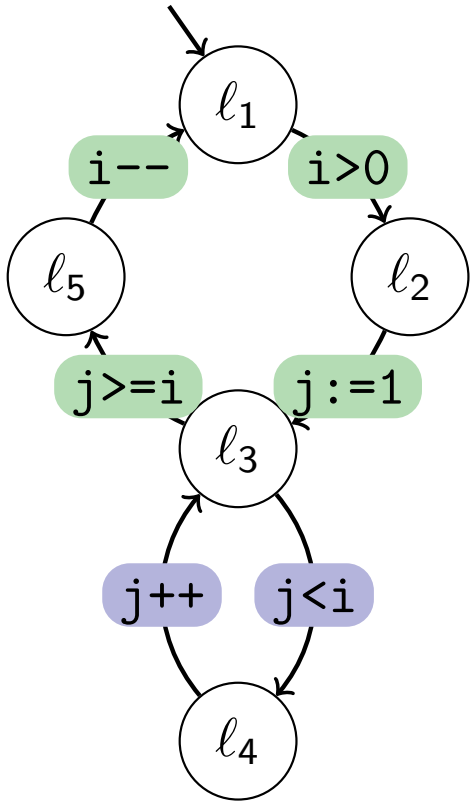
Example: Bubble Sort

```
program sort(int
i, int a[])
ℓ1 while (i>0)
ℓ2       int j:=1
ℓ3       while(j<i)
              if (a[j]>a[i])
                   swap(a,i,j)
ℓ4            j++
ℓ5       i--
```



Example: Bubble Sort

```
program sort(int i)
ℓ1  while (i>0)
ℓ2      int j:=1
ℓ3      while(j<i)
ℓ4          j++
ℓ5      i--
```



quadratic ranking function:

$$f(\mathtt{i},\mathtt{j}) = \mathtt{i}^2 - \mathtt{j}$$

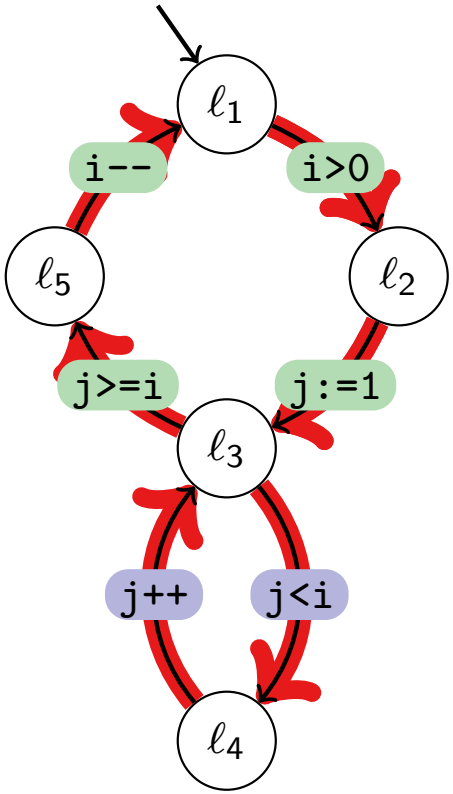
lexicographic ranking function:

$$f(\mathtt{i},\mathtt{j}) = (\mathtt{i}, \mathtt{i} - \mathtt{j})$$

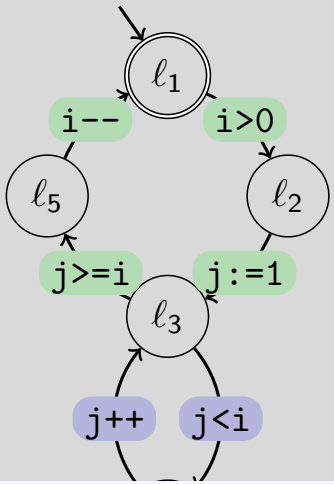
Example: Bubble Sort

single trace $\textsc{Outer}^\omega$
has ranking function $f(\mathtt{i}, \mathtt{j}) = \mathtt{i}$

is also ranking function for
set of traces $(\textsc{Inner}^*.\textsc{Outer})^\omega$



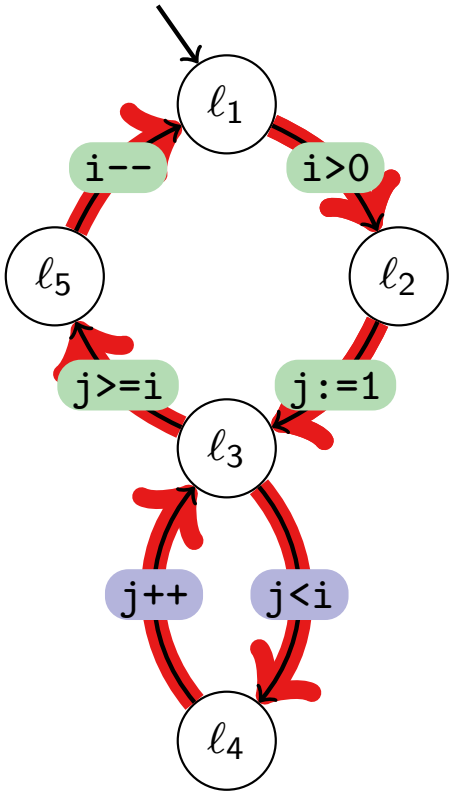
module $\mathcal{P}_1$:
program with fairness constraint
whose set of traces is
$(\textsc{Inner}^*.\textsc{Outer})^\omega$



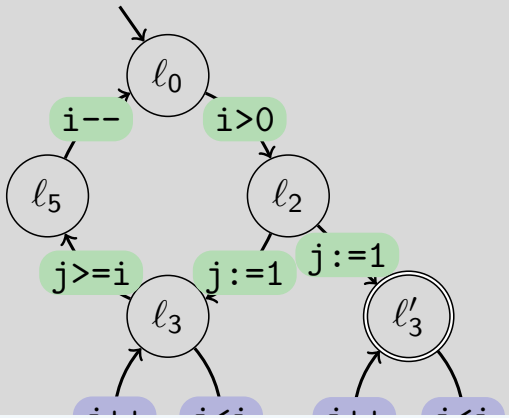
Example: Bubble Sort

new trace $\textsc{Outer}.\textsc{Inner}^\omega$
has ranking function $f(\mathtt{i},\mathtt{j}) = \mathtt{i} - \mathtt{j}$

is also ranking function for
set of traces $(\textsc{Inner} + \textsc{Outer})^*.\textsc{Inner}^\omega$

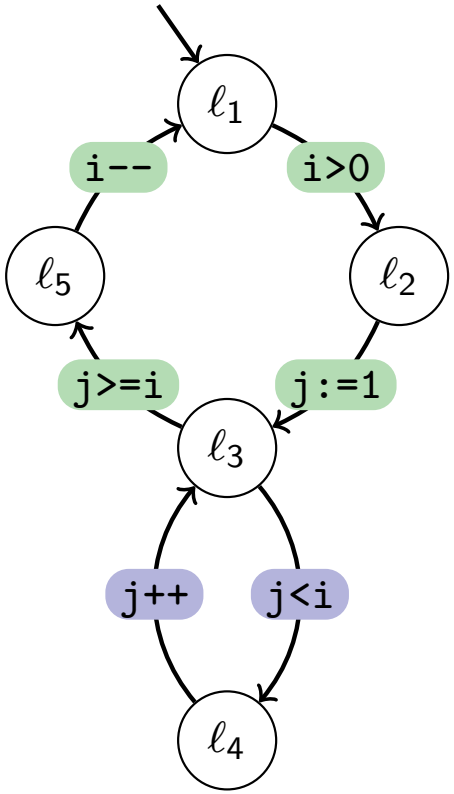


module $\mathcal{P}_2$:
program with fairness constraint whose
set of traces is
$(\textsc{Inner} + \textsc{Outer})^*.\textsc{Inner}^\omega$

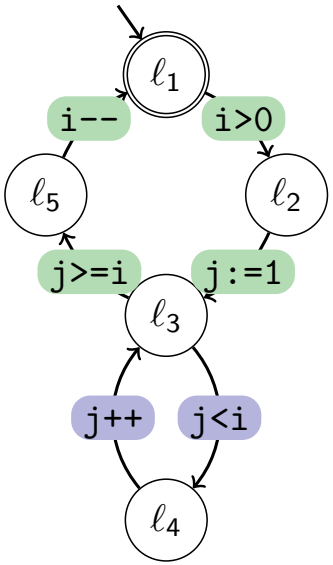


program $\mathcal{P}$ | module $\mathcal{P}_1$ | module $\mathcal{P}_2$

$(\textsc{Outer}\ \textsc{Inner})^\omega + = (\textsc{Inner}^*.\textsc{Outer})^\omega + (\textsc{Inner}\ \textsc{Outer})^*.\textsc{Inner}^\omega +$



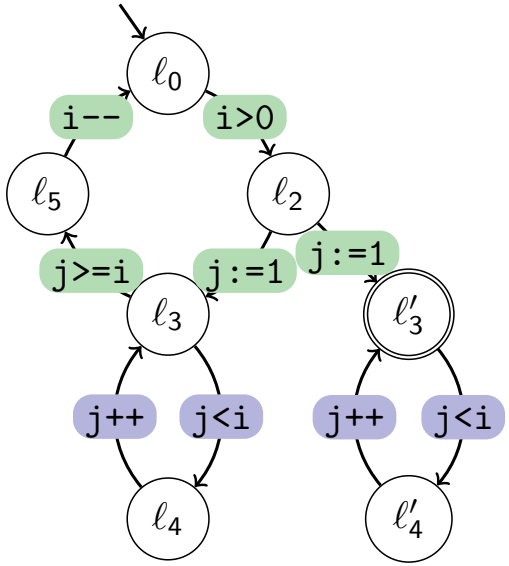
=



ranking function

$f(\mathtt{i},\mathtt{j}) = \mathtt{i}$

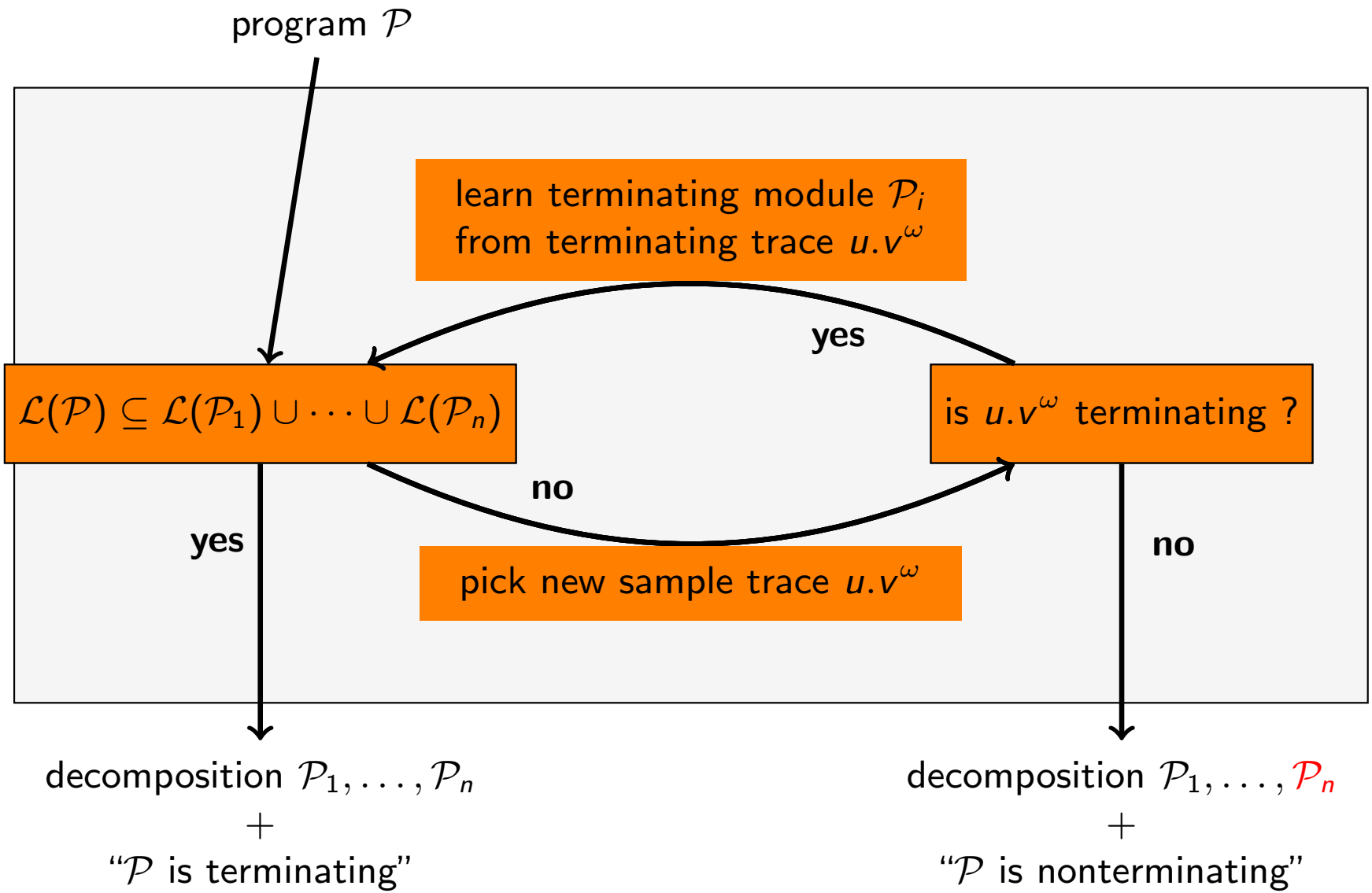
∪



ranking function

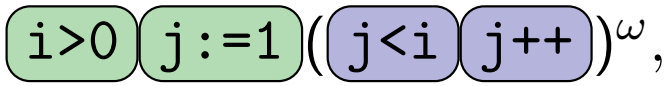
$f(\mathtt{i},\mathtt{j}) = \mathtt{i} - \mathtt{j}$

Algorithm for Construction of Decomposition $\mathcal{P}_1, \ldots, \mathcal{P}_n$

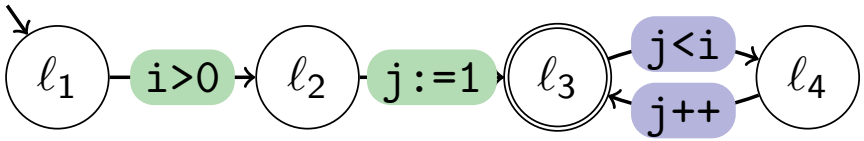


Learn Module from Trace – Example

input: ultimately periodic trace



1. construct trivial module



2. synthesize ranking function

$$f(i,j) = i - j$$

Colón, Sipma	**Synthesis of Linear Ranking Functions**	(TACAS 2001)
Podelski, Rybalchenko	**A complete method for the synthesis of linear ranking functions**	(VMCAI 2004)
Bradley, Manna, Sipma	**Termination Analysis of Integer Linear Loops**	(CONCUR 2005)
Bradley, Manna, Sipma	**Linear ranking with reachability**	(CAV 2005)
Bradley, Manna, Sipma	**The polyranking principle**	(ICALP 2005)
Ben-Amram, Genaim	**Ranking functions for linear-constraint loops**	(POPL 2013)
H., Hoenicke, Leike, Podelski	**Linear Ranking for Linear Lasso Programs**	(ATVA 2013)
Cook, Kroening, Rümmer, Wintersteiger	**Ranking function synthesis for bit-vector relations**	(FMSD 2013)
Leike, H.	**Ranking Templates for Linear Loops**	(TACAS 2014)

Tool Demo

Büchi Automizer

Section 16

LTL Software Model Checking

Outline

Tool Demo

LTL Automizer

References I

[1] Dirk Beyer et al. "Correctness witnesses: exchanging verification results between verifiers". In: *SIGSOFT FSE*. ACM, 2016, pp. 326–337.

[2] Dirk Beyer et al. "Witness validation and stepwise testification across software verifiers". In: *ESEC/SIGSOFT FSE*. ACM, 2015, pp. 721–733.

[3] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. "SMTInterpol: An Interpolating SMT Solver". In: *SPIN*. Vol. 7385. Lecture Notes in Computer Science. Springer, 2012, pp. 248–254.

[4] Edmund M. Clarke et al., eds. *Handbook of Model Checking*. Springer, 2018.

[5] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. "Software Model Checking for People Who Love Automata". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 36–52.

[6] K. Rustan M. Leino. "This is Boogie 2". 2008.

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *TACAS*. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340.