LAST LECTURE

- Bounded Model Checking of Programs by reduction to SMT
- Assignment to variables replaced by equality predicate, arithmetic operators replaced by corresponding functions/predicates in LIA.

Z3

INTRODUCTION

- Z3 is a constraint-solver/theorem-prover developed at Microsoft Research.
- Basic Operation:
  - It takes as input a formula [PL/FOL/SMT].
  - Outputs SAT/UNSAT.
- Supports a whole range of theories (including all theories we have seen).
- Open-source (written in C++)
  - Latest version available at Z3 Github page (https://github.com/Z3Prover/z3).

INPUT/OUTPUT FORMAT

1. APIs for Python, C++, Java, etc.
    - API functions for declaring variables, constants, predicates, functions, and for constructing formula.
    - API functions for accessing a satisfying interpretation (in case of SAT).
2. SMT-LIB 2.0
    - Standard input format for all SMT solvers
    - Formula written in SMT-LIB 2.0 can be directly provided to the Z3 executable.

INPUT FORMAT

- Z3 expects input formula in Many Sorted First Order Logic (MSFOL).
    - ‘sort’ is similar to type. Variables, constants, functions, predicates must be given appropriate types.
    - Built-in sorts: Bool, Integer, Real, Array,…
    - Users can also define new sorts.

SMT-LIB EXAMPLE

$year_0 = 2008 \land$
$g_0 = (days_0 > 365) \land$
$oldDays_0 = days_0 \land$
$g_1 = (IsLeapYear(year_0)) \land$
$g_2 = (days_0 > 366)) \land$
$days_1 = days_0 - 366 \land$
$year_1 = year_0 + 1 \land$
$days_2 = ite(g_1 \&\& g_2, days_1, days_0) \land$
$year_2 = ite(g_1 \&\& g_2, year_1, year_0) \land$
$days_3 = days_0 - 365 \land$
$year_3 = year_0 + 1 \land$
$days_4 = ite(g_1, days_2, days_3) \land$
$year_4 = ite(g_1, year_2, year_3) \land$
$(\neg(days_4 < oldDays_0) \lor \neg(days_4 <= 365))$

```
(declare-const year0 Int)
(declare-const g0 Bool)
(declare-fun IsLeapYear (Int) Bool)
.
.
(assert (= year0 2008))
(assert (= g0 (> days0 365)))
.
.
(assert (or (not (< days4 oldDays0)) (not (<= days4 365))))

(check-sat)
(get-model)
```

TUTORIALS

- For SMT-LIB
  - https://rise4fun.com/Z3/tutorial/guide
- For Python API
  - http://theory.stanford.edu/~nikolaj/programmingz3.html
- Download, Installation instructions
  - https://github.com/Z3Prover/z3

TOPICS NOT COVERED

- Decision procedures for various theories
- First-Order Logic Normal Forms (Clausal Normal Form, Skolem Normal Form), FOL Resolution.
- Nelson-Oppen Method, DPLL(T)
- Extensions of FOL for Verification: Linear Temporal Logic, Computational Tree Logic

COURSE STRUCTURE

CONSTRAINT SOLVERS

- Propositional Logic, SAT solving, DPLL
- First-Order Logic, SMT
- First-Order Theories

DEDUCTIVE VERIFICATION

- Operational Semantics
- Strongest Post-condition, Weakest Pre-condition
- Hoare Logic

MODEL CHECKING AND OTHER VERIFICATION TECHNIQUES

- Predicate Abstraction, CEGAR
- Abstract Interpretation
- Property-directed Reachability
- …

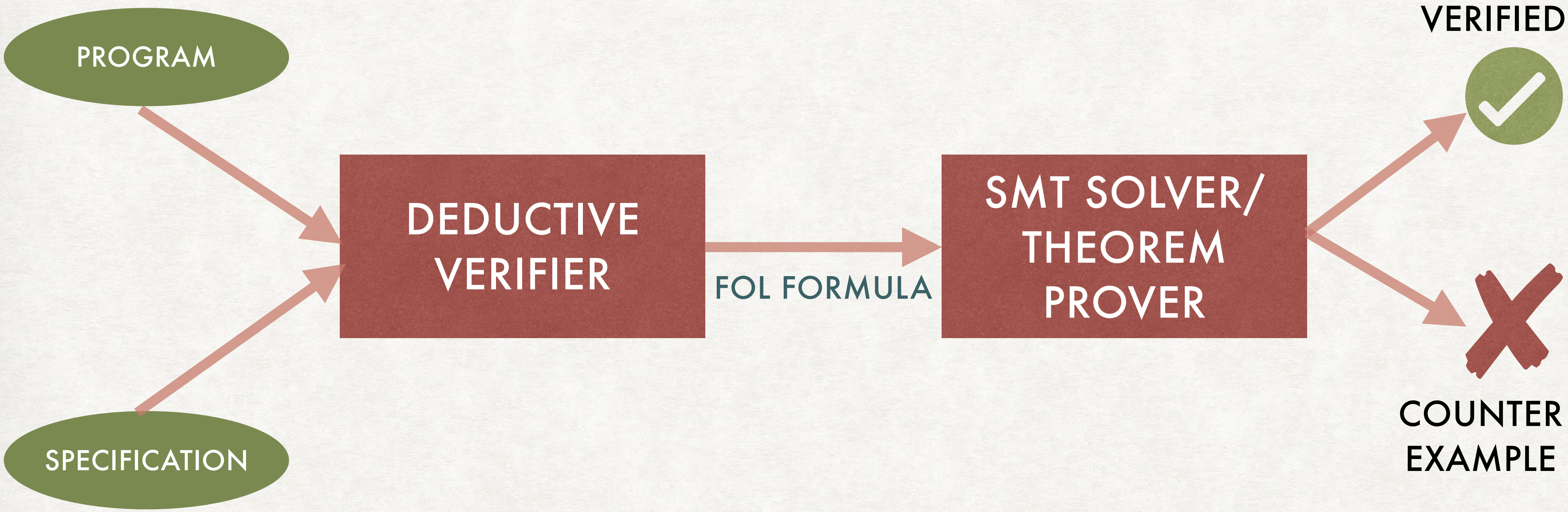
FORMAL SPECIFICATION AND VERIFICATION OF PROGRAMS

INTRODUCTION

- So far we have seen…
  - Syntax, Semantics for Propositional Logic and First-Order Logic and (some examples of) Decision Procedures for Validity/ Satisfiability
  - Underlying engine for Deductive Verification of programs
- Now we will study some well-known schemes to reduce the automated verification problem to the satisfiability problem in first-order logic.

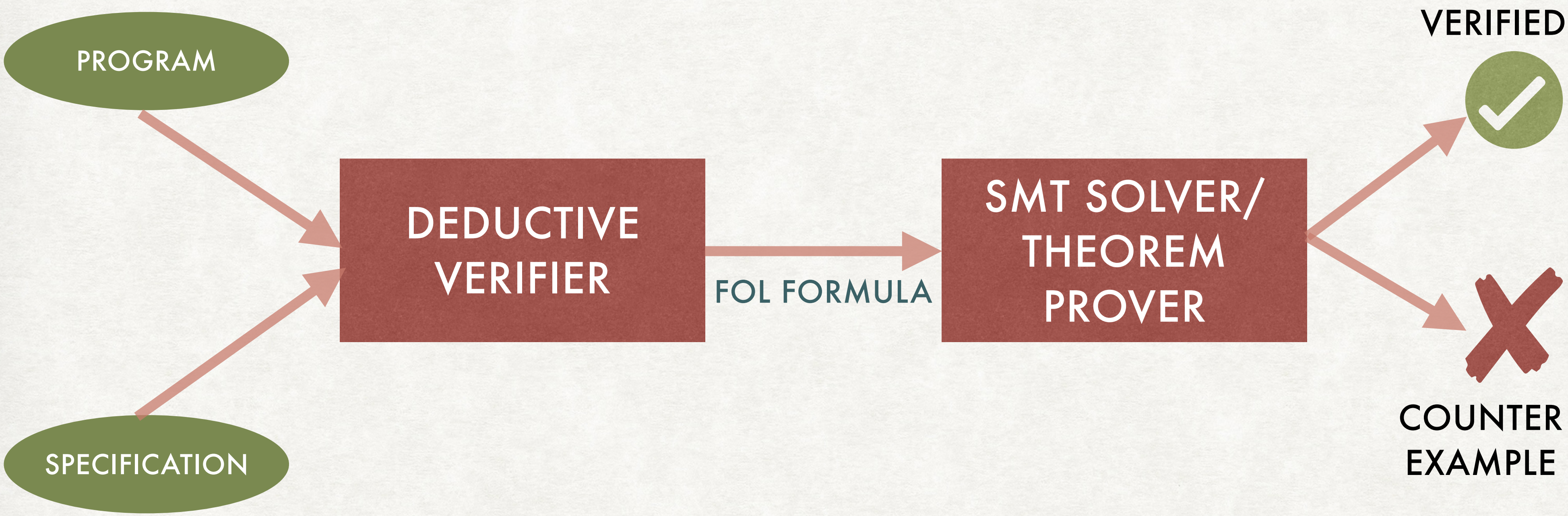
AUTOMATED VERIFICATION

OVERVIEW



AUTOMATED VERIFICATION

OVERVIEW



- Assertions
- Pre-conditions/Post-conditions
- Invariants
- …

IMP

A SMALL IMPERATIVE PROGRAMMING LANGUAGE

- Let V be a set of program variables
- Let $Exp(V)$ be the set of linear expressions, and $\Sigma(V)$ be the set of linear formulae over V
  - $Exp(V)$ are terms in Linear Real Arithmetic
  - $\Sigma(V)$ are formulae in Linear Real Arithmetic
- Examples
  - $3x + 2y \in Exp(\{x, y\})$
  - $x \leq y + z \wedge z = w \in \Sigma(\{x, y, z, w\})$

IMP

A SMALL IMPERATIVE PROGRAMMING LANGUAGE

$\mathtt{exp} \in \mathit{Exp}(V)$

$\mathrm{F} \in \Sigma(V)$

```
c = x := exp | x := havoc
  = | assume(F) | assert(F)
  = | skip | c; c | if(F) then c else c | while(F) do c
```

Assigns a random value

EXAMPLES

```
assume(i = 0 ∧ n ≥ 0);
while(i < n) do
  i := i + 1;
assert(i = n);
```

PRE-CONDITION

POST-CONDITION

EXAMPLES

```
assume(i = 0 ∧ n ≥ 0);
while(i < n) do
  i := i + 1;
assert(i = n);
```

FOL formula in LRA whose
free variables are program variables

EXAMPLES

```
{i = 0 ∧ n ≥ 0}
while(i < n) do
  i := i + 1;
{i = n}
```

EXAMPLES

```
{i = 0 ∧ n ≥ 0}
while(i < n) do
  i := i + 1;
{i = n}
```

```
{Pre-condition}
Program
{Post-condition}
```

EXAMPLES

Linear Search

Input: Array a, Lower limit l, Upper limit u, Element to be searched e

Output: true if element is present, false otherwise

```
i := l;
present := false;
while(i <= u && !present)
{
  if (a[i] == e) then
    present := true;
  else
    i := i + 1;
}
```

EXAMPLES

Linear Search

Input: Array a, Lower limit l, Upper limit u, Element to be searched e

Output: true if element is present, false otherwise

```
assume(?);
i := l;
present := false;
while(i <= u && !present)
{
  if (a[i] == e) then
    present := true;
  else
    i := i + 1;
}
assert(?);
```

EXAMPLES

Linear Search

Input: Array a, Lower limit l, Upper limit u, Element to be searched e

Output: true if element is present, false otherwise

```
assume(l ≥ 0 ∧ u ≤ |a|);
i := l;
present := false;
while(i <= u && !present)
{
  if (a[i] == e) then
    present := true;
  else
    i := i + 1;
}
assert(?);
```

EXAMPLES

Linear Search

Input: Array a, Lower limit l, Upper limit u, Element to be searched e

Output: true if element is present, false otherwise

```
assume(l ≥ 0 ∧ u ≤ |a|);
i := l;
present := false;
while(i <= u && !present)
{
  if (a[i] == e) then
    present := true;
  else
    i := i + 1;
}
assert(present ↔ l ≤ i ≤ u ∧ a[i] = e);
```

EXAMPLES

Linear Search

Input: Array a, Lower limit l, Upper limit u, Element to be searched e

Output: true if element is present, false otherwise

```
assume(l ≥ 0 ∧ u ≤ |a|);
i := l;
present := false;
while(i <= u && !present)
{
  if (a[i] == e) then
    present := true;
  else
    i := i + 1;
}
assert(present ↔ ∃x. l ≤ x ≤ u ∧ a[x] = e);
```