Formal Verification

Announcements

- Lab6 due last night
- No class 4/21 and 4/30
- Lab7 today - due 3/30

Overview

1. DPLL^T Review

2. Simplex Algorithm
   a. Solving theories of integers

3. Specifications
   a. Lab today!

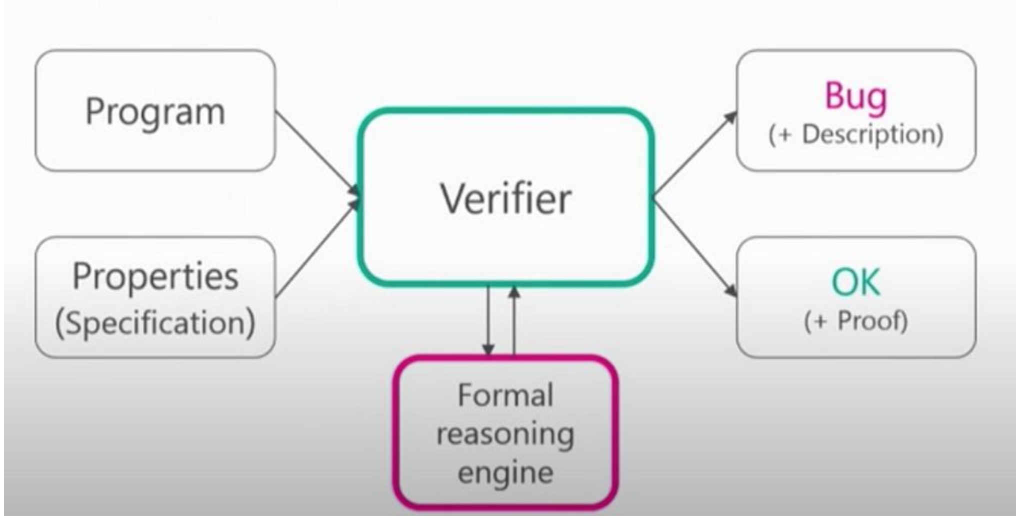
Formal Verification

```
public int addPositive(int x, int y) {

    return x + y;

}
```

Precondition:x >= 0 && y >= 0;

Assertion:

(addPos(x, y) == x + y)



x >= 0 && y >= 0 && ! (addPos(x, y) == x + y)

How does the “formal verification engine” work?

1. SMT solving
   a. Given a specification, translate it to a SMT formula and ask a solver if it’s SAT or UNSAT.
   b. DPLL

2. Deductive reasoning

DPLL Algorithm

Given a boolean formula in CNF form, DPLL decides UNSAT/SAT and gives an interpretation if its SAT

Alternates between two phases:

1. Deduction
   a. Tries to simplify the formula using the laws of logic
2. Search
   a. Searches for an interpretation

DPLL - Deduction Phase

Boolean Constant Propagation

$$(\ell) \wedge C_2 \wedge \cdots C_n$$

Suppose the first clause consists of a single literal (We call this a *unit clause*).

Any interpretation of the formula must assign ℓ to TRUE.

Simplify the formula by substituting TRUE for all ℓs

DPLL

Algorithm 1: DPLL

Data: A formula F in CNF form
Result: $I \models F$ or UNSAT

▷ Boolean constant propagation (BCP)
while *there is a unit clause* (ℓ) *in* F **do**
    Let F be $F[\ell \mapsto \mathsf{true}]$

if F *is* true **then return** SAT

▷ Search
for *every variable* p *in* F **do**
    If DPLL$(F[p \mapsto \mathsf{true}])$ is SAT **then return** SAT
    If DPLL$(F[p \mapsto \mathsf{false}])$ is SAT **then return** SAT
return UNSAT
▷ The model I that is returned by DPLL when the input is SAT is maintained implicitly in the sequence of assignments to variables (of the form $[l \mapsto \cdot]$ and $[p \mapsto \cdot]$)

DPLL Modulo Theories

Boolean Abstraction

We can turn a formula with *linear real arithmetic* into a boolean formula using **boolean abstraction**

$F \triangleq (x \leqslant 0 \lor x \leqslant 10) \land (\neg x \leqslant 0)$ $\Rightarrow$ $F^B \triangleq (p \lor q) \land (\neg p)$

Every unique linear inequality is replaced with a boolean variable

The boolean abstraction of F is denoted F^B

We use superscript T to map boolean formulae back to their theory formulae: $(F^B)^T = F$

Boolean Abstraction

If F^B is UNSAT, then F is UNSAT.

Example: $$F \triangleq (x \leqslant 0 \wedge (\neg x \leqslant 0)$$

$$F^B \triangleq (p) \wedge (\neg p)$$

Boolean Abstraction

If F^B is SAT, then F is not necessarily SAT!

$$F \triangleq x \leqslant 0 \land x \geqslant 10.$$

$F^B = \quad p \land q$

During abstraction, relations between the inequalities are lost. X cannot be <= 0 and >= 10, but p and q have no relation!

SMT Solvers

To solve SMT, we can slightly modify DPLL to $DPLL^T$

- Start by treating the formula as if its completely boolean, then incrementally add more and more *theory* information until we can conclusively say it is SAT or UNSAT
- Requires a boolean abstraction of F
- Requires access to a *theory solver*

DPLL^T

First checks if F^B is UNSAT. Since we know that F will also be UNSAT

Algorithm 2: DPLL^T

Data: A formula F in CNF form over theory T
Result: $I \models F$ or UNSAT

Let F^B be the abstraction of F
while true **do**
- **If** DPLL(F^B) is UNSAT **then return** UNSAT
- Let I be the model returned by DPLL(F^B)
- Assume I is represented as a formula
- **if** I^T *is satisfiable (using a theory solver)* **then**
  - **return** SAT and the model returned by theory solver
- **else**
  - Let F^B be $F^B \wedge \neg I$

Queries a theory solver on the *concrete* output of DPLL. (I is an interpretation)

If SAT, we're done!
IF UNSAT, doesn't necessarily mean it's UNSAT... negate and continue

DPLL^T Example

F: $x \geqslant 10 \wedge (x < 0 \vee y \geqslant 0)$

What is F^B? $p \wedge (q \vee r)$

1. **First iteration of DPLL on F^B**

Returns SAT: { p -> T, q -> T }
I = p ∧ q

2. **Query Theory Solver on I^T**

What is **I**^{T}? $\underbrace{x \geqslant 10}_{p} \wedge \underbrace{x < 0}_{q}$

Theory solver returns **UNSAT**

3. **Rerun DPLL on** F^B ∧ ! I

What is F^B ∧ ! I ? $p \wedge (q \vee r) \wedge \underbrace{(\neg p \vee \neg q)}_{\neg I_1}$

Returns SAT: { p -> T, q -> F, r -> T}

4. **Query Theory Solver on I^T**

What is **I**^{T}?

I^T = (x>= 10) ∧ (x>=0) ∧ (y >= 0)

Theory solver returns { x->10, y-> 0 }

Simplex Algorithm

Simplex Algorithm

- Developed in 1947
- Finds a satisfying assignment that maximizes some objective function
- We are using it to find any satisfying assignment

Simultaneously looks for a model and proof of unsatisfiability.

Starts with some interpretation and continues to update it every iteration

Simplex Form

$$\sum_i c_i \cdot x_i = 0$$

$$\ell_i \leq x_i \leq u_i$$

Translate to Simplex Form

$(x + y \geq 0) \land (-2x + y \geq 2) \land (-10x + y \geq -5)$

Take every inequality and translate it into two clauses: an equality and a bound

$(x + y \geq 0) \Rightarrow \begin{cases} s_1 = x + y \\ s_1 \geq 0 \end{cases}$

$(-10x + y \geq -5) \Rightarrow \begin{cases} s_3 = -10x + y \\ s_3 \geq -5 \end{cases}$

$(-2x + y \geq 2) \Rightarrow \begin{cases} s_2 = -2x + y \\ s_2 \geq 2 \end{cases}$

Simplex Algorithm

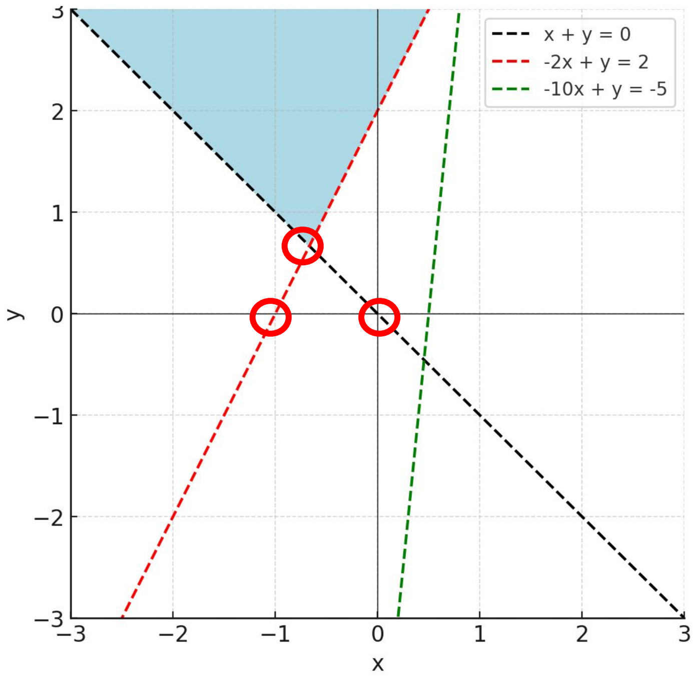
Simultaneously looks for a model and proof of unsatisfiability.

Starts with some interpretation and continues to update it every iteration

Pick a bound ($s_1 \geq 0$) that is not satisfied and modify the interpretation to satisfy it

Simplex Algorithm

$(x + y \geq 0) \wedge (-2x + y \geq 2) \wedge (-10x + y \geq -5)$



Init: $I_0 = \{x \rightarrow 0, y \rightarrow 0\}$

$I_1 = \{x \rightarrow -1, y \rightarrow 0\}$

Basic and Non-Basic Variables

Basic Variables: appear on the left side of an equality. Initially, these are the slack variables (s1, s2, s3)

Non-basic Variables: all other variables

$s_1 = x + y$

$s_1 \geq 0$

$s_2 = -2x + y$

$s_2 \geq 2$

$s_3 = -10x + y$

$s_3 \geq -5$

Algorithm 3: Simplex

Data: A formula F in Simplex form
Result: $I \models F$ or UNSAT

Let I be the interpretation that sets all variables $fv(F)$ to 0
while *true* **do**
  if $I \models F$ **then return** I
  Let x_i be the first basic variable s.t. $I(x_i) < l_i$ or $I(x_i) > u_i$
  if $I(x_i) < l_i$ **then**
    Let x_j be the first non-basic variable s.t.

$$(I(x_j) < u_j \text{ and } c_{ij} > 0) \text{ or } (I(x_j) > l_j \text{ and } c_{ij} < 0)$$

    if *If no such x_j exists* **then return** UNSAT
    $I(x_j) \leftarrow I(x_j) + \frac{l_i - I(x_i)}{c_{ij}}$
  else
    Let x_j be the first non-basic variable s.t.

$$(I(x_j) > l_j \text{ and } c_{ij} > 0) \text{ or } (I(x_j) < u_j \text{ and } c_{ij} < 0)$$

    if *If no such x_j exists* **then return** UNSAT
    $I(x_j) \leftarrow I(x_j) + \frac{u_i - I(x_i)}{c_{ij}}$
  Pivot x_i and x_j

Formal Verification

Bringing it all together...

How do we use SMT to prove that our program functions as expected?

Approaches to Formal Verification

1. Both models and the programs are encoded as a proof
2. Tools take as input a program in a particular language with an **annotation language.** Automatically produces a SMT formula which is fed to a verifier

Specifications

What Are Specifications?

Specifications describe what a program should do

A **Precondition** is a condition that must be true before method execution, and a **Postcondition** is a condition that must be true after method execution.

A **Specification** is a contract between a method and its caller, consisting of a precondition and a postcondition.

JML

JML = Java Modeling Language

Used to specify contracts for Java methods and classes

Annotations look like comments but express **preconditions**, **postconditions**, and **invariants**

Basic Syntax

//@ requires <precondition>;

//@ ensures <postcondition>;

equires: What must be true before method executes

ensures: What will be true after method completes, if preconditions were met

The `\result` Keyword

//@ ensures \result == x + y;

`\result` refers to the value returned by the method

Only used in `ensures` clauses

Example

```
//@ requires x >= 0 && y >= 0;
//@ ensures \result == x + y;
public int addPositive(int x, int y) {
    return x + y;
}
```

Using Logical Operators

You can use && and || in your pre and post conditions

//@ requires x > 0 && y > 0;

//@ ensures \result == x * y || \result == 0;

Example 2

```
//@ requires n >= 0;
//@ ensures (\result == 1 && n == 0) || \result == n * factorial(n - 1);
public int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

Conditional Logic

JML supports conditional expressions using `==>` **(implies)** and `?:` **(ternary-style)**

//@ ensures x > 0 ==> \result > 0;

“If `x > 0` is true, then `\result > 0` must be true.”

//@ ensures (x >= 0) ? \result == x : \result == -x;

“If `x >= 0`, then result is `x`, otherwise it's `-x`.”

Example:

```
//@ ensures (x >= 0) ==> \result == x;

//@ ensures (x < 0) ==> \result == -x;

public int absoluteValue(int x) {

    return (x >= 0) ? x : -x;

}
```

OpenJML

OpenJML translates JML specs into boolean constraints and passes them to an SMT solver

OpenJML

```
public class MathUtils {
    //@ requires x >= 0;
    //@ ensures \result == x + 1;
    public static int increment(int x) {
        return x + 1;
    }
}
```

$(x >= 0) => (result == x + 1)$

$(x >= 0) => ((x + 1) == x + 1)$

Feed the negation to Z3

$(x \geq 0) \land ((x + 1) \neq x + 1)$

UNSAT

Approaches to Formal Verification

1. Program is part of the proof
    a. Offer a higher level of assurance
    b. Annotations often end up repeating much of the program
2. Annotations on normal languages
    a. Easier to adopt

Summary

- DPLLT
  - Sometimes our constraints contain non-boolean variables
- Simplex
  - Solves linear arithmetic constraints
- Formal Verification
  - Annotation based
  - Lab today
- Next class:
  - Formal verification as deductive reasoning with programs as proofs