

# Lecture 10: Presburger arithmetic

David Dill

Department of Computer Science

# Outline

---

- A little more on Rice's theorem
- Presburger arithmetic

## Statement of Rice's theorem

**Thm:** Every non-trivial property  $P$  of the R.E. languages is undecidable.

(a). *Property:* A set of languages (in this case, a set of Turing Machines).

(b). *Trivial:* Either every language has it or no language has it.

## WRONG Applications of Rice's theorem

Rice's theorem does not apply to these

- Whether a TM has less than 7 states (not a language property).
- Whether a TM has a final state (not a language property).
- Whether a TM has a start state (not a language property).
- Whether the language is RE (trivial – all RE languages).
- Whether the language is  $L_d$  (trivial – no RE languages).

## Logical decision procedures

Being able to solve logic problems automatically would be extremely valuable to computer science (and many other fields).

Lots of problems can be reduced to logic problems, so general-purpose logic solvers can be extremely helpful for solving hard problems.

Example: Program verification: Is a program correct?

Unfortunately, we'll say later that many such problems are *provably* not decidable using computers.

Example: Validity of first-order logic formulas:

$$\forall x, y \exists z: (\neg P(x, y) \vee (P(x, z) \wedge P(z, y)))$$

Example: Non-linear arithmetic over the integers.

$$\forall n: n > 2 \Rightarrow \neg \exists x, y, z: (x^n + y^n = z^n)$$

There is an inherent conflict between *expressive power*, which we want so we can encode more problems, and *computational complexity* (including decidability), which is helpful if we want a computer to solve it.

# Presburger arithmetic

Presburger arithmetic is the quantified theory of linear inequalities over the integers.

It is one of the most expressive fragments of arithmetic that is actually decidable (although the *proven* lower bound is  $O(2^{2^n})$ ).

A *term* is an expression representing an integer

- An integer constant  $0, 1, -20$ , etc.
- An integer variable  $X, Y, Z$ , etc.
- Linear combinations of terms: e.g.,  $3 * X + 4 * Y + 5$ .

A *formula* is a predicate (something with a Boolean value):

- $t_1 = t_2, t_1 < t_2, t_1 \leq t_2, t_1 \geq t_2, t_1 > t_2$ , where  $t_1, t_2$  are terms.
- propositional connectives:  $\neg P, P \wedge Q, P \vee Q, P \Rightarrow Q, P \Leftrightarrow Q$  where  $P, Q$  are predicates
- Quantifiers:  $\exists x : P$  and  $\forall x : P$ , where  $P$  is a predicate.

## Simplifying the logic

Since turning terms and formulas into automata is going to be hard, we want to handle as few cases as possible.

Formulas can be simplified by transformations like these:

- Write all logical connectives using  $\wedge$ ,  $\vee$ , and  $\neg$
- $X - Y = X + (-Y)$
- $C * X = X + X + \dots + X$  ( $C$  is an integer constant)
- $X \leq Y \iff \neg(Y < X)$
- $X = Y \iff \neg(X < Y) \wedge \neg(Y < X)$
- $(\dots X + Y < Z \dots) \iff \exists W: W = X + Y \wedge (\dots W < Z \dots)$
- $(\dots X + C < Z \dots) \iff \exists W: W = C \wedge (\dots X + W < Z \dots)$ .
- $(\forall X: P) \iff (\neg \exists X: \neg P)$ .

## Simplifying the logic

We can reduce the logic to these formulas: ( $C$  is a constant),  $X$ ,  $Y$ , and  $Z$  are variables, and  $P$  and  $Q$  are formulas.

- $X = C$  where  $C$  is an integer constant.
- $Y = -X$
- $Z = X + Y$
- $X < Y$
- $\neg F$ , where  $F$  is any formula
- $F \wedge G$ , where  $F$  and  $G$  are formulas.
- $\exists X : F$  where  $F$  is a formula.

# Presburger solutions as a regular language

## Logical terminology:

$X$  is a *bound variable* if it is in the scope of a quantifier. E.g.,  $\forall X : (\dots X \dots)$  in the formula.

If  $X$  is not a *bound variable*, it is a *free variable*.

**Example:**  $\exists X, Y : X = 2 * Y \wedge X = 3 * Z$

## Key ideas:

The *language of a Presburger formula* will be all of the assignments of integers to the free variables of the formula that make it true.

Numbers are represented as bit-strings in *twos-complement*.

I.e., Base 2, negative numbers have a sign bit,  $k$ -bit numbers range from  $2^{k-1}$  to  $2^{k-1} - 1$ . One way to compute unary minus is:  $-x = !x + 1$ .

# Language of a Presburger Formula

The alphabet:

$$\begin{array}{l} X = \left[ \begin{array}{c|c|c|c|c|c} 0 & 1 & 1 & 0 & 1 & 1 \\ \hline 1 & 1 & 0 & 1 & 0 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 & 1 & 1 \end{array} \right] \\ Y = \left[ \begin{array}{c|c|c|c|c|c} 1 & 1 & 0 & 1 & 0 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 & 1 & 1 \end{array} \right] \\ Z = \left[ \begin{array}{c|c|c|c|c|c} 0 & 0 & 0 & 1 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 & 1 & 1 \end{array} \right] \\ W = \left[ \begin{array}{c|c|c|c|c|c} 1 & 1 & 1 & 1 & 1 & 1 \end{array} \right] \end{array}$$

The alphabet is the “vertical slices”  $\left[ \begin{array}{c} 0 \\ 1 \\ 0 \\ 1 \end{array} \right]$ ,  $\left[ \begin{array}{c} 1 \\ 1 \\ 0 \\ 1 \end{array} \right]$ , etc.

bit-strings start with the *least-significant bit first* (but I will write examples MSB-first).

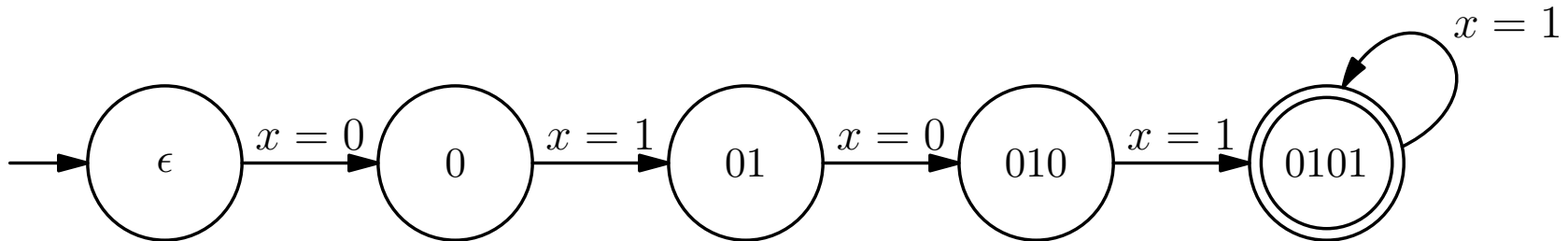
If there are  $m$  free variables, the alphabet is of size  $2^m$ .

**Note:** This forces all bit-strings to be of same length.

**Amazing fact: The language of a Presburger formula is regular!**

## Finite automata for $L(X = C)$

This checks that  $X = 1010$  ( $-6$ )

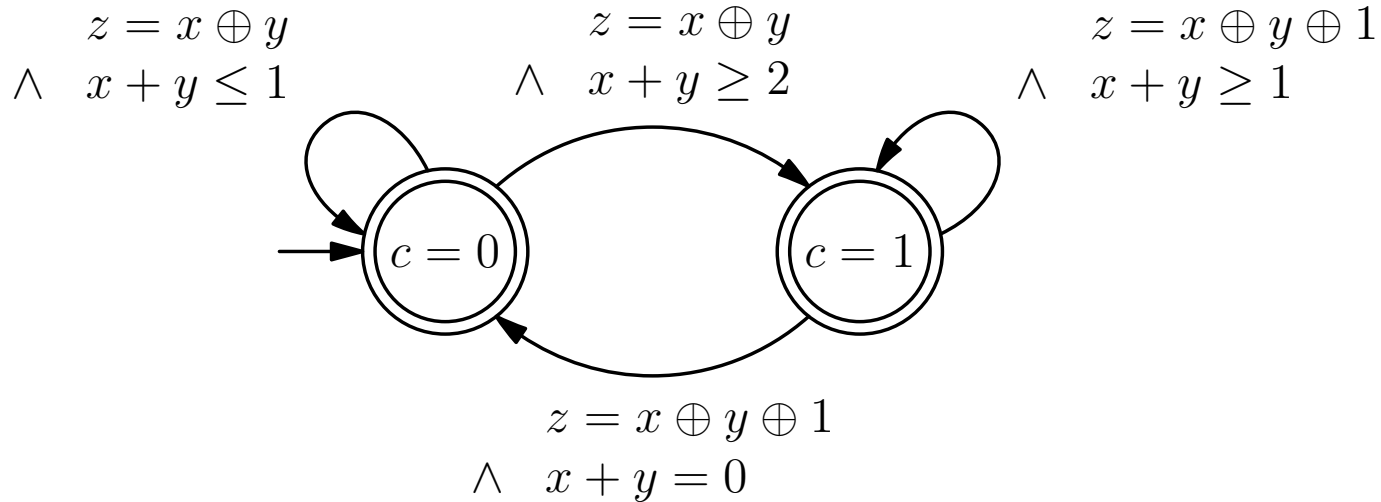


Notation:

- Lower case letter represent individual bits. E.g.,  $x$  is the current bit of the bitvector (string)  $X$ .
- For readability, I wrote logical formulas instead of lists of symbols. In this case, the alphabet is  $\{0, 1\}$ , and  $x = 1$  means the label is 1.
- The “trap” state is omitted.
- It’s a DFA if no symbol satisfies more than one formula out of a state.
- Loop at the end is because any number with all 1’s before 1010 is  $-6$  (e.g., 11111010).

# Finite automaton for $L(Z = X + Y)$

This checks that  $Z = X + Y \pmod{2^k}$ :



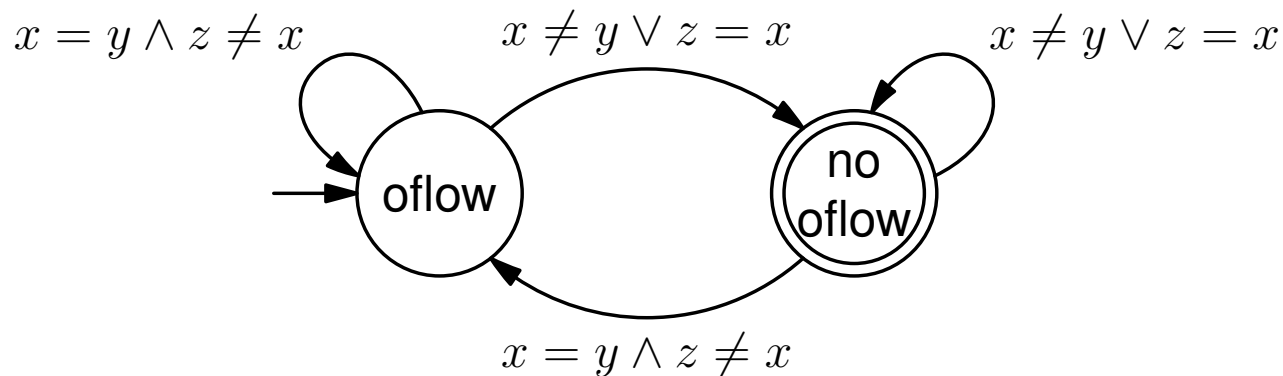
- $z = x \oplus y$  is the list of all symbols  $xyz$  satisfying the formula: 000, 101, 110, 011. In  $x + y \geq 2$ , the symbols are the  $xyz$  values 110 and 111.
- The state remembers the “carry bit” for the bits already processed. That’s all you need to remember to do addition (see: “ripple carry adder”).
- It accepts whenever the  $z$  bit is correct, taking into account the carry.
- If no transition is shown,  $z$  is wrong and it goes to trap state.
- There is nothing special about the sign bit in addition modulo  $2^k$ .

## Disallowing overflow

We need *integer* arithmetic, not  $\text{mod } 2^k$  arithmetic.

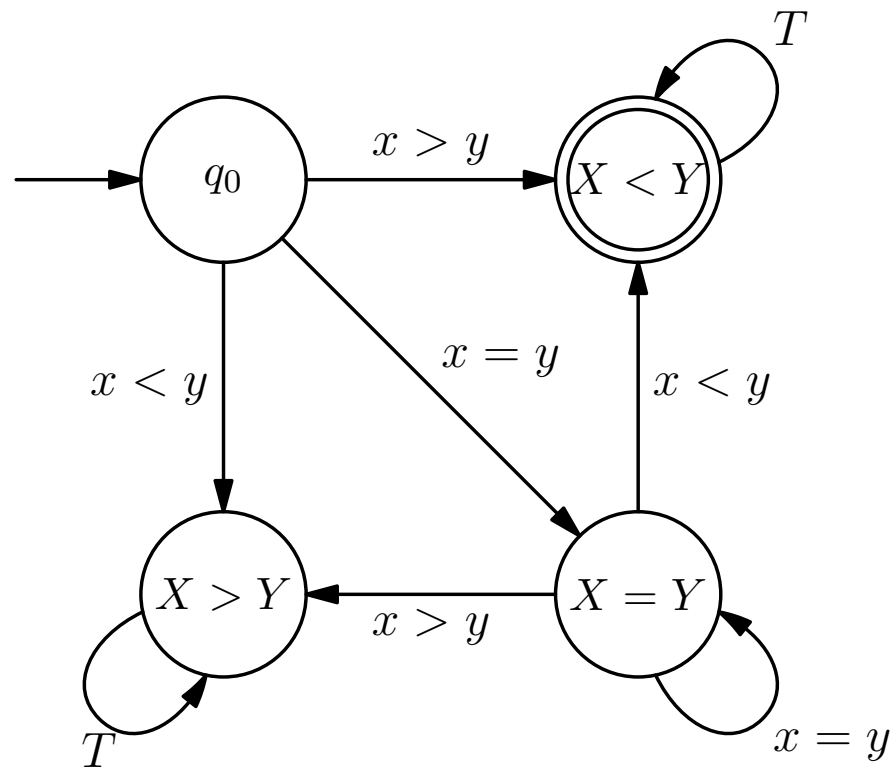
To do that, we need to make sure that all solutions have enough bits that there is no overflow.

We can do that by defining an “overflow checker” DFA. The DFA for  $Z = X + Y$  is the product of this and the previous, so only the solutions where (1)  $Z = X + Y \text{ mod } 2^k$  and (2) there is no overflow are accepted by the product automaton.



## Finite automata for $L(X < Y)$

It's easier to figure out the automaton for  $X < Y$  if we do the most-significant bit first. The LSBs only matter if the MSBs are equal. The transitions out of  $q_0$  may look weird, but that's because a sign bit of 1 means the number is less than a sign bit of 0.



Regular languages are closed under reversal, so we know that the LSB-first automaton exists, but is not as obvious.

## Connectives

Making up the automata is the hardest part. Combining them is conceptually simple (although the automata may be large):

- $L(\neg P) = \overline{L(P)}$
- $L(P \wedge Q) = L(P) \cap L(Q)$

**Detail:**  $\wedge$  is not exactly  $\cap$ . The alphabets of  $L(P)$  and  $L(Q)$  have to be reconciled: The order of variables may change, and we may have to add them. This is very simple to do but notationally tedious.

## Existential quantification

$L(\exists X : P) = h[L(P)]$  where  $h$  is the homomorphism that deletes  $X$  from every symbol:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \mapsto \begin{bmatrix} y \\ z \\ w \end{bmatrix}$$

Effect: Changes DFA to NFA.

Universal quantification:  $\forall x : P(x)$  is  $\neg \exists x : \neg P(x)$ : Compute NFA for  $\exists$ , then determinize (using subset construction) and complement.

## Decision problems

*Satisfiability*: Is it true for some assignment of integers to the free variables? Build the FA, check if the language is non-empty.

*Validity*: Is it true for *all* assignments to free variables? Check whether automaton is universal (or negate formula and test for satisfiability).