First Order Logic (FOL) [1]
http://lcs.ios.ac.cn/~znj/DM2017

Naijun Zhan

April 5, 2017

[1] Special thanks to Profs Hanpin Wang (PKU) and Lijun Zhang (ISCAS) for their courtesy of the slides on this course.

Outline

Algorithm (from wikipedia)

- An algorithm is a set of rules that precisely defines a sequence of operations, which can perform calculation, data processing and automated reasoning tasks.
- An algorithm is an effective method that can be expressed within a finite amount of space and time and in a well-defined formal language for calculating a function. Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing “output" and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input.
- History
  - The concept of algorithm has existed for centuries;
  - what would become the modern algorithm began with attempts to solve the Entscheidungsproblem (the "decision problem") posed by David Hilbert in 1928.
  - Subsequent formalizations were framed as attempts to define “effective calculability" or “effective method"; those formalizations included the Gödel-Herbrand-Kleene recursive functions, Alonzo Church’s lambda calculus, Emil Post’s “Formulation 1", and Alan Turing’s Turing machines.
  - **Giving a formal definition of algorithms, corresponding to the intuitive notion, remains a challenging problem.**
- Church-Turing Thesis: **any real-world computation can be translated into an equivalent computation involving a Turing machine.**
- **An algorithm can be considered to be any sequence of operations that can be simulated by a Turing-complete system.**

Algorithm (Cont'd)

- Expressing algorithms: high-level description, implementation-level description and formal description.
- Complexity analysis: **Formal versus empirical, execution efficiency**.
- Classification
  - By implementation: recursion, logical, serial, parallel, distributed, deterministic vs nondeterministic, exact vs approximation, quantum
  - By design paradigm: brute-force or exhaustive search, divide and conquer, search and enumeration, randomized algorithms.
  - By optimization problems: linear programming, dynamic programming, integer programming, semi-definite programming, the greedy method, the heuristic method
  - By field of study
  - By complexity: space complexity and time complexity.

Pseudo-code

Numbers:

$$Num ::= d \mid dNum \qquad \text{where } d \in \{0, 1, \ldots, 9\}$$

Identifiers:

$$Id ::= aId' \qquad \text{where } a \in A$$
$$Id' ::= \lambda \mid aId' \mid NumId'$$

Numeric expressions:

$$Exp ::= Num \mid Id \mid Id[Exp] \mid \textbf{length}(Id) \mid (Exp) \mid Exp + Exp \mid Exp - Exp \mid Exp * Exp \mid Exp/Exp \mid Id(Exp, \ldots, Exp)$$

Boolean expressions:

$$BExp ::= \textbf{true} \mid \textbf{false} \mid Exp = Exp \mid (BExp) \mid Exp \leq Exp \mid Exp < Exp \mid Exp \geq Exp \mid Exp > Exp \mid \neg BExp \mid BExp \wedge BExp \mid BExp \vee BExp$$

Procedure declaration

$$Pr ::= \textbf{procedure } Id(Id, \ldots, Id)\ P \mid \textbf{procedure } Id(Id, \ldots, Id)\ P; \textbf{return } Exp$$

Programs:

$$P ::= \textbf{skip} \mid Id := Exp \mid Id[Exp] := Exp \mid P; P \mid \textbf{if } BExp \textbf{ then } P \textbf{ else } P \textbf{ fi} \mid \textbf{while } BExp \textbf{ do } P \textbf{ done}$$

Outline

Collatz Conjecture

Algorithm 1: Collatz Conjecture

```
/* The following pseudo-code in PROC terminates when the value of n
   becomes to 1.                                                    */
/* The input n is a natural number, greater than or equal to 1      */
  procedure Collatz(n)
    while n ≠ 1 do
      if even(n) then
        n := n/2
      else
        n := n × 3 + 1
      fi
    done
```

Discussion: **Does this algorithm terminate?**

Algorithm 2: Maximum in a generic array

```
/* The following pseudo-code in PROC returns the maximum value
   occurring in the provided array.                              */
  procedure max(array)
    maxvalue := array[0];
    for i := 1 to length(array) − 1 do
      if array[i] > maxvalue then
        maxvalue := array[i]
      else
        skip
      fi
    done;
    return maxvalue
```

Algorithm 3: Index of a value in an array

```
/* The following pseudo-code in PROC returns the index of the specified
   value if it occurs in the provided array, otherwise length(array) is
   returned.                                                          */
  procedure indexOf(value, array)
    index := length(array);
    for i := 0 to length(array) − 1 do
      if array[i] = value then
        index := i
      else
        skip
      fi
    done;
    return index
```

Algorithm 4: Index of a value in a sorted array

```
/* The following pseudo-code in PROC returns the index of the specified
   value if it occurs in the provided sorted array, otherwise
   length(array) is returned.                                          */
  procedure indexOfSorted(value, array)
    index := length(array);
    low := 0;
    high := length(array) − 1;
    while low < high do
      middle := (low + high)/2;
      if array[middle] < value then
        low := middle + 1
      else
        high := middle
      fi
    done;
    if array[low] = value then
      index := low
    else
      skip
    fi;
    return index
```

Algorithm 5: Swap elements in an array

```
/* The following pseudo-code in PROC swaps the elements at index i and
   j in the provided array.                                          */
  procedure swap(array, i, j)
    temp := array[i];
    array[i] := array[j];
    array[j] := temp
```

Algorithm 6: Bubble sort

```
/* The following pseudo-code in PROC sorts the provided array.       */
  procedure bubbleSort(array)
    for i := 0 to length(array) − 1 do
      for j := 0 to (length(array) − 1) − i do
        if array[j] > array[j + 1] then
          swap(array, j, j + 1)
        else
          skip
        fi
      done
    done
```

Algorithm 7: Gnome sort

```
/* The following pseudo-code in PROC sorts the provided array.      */
  procedure gnomeSort(array)
    i := 0;
    while i < length(array) do
      if i = 0 ∨ array[i − 1] ≤ array[i] then
        i := i + 1
      else
        swap(array, i, i − 1);
        i := i − 1
      fi
    done
```

Exercise

Does this algorithm terminate? please justify your judgement.

Algorithm 8: Insertion sort

```
/* The following pseudo-code in PROC sorts the provided array.      */
  procedure insertionSort(array)
    for i := 1 to length(array) − 1 do
      j := i;
      while j > 0 ∧ array[j − 1] > array[j] do
        swap(array, j, j − 1);
        j := j − 1
      done
    done
```

Algorithm 9: Change Making

```
/* The algorithm makes changes c_1 > c_2 ... > c_r for n cents.          */
  procedure procedureChange(c_1, c_2, ..., c_r)
    for i := 1 to r do
      d_i := 0;
      while n ≥ c_i do
        d_i := d_i + 1;
        n := n − c_i
      done
    done
```

Decidability

- A **decision problem** is any arbitrary yes-or-no question on an infinite set of inputs. Because of this, it is traditional to define the decision problem equivalently as: the set of possible inputs together with the set of inputs for which the problem returns yes.
- The Primality problem: is the instance x a prime number?
- The answer (solution) to any decision problem is just one bit (true or false).

- A problem Q is *decidable* iff there is an algorithm A, such that for each instance q of Q, the computation $A(q)$ stops with an answer.
- A problem Q is semi-decidable iff there is an algorithm A, such that for each instance q of Q, if q holds, then $A(q)$ stops with the positive answer; otherwise, $A(q)$ either stops with the negative answer, *or does not stop*.

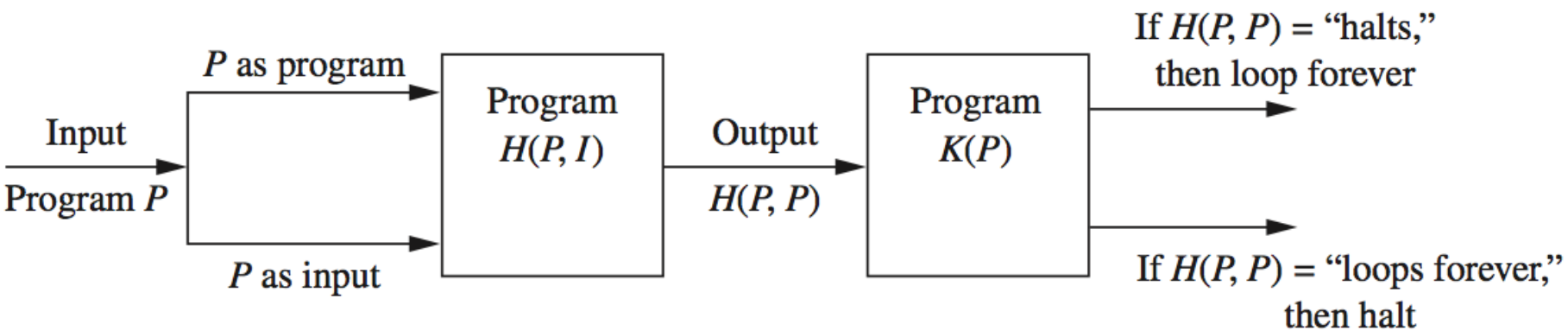
The Halting Problem

Halting problem

It takes as input a computer program and input to the program and determines whether the program will eventually stop when run with this input.

- If the program halts, we have our answer.
- If it is still running after any fixed length of time has elapsed, we do not know whether it will never halt or we just did not wait long enough for it to terminate.

Undecidability of the Halting Problem



The slides are downloadable from
http://www.computational-logic.org/iccl/master/lectures/summer07/sat/slides/dpll.pdf

Outline

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that $|f(x)| \leq C|g(x)|$ whenever $x > k$.

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$ if there are positive constants C and k such that $|f(x)| \geq C|g(x)|$ whenever $x > k$.
Moreover, We say that $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $O(g(x))$ and $\Omega(g(x))$.

Exercise

Show that:

1. Show that $log\ n!$ is $O(nlogn)$.
2. Show that n^2 is not $O(n)$.
3. Suppose that $f_1(x)$ is $O(g_1(x))$ and that $f_2(x)$ is $O(g_2(x))$. Then $(f_1 + f_2)(x)$ is $O(max(|g_1(x)|, |g_2(x)|))$.
4. Suppose that $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$. Then $(f_1 f_2)(x)$ is $O(g_1(x)g_2(x))$.
5. Show that $3x^2 + 8xlogx$ is $\Theta(x^2)$.
6. Assume $a_n \neq 0$. Show that $\sum_{i=0}^{n} a_i x^i$ is $\Theta(x^n)$.

Outline

Space complexity vs time complexity

We are interested in the *time complexity* of the algorithm.

TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

P vs NP

- A problem that is solvable using an algorithm with polynomial worst-case complexity is called tractable. Tractable problems are said to belong to class P.
- Problems for which a solution can be checked in polynomial time are said to belong to the class NP.
- The $P = NP$ problem asks whether NP, the class of problems for which it is possible to check solutions in polynomial time, equals P, the class of tractable problems.
- NP-complete problems: It is an NP problem and if a polynomial time algorithm for solving it were known, then $P = NP$.
- The satisfiability problem is NP-complete. Cook-Levin Theorem

A prize of $1,000,000$ dollars is offered by the Clay Mathematics Institute for its solution.

Outline

Hilbert Program

NOTE: The following material is from Moshe Vardi
Hilbert Program (1922-1930): Formalize mathematics and establish that:

- Mathematics is consistent:
  a mathematical statement and its negation cannot ever both be proved.
- Mathematics is complete:
  all true mathematical statements can be proved.
- Mathematics is decidable: there is a mechanical way to determine whether a given mathematical statement is true or false.

The Demise of Hilbert Program

Gödel:

- Incompleteness of ordinary arithmetic - There is no systematic way of resolving all mathematical questions.
- Impossibility of proving consistency of mathematics
  Gödel (1930): "This sentence is not provable."
- Church and Turing (1936): Unsolvability of first-order logic:
  The set of valid first-order sentences is not computable.

Entscheidungsproblem

Entscheidungsproblem (The Decision Problem) [Hilbert-Ackermann, 1928]: Decide if a given first-order sentence is valid (dually, satisfiable).
Church-Turing Theorem, 1936: The Decision Problem is unsolvable.
Turing, 1936:

- Defined computability in terms of Turing machines (TMs)
- Proved that the termination problem for TMs is unsolvable ("this machine terminates iff it does not terminate")
- Reduced termination to Entscheidungsproblem.

Mathematical Logic - 1936

Logic as Foundations of Mathematics:

- Incomplete (example: Continuum Hypothesis)
- Cannot prove its own consistency
- Unsolvable decision problem
- Unsolvable termination problem
  Can we get some positive results?
- Focus on special cases!

The Fragment-Classification Project

Idea: Identify decidable fragments of first-order logic - (1915-1983)

- Monadic Class (monadic predicates)
- Bernays-Schönfinkel Class ($\exists^*\forall^*$)
- Ackermann Class ($\exists^*\forall\exists^*$)
- Gödel Class ($\exists^*\forall\forall\exists^*$)

Outcome: Very weak classes! What good is first-order logic?

Monadic Logic

Monadic Class: First-order logic with monadic predicates - captures syllogisms.
$\forall x(Man(x) \rightarrow Mortal(x))$
Löwenheim, 1915: The Monadic Class is decidable.

- Proof: Bounded-model property - if a sentence is satisfiable, it is satisfiable in a structure of bounded size.
- Proof technique: quantifier elimination.

Logic of Integer Addition

Integer Addition:

- Domain: N (natural numbers)
- Predicate: $=$
- Addition function: $+$

- $y = 2x : y = x + x$
- $x \leq y : (\exists z)(y = x + z)$
- $x = 0 : (\forall y)(x \leq y)$
- $x = 1 : x \neq 0 \wedge (\forall y)(y = 0 \vee x \leq y)$
- $y = x + 1 : (\exists z)(z = 1 \wedge y = x + z)$

Bottom Line: Theory of Integer Addition can express Integer Programming (integer inequalities) and much more.

Presburger Arithmetics

Mojzesz Presburger, 1929:

- Sound and complete axiomatization of integer addition
- Decidability: There exists an algorithm that decides whether a given first-order sentence in integer-addition theory is true or false.
  - Decidability is shown using quantifier elimination, supplemented by reasoning about arithmetical congruences.
  - Decidability can also be shown using automata-theoretic techniques.

Complexity of Presburger Arithmetics

Complexity Bounds:

- Oppen, 1978: TIME($2^{2^{2^{poly}}}$) upper bound
- Fischer & Rabin, 1974: TIME($2^{2^{lin}}$) lower bound

Rabin, 1974: "Theoretical Impediments to Artificial Intelligence": "the complexity results point to a need for a careful examination of the goals and methods in AI".

Finite Words - Nondeterministic Finite Automata

Input word: $a_0, a_1, ..., a_{n-1}$
Run: $s_0, s_1, ..., s_n$

- $s_0 \in S_0$
- $s_{i+1} \in \rho(s_i, a_i)$ for $i \geq 0$

Acceptance: $s_n \in F$
Recognition: $L(A)$ - words accepted by A.
Fact: NFAs define the class Reg of regular languages.

Logic of Finite Words

View finite word $w = a_0, ..., a_{n-1}$ over alphabet Σ as a mathematical structure:

- Domain: $0, ..., n-1$
- Dyadic predicate: $\leq$
- Monadic predicates: $\{P_a : a \in \Sigma\}$

Monadic Second-Order Logic (MSO):

- Monadic atomic formulas: $P_a(x)$ $(a \in \Sigma)$
- Dyadic atomic formulas: $x < y$
- Set quantifiers: $\exists P, \forall P$
  Example: $(\exists x)((\forall y)(\neg(x < y)) \wedge P_a(x))$ - last letter is a.

[Büchi, Elgot, Trakhtenbrot, 1957-8 (independently)]: MSO $\equiv$ NFA
Both MSO and NFA define the class Reg.
Proof: Effective

- From NFA to MSO ($A \to \varphi_A$)
- From MSO to NFA ($\varphi \to A_\varphi$)

NFA Nonemptiness

Nonemptiness: $L(A) = \emptyset$
Nonemptiness Problem: Decide if given A is nonempty.
Directed Graph $G_A = (S, E)$ of NFA $A = (\Sigma, S, S_0, \rho, F)$:

- Nodes: S
- Edges: $E = \{(s, t) : t \in \rho(s, a) \text{ for some } a \in \Sigma\}$

It holds: A is nonempty iff there is a path in G_A from S_0 to F.
Decidable in time linear in size of A, using breadth-first search or depth-first search.

MSO Satisfiability - Finite Words

Satisfiability: $\mathit{models}(\psi) = \emptyset$
Satisfiability Problem: Decide if given ψ is satisfiable.
It holds: ψ is satisfiable iff A_ψ is nonnempty.
It holds: MSO satisfiability is decidable.

- Translate ψ to A_ψ.
- Check nonemptiness of A_ψ .

Complexity Barrier

Computational Complexity:

- Naive Upper Bound: Nonelementary Growth 2 to the power of the tower of height $O(n)$
- Lower Bound [Stockmeyer, 1974]: Satisfiability of FO over finite words is nonelementary (no bounded-height tower).

Program Verification

- The Dream - Hoare, 1969: "When the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program."
- The Nightmare - De Millo, Lipton, and Perlis, 1979: "We believe that . . . program verification is bound to fail. We cannot see how it is going to be able to affect anyone's confidence about programs."
- "... software verification ... has been the Holy Grail of computer science for many decades but not in some very key areas, for example, driver verification we are building tools that can do actual proof about the software and how it works in order to guarantee the reliability." (Bill Gates, keynote address at Winhec 2002)
- Hoare has pose a grand challenge: "The verification challenge is to achieve a significant body of verified programs that have precise external specifications, complete internal specifications, machine-checked proofs of correctness with respect to a sound theory of programming."
- NICTA seL4 using Isabelle/HOL, INRIA CompCert using Coq

Logic in Computer Science: c. 1980

Status: Logic in CS is not too useful!

- First-order logic is undecidable.
- The decidable fragments are either too weak or too intractable.
- Even Boolean logic is intractable.
- Program verification is hopeless.

Post 1980: From Irrelevance to Relevance

A Logical Revolution:

- Relational databases
- Boolean reasoning
- Model checking
- Termination checking
- ...

The Temporal Logic of Programs

Crux: Need to specify ongoing behavior rather than input/output relation! "Temporal logic to the rescue" [Pnueli, 1977]:

- Linear temporal logic (LTL) as a logic for the specification of non-terminating programs
- Model checking via reduction to MSO
  But: nonelementary complexity!

In 1996, Pnueli received the Turing Award for seminal work introducing temporal logic into computing science and for outstanding contributions to program and systems verification.

Examples

- always not (CS1 and CS2): safety
- always (Request implies eventually Grant): liveness
- always (Request implies (Request until Grant)): liveness

Model Checking

"Algorithmic verification" [Clarke & Emerson, 1981, Queille & Sifakis, 1982]: Model checking programs of size m wrt CTL formulas of size n can be done in time mn.
Linear-Time Response [Lichtenstein & Pnueli, 1985]: Model checking programs of size m wrt LTL formulas of size n can be done in time $m2^{O(n)}$ (tableau heuristics).
Seemingly:

- Automata: non-elementary
- Tableaux: exponential

Back to Automata

Exponential-Compilation Theorem [Vardi & Wolper, 1983-1986]: Given an LTL formula φ of size n, one can construct an automaton A_φ of size $2^{O(n)}$ such that a trace σ satisfies φ if and only if σ is accepted by A_φ. Automata-Theoretic Algorithms:

- LTL Satisfiability: φ is satisfiable iff $L(A_\varphi) = \emptyset$ (PSPACE)
- LTL Model Checking: $M \models \varphi$ iff $L(M \times A_{\neg\varphi}) = \emptyset$ ($m2^{O(n)}$)

Today: Widespread industrial usage
Industrial Languages: *PSL*, *SVA* (IEEE standards)

Solving the Unsolvable

B. Cook, A. Podelski, and A. Rybalchenko, 2011:"in contrast to popular belief, proving termination is not always impossible"

- The Terminator tool can prove termination or divergence of many Microsoft programs.
- Tool is not guaranteed to terminate! Explanation:
- Most real-life programs, if they terminate, do so for rather simple reasons.
- Programmers almost never conceive of very deep and sophisticated reasons for termination.

Logic: from failure to success

Key Lessons:

- Algorithms
- Heuristics
- Experimentation
- Tools and systems

Key Insight: Do not be scared of worst-case complexity.

- It barks, but it does not necessarily bite!