Complexity Theory and Logic

Lecture Notes from CSE 532

Richard Ladner
Department of Computer Science and Engineering
University of Washington
Seattle, Washington, U.S.A. 98195

Winter 1998

Contents

Preface

These are the lecture notes from CSE 532, a graduate algorithms course I taught at the University of Washington in Winter 1998. The course covered some of the foundation of computability, computational complexity and logic. These notes are not intended to be a survey of that area, but a course showing the connections between these fields. Some results are not found in standard texts in computational complexity or logic. For these we have provided references.

I am grateful to the students who attended faithfully and took the notes that make up this series of lectures.

Lecture 1

History of Complexity and Logic

January 6, 1998
Lecturer: Richard Ladner
Notes: Vass Litvinov

1.1. Administrativia

"Complexity and Logic"
Richard Ladner
ladner@cs, Sieg 311, 543-9347
Office hours: whenever the door is open/by appointment

Text: *Computational Complexity* by Papadimitriou [12] (required)
Read Chapters 1, 2, 3, 7

Grades based on:

- problem sets
- class notes
- oral presentation (Finals week). Options:
  - critical analysis of research papers
  - critical analysis of a working verification system
- class participation

Topics

- Logic
  - Propositional Logic
  - First Order Logic
  - Temporal Logic (has to do with programs)
  - Logic of Knowledge (has to do with distributed programs)
- Complexity

- Complexity Classes
- Alternating Computation
- Interactive Proof Systems
- Undecidability of Logics
- Complexity of Logics

1.2. Connections Between Complexity and Logics

1900s Hilbert proposed that mathematics have a *logical foundation*, a formalism behind it. He believed that within the formalism every true statement could be "determined." So called *"Hilbert's Plan."*

1910s Russell and **Whitehead** published "Principia Mathematica," the first attempt to determine the truth of all mathematical formulas.

1920s Presburger proved that every true statement in arithmetic with addition only could be determined.

1930s Gödel showed that the class of consequences of a set of axioms for arithmetic with + and * was undecidable.

Turing and **Church** and others proposed formal models of computation (*Turing Machine, Lambda Calculus, ...*). They all turned to be equivalent. *Universal Turing Machine* was conceived. Halting problem also turned to be undecidable. *Turing and Church Thesis* became accepted.

1940s Post defined formally the notion of *reducibility* between problems.

Many problems were shown to be undecidable using reducibility arguments.

Mathematical logic flourished.

Working computers were born in wake of WWII.

1950s Rabin and **Scott** built the theory of *finite automata* (nondeterministic). Most problems about them were shown decidable.

This marked the birth of theoretical computer science.

1960s Hartmanis and **Stearns** invented modern *computational complexity* based on the Turing Machine. Time and space complexity became standards.

Hoare and **Floyd** proposed *formal verification* of programs based on logical reasoning. (Like the "Hilbert's Plan.")

1970s Cook (and Levin?) invented *NP-completeness.* SAT was shown to be NP-complete. **Karp** and **Garey** and **Johnson** helped make NP-completeness proofs an industry. Almost all proofs were reductions.

Complexity theorists ignored program verification with some good reason. **Meyer** and **Stockmeyer** and others showed exponential and worse lower bounds on logical theories.

Stockmeyer and **Kozen** invented *Alternating Turing Machines.*

Second half of 1970s: formal verification was taken seriously. **Pnueli** applied temporal logic to program verification. **Pratt** invented *dynamic logic*, the right way to look at Hoare's logic.

1980s Complexity theory became a standard part of Computer Science undergraduate curriculum.

Difficult proofs for weak lower bounds became popular.

Goldwasser and **Mihali** and **Rackoff** invented interactive proof systems as a basis for cryptography. Randomizization became popular.

Parallel complelxity became a rage, technology-driven.

Some theorists continued to work on logic in spite of being ignored by complexity theorists.

1990s Trends from 1980s continue, with some changes.

Approximation algorithms and lower bounds for NP-complete problems became popular. Complexity theory is full blown, but the results became more diffuse, more technical; no breakthroughs in the grand approach.

Practical verification systems begin to emerge. VIS, FormalCheck, SPIN, SMV, others. Mostly verification of hardware; complexity theory is ignored, but some of the systems adopted ideas from the logic area.

Other Connections between Logic and Computer Science

- LISP - functional programming evolved from λ-calculus
- Type theory - meta-logic of programs
- Prolog - logic programming
- Theorem proving, symbolic computation
- Hardware design
- Relational databases
- Semantics of programming languages
- AI

Lecture 2

Review of Computability and Complexity

January 8, 1998
Lecturer: Richard Ladner
Notes: Jayram S. Thathachar

In this lecture we will review the standard literature on computability and complexity. The notation used is almost consistent with Papadimitriou's book [12]. Another great resource is the classic book by Hopcroft and Ullman [9].

2.1. Turing Machines

A (deterministic) *Turing machine* T is a computing device with a finite instruction set and a head reading a one-way infinite tape (extending to the right) whose squares are filled with symbols from some appropriate alphabet. The tape contains a finite number of non-blank symbols appearing in a finite segment of the left portion of the tape. The machine operates in steps. In each step, the head reads the symbol from the square it is positioned on, possibly changes it, and then possibly moves right or left, all of which is done according to the finite instruction set. Formally, it can be described as a a 5-tuple $(K, \Sigma, \Gamma, \delta, s)$, where

- K is the set of states,
- Σ is the input alphabet,
- $\Gamma \supseteq \Sigma$ is the tape alphabet which includes the end-marker symbol (present in the leftmost square of the tape) and the blank symbol,
- s is the start state, and
- $\delta : K \times \Gamma \to (K \cup \{Y, N, H\}) \times \Gamma \times D$ is the transition function describing the operation of the machine in one step. Here $D = \{L, R, S\}$ denotes the direction moved, and $\{Y, N, H\}$ denotes the set of halting states (Y is accept, N is reject and H is halt without accept or reject).

We will adopt certain conventions about the operation of the Turing machine that make our descriptions easier, e.g. the Turing machine is not allowed to move off the left end of the tape.

A *configuration* of the Turing machine at any instant is a description of the contents of the tape, head position and state at that time. Formally it is given by the tuple (q, u, v), where q is the state, and u and v denote the contents of the tape to the left and right of the head. We will make v finite by stopping at the square after which there are only blank symbols on the tape. The notation $C \xrightarrow{T} D$ describes the move from configuration C to D in one step according to δ, and $C \xrightarrow{T^m} D$ describes a move of m steps.

The language accepted by T is the set of input strings in Σ^* which causes T to end up in the accept state Y. In other words, $x \in L(T)$ if and only if $\text{Init}(x) \xrightarrow{T^*}$ Accepting configuration. The (partial) function computed by T on any string is given by the tape contents if and when T ends in the halt state H on that string.

We can generalize the one-tape model to multi-tapes, in which case we can distinguish between input tapes, which are read-only, output tapes, which are write-only and "move right-only and add symbol", and work-tapes, which have no restrictions. Another generalization is a *non-deterministic Turing machine* for which the transition is described as a relation

$$\Delta \subseteq K \times \Gamma \times (K \cup \{Y, N, H\}) \times \Gamma \times D,$$

thereby allowing multiple potential moves in a single step.

2.2. Computability

Some definitions on types of languages:

Decidable: Language for which there is a Turing machine that halts on all inputs in Σ^* and accepts those and only those strings that are in the language. Also known as *computable.*

Recursively Enumerable (r.e.): Language for which there is a Turing machine that accepts a string if and only if it is in the language.

We can also describe Turing machines by strings which allows a *universal* Turing machine to be constructed that can simulate any Turing machine on any input. Thus the language

$$K = \{x : \text{Turing machine described by } x \text{ halts on input } x\}$$

is r.e. However,

Proposition 2.1: K is undecidable.

The key notion for proving undecidability results is *reduction.* In the simplest form we have *many-one* reduction, where $L \leq_m L'$, if there is a computable function f such that $x \in L$ if and only if $f(x) \in L'$. More generally, we can have *Turing reduction* $L \leq_T L'$. This means that there is a Turing machine deciding L that has an *oracle* tape that can answer questions about membership in L'. (The many-one reduction is a special case where the Turing machine just computes $f(x)$ and asks just one question about the membership of $f(x)$ in L' and halts with the same answer that it obtains on the oracle tape.)

The *Church-Turing* thesis states that any finite computing device is no more expressive than Turing machines. This includes RAMs, PRAMS, supercomputers with gazillions of processors, and the abacus.

2.3. Complexity

Time Complexity

Some more definitions:

TIME($t(n)$) denotes the set of languages that are decidable by Turing machines that always halt in at most $t(n)$ steps on inputs of length n. Similarly, NTIME($t(n)$) for non-deterministic Turing machines. (This means that on any string, every computation path halts in time $t(n)$ and at least one of them ends in an accepting configuration if and only the string is in the language.) The two famous complexity classes are

$$\begin{aligned} P &= \bigcup_k \mathrm{TIME}(n^k) \\ NP &= \bigcup_k \mathrm{NTIME}(n^k), \end{aligned}$$

and the big open question is

Does $P = NP$?

Similar to computability, there are analogous notions of reduction for complexity classes. Thus, we have many-one *polynomial-time* reduction, where $L \leq_m^P L'$ if there is a computable function f running in polynomial time such that $x \in L$ if and only if $f(x) \in L'$. This is a useful technique for showing that languages are equally hard (or easy) because $L' \in P \Rightarrow L \in P$ and $L' \in NP \Rightarrow L \in NP$.

Space Complexity

Here we deal with multi-tape Turing machines having input and work tapes. The space used in a computation is the total amount of cells used in all of the work tapes. (We don't count the input tape because we can now talk about sub-linear space.) Thus, we can define SPACE($s(n)$) and NSPACE($s(n)$). We define $PSPACE = \bigcup_k SPACE(n^k)$, $L = SPACE(\log n)$ and $NL = NSPACE(\log n)$

Analogously, we have *log-space* reductions, where $L \leq_m^{\log} L'$ if there is a computable function f using logarithmic space such that $x \in L$ if and only if $f(x) \in L'$.

Relations between Complexity Classes

For $t(n) \geq n$, and $s(n) \geq \log n$, we have

Proposition 2.2:

$$\mathrm{TIME}(t(n)) \subseteq \mathrm{NTIME}(t(n)) \subseteq \bigcup_c \mathrm{TIME}(2^{ct(n)})$$

$$\mathrm{SPACE}(s(n)) \subseteq \mathrm{NSPACE}(s(n)) \subseteq \mathrm{SPACE}(s^2(n))$$

For the sake of motivating these results, we prove below one of the above results, namely NSPACE($s(n)$) $\subseteq$ SPACE($s^2(n)$), also known as *Savitch's Theorem*.

Proof: Let T be a $s(n)$-space bounded non-deterministic Turing machine.

We define a *m-configuration* as (q, i, j, α), where q is the state, i is the input head position, j is the work-tape head position and α is the contents of the work tape which has exactly m cells used. Thus, $q \in K$, $0 \le i \le n+1$, $0 \le j \le m$, and $\alpha \in \Gamma^m$. Note that the length of a m-configuration is $O(\log n + m)$.

For C and D which are m-configurations, let $\text{REACH}(m, C, D, k) = \text{true}$ if and only if $C \xrightarrow{T^{\le 2^k}} D$ and every intermediate configuration is a m-configuration. In m amount of storage, T makes no more than 2^{cm} steps, where c is independent of m. Therefore, x is accepted by T if and only if $\text{REACH}(s(|x|), \text{Init}_{s(n)}(x), A, c \cdot s(|x|)) = \text{true}$, for some accepting configuration A, where Init_m is the initial configuration with exactly m work tape cells assigned blank.

The proof will be completed in the next lecture. $\square$

Lecture 3

Time and Space Complexity

January 13, 1998
Lecturer: Richard Ladner
Notes: Dan Fasulo

3.1. Savitch's Theorem

Recall that in the previous lecture, we defined a reachability function REACH such that, for a given Turing machine T and m-configurations C and D of T, REACH(m, C, D, k) is true if and only if state D can be reached from state C in at most 2^k steps and using a set of at most m work tape cells. Define $\text{Init}_m(x)$ to be the initial m-configuration for T in which T is at a start state, x is the input string, and m is the number of blank spaces available on the work tape. Thus, for a given Turing maching T and input x,

$$x \in L(T) \Leftrightarrow \text{REACH}(s(|x|), \text{Init}_{s(|x|)}(x), A, c \cdot s(|x|)),$$

where A is an accept state of T and c is a positive integer. We now show that the REACH function can be computed in an efficient amount of space by a deterministic algorithm.

Theorem 3.1 (Savitch):

$$\text{NSPACE}(s(n)) \subseteq \text{SPACE}(s^2(n)).$$

Proof: By our observation above, we only need to show that we can deterministically compute the REACH function in $O(s^2(n))$ space; then we will have a space-efficient algorithm which accepts x if T accepts x. Let $n = |x|$ and assume that we can use m work tape cells for computation. Also assume that we have a representation of T as a directed graph in which the vertices correspond to m-configurations and there is an edge from configuration A to configuration B if and only if B can be reached from A in one step. We now give the algorithm for REACH.

```
/* Decides if D can be reached from C in 2^k time with out exceeding m
   work tape cells. We assume T and x are fixed. */

algorithm reach(m : integer, C, D : m-configuration, k : integer)
{
   if k = 0 then
      if D reachable from C in 1 step or C = D then
         return true;
      else
```

```
         return false;
   else (k > 0)
      for each m-configuration E
         if reach(m, C, E, k-1) and reach(m, E, D, k-1) then
            return true;
      return false;
}
```

We now show that algorithm `reach` requires less than $s^2(n)$ units of storage if both m and k are $O(s(n))$. Observe that each m-configuration contains m units of tape storage, a number of size $\log n$ representing the index into the input string, and a number of size $\log m$ representing the index into the work area. The storage of each instance of the `reach` algorithm uses two such configurations plus $\log k$ storage to represent k. Thus, it takes a total of $O(m + \log n + \log k)$ storage. If we imagine a machine running the `reach` algorithm, there would be at most k instances of the reach function on the machine's stack at any given time. Hence, the total storage required would be $O(k(m + \log n + \log k))$. If we m and k are $O(s(n))$ then because $s(n) \geq \log n$ then the total storage is

$$O(s(n)(s(n) + \log n + \log\log s(n))) = O(s^2(n)).$$

This analysis, however, requires that we know $s(n)$ so that we can set m and k to be the appropriate size. If $s(n)$ is not known but the input x is guaranteed to be in the set, then we can simply let $m = 1, 2, \ldots$, until a value which works is found. However, if x is not necessarily in the set, then a more sophisticated approach is required. We define a new algorithm `safe` which determines if a given amount of storage is sufficient for the computation, regardless of whether the input is accepted or not. It can be seen that `safe` requires approximately the same amount of storage as `reach`.

```
/* Determines if T, starting in configuration C, does not
   exceed m tape cells in <= 2^k moves. */

algorithm safe(m : integer, C : m-configuration, k : integer)
{
   if k = 0 then
      if in one move of T from C, T exceeds the workspace then
         return false;
      else
         return true;
   else
      return (safe(m, C, k-1) and (for all m-configurations D)
              (reach(m, C, D, k-1) implies safe(m, D, k-1)));
}
```

We can then use the `safe` and `reach` algorithms to give a deterministic procedure for determining if $x \in L(T)$ in a space-efficient manner:

```
/* Deterministically simulates NTM T on input x in a space-
   efficient manner. */
```

```
algorithm deterministic_T(x : string)
{
   m : integer;
   enough_space : boolean;

   m = 0;
   enough_space = false;

   repeat
      m = m + 1;
      enough_space = enough_space or safe(m, Init(m,x), c*m);
   until enough_space;

   for some accepting m-configurations A
      if reach(Init(m,x), A, c * m) then
         return true;

   return false;
}
```

□

3.2. Linear Speed-up

We begin by presenting two results that show that one can always construct a faster or more space efficient Turing machine for a given problem, at the cost of creating a significantly larger machine. The first two results are known as the *Linear Speed-up Theorems.*

Theorem 3.2 (Linear Speed-up): For $\epsilon > 0$

$$\begin{aligned} \text{[N]SPACE}(s(n)) &= \text{[N]SPACE}(\epsilon \cdot s(n)), \\ \text{[N]TIME}(t(n)) &= \text{[N]TIME}(n + 2 + \epsilon \cdot t(n)). \end{aligned}$$

The idea behind the proof is that given T, we can create a new machine T' such that each tape cell of T' encodes m tape cells in T. This obviously reduces the number of cells required to store the information. It also leads to a speedup in the machine. We create a transition function for T' that simulate T for at least m move in just 8 moves of T'. In four moves of T', it accumulate $3m$ characters of T where the head of T is in the middle third of the characters. In 0 moves of T' it computes what would happen to these characters when the head of T would move off of them. This would simulate at least 8 moves of T. Finally, T' restores the new $3m$ characters to the 3 cells of T' and move the head of T' to be under where the head of T is. This restoration takes another 4 moves of T'.

3.3. Tape Reduction

Let [N]SPACE$_k(s(n))$ be the class of problems which can be solved in [non]deterministic space $s(n)$ by a machine with k work tapes, and, analogously, [N]TIME$_k(t(n))$ be the class of problems which can be solved in [non]deterministic time $t(n)$ by a k-tape machine.

Theorem 3.3 (Tape Reduction to One Tape): If $t(n) > n$ then

$$\begin{aligned}[\mathrm{N}]\mathrm{SPACE}_k(s(n)) &= [\mathrm{N}]\mathrm{SPACE}_1(s(n)),\\ [\mathrm{N}]\mathrm{TIME}_k(t(n)) &= [\mathrm{N}]\mathrm{TIME}_1(t^2(n)).\end{aligned}$$

The idea behind of the proof of this theorem is that the characters on each tape at index i in the multi-tape machine can be grouped into a single character on a 1-tape machine. The head position for each tape is also encoded in the characters of the 1-tape machine. In this way the storage stays the same, but the time will grow quadratically. Each may take as many at $t(n)$ steps of 1-tape machine to simulate one step of the multi-tape machine. It takes that long to find out what is under the heads and update their contents.

There is a very clever construction that yields a better time simulation if the simulating machine is allowed two tapes instead of one.

Theorem 3.4 (Tape Reduction to Two Tapes): If $t(n) > n$ then

$$[\mathrm{N}]\mathrm{TIME}_k(t(n)) = [\mathrm{N}]\mathrm{TIME}_2(t(n)\log t(n)).$$

3.4. Hierarchy

Finally, we begin looking at hierarchy theorems that demonstrate that complexity classes get truly larger if more and more space or time is allowed for computing. We say that $s(n)$ is *tapeconstructible* if and only if there exists T such that T uses exactly $s(n)$ tape cells on an input a^n and then halts.

Theorem 3.5 (Space-Hierarchy): If $s_2(n)$ is tape constructible, $\lim_{n\to\infty} \frac{s_1(n)}{s_2(n)} = 0$, and $s_1(n)$ and $s_2(n)$ are both greater than $\log_2 n$, then

$$\mathrm{SPACE}(s_1(n)) \subset \mathrm{SPACE}(s_2(n))^1.$$

Interestingly there is a gap theorem that demonstrates that some pathological behavior can happen if space bounds are not tape constructible.

Theorem 3.6 (Gap): For some computable function $s(n)$,

$$\mathrm{SPACE}(s(n)) = \mathrm{SPACE}(2^{s(n)}).$$

[1]We reserve the symbol "$\subset$" to mean proper subset and the symbol "$\subseteq$" for plain subset.

Lecture 4

Complexity Hierarchies & Alternation

January 15, 1998
Lecturer: Richard Ladner
Notes: Jim Fix

In this lecture we will complete our presentation of space and time complexity hierarchies, and introduce a more powerful Turing machine variant.

4.1. Space and Time Hierarchies

In the last lecture, we stated the Space Hierarchy Theorem, which we will now prove.

Proof:[Space-Hierarchy] We use a diagonalization argument: we construct a $s_2(n)$ space bounded Turing machine T such that for any $s_1(n)$ space bounded Turing machine T' there is an input x such that $x \in L(T')$ exactly when $x \notin L(T)$. Thus, we will demonstrate that $L(T) \notin \text{SPACE}(s_1(n))$. Our construction of T relies on the following fact: if T' is a Turing machine that runs in space $s_1(n)$, then there is a binary description e for T' such that the universal Turing machine, given e and an input x, simulates T' using only $cs_1(n)$ space. In addition, we assume that if e is a binary description of T', then so are the strings $e0, e00$ and so on.

Let T be a Turing machine that, for any input string x, does the following:

1. Lays out $s_2(|x|)$ storage cells. T never exceeds this amount of storage.
2. Checks if x is the binary description of a Turing machine. If not, T rejects the input.
3. Simulates the Turing machine with description x on x in the allotted storage without exceeding $2^{s_2(|x|)}$ time. This requires the maintenance of a time step counter that uses an additional $s_2(|x|)$ space.
4. If the simulation accepts x within the allotted space and time, then T rejects x. Otherwise, T accepts x.

Using linear speedup, it is clear that $L(T)$ can be accepted in space $s_2(n)$. We now argue that $L(T)$ cannot be accepted by any $s_1(n)$ space bounded Turing machine.

Let T' be a Turing machine that runs in space $s_1(n)$ and e be a minimal encoding of T'. On input e, T simulates T' in space $cs_1(|e|)$ and time $2^{ds_1(|e|)}$ for some $0 < c < d$. Choose n so that $s_2(n) > ds_1(n)$ and $n \geq |e|$, hence $s_2(n) > cs_1(n)$ and $2^{s_2(n)} > 2^{ds_1(n)}$. Let $x = e0^{n-|e|}$ and consider the result of T on input x. T will have enough storage and time to simulate T', and will accept x if and only if T' would reject x. Hence, $x \in L(T') \Leftrightarrow x \notin L(T)$. $\square$

There is an analogous hierarchy result for time complexity, demonstrating that increasing compute time yields more powerful complexity classes. A function $t(n)$ is *time constructible* if there is a Turing machine that runs in exactly $t(n)$ steps on input a^n.

Theorem 4.1 (Time Hierarchy): If $t_2(n)$ is time constructible and $\lim_{n\to\infty} \frac{t_1(n)\log t_1(n)}{t_2(n)} = 0$, then

$$\mathrm{TIME}(t_1(n)) \subset \mathrm{TIME}(t_2(n)).$$

The proof of this theorem is very similar to Theorem 3.5, and uses the fact that a universal Turing machine can simulate any $t_1(n)$ time bounded machine using only $ct_1(n)\log t_1(n)$ time.

The space and time hierarchy theorems can be quite useful, for example, we have the following applications:

Corollary 4.2:

$$\mathrm{SPACE}(n) \subset \mathrm{SPACE}(n^2) \subset \mathrm{SPACE}(n^3) \subset \cdots,$$

$$\mathrm{TIME}(n) \subset \mathrm{TIME}(n^2) \subset \mathrm{TIME}(n^3) \subset \cdots.$$

There are similar nondeterministic hierarchies, but for "practical" purposes only the deterministic hierarchies are useful.

4.2. Alternation

An *alternating Turing machine* is a generalization of deterministic and nondeterministic Turing machines. Formally, an alternating Turing machine is a 7-tuple $T = (K, K_E, K_U, \Sigma, \Gamma, \Delta, s)$ defined as in nondeterministic Turing machines with the addition that the set of states K is partitioned into the *existential states* K_E and the *universal states* K_U, hence

- $K_E \cup K_U = K$ and $K_E \cap K_U = \emptyset$.

We define machine configurations as before. A configuration that is in an existential (universal) state is called an *existential (universal) configuration.* From Δ, we can derive the one step relation $\xrightarrow{T}$ on configuration pairs as we did for nondeterministic Turing machines.

For alternating Turing machines, we define a new notion for language acceptance. Given an alternating Turing machine T, a *computation tree* of T is a tree whose nodes are configurations of T. The children of a node of configuration C in the tree are all configurations D such that $C \xrightarrow{T} D$. A computation tree is *accepting* if

- it is a single node and its root configuration is accepting, or
- it has more that one node and either
  - the root configuration is existential and at least one of its subtrees is accepting, or
  - the root configuration is universal and all if its subtrees are accepting.

The language accepted by an alternating Turing machine T is the set of strings $x \in L(T)$ such that the computation tree with root labeled $\mathrm{Init}(x)$ is finite and accepting.

4.3. Complexity of Alternation

ATIME($t(n)$) is the class of languages accepted by alternating Turing machines whose computation trees have depth at most $t(n)$ on an input of length n. ASPACE($s(n)$) is the class of languages accepted by alternating Turing machines whose computation trees are finite and whose configurations use at most $s(n)$ storage. It is common to define

$$\mathrm{AP} = \bigcup_k \mathrm{ATIME}(n^k).$$

Space bounded alternating Turing machines are related to deterministic time bounded Turing machines by the remarkable equivalence:

Theorem 4.3 (Alternating Space): If $s(n) \geq \log n$ then

$$\mathrm{ASPACE}(s(n)) = \bigcup_c \mathrm{TIME}(2^{cs(n)}).$$

This theorem provides interesting alternative characterizations for deterministic time complexity classes, most notably:

Corollary 4.4:

$$\mathrm{ASPACE}(\log n) = P$$

$$\mathrm{ASPACE}(n) = \textstyle\bigcup_c \mathrm{TIME}(2^{cn}) = \mathrm{EXPTIME}.$$

Proof:[Alternating Space Theorem]

($\subseteq$) For now, we assume that $s(n)$ is tape constructible. We will correct this later. Let T be an alternating Turing machine that runs in space $s(n)$. T can be in at most $2^{c_1 s(n)}$ configurations. Consider the directed graph G whose nodes are configurations of T and edges are the successor relation. Define a deterministic Turing machine T' which does the following on an input x of length n:

1. write G on the tape using $2^{c_2 s(n)}$ cells
2. mark all accepting configurations with 1 and all rejecting configurations with 0
3. repeat the following until Init(x) is marked
   (a) For each unmarked exitential configuration, if all its successors are marked, then mark it 0 if all its successor are marked 0 and mark it 1 if one of its successors is marked 1.
   (b) For each unmarked universal configuration, if all its successors are marked, then mark it 1 if all its successor are marked 1 and mark it 0 if one of its successors is marked 0.
4. If Init(x) is marked with 1 accept, otherwise reject.

T' will accept x if and only if T would accept x. Steps 1 and 2 each use no more than $2^{O(s(n))}$ time. Step 3 will make at most $2^{O(s(n))}$ iterations, since at least one configuration will be newly marked in step 3a. Step 3a can naively scan through all $2^{O(s(n))}$ configurations, hence step 3 uses no more than $2^{O(s(n))}$ time over all iterations.

To remove the space constructibility requirement on $s(n)$, we have T' guess the amount of space used by T on x. It does this by trying increasing space bounds $s = 1, 2$ and so on. For each value of s, it lays out the graph G as above, however G is modified slightly:

- The nodes of G are the configurations of T that use space s plus an additional error node E.
- The edges are based on the successor relation, however, if a configuration C has a successor configuration that uses more than s space, then the node for C has an edge into E instead.

We initialize the marking by marking the accepting configuration 1 and the rejecting configuration 0, but E remains unmarked. The marking is then done as described above. If $Init(x)$ gets marked, then T' accepts or rejects accordingly, otherwise the next value of s is tested. Note that if $Init(x)$ does not get marked then a configuration reachable from $Init(x)$ attempts to exceed the allotted space s.

($\supseteq$) This will be covered in the next lecture.

$\square$

Lecture 5

Alternating Space & Time Theorems

January 20, 1998
Lecturer: Richard Ladner
Notes: Sumeet Sobti

In this lecture, we complete the proof of the Alternating Space Theorem. We also prove the Alternating Time Theorem. Thereafter, we look at a more powerful variant of alternating Turing machines.

5.1. Alternating Space Theorem

In the last lecture, we proved one half of the Alternating Space Theorem. Here we complete the proof.

Theorem 5.1 (Alternating Space): If $s(n) \geq \log n$, then

$$\mathrm{ASPACE}(s(n)) = \bigcup_c \mathrm{TIME}(2^{cs(n)}).$$

Proof:

($\subseteq$) Proved in the last lecture.

($\supseteq$) Suppose $T = (K, \Sigma, \Gamma, \delta, s)$ is a single-tape deterministic Turing machine that takes at most $2^{cs(n)}$ steps on an input of length n. (Note that we are not loosing any generality in assuming T to be a single-tape Turing machine. See Tape-Reduction theorem).

For now, lets assume that the $2^{cs(n)}$ time bound is known. (Later we'll see how the proof can be modified to work without this assumption.)

Lets represent the configurations of the machine by strings of the form $\#\alpha q\beta\#\#$, where q denotes the state of the machine, $\alpha\beta$ is the tape contents ($|\alpha\beta| = 2^{cs(n)}$) and α is the string on the portion of the tape that lies to the left of the head. Thus, each machine configuration is represented by a string of length $m = 2^{cs(n)} + 4$.

We now design an ASPACE($s(n)$) algorithm to decide membership in $L(T)$.

If D is the configuration that follows from C in one step, then for any $2 \leq k \leq m-2$, the k^{th} symbol of WD is fully determined by the $(k-1)^{th}, k^{th}, (k+1)^{th}$ and $(k+1)^{th}$ symbols of C. Thus we can define a predicate F in the following way.

$F(\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma)$ is true if and only if σ must be the k^{th} symbol in the succeeding configuration if $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ are respectively the $(k-1)^{th}, k^{th}, (k+1)^{th}$ and $(k+2)^{th}$ symbols in the preceding configuration.

Also it is easy to see that F can be deterministically and computed in constant time.

Now, consider a fixed input string x of length n. Define the predicate R_x in the following manner. Let x_i be the i-th character of x.

$R_x(t, k, \sigma)$ is true iff σ is the k^{th} symbol of the t^{th} configuration of T when T runs on input x.

An *alternating algorithm* for computing $R_x(t, k, \sigma)$:

- if $(t = 0)$ then return

$$\begin{aligned}
&[(k = 1 \wedge \sigma = \#) \vee \\
&(k = 2 \wedge \sigma = s) \vee \\
&(k = 3 \wedge \sigma = x_1) \vee \ldots \vee (k = n + 2 \wedge \sigma = x_n) \vee \\
&(n + 3 \leq k \leq m - 2 \wedge \sigma = \sqcup) \vee \\
&(m - 1 \leq k \leq m \wedge \sigma = \#)]
\end{aligned}$$

- if $(t > 0)$ then return

$$\begin{aligned}
&[((k = 1 \vee m - 1 \leq k \leq m) \wedge \sigma = \#) \vee \\
&(2 \leq k \leq m - 2 \wedge \exists \sigma_1, \sigma_2, \sigma_3, \sigma_4 (F(\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma) \wedge \\
&\qquad R_x(t - 1, k - 1, \sigma_1) \wedge \\
&\qquad R_x(t - 1, k, \sigma_2) \wedge \\
&\qquad R_x(t - 1, k + 1, \sigma_3) \wedge \\
&\qquad R_x(t - 1, k + 2, \sigma_4)))]
\end{aligned}$$

The correctness of the above algorithm is obvious. An important thing to note here is the way this algorithm will be executed by an alternating TM. Lets see in a little detail how an alternating TM may execute this algorithm.

Suppose the alternating TM wants to compute the value of a formula $\psi_1 \vee \psi_2$. What it does is that it goes to an existential state (if it is not in one already), and 'forks' out into two branches. It computes ψ_1 on one branch and ψ_2 on the other. Similarly if it wants to compute $\psi_1 \wedge \psi_2$, it goes to a universal state and forks out into two branches.

If it needs to compute $\exists x : \psi$, then it goes to an existential state and forks out into a number of branches. On each branch, it takes a different value of x and computes ψ. Similar is the computation of $\forall x : \psi$.

When the above algorithm is executed in this manner by an alternating TM, it can be easily seen that the only space it requires is to store the parameters t, k and σ. Now $0 \leq t \leq 2^{cs(n)}$, $1 \leq k \leq m$ and $\sigma \in K \cup \Gamma \cup \{\#\}$. Thus the parameters can be stored in $O(s(n))$ space. So, the above algorithm is an ASPACE($s(n)$) algorithm.

Using the above algorithm, we can easily write an ASPACE($s(n)$) algorithm for deciding membership in $L(T)$.

An *alternating algorithm* for deciding $L(T)$:

Input x.

- return $\exists t, k : (0 \leq t \leq m \wedge 1 \leq k \leq m \wedge R_x(t, k, Y))$ where Y is the accepting state of T.

To remove the assumption about apriori knowledge of $s(n)$, we stylize T in a particular manner. We let T have a unique accepting and a unique rejecting configuration. For example, we may stipulate that T always halts with its tape contents erased and its head at the left end of the tape. In this case, the unique accepting configuration will be $\#Y \sqcup \sqcup \ldots \sqcup \#\#$, where Y is the accepting state. Similarly, the rejecting state will also be unique. Now, one can cycle through $m = 3, 4, \ldots$ until either the accepting or the rejecting configuration is found. The details are left as an exercise to the students.

□

5.2. Alternating Time Theorem

Theorem 5.2 (Alternating Time): If $t(n) \geq n$ and $s(n) \geq n$, then

$$\text{ATIME}(t(n)) \subseteq \text{SPACE}(t(n))$$

$$\text{NSPACE}(s(n)) \subseteq \text{ATIME}(s^2(n))$$

Proof: First we show that ATIME$(t(n)) \subseteq$ SPACE$(t(n))$. Suppose T is a $t(n)$ time bounded alternating TM, which on every input makes at least one move. W.l.o.g., assume that at each step, T has exactly d choices for the next move (where d is a constant).

Fix a given input x of length n. Now, by definition of acceptance in alternating TMs, $x \in L(T)$ iff the computation tree of T on x is an accepting tree. Thus, it would suffice if we can give an SPACE$(t(n))$ deterministic algorithm to test if the computation tree of T on x is accepting or not. This is exactly what we are going to do. We will a give *space-efficient* evaluation procedure for computation trees of T.

Suppose R is the computation tree of T on input string x. A computation path starting from the root of R (i.e. Init(x)) can be represented as a sequence of integers $\sigma = i_1 i_2 \ldots i_k$, where i_j indicates which of the d possible moves was made at the j^{th} step in the computation. Thus, $\forall j : 1 \leq i_j \leq d$. Also since T is a $t(n)$-time bounded machine, no computation path can be mose than $t(n)$ steps long. Therefore, $k \leq t(n)$.

For any path $\sigma = i_1 i_2 \ldots i_k$, we define the following premitives.

$last(\sigma) = \lambda$ if $k = 0$, and i_k if $k \geq 1$.
$back(\sigma) = i_1 i_2 \ldots i_{k-1}$.
$down(\sigma) = i_1 i_2 \ldots i_k 1$.
$conf(\sigma) =$ the configuration T reaches by following path σ.

Algorithm for evaluating computation tree of T on input x:

$\sigma = \lambda, b =?, v =?$.
while $\sigma \neq \lambda$ or $b \neq \lambda$ or $v =?$ do

$v =?$:
$conf(\sigma)$ is accepting: $b = last(\sigma)$; $\sigma = back(\sigma)$; $v = T$
$conf(\sigma)$ is rejecting: $b = last(\sigma)$; $\sigma = back(\sigma)$; $v = F$
else: $\sigma = down(\sigma)$

$v = T$:
$conf(\sigma)$ is existential: $b = last(\sigma)$; $\sigma = back(\sigma)$; $v = T$
$conf(\sigma)$ is universal $\wedge b = d$: $b = last(\sigma)$; $\sigma = back(\sigma)$; $v = T$
$conf(\sigma)$ is universal $\wedge b \neq d$: $b =?$; $\sigma = \sigma(b+1)$; $v =?$

$v = F$:
$conf(\sigma)$ is universal: $b = last(\sigma)$;$\sigma = back(\sigma)$; $v = F$
$conf(\sigma)$ is existential $\wedge b = d$: $b = last(\sigma)$; $\sigma = back(\sigma)$; $v = F$
$conf(\sigma)$ is existential $\wedge b \neq d$: $b =?$; $\sigma = \sigma(b+1)$; $v =?$

endwhile

When $\sigma = \lambda$ and $b = \lambda$ the computation tree has been searched and there is a value for v. The value of $v = T$ if and only if the input is accepted.

The correctness of the algorithm can be seen because of several invariants. First, the current node in the tree being searched is represented by the path σ. Second, if the search is moving toward the root, then b is the branch just taken and v is the value of the subtree of that branch. The algorithm requires space for storing the path σ and for computing the $conf(\sigma)$ function. But since T is a $t(n)$ time bounded machine, this is $O(t(n))$ space. Hence, the algorithm works in $O(t(n))$ space.

The proof of "NSPACE($s(n)$) $\subseteq$ ATIME($s^2(n)$)" is similar to that of Savitch's theorem. Remember, in the proof of Savitch's theorem, we needed to compute

$$\exists E : \text{REACH}(m, C, E, k-1) \wedge \text{REACH}(m, E, D, k-1)$$

There the machine used a stack to implement the recursion in the computation. Here the alternating machine computes the formula in the manner described earlier (See the discussion in proof of the Alternating Space Theorem).

Another point where the proof needs modification is part where the algorithm goes through all $m = 1, 2, 3, 4, 5, \ldots$ till it finds a *safe* m. But this approach takes too much time. Here we need to make m go along powers of 2, (i.e. $m = 1, 2, 4, 8, \ldots$).

Actually, it is not surprising to see that the proof shares so many ideas with the proof of Savitch's theorem. Savitch's theorem almost trivially follows from the Alternating Time theorem.

□

An immediate consequence of the Alternating Time theorem is the following corollary.

Corollary 5.3: AP = PSPACE

5.3. A Variant On Alternating Turing Machines

In this section, we describe a variant on alternating Turing machines. This model is more powerful than the ordinary alternating TMs that we have been working with so far. The power of this model comes from the fact that it has *random access* on the input tape.

More concretely speaking, in this model the alternating TM has an input tape, an index tape and a work tape. It has two heads, one each for the index tape and the work tape. It doesn't need to have a head for the input tape. The reason is that the input is read indirectly using the index tape. Its next move depends on the state, symbol under the head on work tape and the symbol on the input tape that is indexed by the index currently written on the index tape.

Random access on the input tape allows us to prove a stronger version of the alternating time theorem. For this model, we can prove the alternating time theorem for any $t(n) \geq log(n), s(n) \geq log(n)$.

5.4. Complements Of Complexity Classes

For any class C, coC is defined as the class of languages whose complements are in C. For example,

Definition 5.4: $\text{coNTIME}(t(n)) = \{\overline{L} : L \in \text{NTIME}(t(n))\}$

Definition 5.5: $\text{coNSPACE}(s(n)) = \{\overline{L} : L \in \text{NSPACE}(s(n))\}$

Some facts:

$$\text{coNP} = \bigcup_k \text{coNTIME}(n^k)$$

$$\text{coNPSPACE} = \text{NPSPACE} = \text{PSPACE}$$

Open Problem 5.6: $\text{coNP} = \text{NP}$?

Theorem 5.7 (Immerman-Szelepscényi): If $s(n)$ is tape constructible and $s(n) \geq log(n)$, then

$$\text{NSPACE}(s(n)) = \text{coNSPACE}(s(n))$$

Lecture 6

The Immerman-Szelepscényi Theorem

January 22, 1998
Lecturer: Richard Ladner
Notes: Andrew Keller

6.1. Immerman-Szelepscényi Theorem

Theorem 6.1 (Immerman-Szelepscényi): If $s(n) \geq \log n$ and tape constructible, then

$$\text{NSPACE}(s(n)) = \text{CoNSPACE}(s(n)).$$

Proof:
It suffices to show $L \in \text{NSPACE}(s(n)) \Rightarrow \overline{L} \in \text{NSPACE}(s(n))$. Let T be a $s(n)$ space bounded nondeterministic Turing Machine. Fix x to be an input of length n. Define three nondeterministic space $s(n)$ bounded partial functions:

1. Count(k) = number of configurations reachable from the initial configuration in $\leq k$ steps of T. Count is a total function.
2. Reach(C, k) = true if and only if configuration C is reachable from the initial configuration $Init(x)$ in $\leq k$ steps of T. Reach is only defined when it is true.
3. NoReach(C, k) = true if and only if configuration C is not reachable from the initial configuration $Init(x)$ in $\leq k$ steps of T. NoReach is only defined when it is true.

Definition 6.2: A partial function $f(x)$ is *non-deterministic computable* in space $s(n)$ (or time $t(n)$) if there is a nondeterministic Turing Machine with output tape such that for all x:

- all computations run in allotted space $s(n)$ (or time $t(n)$),
- if $f(x)$ is defined then there is at least one computation that leads to acceptance with $f(x)$ on the output tape,
- if $f(x)$ is defined then any output produced in an accept state is $f(x)$.

Using the above three nondeterministic functions, an algorithm for $\overline{L}$ can be written:

```
for all configurations C
    if C is accepting then NoReach(C,2**(cs(n)))
```

The algorithms for the three functions are:

- Algorithm for Reach(C, k):

```
run T nondeterministically from Init(x) for <= k steps
if C is reached, return true
```

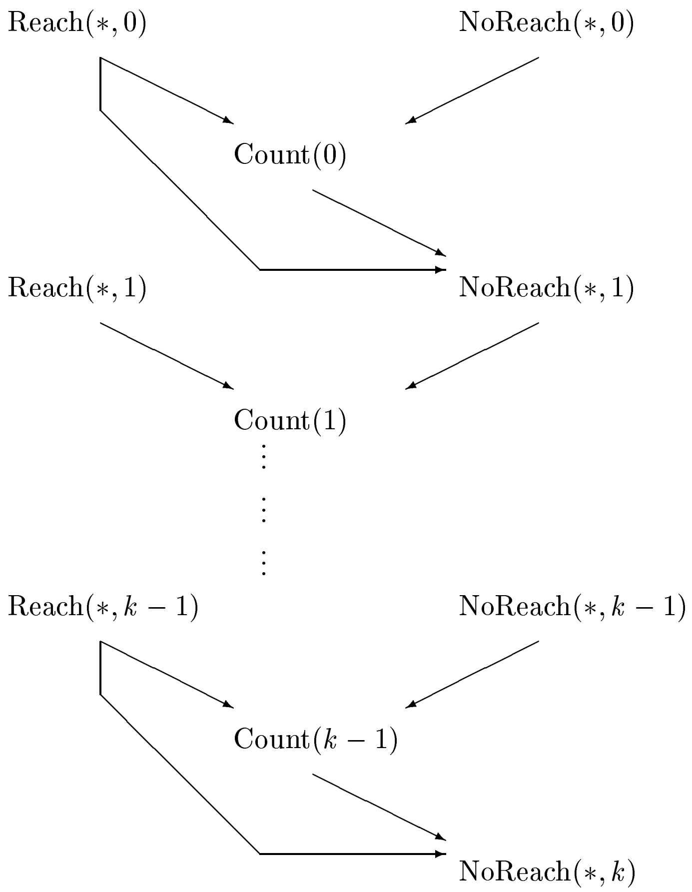
- Algorithm for NoReach(C, k):

```
if k = 0
   if C <> Init(x) return true
   else return false
if k > 0
   m := 0
   for all D
      if Reach(D, k-1) and C is not reachable from D in <= 1 step then m := m+1
   if m = Count(k-1) return true
```

- Algorithm for Count(k):

```
m := n := 0
for all C do one of:
     if Reach(C,k) then m := m+1
     if NoReach(C,k) then n := n+1
if m+n=M where M is the total number of configurations, return m
```

To compute NoReach(k), one first needs Reach($k-1$) and Count($k-1$), and to compute Count($k-1$), one first needs Reach($k-1$) and NoReach($k-1$). One can thus compute NoReach(C, k) by iteratively computing Count(0), Count(1),.....,Count($k-1$). The following figure describes the dependencies among the partial functions.



The algorithm for $\overline{L}$ executes the computation of NoReach$(C, 2^{cs(n)})$ in such a manner. Computing Reach(C, k) requires $O(s(n) + \log n + \log k) = O(s(n) + logk)$ space. Furthermore, if both NoReach$(C, k-1)$ and Count$(k-1)$ also require only $O(s(n) + \log k)$ space, then so will NoReach(C, k). Since computing NoReach$(C, 0)$ and Count(0) satisfy that space requirement, one may conclude by induction that computing NoReach(C, k) requires $O(s(n) + \log k)$ space, and that $\overline{L}$ can be computed in space $O(s(n) + \log 2^{cs(n)}) = O(s(n))$ space.

□

6.2. Propositional Logic

We now begin our first unit on logic. Papadimitriou [12] and Enderton [4] are both good resources for the propositional and first-order logic.

Boolean values T,F (constants)

Boolean variables X_1, X_2, ...

Operators $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow, \oplus, \equiv$

Syntax for Boolean Formulas e.g. $x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$

Operator Precedence $\neg, \wedge, \Rightarrow, \equiv, \vee, \oplus$

The semantics of a formula is an interpretation of its variables. To be specific, an assignment σ maps variables to Boolean values: $\sigma(X_i) \in \{T, F\}$. For the constants T and F, $\sigma(T) = \mathrm{T}$, and $\sigma(F) = \mathrm{F}$ σ can be extended to all formulas in the natural way where each operation symbol is interpreted by the correct operator, e.g.

$$\sigma(\alpha \wedge \beta) = \sigma(\alpha) \wedge \sigma(\beta)$$

Define $\sigma \models \alpha$ if and only if $\sigma(\alpha) = \mathrm{T}$. Further, α is *valid* if $\sigma(\alpha) = \mathrm{T}$ for all σ, or $\sigma \models \alpha$ for all σ σ is *satisfiable* if $\sigma(\alpha) = \mathrm{T}$ for some σ

We have the following logic relationship between satisfiability and validity.

$$\models \alpha \Leftrightarrow (\neg\alpha \text{ is } \textit{not} \text{ satisfiable})$$

Naturally there are many laws about logical identities. Here are a few.

$$\models (\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta)$$

$$\models \neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$$

$$\models \alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\beta \vee \gamma)$$

A statement is in *Disjunctive Normal Form* (DNF) if it is expressed as:

$$\bigvee_{i=1}^{m} (\bigwedge_{j=1}^{m_i} z_{ij})$$

where z_{ij} are literals (T,F,X,$\neg X$, etc). For example: $(x \wedge y) \vee (x \wedge \neg y)$.

A statement is in *Conjunctive Normal Form* (CNF) if it is expressed as:

$$\bigwedge_{i=1}^{m} (\bigvee_{j=1}^{m_i} z_{ij}).$$

Theorem 6.3 (Basic Theorem): For all α there is a β in CNF and γ in DNF and

$$\models \alpha \equiv \beta \equiv \gamma$$

Proof: Let α be a statement with variables X_1, X_2, ..., X_k. Then one can construct a truth table in which the value of $\sigma(\alpha)$ is tabulated for all 2^k possible combinations of $\sigma(X_1)$, $\sigma(X_2)$, ..., $\sigma(X_k)$. The truth table for $\sigma(\alpha)$ is:

assignment	*variable*				
	X_1	X_2	...	X_k	$\sigma(\alpha)$
σ_1	T	T	...	T	T
σ_2	T	T	...	F	F
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
σ_{2^k}	F	F	...	F	T

Define a conjunction C related to σ,

$$C(\sigma) = z_1 \wedge z_2 \wedge ...z_k$$

where $z_i = \{\ x_i$ if $\sigma(x_i) = $ T, $\neg x_i$ if $\sigma(x_i) = $ F$\}$.

Similarly, define a disjunction D related to σ,

$$D(\sigma) = \neg z_1 \vee \neg z_2 \vee ...\neg z_k$$

where double negatives are removed.

Then,

$$\gamma = \bigvee_{\sigma:\sigma(\alpha)=T} C(\sigma),$$

$$\beta = \bigwedge_{\sigma:\sigma(\alpha)=F} D(\sigma).$$

$\square$

Lecture 7

Satisfiability and Quantified Boolean Formulas

January 27, 1998
Lecturer: Richard Ladner
Notes: Mark Friedman

Today we have

- the reduction of SAT to 3-CNFSAT,
- the Cook-Levin Theorem showing SAT to be NP-complete, and
- the similar theorem showing Quantified Boolean Formulas to be PSPACE-complete.

7.1. Propositional Logic Continued

Theorem 7.1: SAT $\leq_m^P$ 3-CNFSAT.

For any formula α, we will construct, in polynomial time, a formula $F(\alpha)$ that is in conjunctive normal form, with at most three literals per clause, which is satisfiable if and only if α is. We will introduce a new variable V_β for each subformula β and write down clause $C(\beta)$ which relates V_β and β.

- $C(x)$ is $V_x \equiv x$
- $C(\alpha \wedge \beta)$ is $V_{\alpha\wedge\beta} \equiv V_\alpha \wedge V_\beta \wedge C(\alpha) \wedge C(\beta)$
- $C(\alpha \vee \beta)$ is $(V_{\alpha\vee\beta} \equiv V_\alpha \vee V_\beta) \wedge C(\alpha) \wedge C(\beta)$
- $C(\alpha \equiv \beta)$ is $(V_{\alpha\equiv\beta} \equiv (V_\alpha \equiv V_\beta)) \wedge C(\alpha) \wedge C(\beta)$
- $C(\neg\alpha)$ is $(V_{\neg\alpha} \equiv \neg V_\alpha) \wedge C(\alpha)$

The other operators are similarly done.

For example, let α be $(x \vee y) \wedge (\neg x \vee y)$. Then $C(\alpha)$ is

$$(V_\alpha \equiv V_{x\vee y} \wedge V_{\neg x\vee y})\wedge$$
$$(V_{x\vee y} \equiv V_x \vee V_y) \wedge$$

$$(V_{\neg x \vee y} \equiv V_{\neg x} \vee V_y) \wedge$$
$$(V_{\neg x} \equiv \neg V_x) \wedge$$
$$(V_x \equiv x) \wedge$$
$$(V_y \equiv y).$$

Define: $F(\alpha) = V_\alpha \wedge C(\alpha)$.

F is in 3-CNF. Each RHS in the recursive definition of C is a conjunction of 3-CNF formulas and a single new clause, which has at most three variables. This clause can be replaced by its 3-CNF-conversion in constant space.

We now show that α is satisfiable if and only if $F(\alpha)$ is satisfiable.

This follows from the following two observations:

1. $\tau(C(\alpha)) = T$ if and only if $\tau(V_\beta) = \tau(\beta)$ for all subformulas β of α.
2. Given an assignment σ of T or F to the original variables, you can extend that assignment naturally to σ' such that $\sigma'(C(\beta)) = T$ for all subformulas β of α.

Now to finish the proof if α is satisfiable by an assignment σ, then extend σ to σ' as in 2. By 1, $\sigma(V_\alpha \wedge C(\alpha)) = T$. If $\sigma(F(\alpha)) = T$ then then both $\sigma(V_\alpha) = T$ and $\sigma(C(\alpha)) = T$. By 1, $\sigma(\alpha) = T$.

It may be helpful to view α as a network of gates. Then it is easy to see that the V_i label intermediate values. $F(\alpha)$ now merely 'says' that α is satisfied exactly when its output is true and each gate works properly.

7.2. The Cook-Levin Theorem

Theorem 7.2: SAT is NP-complete.

SAT is clearly in NP. Given a propositional formula, and an assignment, you can check the truth value of the formula in polynomial time.

Suppose we have a nondeterministic Turing machine T which decides L, in n^k time. We will show completeness by constructing, in polynomial time, a formula that is true if and only if T accepts.

Fix input w, which is of length n. The formula F will have variables $x_{\sigma ij}$ representing "the ith symbol of the jth configuration is the letter σ." For simplicity we will often drop the last subscript when the configuration is understood. We will not distinguish between a configuration X and the vector of variables representing it.

We constrain T to be a nondeterministic Turing machine with one tape and a single accepting state, from which there is a self-transition that does nothing. A computation can be of length at most n^k, so a configuration will look like:

$$\#\sigma_1\sigma_2 \ldots \sigma_h q \sigma_{h+1}\sigma_{h+2} \ldots \sigma_{m-3}\#$$

where $m = cn^k + d$. m is the length of a configuration, and an upper bound on the number of steps in the shortest accepting computation of T. If we define $\Gamma' = \Gamma \cup Q \cup \{\#\}$, a configuration is a string in Γ'^*.

We can define a six-place relation $R \subseteq \Gamma'^6$ which identifies all legal triples of successive symbols in two successive configurations of a legal computation in T. That is, we claim that the legality constraint for two successive turing machine configurations can be expressed as a conjunction of constraints on successive triples, such as in C and C' below.

```
C       xqyzxyzxyzXYZxyzxyz
C'      xwpzxyzxyzXYZxyzxyz
```

The relation R indicates that when a machine moves, it may only change symbols in the three-symbol window surrounding the state, and only by the legal rules in the finite state control of T.

- $R(aqbacp)$ whenever $(qbpcR)$ is a transition of T.
- $R(aqbpcb)$ whenever $(qbpcL)$ is a transition of T.
- $R(abcdbe)$ whenever a, b, c are not symbols in Q.
- $R(\#ab\#cd)$ for all symbols a, b, c, d.
- $R(ab\#cd\#)$ for all symbols a, b, c, d.
- $R(aAbaAb)$ where A is the accept state.

Now we can write down declaratively the constraints on variables that describe an accepting computation of input w.

- x is a string, with exactly one symbol per postion.

$$S(x) = \left(\bigwedge_{i=1}^{m} \bigvee_{\sigma \in \Gamma'} x_{\sigma i} \right) \wedge \left(\bigwedge_{i=1}^{m-2} \bigwedge_{\sigma \in \Gamma'} \bigwedge_{\substack{\tau \in \Gamma' \\ \tau \neq \sigma}} x_{\sigma i} \Rightarrow x_{\tau i} \right)$$

- T can compute y from x in one step.

$$C(x,y) = \bigwedge_{i=1}^{m-2} \bigvee_{\substack{\sigma_0,\sigma_1,\sigma_2,\tau_0,\tau_1\tau_2 \in \Gamma' \\ R(\sigma_0\sigma_1\sigma_2\tau_0\tau_1\tau_2)}} x_{\sigma_0 i} \wedge x_{\sigma_1 (i+1)} \wedge x_{\sigma_2 (i+2)} \wedge y_{\tau_0 i} \wedge y_{\tau_1 (i+1)} \wedge y_{\tau_2 (i+2)}$$

- x_0 is the initial configuration of T on input $w = w_1 \ldots w_n$, where s is the start state of T, and the rest of the tape is blank.

$$Init_w(x) = x_{\#1} \wedge x_{\#m} \wedge x_{s2} \wedge x_{w_1 3} \wedge \ldots \wedge x_{w_n (n+2)} \bigwedge_{i=n+3}^{m-1} x_{\sqcup i}$$

Now we define one big hairy formula F_w, which 'says' that at time m we accept, at time zero we have the initial configuration, and every step is legal.

$$F_w = Init_w(x_0) \wedge \bigvee_{i=1}^{m-1} x_{Aim} \wedge \bigwedge_{t=0}^{m} S(x_t) \wedge \bigwedge_{t=0}^{m-1} C(x_t, x_{t+1})$$

Note that F_w is quadratic in w, and is in CNF, if you distribute the disjunction in $C(x,y)$. In reality, there is a logarithmic expansion factor as well in the worst case, because the number of new variables is proportional to the length of α, But the number of different variables in α may be very small. So encoding these different symbols incurs a logarithmic penalty.

7.3. Quantified Boolean Formulas

A SAT problem maps to a problem of the form $\exists\overline{x}F$ for a vector of boolean variables $\overline{x}$ and a propositional formula F. *Quantified Boolean Formulas*, or QBF, are of the form $Q\overline{x}F$, where Q is a quantifier ($\exists$ or $\forall$), and F is a propositional *or* QBF formula. One example of a QBF is

$$\exists x \exists y \forall z (x \equiv z \vee y \equiv z)$$

A few side notes. We will be lax about syntactic parentheses rules. The quantifiers do not need to be all at the beginning. When they are, the QBF is said to be in *prenex* form. To convert to prenex form is easy. First rename away any quantified variable from all other variables to prevent variable capture. Then distribute $\vee$'s and $\wedge$'s over quantifiers. For a quantifier to cross a negation, it must flip from an existential to a universal, or vice versa. For example, the prenex form of $\forall y \exists x (x \wedge \neg(\exists x (x \equiv y))$ is

$$\forall y \exists x \forall z (x \wedge \neg(z \equiv y))$$

A QBF is well-formed if all variables are quantified. We adopt the convention that any unquantified variable is implicitly existentially quantified, and then all QBF formulae are well-formed. In QBF there is no distinction between truth and validity. We can define what it means to be true by enumerating the possible combinations of truth assignments to quantifiers, going left to right. Define B_ω to be the set of all true quantified Boolean formulas.

Theorem 7.3: B_ω is PSPACE-complete.

AP = PSPACE, so we will show B_ω is AP-complete. B_ω is in AP. Reading the input formula left to right, when we reach $\exists x$, we branch existentially and choose a truth assignment for x. Similarly, when we reach $\forall y$, we branch universally and choose a truth assignment for y. Then we check the formula for truth, given truth assignments for all the variables.

Completeness. Now we proceed as in Cook's theorem. Let T be an alternating Turing machine that runs in n^k time, on input x of length n. The initial state is existential. Without loss of generality, we can add $O(n^k)$ length to the computation to make every state alternate.

We use the same boolean variables, and configurations as in Cook's theorem. We also use *Init* and C as defined there. We define the ACCEPT predicate on configurations recursively:

- T accepts,

$$ACCEPT_m = \bigvee_{i=1}^{m} x_{Aim}.$$

- A state is accepting if its successor state(s) is(are),

  If t is even,

$$ACCEPT_t(x_t) = \exists x_{t+1}(C(x_t, x_{t+1}) \wedge ACCEPT_{t+1}(x_{t+1})).$$

  If t is odd,

$$ACCEPT_t(x_t) = \forall x_{t+1}(C(x_t, x_{t+1}) \Rightarrow ACCEPT_{t+1}(x_{t+1})).$$

Now we write down a big hairy QBF that 'says' whether the computation of T accepts input w.

$$F_w = \exists x_0(Init_w(x_0) \wedge ACCEPT_0(x_0)).$$

Lecture 8

Propositional Proof Systems

February 3, 1998
Lecturer: Richard Ladner
Notes: Nicholas Bone

8.1. Rules of Inference

Shoenfield defined a proof system $\mathcal{S}$ based on the following five rules of inference. The formulas above the line represent an antecedent—or premise—and the formulas below the line represent the consequent—or conclusion— we may draw from these.

1. $\dfrac{}{\neg\alpha \vee \alpha}$
2. $\dfrac{\alpha}{\alpha \vee \beta}$
3. $\dfrac{\alpha \vee \alpha}{\alpha}$
4. $\dfrac{(\alpha \vee \beta) \vee \gamma}{\alpha \vee (\beta \vee \gamma)}$
5. $\dfrac{\alpha \vee \beta, \neg\alpha \vee \gamma}{\beta \vee \gamma}$ (Cut Rule)

Definitions: In the following definitions, $\mathcal{Z}$ is an arbitrary proof system and α and β are formulas.

- $\vdash_{\mathcal{Z}} \alpha$ if $\exists \beta_1, \ldots, \beta_k$, where $\beta_k = \alpha$ and $\forall i \leq k \exists i_1, \ldots, i_m < i$ such that $\dfrac{\beta_{i_1}, \ldots, \beta_{i_m}}{\beta_i}$ is an instance of a rule in $\mathcal{Z}$. In other words, there is a proof of α in $\mathcal{Z}$.
- $\models \alpha$ if α is valid (true under any assignment).
- $\mathcal{Z}$ is *sound* if $\forall \alpha, \vdash_{\mathcal{Z}} \alpha \Rightarrow \models \alpha$.
- $\mathcal{Z}$ is *complete* if $\forall \alpha, \models \alpha \Rightarrow \vdash_{\mathcal{Z}} \alpha$.

Example: The derivation of $\dfrac{\alpha, \alpha \Rightarrow \beta}{\beta}$ (Modus Ponens).

1	α	from the antecedent
2	$\neg\alpha \vee \beta$	$(\equiv \alpha \Rightarrow \beta)$ from the antecedent
3	$\alpha \vee \beta$	applying rule 2 to statement 1
4	$\beta \vee \beta$	applying rule 5 to statements 2 and 3
5	β	applying rule 3 to statement 4

Theorem 8.1: $\mathcal{S}$ is both sound and complete.

Proof omitted.

8.1.1. Axioms

Let Γ be a set of axioms (not necessarily valid), $\mathcal{Z}$ a proof system, and α and β formulas. Then we have the following additional definitions:

- $\Gamma \vdash_{\mathcal{Z}} \alpha$ if $\exists \beta_1, \ldots, \beta_k$, where $\beta_k = \alpha$ and $\forall i \leq k$ either $\beta_i \in \Gamma$ or $\exists i_1, \ldots, i_m < i$ such that $\dfrac{\beta_{i_1}, \ldots, \beta_{i_m}}{\beta_i}$ is an instance of a rule in $\mathcal{Z}$. In effect, Γ adds the set of rules $\{\dfrac{-}{\gamma} : \gamma \in \Gamma\}$ to our proof system.
- $\Gamma \models \alpha$ if α is valid relative to Γ (any assignment which satisfies all formulas of Γ also satisfies α).
- $\mathcal{Z}$ is *strongly sound* if $\forall \Gamma \forall \alpha, \Gamma \vdash_{\mathcal{Z}} \alpha \Rightarrow \Gamma \models \alpha$.
- $\mathcal{Z}$ is *strongly complete* if $\forall \Gamma \forall \alpha, \Gamma \models \alpha \Rightarrow \Gamma \vdash_{\mathcal{Z}} \alpha$.

Theorem 8.2: $\mathcal{S}$ is strongly sound and strongly complete.

Proof omitted.

Theorem 8.3: Deduction Theorem for $\mathcal{S}$: Let Γ be a set of formulas, and α and β be formulas. Then $\Gamma \cup \{\alpha\} \vdash_{\mathcal{S}} \beta$ if and only if $\Gamma \vdash_{\mathcal{S}} \alpha \Rightarrow \beta$.

This can be proved by induction on proof length.

8.2. Algorithms for Validity

Proof systems are pretty, but they don't give insight into how to find a proof. The following are three algorithmic approaches for determining the validity of Boolean formulas.

Remark: $\models \alpha \Leftrightarrow \neg\alpha$ is not satisfiable.

8.2.1. Basic Truth Table Approach

Given a formula, α, for each possible assignment, σ, check if $\sigma \models \alpha$. This procedure takes exponential time.

8.2.2. Davis-Putnam Approach

First a notational definition: Let α be a formula and x be a variable appearing in α. Then $\alpha(x = T)$ is the formula obtained by replacing every occurrence of x in α with T, and simplifying using rules to remove T and F from formulas. Removal rules include replacing $T \vee \beta$ with T, $T \wedge \beta$ with β and so on. The expression $\alpha(x = F)$ is defined similarly.

The Davis-Putnam algorithm is defined as follows:

- $DP(T)$ =true; $DP(F)$ =false
- For a formula α of $n > 0$ variables, choose a variable x in α and return $DP(\alpha(x = T)) \wedge DP(\alpha(x = F))$. Note that the formulas $\alpha(x = T)$ and $\alpha(x = F)$ have no more than $n - 1$ variables.

This algorithm can also run in exponential time.

8.2.3. Resolution

More notation: If a is a literal, then define $\overline{a}$ to be the negation of a. I.e., if a is the assertion of the variable x, then $\overline{a}$ is the assertion of $\neg x$, and vice versa.

Disjunctive clauses $a_1 \vee a_2 \vee \cdots \vee a_n$ are represented as sets $\{a_1, a_2, \ldots, a_n\}$. A CNF-formula $\bigwedge_{i=1}^{m} \bigvee_{j=1}^{n_i} a_{ij}$ is represented as a set of sets $\{\{a_{11}, a_{12}, \ldots, a_{1n_1}\}, \ldots, \{a_{m1}, a_{m2}, \ldots, a_{mn_m}\}\}$.

A CNF-formula $\mathcal{F} = \{C_1, C_2, \ldots, C_n\}$ is satisfiable if there is a set S of literals with the following two properties:

1. $a \in S \Rightarrow \overline{a} \notin S$
2. $\forall i, S \cap C_i \neq \emptyset$

Such an S is called a *partial satisfying assignment.*

Let C and D be two clauses such that $a \in C$ and $\overline{a} \in D$. Then the *resolvent* of C and D by a is $(C - \{a\}) \cup (D - \{\overline{a}\})$ (essentially the same as Shoenfield's rule 5).

Lemma 8.4: Let $\mathcal{F}$ be a CNF-formula, and let C and D be two clauses in $\mathcal{F}$ such that $a \in C$ and $\overline{a} \in D$. Then $\mathcal{F}$ is satisfiable $\Leftrightarrow \mathcal{F} \cup \{E\}$ is satisfiable, where E is the resolvent of C and D by a.

Proof: ($\Leftarrow$) Trivial.

($\Rightarrow$) Let S be a partially satisfying assignment for $\mathcal{F}$. S cannot contain both a and $\overline{a}$. If $a \notin S$ then $(C - \{a\}) \cap S \neq \emptyset$, so $E \cap S \neq \emptyset$. If $\overline{a} \notin S$ then $(D - \{\overline{a}\}) \cap S \neq \emptyset$, so $E \cap S \neq \emptyset$. □

Let $\mathcal{F}$ be a CNF-formula. Construct a chain $\mathcal{F} = \mathcal{F}_0 \subseteq \mathcal{F}_1 \subseteq \cdots$, where each $\mathcal{F}_i$ is obtained by applying resolution to two clauses of $\mathcal{F}_{i-1}$. The chain eventually terminates with some $\mathcal{F}_m$. $\mathcal{F}_m$ is called the *resolution closure* of $\mathcal{F}$, and is denoted $r(\mathcal{F})$.

Theorem 8.5: Let $\mathcal{F}$ be a CNF-formula. Then $\mathcal{F}$ is satisfiable $\Leftrightarrow \emptyset \notin r(\mathcal{F})$.

Proof: ($\Rightarrow$) Follows from 8.4 and the second condition for satisfiability.

($\Leftarrow$) We need only show that $r(\mathcal{F})$ is satisfiable; 8.4 will take care of the rest.

Assume $\emptyset \notin r(\mathcal{F})$, and let $x_1, x_2, \ldots, x_n$ be the variables of $\mathcal{F}$. Define $\tilde{C} = \{\overline{a} : a \in C\}$ for any clause C. We now define a sequence $S_0, S_1, \ldots S_n$. Let $S_0 = \emptyset$, and for $i = 1$ to n define

$$S_i = \begin{cases} S_{i-1} \cup \{x_i\} & \text{if there exists } C \in r(\mathcal{F}) \text{ such that } \tilde{C} \subseteq S_{i-1} \cup \{\neg x_i\} \\ S_{i-1} \cup \{\neg x_i\} & \text{otherwise.} \end{cases}$$

We will show that S_n satisfies all the clauses in $\mathcal{F}$ and has no complementary literals.

By the construction, S_n cannot contain complementary literals, so it meets the first condition for being a partial satisfying assignment for $\mathcal{F}$. Now we shall prove that S_n meets the second condition by showing that for all $C \in r(\mathcal{F})$, $S_n \cap C \neq \emptyset$. It suffices to show that for all $C \in r(\mathcal{F})$, $\tilde{C} \not\subseteq S_n$. This is done in the following claim:

Claim: $\forall i \forall C \in r(\mathcal{F}), \tilde{C} \not\subseteq S_i$.

Proof of Claim: The proof is by induction on i:

Basis: $i = 0$. Let $C \in r(\mathcal{F})$. By assumption, $C \neq \emptyset$, so $\tilde{C} \neq \emptyset$, and hence $\tilde{C} \not\subseteq S_0 = \emptyset$.

Inductive step: Assume claim holds for $i - 1$. There are two cases to consider depending on whether or not x_i was chosen to be added to S_{i-1} to form S_i.

Case 1. $S_i = S_{i-1} \cup \{\neg x_i\}$. In this case we have for all $C \in r(\mathcal{F})$, $\tilde{C} \not\subseteq S_{i-1} \cup \{\neg x_i\} = S_i$. Thus we are done.

Case 2. $S_i = S_{i-1} \cup \{x_i\}$. In this case there is $C \in r(\mathcal{F})$ such that $\tilde{C} \subseteq S_{i-1} \cup \{\neg x_i\}$. By the inductive hypothesis $\tilde{C} \not\subseteq S_{i-1}$. Hence, $x_i \in C$.

For purposes of coming to a contradiction assume there is $D \in r(\mathcal{F})$ such that $\tilde{D} \subseteq S_i$. By the inductive hypothesis, $\tilde{D} \not\subseteq S_{i-1}$. Since $S_i = S_{i-1} \cup \{x_i\}$ then $\neg x_i \in D$. Thus, the set $E = (C - \{x_i\}) \cup (D - \{\neg x_i\})$ must also be in $r(\mathcal{F})$. However, both both $\tilde{C} - \{\neg x_i\}$ and $\tilde{D} - \{x_i\}$ are contained in S_{i-1}. We have $\tilde{E} = \tilde{C} - \{\neg x_i\} \cup \tilde{D} - \{x_i\}$. Hence, $\tilde{E} \subseteq S_{i-1}$, contradicting the inductive hypothesis. $\square$

Lecture 9

Propositional Modal Logic

February 5, 1998
Lecturer: Richard Ladner
Notes: Todd Millstein

9.1. Syntax and Semantics

We will be discussing the modal logic K, which is named for Saul Kripke, who analyzed it in 1963. The logic is propositional logic augmented with a new unary operator, denoted $\Box$. We can informally understand $\Box$ to mean several different things: "it is necessary that ..., ' "I know that ...," "I believe that ...," "it is true in all next states that ..."

Syntax: If α is a well-formed formula in K, then so is $\Box\alpha$.

We introduce a K-*model*, which is essentially a special kind of directed graph. Each node represents a "world" or " state" in which every boolean variable has an assignment of either true or false. An edge from state s_1 to state s_2 means that we can get from s_1 to s_2.

More formally, a K-model M is a triple (S, π, R) where

- S is a (possibly infinite) set of world states.
- π is a function mapping each world state in S to a truth assignment for all the boolean variables.
- $R \subseteq S \times S$ is a relation on S.

Semantics: Given a K-model $M = (S, \pi, R)$ and a state $s \in S$, we define the semantics of a modal logic formula inductively as follows (x denotes a boolean variable, and α and β denote arbitrary modal logic formulas):

- $M, s \models x$ if $\pi(s)(x) = \mathrm{T}$
- $M, s \models \neg\alpha$ if $M, s \not\models \alpha$
- $M, s \models \alpha \vee \beta$ if $M, s \models \alpha$ or $M, s \models \beta$
- $M, s \models \Box\alpha$ if $\forall t \in S, (s,t) \in R \Rightarrow M, t \models \alpha$

The other propositional connectives $(\wedge, \Rightarrow, \equiv)$ can be defined in terms of the above connectives in the usual way. In addition, we can define a new operator, $\Diamond$, defined as follows: $\Diamond\alpha \equiv \neg\Box\neg\alpha$. The intuitive meaning is that we can get to a state where α is true from the current state.

As an example, consider the K-model $M = (S, \pi, R)$ defined as follows:

- $S = \{s_1, s_2, s_3\}$

- $\pi = \{(s_1, \{(x, T), (y, T)\}), (s_2, \{(x, F), (y, T)\}), (s_3, \{(x, T), (y, T)\})\}$
- $R = \{(s_1, s_2), (s_1, s_3)\}$

In this example, $M, s_1 \models y \wedge \Box y$, but $M, s_1 \not\models x \wedge \Box x$. Also, $M, s_1 \models \Diamond x$.

9.2. K-validity and K-satisfiability

As in propositional logic, we can define notions of validity and satisfiability. K-validity is defined as follows:

$$\models_K \alpha \text{ if for all } K\text{-models } M = (S, \pi, R) \text{ and states } s \in S,\ M, s \models \alpha$$

Correspondingly, we say that a formula α is K-satisfiable if there exist a K-model $M = (S, \pi, R)$ and a state $s \in S$ such that $M, s \models \alpha$. Finally, K-model checking is the problem of deciding, given a K-model $M = (S, \pi, R)$, a state $s \in S$, and a formula α, whether $M, s \models \alpha$.

An example of K-validity is $\models_K (\Box\alpha \wedge \Box(\alpha \Rightarrow \beta)) \Rightarrow \Box\beta$. A non-example of K-validity is $\not\models_K (\Diamond\alpha \wedge \Diamond(\alpha \Rightarrow \beta)) \Rightarrow \Diamond B$.

Finally, a meta-rule relating the K-validity of two formulas is the "Rule of Necessity": if $\models_K \alpha$ then $\models_K \Box\alpha$.

Now we prove a complexity result about K-satisfiability that was shown by Ladner in 1977 [11]

Theorem 9.1: K-satisfiability is PSPACE-complete for log-space reductions.

Proof: We don't show here that K-satisfiability $\in$ PSPACE because it is a fairly difficult proof. Therefore, we only prove that any language $L \in$ PSPACE can be reduced to K-satisfiability in log-space. In particular, we'll show that B_ω, the set of true Quantified Boolean Formulas, can be log-space reduced to K-satisfiability. The result then follows from the fact that B_ω is PSPACE-complete.

Assume we are given a quantified boolean formula $F_{B_\omega} = Q_1 x_1 \ldots Q_k x_k A$, where each $Q_i \in \{\exists, \forall\}$. Further assume the formula is in prenex form, so A contains no quantifiers. We will create a modal logic formula F_K which is K-satisfiable if and only if F_{B_ω} is true.

The informal idea of the construction is as follows: The modal logic formula which we build will force a certain structure on any K-model and state s which K-satisfies it. Starting from s, the K-model's branching must mimic the order of quantifiers in F_{B_ω}. In particular, the construction will ensure that, when a universal quantifier is encountered, the K-model will be forced to branch from the current state, setting the associated quantified variable to true and false respectively on each branch. Similarly, for every existential quantifier encountered, the K-model will be forced to choose an appropriate value for the associated variable at that point in the K-model.

In this way, the construction requires k levels of states above s, one level per quantifier of F_{B_ω}. The levels are indexed 0 to k, with s at level 0. At level k, we simply need to ensure that the boolean formula A holds. To keep track of the properties that must hold at each level, we introduce a set of k *level variables* $y_0, y_1, \ldots, y_k$. The construction enforces that y_i is true iff we are currently at level i in the K-model.

A convenient definition for the construction of F_K is the following:

$$\Box^{(m)}\alpha = \alpha \wedge \Box\alpha \wedge \Box\Box\alpha \ldots \wedge \Box^m\alpha$$

Now we formally describe the construction. F_K is defined to be the conjunction of the following modal logic formulas (some of these are actually modal logic schemas, to be instantiated for all i such that $0 \leq i < k$):

1. $y_0 \wedge \neg y_1 \wedge \ldots \wedge \neg y_k$ This ensures that the initial state s is considered to be in level 0.

2. $\Box^{(k)}(y_i \Rightarrow \Box(\neg y_0 \wedge \ldots \wedge \neg y_i \wedge y_{i+1} \wedge \neg y_{i+2} \wedge \ldots \wedge \neg y_k))$ This ensures that y_i and only y_i is true at level i.

3. $\Box^{(k)}(y_i \Rightarrow ((x_i \Rightarrow \Box^{(k)} x_i) \wedge (\neg x_i \Rightarrow \Box^{(k)} \neg x_i)))$ This ensures that the truth value of x_i at some state in level i remains the truth value of x_i for all transitively reachable states in subsequent levels.

4. $\Box^{(k)}(y_i \Rightarrow (\Diamond(y_{i+1} \wedge x_{i+1}) \vee \Diamond(y_{i+1} \wedge \neg x_{i+1})))$, for all i such that $Q_{i+1} = \exists$. This ensures that each state at level i branches to at least one state in the $(i+1)$st level. This thereby mimics the behavior of the existentially quantified variable x_{i+1} of F_{B_ω}.

5. $\Box^{(k)}(y_i \Rightarrow (\Diamond(y_{i+1} \wedge x_{i+1}) \wedge \Diamond(y_{i+1} \wedge \neg x_{i+1})))$, for all i such that $Q_{i+1} = \forall$. This ensures that each state at level i branches to at least two states in the $(i+1)$st level: one which sets x_{i+1} to true and another which sets it to false. This thereby mimics the behavior of the universally quantified variable x_{i+1} of F_{B_ω}.

6. $\Box^{(k)}(y_k \Rightarrow A)$ This ensures that A holds at the last level.

This completes the construction. By the comments next to each subformula above, the correctness of the construction should be intuitively clear. We can easily see that each of the subformulas above are polynomial size with respect to the length of F_{B_ω}. Further, these formulas can be generated by a machine using only space logarithmic in the length of F_{B_ω}, so the reduction is log-space. □

Exercise 9.2 (Thought Problem): Build a modal logic formula F such that F is K-satisfiable and all K-models which K-satisfy F have a number of states exponential in the length of F.

Solution: We can use ideas from the construction in the above theorem. In particular, the simulation of a universal quantifier requires the K-satisfying K-model to double in size because it must branch to at least two states: one in which the quantified variable is true and one in which it is false. Intuitively, a succession of such branches will give us the exponential size.

In particular, define F to be the conjunction of the following modal logic formulas:

1. $y_0 \wedge \neg y_1 \wedge \ldots \wedge \neg y_k$ This ensures that the state s which K-satisfies F is considered to be in level 0.

2. $\Box^{(k)}(y_i \Rightarrow \Box(\neg y_0 \wedge \ldots \wedge \neg y_i \wedge y_{i+1} \wedge \neg y_{i+2} \wedge \ldots \wedge \neg y_k))$ This ensures that all states that are i edges away from s are considered to be in level i.

3. $\Box^{(k)}(y_i \Rightarrow (\Diamond x_{i+1} \wedge \Diamond \neg x_{i+1}))$ This ensures that a K-satisfying K-model must do a "universal branch" from each state at level i. That is, from each state at level i, there is an edge to a state in which x_{i+1} is true and an edge to a state in which x_{i+1} is false.

4. $\Box^{(k)}(y_i \Rightarrow ((x_i \Rightarrow \Box^{(k)} x_i) \wedge (\neg x_i \Rightarrow \Box^{(k)} \neg x_i)))$ This ensures that the truth value of variable x_i at level i remains the truth value of x_i for all subsequent levels.

This completes the construction. To K-satisfy this formula, a K-model has to have a state s from which we can transitively reach one state in level k per possible truth assignment of the variables $x_1, \ldots, x_k$. Since there are 2^k such assignments, this gives us the exponential lower bound.

One caveat about this construction is that, although a K-satisfying K-model will be exponential in the number of variables of F, F is not linear in the number of variables. For example, the formulas in the above construction that start with $\Box^{(k)}$ have size at least $\Omega(k^2)$. Therefore, a challenge problem is to find a construction such that the K-model is still exponential, but F is linear in the number of variables.

A closely related logic is the so-called logic of knowledge where the modal operator is modeled by an equivalence relation. Logic of knowledge is investigated with its application in the excellent book by Fagin *et al.* [5].

Lecture 10

K-satisfiability and Linear Time Logic

February 10, 1998
Lecturer: Richard Ladner
Notes: Jason Hartline

10.1. K-Satisfiability is in PSPACE

What is decribed here is basically a space efficient version of the original algorithm of Kripke [10].

Theorem 10.1: K-satisfiability is in PSPACE.

Proof: We will be using a Tableau method which is often used Modal and Temporal Logic. We will be using $(\mathcal{T}, \mathcal{F}, \mathcal{T}', \mathcal{F}')$. Intuitively these sets represent:

$$\begin{aligned} \alpha \in \mathcal{T} &\Rightarrow \alpha \text{ is true.} \\ \alpha \in \mathcal{F} &\Rightarrow \alpha \text{ is false.} \\ \alpha \in \mathcal{T}' &\Rightarrow \alpha \text{ is true in every next world.} \\ \alpha \in \mathcal{F}' &\Rightarrow \alpha \text{ is false in some next world.} \end{aligned}$$

K-SAT$(\mathcal{T}, \mathcal{F}, \mathcal{T}', \mathcal{F})$ returns true if and only if there is a K-model, M, and a state, s, such that

$$M, s \models \bigwedge_{\alpha \in \mathcal{T}} \alpha \wedge \bigwedge_{\alpha \in \mathcal{F}} \neg\alpha \wedge \bigwedge_{\alpha \in \mathcal{T}'} \Box\alpha \wedge \bigwedge_{\alpha \in \mathcal{F}'} \neg\Box\alpha$$

We now give the following alternating polynomial time algorithm for *K-SAT*. *VAR* is the set of variables.

K-SAT$(\mathcal{T}, \mathcal{F}, \mathcal{T}', \mathcal{F}')$
    if $\mathcal{T} \cup \mathcal{F} \not\subseteq VAR$
        choose $\alpha \in (\mathcal{T} \cup \mathcal{F}) - VAR$
        case
            $\alpha = \neg\beta$ and $\alpha \in \mathcal{T}$: return *K-SAT*$(\mathcal{T} - \{\alpha\}, \mathcal{F} \cup \{\beta\}, \mathcal{T}', \mathcal{F}')$
            $\alpha = \neg\beta$ and $\alpha \in \mathcal{F}$: return *K-SAT*$(\mathcal{T} \cup \{\beta\}, \mathcal{F} - \{\alpha\}, \mathcal{T}', \mathcal{F}')$
            $\alpha = \beta \vee \gamma$ and $\alpha \in \mathcal{T}$:
                return *K-SAT*$(\mathcal{T} - \{\alpha\} \cup \{\beta\}, \mathcal{F}, \mathcal{T}', \mathcal{F}') \vee$ *K-SAT*$(\mathcal{T} - \{\alpha\} \cup \{\gamma\}, \mathcal{F}, \mathcal{T}', \mathcal{F}')$
            $\alpha = \beta \vee \gamma$ and $\alpha \in \mathcal{F}$: return *K-SAT*$(\mathcal{T}, \mathcal{F} - \{\alpha\} \cup \{\beta, \gamma\}, \mathcal{T}', \mathcal{F}')$
            $\alpha = \Box\beta$ and $\alpha \in \mathcal{T}$: return *K-SAT*$(\mathcal{T} - \{\alpha\}, \mathcal{F}, \mathcal{T}' \cup \{\beta\}, \mathcal{F}')$
            $\alpha = \Box\beta$ and $\alpha \in \mathcal{F}$: return *K-SAT*$(\mathcal{T}, \mathcal{F} - \{\alpha\}, \mathcal{T}', \mathcal{F}' \cup \{\beta\})$

endcase
else if $\mathcal{T} \cap \mathcal{F} \neq \emptyset$
return *false*
else
return $\bigwedge_{\beta \in \mathcal{F}'} \mathit{K\text{-}SAT}(\mathcal{T}', \{\beta\}, \emptyset, \emptyset)$ (true if $\mathcal{F}' = \emptyset$)
fi

To see if statement α is satisfiable, we then call $\mathit{K\text{-}SAT}(\{\alpha\}, \emptyset, \emptyset.\emptyset)$.

This algorithm is polynomial time because it reduces the size of $|\mathcal{T}| + |\mathcal{F}| + |\mathcal{T}'| + |\mathcal{F}'|$ by 1 at least once every two recursive steps. $|\mathcal{X}|$ denotes the sum of the lengths of the formulas in $\mathcal{X}$. $\square$

10.1.1. Example

$\neg(\Box x \vee \Box\neg x)$ is satisfiable:

$$
\begin{array}{cccc}
\mathcal{T} & \mathcal{F} & \mathcal{T}' & \mathcal{F}' \\
\{\neg(\Box x \vee \Box\neg x)\} & \emptyset & \emptyset & \emptyset \\
 & \downarrow & & \\
\emptyset & \{\Box x \vee \Box\neg x\} & \emptyset & \emptyset \\
 & \downarrow & & \\
\emptyset & \{\Box x, \Box\neg x\} & \emptyset & \emptyset \\
 & \downarrow & & \\
\emptyset & \{\Box\neg x\} & \emptyset & \{x\} \\
 & \downarrow & & \\
\emptyset & \emptyset & \emptyset & \{x, \neg x\} \\
 & \swarrow & \searrow &
\end{array}
$$

$$
\begin{array}{cccc}
\emptyset & \{x\} & \emptyset & \emptyset \\
 & \downarrow & & \\
 & \mathit{true} & &
\end{array}
\qquad
\begin{array}{cccc}
\emptyset & \{\neg x\} & \emptyset & \emptyset \\
 & \downarrow & & \\
\{x\} & \emptyset & \emptyset & \emptyset \\
 & \downarrow & & \\
 & \mathit{true} & &
\end{array}
$$

10.1.2. K-model checking is in P (in the size of the model and formula).

Make a tableau for each state in the model and fill in whether a particular statement is true or false at that state starting inductively with small length statements.

10.2. Linear Time Logic: *LTL*

A good resource for temporal logic is Emerson's article in the *Handbook of Theoretical Computer Science, Vol. B, Formal Models and Semantics* [2]. A fine paper that contains most of the results covered here about linear time temporal logic is by Sistla and Clarke [14]

10.2.1. The *LTL* Model

Our model is a sequence of states such that every state looks like every other state. We will introduce two new unary operators and one new binary operator:

$$\circ \rightarrow \circ \rightarrow \circ \rightarrow \circ \rightarrow \circ \rightarrow \circ \rightarrow \circ \rightarrow \circ \rightarrow \circ \rightarrow \cdots$$

$$\begin{array}{rcl} G\alpha & \Rightarrow & \textit{globally } \alpha \text{ is true.} \\ X\alpha & \Rightarrow & \text{in the } \textit{next} \text{ state } \alpha \text{ is true.} \\ \alpha U\beta & \Rightarrow & \alpha \text{ is true } \textit{until } \beta \text{ is true.} \end{array}$$

Relative to our model, these operators represent the following scenarios.

$$\begin{array}{rcccccccccccccc}
 & & & & G\alpha & & & & & & & & & & \\
G\alpha & : & \circ & \rightarrow & \circ & \rightarrow & \circ & \rightarrow & \circ & \cdots & & & & & \\
 & & & & \alpha & & \alpha & & \alpha & & & & & & \\
 & & & & X\alpha & & & & & & & & & & \\
X\alpha & : & \circ & \rightarrow & \circ & \rightarrow & \circ & \rightarrow & \circ & \cdots & & & & & \\
 & & & & & & \alpha & & & & & & & & \\
 & & & & \alpha U\beta & & & & & & & & & & \\
\alpha U\beta & : & \circ & \rightarrow & \circ & \rightarrow & \circ & \cdots & \circ & \rightarrow & \circ & \rightarrow & \circ & \cdots \\
 & & & & \alpha & & \alpha & & \alpha & & \beta & & & \\
 & & & & \alpha U\beta & & & & & & & & & & \\
 & & \circ & \rightarrow & \circ & \rightarrow & \circ & \cdots & & & & & & & \\
 & & & & \beta & & & & & & & & & &
\end{array}$$

10.2.2. Semantics

An *LTL* model is $M = (S, \pi, x)$ where S is a set of states, π is an assignment that takes $s \in S$ to variable assignments, and x is an infinite path through the states $s_0 s_1 s_2 \ldots$.

$$x^i \stackrel{\text{def}}{=} s_i s_{i+1} s_{i+2} \ldots$$

This can then be defined inductively as:

$$\begin{array}{rcl} x^0 & = & x \\ x^i & = & s_i x^{i+1} \end{array}$$

We will use the following semantics. Note that since every state is like any other state we can pick our starting state to be s_0.

$$\begin{array}{rcl} M, x & \models & z \text{ if } \pi(s_0)(z) = \textit{true} \\ M, x & \models & \alpha \vee \beta \text{ if } M, x \models \alpha \text{ or } M, x \models \beta \end{array}$$

$$\begin{aligned}
M, x &\models \neg\alpha \text{ if } M, x \not\models \alpha \\
M, x &\models G\alpha \text{ if } \forall i \geq 0\ M, x^i \models \alpha \\
M, x &\models X\alpha \text{ if } M, x^1 \models \alpha \\
M, x &\models \alpha U \beta \text{ if } \exists j \geq 0\ M, x^j \models \beta \text{ and } \forall i < j\ M, x^i \models \alpha
\end{aligned}$$

An *LTL* statement α is valid, denoted $\models_{\text{LTL}} \alpha$, if and only if, for all $M - (S, \pi, x)$, $M, x \models \alpha$. Now we will introduce some new operators that are defined in terms of G, and U.

$$\begin{aligned}
F\alpha &\stackrel{\text{def}}{=} \neg G \neg\alpha \quad \text{“eventually } \alpha\text{”} \\
\alpha B \beta &\stackrel{\text{def}}{=} \neg(\neg\alpha U \beta) \quad \text{“}\alpha \text{ before } \beta\text{”}
\end{aligned}$$

Curiously: $\models_{\text{LTL}} F\alpha \equiv \mathit{true}\, U \alpha$

10.2.3. *LTL* validities

The following validities push negations towards variables to transform a formula into a “positive” form where only variables are negated.

$$\begin{aligned}
&\models_{\text{LTL}} \neg G\alpha \equiv F\neg\alpha \\
&\models_{\text{LTL}} \neg F\alpha \equiv G\neg\alpha \\
&\models_{\text{LTL}} \neg(\alpha U \beta) \equiv \neg\alpha B \beta \\
&\models_{\text{LTL}} \neg X\alpha \equiv X\neg\alpha
\end{aligned}$$

Here are some more validities of interest:

$$\begin{aligned}
&\models_{\text{LTL}} FF\alpha \equiv F\alpha \\
&\models_{\text{LTL}} GG\alpha \equiv G\alpha \\
&\models_{\text{LTL}} \alpha U \beta \equiv \beta \vee [X(\alpha U \beta) \wedge \alpha] \\
&\models_{\text{LTL}} \alpha B \beta \equiv \neg\beta \wedge [\alpha \vee X(\alpha B \beta)] \\
&\models_{\text{LTL}} G\alpha \equiv \alpha \wedge XG\alpha \\
&\models_{\text{LTL}} F\alpha \equiv \alpha \vee XF\alpha
\end{aligned}$$

Lecture 11

Linear Time Logic

February 12, 1998
Lecturer: Richard Ladner
Notes: Vassily Litvinov

11.1. LTL-satisfiability is PSPACE-complete

We will prove the theorem stated in the previous lecture:

Theorem 11.1: LTL-satisfiability is PSPACE-complete for log-space reductions.

Proof:

PSPACE-hardness. Given a Turing machine $T = (S, \Sigma, \Gamma, \delta, s)$ that runs in polynomial space, construct an LTL formula α which is satisfiable if and only if T accepts its input. We will assume that some $A \in S$ is the only accepting state and that once T gets in A it stays in it and keeps running forever.

Intuitively a satisfying model will represent an infinite sequence of configurations of T: $\#c_0\#c_1\#c_2\ldots$ where each c_i has the same length polynomial in the length of input. The LTL formula we will build will check that this string is a legal computation of T. The following properties need to be checked:

- c_0 is the initial configuration
- c_{i+1} follows c_i correctly
- some c_i is an accepting configuration

Define $\Gamma' = \Gamma \cup S \cup \{\#\}$; $m = |c_i| + 2$ is the length of a configuration; $w = w_1 \ldots w_n$ is the input. States in the model will represent individual characters σ_i which together form the infinite string of T's configurations: $\sigma_0\sigma_1\sigma_2\ldots \equiv \#c_0\#c_1\ldots$. The properties listed above can be restated in terms of $\sigma_0\sigma_1\sigma_2\ldots$ as follows:

- $\sigma_0 \ldots \sigma_m$ is the initial configuration
- $\forall i. f(\sigma_i, \sigma_{i+1}, \sigma_{i+2}, \sigma_{i+3}) = \sigma_{i+m}$, where f is the function that computes a symbol in the subsequent configuration of T given the corresponding four-window in the previous configuration
- $\exists i. \sigma_i = A$

All we need to do is just check these properties, because the only way for them to hold is when they represent an accepting computation of T.

The variables in our model will be z_σ for all $\sigma \in \Gamma'$. If z_τ is true in the ith state in the path, then $\sigma_i = \tau$. Now build the formula to be satisfied as the conjunction of the following:

- one symbol per position:

$$G \left(\bigvee_{\sigma\in\Gamma'} z_\sigma \wedge \bigwedge_{\sigma\in\Gamma'} \bigwedge_{\tau\in\Gamma'-\{\sigma\}} (z_\sigma \Rightarrow \neg z_\tau) \right)$$

- the first m states model the initial configuration:

$$z_\# \wedge X z_S \wedge X^2\, z_{w_1} \wedge \ldots \wedge X^{n+1}\, z_{w_n} \wedge X^{n+2}\, z_\sqcup \wedge \ldots \wedge X^{m-2}\, z_\sqcup \wedge X^{m-1}\, z_\#$$

- the computation of T is modeled correctly:

$$G \left(\bigwedge_{\sigma_0,\sigma_1,\sigma_2,\sigma_3\in\Gamma'} (z_{\sigma_0} \wedge X z_{\sigma_1} \wedge X^2\, z_{\sigma_2} \wedge X^3\, z_{\sigma_3} \Rightarrow X^m\, z_{f(\sigma_0,\sigma_1,\sigma_2,\sigma_3)}) \right)$$

- there is an accepting configuration

$$F\, z_A$$

This formula can be produced in log space. So, PSPACE-hardness has been shown.

LTL-SAT is in PSPACE. Given an LTL formula α, check if it is satisfiable. We will use a tableau method. The nondeterministic algorithm will discover an infinite sequence of states (that satisfies the formula) as a finite sequence with a loop. These days they use automata on infinite strings or trees (for branching time logic). We assume that we have only X and U as modal operators because G can be defined in terms of U.

Definition 11.2: A *tableau* for α is a pair $(\mathcal{T}, \mathcal{F})$ with the following properties:

- every subformula of α is in $\mathcal{T} \cup \mathcal{F}$ ("completeness")
- $\mathcal{T} \cap \mathcal{F} = \phi$ ("consistency")
- ("semantic consistency")

  $\neg\beta \in \mathcal{T} \Rightarrow \beta \in \mathcal{F}$
  $\neg\beta \in \mathcal{F} \Rightarrow \beta \in \mathcal{T}$
  $\beta \vee \gamma \in \mathcal{T} \Rightarrow \beta \in \mathcal{T}$ or $\gamma \in \mathcal{T}$
  $\beta \vee \gamma \in \mathcal{F} \Rightarrow \beta, \gamma \in \mathcal{F}$
  (remember $\beta\, U\, \gamma \equiv \gamma \vee (\beta \wedge X(\beta\, U\, \gamma))$
  $\beta\, U\, \gamma \in \mathcal{T}$ and $\gamma \in \mathcal{F} \Rightarrow \beta \in \mathcal{T}$

$\beta\ U\ \gamma \in \mathcal{F} \Rightarrow \gamma \in \mathcal{F}$

(other operators can be expressed in terms of the ones above)

- define relation $Next$ on two tableaus: $Next((\mathcal{T}, \mathcal{F}), (\mathcal{T}', \mathcal{F}'))$ if the following conditions hold:

  $X\alpha \in \mathcal{T} \Rightarrow \alpha \in \mathcal{T}'$

  $X\alpha \in \mathcal{F} \Rightarrow \alpha \in \mathcal{F}'$

  $\beta\ U\ \gamma \in \mathcal{T}$ and $\gamma \in \mathcal{F} \Rightarrow \beta\ U\ \gamma \in \mathcal{T}'$

  $\beta\ U\ \gamma \in \mathcal{F}$ and $\beta \in \mathcal{T} \Rightarrow \beta\ U\ \gamma \in \mathcal{F}'$

We will also define $untils(\mathcal{T})$ to be all U-formulas in $\mathcal{T}$.

To check if a is satisfiable, our algorithm will nondeterministically guess the start state as some tableau $(\mathcal{T}_0, \mathcal{F}_0)$ and the loop state $(\mathcal{T}_L, \mathcal{F}_L)$. It will then check that there is a path from $(\mathcal{T}_0, \mathcal{F}_0)$ to $(\mathcal{T}_L, \mathcal{F}_L)$ so that $Next$ holds on all pairs of adjacent tableaus. For each $\beta\ U\ \gamma \in untils(\mathcal{T}_L)$ there needs to be a state that satisfies γ. So the algorithm will make sure that there exists a loop around $(\mathcal{T}_L, \mathcal{F}_L)$ that does that.

More precisely, the algorithm does the following:

Guessa tableau $(\mathcal{T}_0, \mathcal{F}_0)$ such that $\alpha \in \mathcal{T}_0$ (and the tableau properties hold)
Guess a tableau $(\mathcal{T}_L, \mathcal{F}_L)$;
$(\mathcal{T}, \mathcal{F}) := (\mathcal{T}_0, \mathcal{F}_0)$;
while $(\mathcal{T}, \mathcal{F}) \neq (\mathcal{T}_L, \mathcal{F}_L)$ do
    $(\mathcal{T}, \mathcal{F}) := MOVE((\mathcal{T}, \mathcal{F}))$;
$(\mathcal{T}, \mathcal{F}) := MOVE((\mathcal{T}, \mathcal{F}))$;
$E := untils(\mathcal{T}_L)$;
while not ($E = \phi$ and $(\mathcal{T}, \mathcal{F}) = (\mathcal{T}_L, \mathcal{F}_L)$) do
    if $\beta\ U\ \gamma \in E$ and $\gamma \in \mathcal{T}$ then $E := E - \{\beta\ U\ \gamma\}$;
    $(\mathcal{T}, \mathcal{F})$:= $MOVE((\mathcal{T}, \mathcal{F}))$;
Accept

The operation $MOVE((\mathcal{T}, \mathcal{F}))$ is defined by the following procedure:

Guess a tableau $(\mathcal{T}', \mathcal{F}')$;
if$Next((\mathcal{T}, \mathcal{F}), (\mathcal{T}', \mathcal{F}'))$ then return $(\mathcal{T}', \mathcal{F}')$ else halt (rejecting this path)

Problem: this algorithm may not terminate. Solution: add a counter to count up to the number of all possible tableaus. If the counter is up and the `while`-loop is not done yet, it means that the loop is infinite and the input is to be rejected. (For the second `while`-loop we need to restart the counter every time a formula is removed from E.

□

Example 11.3: Here's an example of a run of the algorithm. Let

$$\alpha = z\ U\ X\neg z \qquad - \textit{a little bugger}$$

the start state :	$\mathcal{T}_0$	$\mathcal{F}_0$
	$z\ U\ X\neg z$	$X\neg z$
	z	$\neg z$
the loop state :	$\mathcal{T}_L$	$\mathcal{F}_L$
	$z\ U\ X\neg z$	
	z	$\neg z$
	$X\neg z$	

The algorithm proceeds with states as follows:

the start state :	$\mathcal{T}_0$	$\mathcal{F}_0$
	$z\ U\ X\neg z$	$X\neg z$
	z	$\neg z$
next hit the loop state :	$\mathcal{T}$	$\mathcal{F}$
$E = \{z\ U\ X\neg z\}$	$z\ U\ X\neg z$	
	z	$\neg z$
	$X\neg z$	
one more state in the loop :	$\mathcal{T}$	$\mathcal{F}$
$E := E - \{z\ U\ X\neg z\} \to \phi$	$z\ U\ X\neg z$	$X\neg z$
	$\neg z$	z
came back to the loop state :	$\mathcal{T}$	$\mathcal{F}$
	$z\ U\ X\neg z$	
	z	$\neg z$
	$X\neg z$	

11.2. Model Checking

Remember Kripke model is (S, π, R); S is an (infinite) set of states, π is the truth assignment for variables at every state, R is a relation between states in S. Note that R defines a directed graph with states as nodes and thus defines paths as sequences of states.

Definition 11.4: LTL-Model Checking.
$M, s_0 \models \alpha$ if for every infinite path x starting at s_0, $M, x \models \alpha$. (Finite paths are ignored, the assumption being that in actual applications no path ends.)

Theorem 11.5: LTL-Model Checking is PSPACE-complete.

The proof is rather difficult. Compare with K-model checking which is in P.

11.3. Branching Time Logic

X, U, G - path operators.
E ("exists") and A ("forall") are new state operators.
State Formulas:

- include variables (denoted by z's)
- are closed under $\vee$, $\neg$ and under E, A

Path Formulas:

- include state formulas
- are closed under X, U, G

Semantics of state operators.

- $M, s \models z$ if $\pi(s)(z) = T$
- $M, s \models \neg\alpha$ same as before
- $M, s \models \alpha \vee \beta$ same as before
- $M, s \models E\alpha$ if there exists path x starting at s such that $M, x \models \alpha$ (LTL-relation)
- $M, s \models A\alpha$ if for all paths x starting at s it is true that $M, x \models \alpha$

Semantics of path operators. These are exactly as in LTL.

Lecture 12

CTL and First-order Predicate Logic

February 17, 1998
Lecturer: Richard Ladner
Notes: Todd Millstein

12.1. Computational Tree Logic

12.1.1. CTL and CTL*

The logics CTL and CTL* were first defined formally by Clark, Emerson, and Sistla [1]. CTL includes two kinds of formulas. The *path* formulas talk about paths in a model. These formulas are closed under the linear-time logic operators X, U, and G. The *state* formulas talk about states in the model. These formulas are closed under the ordinary boolean connectives as well as two new operators, E ("exists") and A ("for all"). The semantics of these operators were given in the previous lecture.

In CTL, there is a restriction that each path operator in a formula must be immediately preceded by a state operator. This ensures that path formulas can never be evaluated with respect to a state, but only with respect to a path. CTL* drops this restriction, providing a semantics for evaluating a path formula with respect to a state.

As an example, consider the formula $AGEF\alpha$. This is in CTL because each path operator (G and F) is directly preceded by a state operator (A and E, respectively). The formula is true in model M at state s if from every state on every path from s (AG), there exists a path on which α is eventually true ($EF\alpha$).

On the other hand, consider the formula $AGF\alpha$. This is not in CTL because G directly precedes F, but it is in CTL*. The formula is true in model M at state s if from every state on every path from s (AG), α is eventually true ($F\alpha$) *on that same path*.

12.1.2. Complexity Results

The following table summarizes some complexity results for the temporal logics we have discussed.

	model checking	*satisfiability*
LTL	PSPACE-complete [14]	PSPACE-complete [14]
CTL	P [1]	EXPTIME-complete [8]
CTL*	PSPACE-complete [1]	2-EXPTIME-complete [3, 15]

As we can see, the syntactic restriction imposed by CTL makes it tractable, as model-checking takes polynomial time instead of polynomial space. We will informally discuss this algorithm. Like other model-checking algorithms we've seen, it uses the *tableau* method.

We are given a formula α to check, a model $M = (S, \pi, R)$ (defined as in previous lectures), and a state s_0 in the model. Each state $s \in S$ has a set T_s of true formulas and F_s of false formulas. These sets initially contain the true and false variables at that state, respectively, and we use this information to derive truth values for all subformulas of α at each state.

As an example, here are the rules for deriving the truth value of $E(\beta U \gamma)$ in some state s:

- $\gamma \in T_s \longrightarrow E(\beta U \gamma) \in T_s$
- $\beta \in F_s, \gamma \in F_s \longrightarrow E(\beta U \gamma) \in F_s$
- $(\beta \in T_s, \exists s'((s,s') \in R, E(\beta U \gamma) \in T_{s'})) \longrightarrow E(\beta U \gamma) \in T_s$
- $(\gamma \in F_s, \forall s'((s,s') \in R \Rightarrow E(\beta U \gamma) \in F_{s'})) \longrightarrow E(\beta U \gamma) \in F_s$

There are similar rules for the ordinary boolean connectives and for operators of the form $EX\beta$, $AX\beta$, and $A(\beta U \gamma)$ (all other operators can be defined in terms of these operators). The algorithm applies the rules until no more can be applied and then checks whether $\alpha \in T_{s_0}$. This can be done in polynomial time.

There are two final comments about modal and temporal logics, regarding areas that we have not explored. First, although we have stressed algorithms such as the above one, there are also classical proof systems for K and CTL (and LTL? and CTL*?). Second, there is a close connection between modal and temporal logics and automata on infinite objects.

12.2. First-order Predicate Logic

An excellent resource for the classical theory of first-order logic is Enderton [4].

12.2.1. Syntax

We will be studying first-order predicate logic with equality. That is, the symbol "=" always denotes equality. The basic elements of the logic are:

- individual variables $x, y, z, \ldots$ These can take on arbitrary values in some domain.
- function symbols $f, g, h, \ldots$ Each function has an associated arity. Note that a 0-ary function is simply a constant.
- relation symbols $R, S, T, \ldots$ Each relation has an associated arity. Note that the two 0-ary relations are simply the booleans *true* and *false*. Further, a unary relation is just a set.

¿From these primitives, we can build the syntax of the language inductively. *Terms* are defined as follows:

- Any variable is a term.
- If $t_1, t_2, \ldots, t_k$ are terms and f is a k-ary function symbol, then $f(t_1, t_2, \ldots, t_k)$ is a term.

From terms, we can define *formulas* in the logic:

- If $t_1, t_2, \ldots, t_k$ are terms and R is a k-ary relation symbol, then $R(t_1, t_2, \ldots, t_k)$ is a formula.

- If t_1 and t_2 are terms, then $t_1 = t_2$ is a formula.
- If α is a formula, then so is $\neg\alpha$.
- If α and β are formulas, then so is $\alpha \vee \beta$.
- If α is a formula, then so is $\exists x.\alpha$.

Finally, a *sentence* is a formula containing no free variables.

12.2.2. Semantics

A *language* L is a set of function and relation symbols. A *structure* (or *model*, or *interpretation*) for a language L is a pair $M = (U^M, I^M)$, where $U^M \neq \phi$. U^M defines the domain over which variables range, and I^M gives semantics for each of the function and relation symbols in L.

For example, consider the language $L = \{+, <, 0\}$. We can define a model $\mathbb{Z} = (U^{\mathbb{Z}}, I^{\mathbb{Z}})$, where $U^{\mathbb{Z}} = \{\dots, -1, 0, 1, \dots\}$ and $I^{\mathbb{Z}}$ defines $+^{\mathbb{Z}}$ to be integer addition, $<^{\mathbb{Z}}$ to be integer less-than, and $0^{\mathbb{Z}}$ to be the integer constant 0. Clearly $\mathbb{Z}$ is a model for the integers.

Because formulas can contain free variables, we need an associated *assignment* σ in a model M, which maps each variable to an element of U^M. The assignment σ can then be inductively extended in the obvious way to interpret all terms:

$$\sigma(f(t_1, t_2, \dots, t_k)) = f^M(\sigma(t_1), \sigma(t_2), \dots, \sigma(t_k))$$

Now we can inductively define the semantics of a formula α under a given model M and assignment σ:

- $M, \sigma \models R(t_1, t_2, \dots, t_k)$ if $R^M(\sigma(t_1), \sigma(t_2), \dots, \sigma(t_k))$
- $M, \sigma \models t_1 = t_2$ if $\sigma(t_1) = \sigma(t_2)$
- $M, \sigma \models \neg\beta$ if $M, \sigma \not\models \beta$
- $M, \sigma \models \beta \vee \gamma$ if $M, \sigma \models \beta$ or $M, \sigma \models \gamma$
- $M, \sigma \models \exists x.\beta$ if there is an assignment τ such that $\tau(y) = \sigma(y)$ for all $y \neq x$, and $M, \tau \models \beta$

If α is a sentence, then σ is irrelevant since α has no free variables, so we simply write $M \models \alpha$. A sentence α is *valid* if for all models M, $M \models \alpha$. A sentence α is *satisfiable* if for some model M, $M \models \alpha$.

Let Γ be a set of sentences. We write $M \models \Gamma$ if for all $\alpha \in \Gamma, M \models \alpha$. We write $\Gamma \models \alpha$ if for all models M such that $M \models \Gamma$, $M \models \alpha$. Define the *consequences* $CN(\Gamma)$ of a set of sentences Γ as $\{\alpha : \Gamma \models \alpha\}$. If $\Gamma = CN(\Gamma)$, then Γ is called a *theory*. A theory Γ is *consistent* if it has a model. A theory Γ is *complete* if for all α, either $\alpha \in \Gamma$ or $\neg\alpha \in \Gamma$.

Define the theory $Th(M)$ of a model M as $\{\alpha : M \models \alpha\}$.

Proposition 12.1: Γ is consistent and complete if and only if $\Gamma = Th(M)$ for some model M.

Proof: $\Rightarrow$: Suppose Γ is consistent and complete. Since it's consistent, it has some model M. Therefore, $\Gamma \subseteq Th(M)$. Suppose $\Gamma \neq Th(M)$, so there is some sentence α such that $\alpha \in Th(M)$ but $\alpha \notin \Gamma$. Since Γ is complete, this means that $\neg\alpha \in \Gamma$, but then $\neg\alpha \in Th(M)$. So both α and $\neg\alpha$ are in $Th(M)$, which cannot occur according to the semantics presented above. Therefore, we have a contradiction, so it must be the case that $\Gamma = Th(M)$.

$\Leftarrow$: Suppose $\Gamma = Th(M)$. Then by the definition of $Th(M)$, $M \models \Gamma$, so Γ is consistent. Further, by the semantics presented above, we know that for all sentences α, either $M \models \alpha$ or $M \models \neg\alpha$. Therefore $Th(M)$ is complete, so Γ is complete as well. $\square$

In the next lecture, we will discuss the following three theorems:

Theorem 12.2 (Completeness): If Γ is recursively enumerable, then so is $CN(\Gamma)$.

Theorem 12.3 (Compactness): If Γ is inconsistent, then some finite subset of Γ is inconsistent.

Theorem 12.4 (Lowenheim-Skolem): If Γ is countable and consistent, then Γ has a countable model.

These theorems will be proven by the *Herbrand universe* method.

Lecture 13

The Herbrand Universe Construction

February 19, 1998
Lecturer: Richard Ladner
Notes: Marc T. Friedman

13.1. The Theorem and its Implications

Theorem 13.1: Given Γ, a recursively enumerable set of first-order sentences, we construct a set H_Γ of *propositional* formulas which is satisfiable if and only if Γ is. This effective construction builds a Turing machine generating H_Γ given one generating Γ. Furthermore, if Γ is satisfiable, then H_Γ will completely describe a countable model for Γ, which can easily be extracted from a model for H_Γ.

This theorem has a number of remarkable consequences.

- **Compactness.** Γ is satisfiable if and only if all of its finite subsets are. This is often stated in the contrapositive: if Γ is unsatisfiable, so is some finite subset of Γ. This follows from the Herbrand universe construction by virtue of the fact that any unsatisfiable subset of H_Γ is constructed (by a process described below) from a finite subset of Γ, and there is a compactness theorem for propositional logic.
- **Lowenheim-Skolem Theorem.** If Γ is satisfiable, then it is so in a countable (possibly finite) model. In some sense, this exposes a weakness of first-order logic in that it cannot express any property of uncountable sets that distinguish them from countable ones. For instance, the first-order theory of real numbers $Th(\Re) = (\Re, +, *, <, 0, 1)$ has a countable "non-standard" model, model. For example the first-order theory cannot talk about arbitrary functions. Talking about functions is vital for discussions about derivatives and integrals.
- **Completeness.** if Γ is recursively enumerable so is $CN(\Gamma)$. The more familiar statement is that if $\Gamma \models \alpha$ then $\Gamma \vdash \alpha$. The more familiar statement follows from the fact that the Herbrand universe construction defines a proof system for first-order logic by the observation that $\Gamma \cup \{\neg\alpha\}$ is not satisfiable if and only if $\Gamma \models \alpha$. If $\Gamma \cup \{\neg\alpha\}$ is not satisfiable then there is a finite subset of H_Γ that is not satisfiable. Such a finite subset can be found as part of the construction of H_Γ.

13.2. The construction

Given Γ, a countable set of first-order sentences.

- Construct P_Γ, the *prenex* form of sentences in Γ. Rename away any variables whose scopes overlap, and move the quantifiers forward. For instance, change

$$\forall x \forall y (R(x,y) \land \exists y P(y))$$

to

$$\forall x \forall y \exists z (R(x,y) \land P(z)).$$

P_Γ is satisfiable if and only if Γ is, by the same model.

- Construct S_Γ, the *Skolemization* of P_Γ, to eliminate existential quantifiers while introducing new function symbols. For instance, if α is some formula with free variables x_1, x_2, x_3, x_4, to convert the sentence

$$\forall x_1 \exists x_2 \forall x_3 \exists x_4 \alpha(x_1, x_2, x_3, x_4)$$

introduce two new functions $f()$ and $g()$, resulting in

$$\forall x_1 \forall x_3 \alpha(x_1, f(x_1), x_3, g(x_1, x_3)).$$

S_Γ is satisfiable if and only if P_Γ is, since any model of P_Γ will have an object assigned to each universally quantified variable which depends only on the preceding quantified variables, which assures us that functions f and g exist. On the other hand, if you can find such functions, then the existentially quantified variables clearly exist.

- Add the axioms of equality to S_Γ to construct E_Γ. These rules completely characterize the behavior of equality as an equivalence relation on, and how it interacts with function and relation symbols:[1]

  1. $\forall x (x = x)$
  2. $\forall x \forall y (x = y \Rightarrow y = x)$
  3. $\forall x \forall y \forall z (x = y \land y = z \Rightarrow x = z)$
  4. Relations respect equality. Adopt the shorthand $\bar{x}$ for the vector of variables $x_1, x_2, \ldots, x_k$.

$$\forall \bar{x} \forall \bar{y} (\bar{y} = \bar{x} \Rightarrow (R(\bar{x}) \equiv R(\bar{y}))$$

In other words, given equal parameters, the boolean value R returns must be the same.

[1]Throughout these notes, the equals sign will be used for many different things. It should be clear enough from context whether we intend the equals sign in the formal first-order language, equality of objects, or equality in the presentation metalanguage. Boolean equivalence is distinct, however.

5. Functions respect equality.

$$\forall \bar{x} \forall \bar{y} (\bar{y} = \bar{x} \Rightarrow (f(\bar{x}) = f(\bar{y}))$$

   That is, given equal arguments, the objects f refers to must be the identical object in the universe.

E_Γ is satisfiable whenever S_Γ is, since it merely states that equality is interpreted correctly. Note that there are no existential quantifiers in the equality axioms.

- Construct H_Γ, the instances of Γ, from E_Γ. In essence we replace universal quantification with enumeration. First introduce one constant symbol, c. Then T_Γ is the set of all terms constructed from c and the function symbols, *e.g.*

$$T_\Gamma = \{c, f(c), f(f(c)), g(c,c), g(c, f(c)), \ldots\}.$$

  Now

$$H_\Gamma = \{\alpha(t_1, \ldots, t_k) | \forall x_1 \forall x_2 \ldots \forall x_k \alpha(x_1, x_2, \ldots, x_k) \in E_\Gamma \text{ and } t_i \in T_\Gamma .\}$$

  H_Γ contains relations and functions and c, but no quantifiers. So think of each relation such as $R(g(c, f(c)), c)$ or $f(c) = f(f(c))$ as a distinct proposition.

13.3. Correctness.

Claim: H_Γ is satisfiable *as a propositional formula* if and only if E_Γ is satisfiable as a set of first order sentences.

First a few words of explanation. A term or relation is *ground* if it has no variables. The propositions in H_Γ are ground relations, or propositions, like $R(g(c, f(c)), c)$ or $f(c) = f(f(c))$. Each is a sentence which is either true or false. We can enumerate the ground relations, and associate with each a boolean variable, such as $v_{R(g(c,f(c)),c)}$. Then H_Γ can be viewed as boolean propositional formula on these new boolean variables.

Proof: Assume E_Γ is satisfiable, so there is a model $\mathcal{M} \models E_\Gamma$. Choose $c^{\mathcal{M}} \in \mathcal{U}^{\mathcal{M}}$. Now all $t \in T_\Gamma$ are interpreted as an element $t^{\mathcal{M}} \in \mathcal{U}^{\mathcal{M}}$ recursively. For example:

$$f(f(c))^{\mathcal{M}} = f^{\mathcal{M}}(f^{\mathcal{M}}(c^{\mathcal{M}})).$$

Now we define an assignment σ from the propositional variables of H_Γ into $\{T, F\}$:

$$\sigma(R(t_1, t_2)) = R^{\mathcal{M}}(t_1^{\mathcal{M}}, t_2^{\mathcal{M}})$$

$$\sigma(t_1 = t_2) = t_1^{\mathcal{M}} = t_2^{\mathcal{M}}$$ [2]

[2]Note the particularly perverse abuse of equals signs. The first is a symbol in the formula, the second is part of the metalanguage defining σ, and the third is equality in the universe.

Now we claim that for all ground quantifier-free sentences α,

$$\mathcal{M}, c = c^{\mathcal{M}} \models \alpha \Leftrightarrow \sigma \models \alpha. \tag{13.1}$$

Note that we write $\mathcal{M}, c = c^{\mathcal{M}}$ because the model $\mathcal{M}$ does not contain an interpretation for the symbol c. Thus, the notation simply extends the model $\mathcal{M}$ to include an interpretation for c. The equivalence (17.1) transparently true by induction on the structure of α and the definition of σ.

To finish this direction let $\alpha(t_1, t_2, \ldots, t_k) \in H_\Gamma$. We have the follow sequence of consequences:

$\mathcal{M} \models \forall x_1 \forall x_2 \ldots \forall x_k \alpha(x_1, x_2, ..., x_k)$ because $\forall x_1 \forall x_2 \ldots \forall x_k \alpha(x_1, x_2, ..., x_k) \in E_\Gamma$
$\mathcal{M}, c = c^{\mathcal{M}} \models \alpha(t_1, t_2, \ldots, t_k)$ because the formula is universal
$\sigma \models \alpha(t_1, t_2, \ldots, t_k)$ by the equivalence (17.1).

To prove the other direction, let $\sigma \models H_\Gamma$. Now, we define an equivalence relation $\sim$ on terms in T_Γ. $t_1 \sim t_2$ if $\sigma(t_1 = t_2) = T$. The equality axioms 1-3 are used to show that $\sim$ is an equivalence relation Also define $[t] = \{t' | t \sim t'\}$. Then we define the model $\mathcal{M} = (\mathcal{U}^{\mathcal{M}}, I^{\mathcal{M}})$ where the language of $\mathcal{M}$ contains all the symbols of E_Γ and c. The universe contains the equivalence classes over terms:

$$\mathcal{U}^{\mathcal{M}} = \{[t] | t \in T_\Gamma\}.$$

The relations are well-defined on equivalence classes, by the fourth axiom of equality:

$$R^{\mathcal{M}}([t_1], [t_2], \ldots, [t_k]) = \sigma(R(t_1, t_2, \ldots, t_k)).$$

The functions are well-defined on equivalence classes by the fifth axiom of equality:

$$f^{\mathcal{M}}([t_1], [t_2], \ldots, [t_k]) = [f(t_1, t_2, \ldots, t_k)].$$

Then we claim that for all quantifier-free ground formulas α,

$$\mathcal{M} \models \alpha \Leftrightarrow \sigma \models \alpha. \tag{13.2}$$

This proof is also a mechanical structural induction. Note further that this model contains an interpretation of the constant c, since α may contains c's. To obtain a model for E_Γ we remove the interpretation for the symbol c to yield a model $\mathcal{M}'$. Let $\forall x_1 \forall x_2 \ldots \forall x_k \alpha(x_1, x_2, ..., x_k)$ be a sentence in E_Γ. We have the following sequence of consequences:

$\sigma \models \alpha(t_1, t_2, \ldots, t_k)$ for all $t_1, t_2, \ldots, t_k \in T_\Gamma$ because $\alpha(t_1, t_2, \ldots, t_k) \models H_\Gamma$
$\mathcal{M} \models \alpha(t_1, t_2, \ldots, t_k)$ for all $t_1, t_2, \ldots, t_k \in T_\Gamma$ by equivalence (17.2)
$\mathcal{M}, x_1 = [t_1], \ldots, x_k = [t_k] \models \alpha(x_1, x_2, ..., x_k)$ for all $[t_1], [t_2], \ldots, [t_k] \in \mathcal{U}^{\mathcal{M}}$ by the semantics
$\mathcal{M} \models \forall x_1 \forall x_2 \ldots \forall x_k \alpha(x_1, x_2, ..., x_k)$ by the semantics
$\mathcal{M}' \models \forall x_1 \forall x_2 \ldots \forall x_k \alpha(x_1, x_2, ..., x_k)$ because $\forall x_1 \forall x_2 \ldots \forall x_k \alpha(x_1, x_2, ..., x_k)$ does not contain c.

Note further that by eliminating the interpretations for the symbols introduced by skolemization and the symbol c, $\mathcal{M}$ contains a model for the original Γ. This model is countable because there are only countably many terms in T_Γ.

Claim: The construction is effective. If Γ is recursively enumerable then H_Γ is recursively enumerable.

If Γ is recursively enumerable then you can generate its elements in sequence. As you generate them, maintain (perhaps each on a separate tape) a counter n of elements of Γ, the set of function symbols F, the set of relation symbols R, and the set E_Γ. Each time you generate a new sentence of Γ, update the lists. Add to E_Γ the prenex, Skolemized version of the sentence, and any equality axioms for the new function and relation symbols in it. On another tape, build T_Γ in lexicographic order, but only up to those terms of length n. When a new function symbol arises, extend T_Γ to include the terms of length n that contain the new symbol. On the last tape, maintain the elements of H_Γ of length $\leq n$ from E_Γ and T_Γ. You may have to extend these periodically to include formulas with a new symbol that arises from the generation of Γ. $\square$

Lecture 14

Consequences of the Herbrand Universe Construction

February 24, 1998
Lecturer: Richard Ladner
Notes: Sumeet Sobti

14.1. Herbrand Universe Construction: An Example

Theorem-provers almost invariably use the Herbrand universe construction. The line of action is the following. Suppose we need to show that $\Gamma \models \alpha$. Now it suffices to show that $\Gamma \cup \{\neg\alpha\}$ is not satisfiable. For this, we construct $H_{\Gamma \cup \{\neg\alpha\}}$ (in fact, we construct only a part of it) and look for propositional unsatisfiability there. Lets we look at a simple example of theorem-proving by Herbrand universe construction.

Suppose we want to prove the following statement. If s is a 1-1 onto function and t is the left inverse of s, then s is the left inverse of t. (Note that t is the left inverse of s if and only if $\forall x, t(s(x)) = x$.)

The above statement can be formulated as $\Gamma \models \alpha$, where

$$\Gamma = \{\forall x \forall y (s(x) = s(y) \Longrightarrow x = y), \forall y \exists x (s(x) = y), \forall x (t(s(x)) = x)\}$$

$$\alpha = \forall x (s(t(x)) = x)$$

The first two formulae in Γ say that s is a 1-1 onto function. The third one says that t is the left inverse of s. The formula α says that s is the left inverse of t.

Thus,

$$\Gamma \cup \{\neg\alpha\} = \{\forall x \forall y (s(x) = s(y) \Rightarrow x = y), \forall y \exists x (s(x) = y), \forall x (t(s(x)) = x), \exists x (\neg(s(t(x)) = x))\}$$

Let $\Delta = \Gamma \cup \{\neg\alpha\}$. Now, we construct as much of H_Δ as we need. Clearly, Δ is already in prenex form. For *Skolemization*, we introduce a unary function g for the second sentence and a 0-ary function (i.e. a constant) c for the fourth sentence. Also adding the equality axioms, we get the following sentences in E_Δ.

1. $\forall x \forall y (s(x) = s(y) \Rightarrow x = y)$

2. $\forall y (s(g(y)) = y)$

3. $\forall x (t(s(x)) = x)$

4. $\neg(s(t(c)) = c)$

5. $\forall x(x = x)$

6. $\forall x \forall y(x = y \Rightarrow y = x)$

7. $\forall x \forall y \forall z(x = y \wedge y = z \Rightarrow x = z)$

8. $\forall x \forall y(x = y \Rightarrow (s(x) = s(y) \wedge t(x) = t(y) \wedge g(x) = g(y)))$

Now since we have already introduced a constant symbol c, we can construct the set of all terms as

$$T_\Delta = \{c, g(c), t(c), s(c), g(s(c)), \ldots\}$$

Now lets enumerate some formulas of H_Δ.

a. $s(g(c)) = c$ {*Using 2* }

b. $t(s(g(c))) = t(c)$ {*Using 8 and a* }

c. $t(s(g(c))) = g(c)$ {*Using 3*}

d. $t(c) = g(c)$ {*Using 6, 7, b and c*}

e. $s(t(c)) = s(g(c))$ {*Using 8 and d*}

f. $s(t(c)) = c$ {*Using 7, a and e*}

g. $\neg(s(t(c)) = c)$ {*Using 4*}

h. F {*Using f and g*}

Hence we conclude that $\Delta = \Gamma \cup \{\neg\alpha\}$ is not satisfiable.

Of course, for mechanization, we need strategies for making substitutions. Unification is one method of selecting substitutions of terms for variables. In order to combine the Herbrand construction with the propositional resolution procedure it is necessary to look for good substitutions where the result for example yields a literal in one clause and its complement in another.

14.2. Another Consequence of Herbrand Construction

Theorem 14.1: If Γ is a set of sentences (on a finite language) that has infinitely many finite models, then it has an infinite model.

Proof: Define formula $\alpha_i = \exists x_1 \ldots \exists x_i(\bigvee_{j \neq k} \neg(x_j = x_k))$. Thus any model for α_i will have at least i elements. Let $\Delta = \Gamma \cup \{\alpha_i \mid i \geq 1\}$.
Claim: Every finite subset of Δ is satisfiable.
Proof: Suppose Φ is a finite subset of Δ. Let k be the maximum index such that $\alpha_k \in \Phi$. Now, if Γ doesn't belong to Φ, then any model of size at least k will satisfy Φ. If $\Gamma \in \Phi$, then any model M that satifsies Γ and has size at least k will satisfy Φ. Existence of such an M is ensured by the fact that Γ has infinitely many finite models. Thus, in any case, we see that Φ is satisfiable.

By the compactness theorem, we conclude that Δ is satisfiable. Now since $\forall i \geq 1 : \alpha_i \in \Delta$, Δ can't be satisfied by a finite model. Thus, Δ and hence Γ is satisfied by an infinite model. $\square$

14.3. Graphs as Models

A directed graph $G = (V, E)$ can be seen as a model with universe V and a binary relation E defined on elements of V. A sentence ϕ may or may not be satisfied by G. Thus, ϕ defines a class of graphs namely, the graphs which satisfy ϕ. In other words, ϕ defines a *property* on graphs. For example,

- $\phi = \forall x(\neg E(x, x))$ represents the property of *not having any self-loops.*
- $\phi = \forall x \forall y(E(x, y) \Rightarrow \neg E(y, x))$ represents the *anti-symmetry* property.
- $\phi = \forall x \forall y(\neg(x = y) \Rightarrow E(x, y))$ is only satisfied by *cliques.*

Now the question comes "Can we describe all properties of graphs as first-order sentences?" And the answer is "no" as proved in the next theorem.

Theorem 14.2: The property of being *strongly-connected* is not a first-order property of directed graphs.

Proof: Assume that sentence Ψ_{SC} represents the property of being *strongly-connected.* Define sentences Ψ_{SL}, Ψ_{IN} and Ψ_{OUT} as follows.

- Let $\Psi_{SL} = \forall x(\neg E(x, x))$. Thus, Ψ_{SL} states the property of having *no self-loops.*
- Let $\Psi_{OUT} = \forall x \forall y \forall z(E(x, y) \wedge E(x, z) \Rightarrow y = z)$. Thus, Ψ_{OUT} states the property that each vertex in the graph has outdegree 1.
- Let $\Psi_{IN} = \forall x \forall y \forall z(E(y, x) \wedge E(z, x) \Rightarrow y = z)$. Thus, Ψ_{IN} states the property that each vertex in the graph has indegree 1.

Let $\Psi = \Psi_{SC} \wedge \Psi_{SL} \wedge \Psi_{OUT} \wedge \Psi_{IN}$. Thus Ψ describes the class of graphs that are strongly connected, have no self loops and have all vertices of indegree and outdegree 1. This is exactly the class of *cycle* graphs (of finite sizes). Clearly, there are infinitely many *cycle* graphs. Thus, by the previous theorem, there must be an infinite graph satisfying Ψ. But this is impossible. Hence we have a contradiction and the desired result follows. $\square$

14.4. Valid First-Order Sentences

The set of valid propositional sentences is computable. What about the set of valid first-order sentences? As we see below, this is recursively enumerable, but not computable.

Note that $CN(\phi) = \{\alpha \mid \phi \models \alpha\}$ is exactly the set of valid first-order sentences. Now since ϕ is recursively enumerable, so is $CN(\phi)$. The following theorem shows that the set of valid first-order sentences is not computable.

Theorem 14.3: The set of valid first-order sentences is not computable.

Proof: A sentence α is satisfiable if and only if $\neg\alpha$ is not valid. Thus computability of the set of valid first-order sentences implies computability of the set of satisfiable first-order sentences.

Suppose $T = (S, \{a\}, \Gamma, \delta, s)$ is a Turing machine that accepts a non-computable subset of $\{a\}^*$. Let a^n be any input to T. Assume that if T does not accept x, then it never halts. Below we give an algorithmic construction of a sentence α_n such that α_n is satisfiable if and only if T does not accept x. Thus, by the observation made in the previous paragraph, the desired result will follow.

We represent a configuration of T on a^n as a string $\#\beta q\gamma$ where q is the current state, β is the tape contents to the left of the head and γ is the tape contents to the right of the head (including the symbol under the head). Let $\Delta = S \cup \Gamma \cup \{\#\}$ and $k = |\Delta|$. Thus, all symbols in any configuration belong to Δ. As we have seen before, there is a function $f : \Delta^4 \to \Delta$ such that if $\sigma_0, \sigma_1, \sigma_2$ and σ_3 are symbols in $i^{th}, (i+1)^{th}, (i+2)^{th}$ and $(i+3)^{th}$ positions of some configuration respectively, then the $(i+1)^{th}$ symbol in the succeeding configuration is $f(\sigma_0, \sigma_1, \sigma_2, \sigma_3)$.

Let $L = \{0, s, P\}$, where s is a unary function, P is a 3-ary relation and 0 is a constant.

Denote $s^i(0)$ by $\underline{i}$. $\underline{0}$ will denote 0 only. Lets fix an ordering of the symbols in Δ. If σ is the i^{th} symbol in that ordering, then we also denote $\underline{i}$ by $\underline{\sigma}$.

The sentence α_n that we construct is the conjunction of the following.

1. $\forall x(\neg(s(x) = 0))$
2. $\forall x \forall y(s(x) = s(y) \Rightarrow x = y)$
3. $\forall x \forall y \forall z(P(x, y, z) \Rightarrow (z = \underline{0} \vee z = \underline{1} \vee \ldots \vee z = \underline{k-1}))$
4. $\forall x \forall y \forall z \forall z'(P(x, y, z) \wedge P(x, y, z') \Rightarrow z = z')$
5. $\forall y(P(\underline{0}, y, \underline{\#}))$
6. $P(\underline{1}, \underline{0}, \underline{s}) \wedge \bigwedge_{i=2}^{n+1} P(\underline{i}, \underline{0}, \underline{a}) \wedge (\forall x((\bigwedge_{i=0}^{n+1} \neg(x = \underline{i})) \Rightarrow P(x, \underline{0}, \underline{\sqcup})))$
7. $\forall x \forall y(\bigwedge_{\sigma_0,\sigma_1,\sigma_2,\sigma_3 \in \Delta}(P(x, y, \underline{\sigma_0}) \wedge P(s(x), y, \underline{\sigma_1}) \wedge P(s(s(x)), y, \underline{\sigma_2}) \wedge P(s(s(s(x))), y, \underline{\sigma_3}) \Rightarrow P(s(x), s(y), \underline{f}(\sigma_0, \sigma_1, \sigma_2, \sigma_3))))$
8. $\forall x \forall y(\neg P(x, y, \underline{A}))$, where A is the accept state.

Claim: α_n is satisfiable $\Longleftrightarrow T$ does not accept a^n.
Proof: ($\Leftarrow$) Suppose T does not accept a^n. Define a model $M = (U^M, I^M)$ as follows. Let $U^M = \{0, 1, 2, \ldots\}$, the set of non-negative integers. I^M defines s^M to be the successor relation among the integers in U^M. Relation P^M is defined as: $P^M(x, y, z)$ if and only if "the symbol in position x of the y^{th} configuration of T (when run on a^n) is the z^{th} symbol of Δ (in the ordering fixed above)". Now it is easy to verify that all of the above sentences are satisfied by M. Sentence 1 states that 0 is not the successor of any other element of the universe. Sentence 2 states that each element has a unique successor. Sentence 3 states that in each position of each configuration of T, there is some symbol of Δ. Sentence 4 states that this symbol has got to be unique. Sentence 5 states that the first symbol of all configurations is $\#$. Sentence 6 states that the 0^{th} configuration is the initial configuration of T on a^n. Sentence 7 states that all the transitions between successive configurations are valid. Sentence 8 states that machine T never reaches the accept state. It is easy to see that M satisfies all of these. Thus, α_n is satisfiable.

($\Rightarrow$) Suppose $M = (U^M, I^M)$ is a satisfying model for α_n. Then, sentences 1 and 2 ensure that the elements $\underline{0}, \underline{1}, \underline{2}, \ldots$ in U^M are in a 1-1 correspondence with non-negative integers. (Remember that $\underline{i}$ denotes $s^i(0)$). Now since sentences in α_n only have universal quantifiers, we can restrict M

to obtain model $M' = (U^{M'}, I^{M'})$, where $U^{M'} = \{\underline{0}, \underline{1}, \underline{2}, \ldots\}$ and $I^{M'}$ defines $s^{M'}$ and $P^{M'}$ to be restrictions of s^M and P^M respectively to $U^{M'}$. It is easy to see that M' also satisfies α_n. Sentences 3 through 7 ensure that $P^{M'}$ must now directly correspond to the *transition relation* of T on a^n. Sentence 8 ensures that T does not accept a^n. Hence the claim follows.

$\square$

Lecture 15

Gödel's Incompleteness Theorem

February 26, 1998
Lecturer: Richard Ladner
Notes: Andrew Keller

15.1. Gödel's Incompleteness Theorem

Let $\mathcal{N} = (N, s, +, \cdot, ^\wedge, <, 0)$ where N are the natural numbers, s, the successor function, $^\wedge$, exponentiation, etc.

Theorem 15.1 (Gödel's Incompleteness Theorem): Th($\mathcal{N}$) is *not* recursively enumerable.

In fact, it turns out that Th($\mathcal{N}$) is beyond the arithmetic hierarchy. The arithmetic hierarchy can be defined by languages reducible to $\emptyset^{(k)}$ where $\emptyset^{(0)} = \emptyset$ and $\emptyset^{(k+1)}$ is the set of all oracle Turing machines encoding $\hat{T}$ such that T accepts $\hat{T}$ with oracle $\emptyset^{(k)}$.

Why is it called the "incompleteness" theorem? Suppose there were recursively enumerable set of arithmetic axioms, $\mathcal{A}$, such that CN($\mathcal{A}$) $\subseteq$ Th($\mathcal{N}$). The Incompleteness Theorem dictates that Th($\mathcal{N}$) is *not* recursively enumerable, so CN($\mathcal{A}$), which is recursively enumerable, cannot be equal to Th($\mathcal{N}$). Hence there must be true statements of arithmetic that cannot be proven from the axioms $\mathcal{A}$. This killed Hilbert's plan for a mechanized system of deriving truths, and is thought to be one of the ten most significant theorems of our century (*USA Today* readers poll, 1996).

The proof described below is simpler than Gödel's original. (For example, he didn't use exponentiation or even Turing machines, but instead invented his own theory of computation and used a diagonalization argument.) It is a reduction from a non-r.e. language to $\{\alpha : \mathcal{N} \models \alpha\}$. Given Turing Machine T which recognizes the *non-recursive* language $L(T)$, this reduction on input x constructs formula α_x such that $x \in \overline{L(T)} \Leftrightarrow \mathcal{N} \models \alpha_x$.

Proof: We start with a Turing Machine $T = (S, \Sigma, \Gamma, \delta, s)$. Let $\Delta = S \cup \Gamma \cup \{\#\}$. We must now make a connection between strings and numbers such that each string $s \in \Delta^*$ corresponds to a unique number. Given that the cardinality of Δ is d, one denotes each letter $a \in \Delta$ by the number $\underline{a}$ where $\{\underline{a} : a \in \Delta\} = \{0, 1, 2, \ldots d-1\}$. The number x corresponding to string s $= \sigma_0, \sigma_1, \ldots, \sigma_{k-1}$ is then given by: $x = \sum_{i=0}^{k-1} \underline{\sigma}_i d^i$. The number x is thus a d-ary numerical representation of s. We assume that $\underline{\#} \neq 0$.

A *partial computation* of T has the form $\#C_0\#C_1 \cdots C_k\#$, where the C_i, $0 \leq i \leq k$ are configurations of T of the same length, each following legally from its predecessor, and C_0 is the intial configuration. This computation then comprises a string $\sigma_0\sigma_1 \cdots \sigma_m$ with d-ary numerical representation $n = \sum_{i=0}^{m-1} \underline{\sigma}_i d^i$.

The partial computation of T upon input $x = x_1 x_2 \cdots x_t$ must have the following three properties:

1. $\sigma_0 = \#, \sigma_1 = s, \sigma_2 = x_1, \ldots, \sigma_{t+1} = x_t, \sigma_{t+2} = \sqcup, \ldots, \sigma_{l-1} = \sqcup, \sigma_l = \#$
2. $\forall p,\ 0 \leq p \leq n - l - 1,\ \sigma_{p+l+1} = f_T(\sigma_p, \sigma_{p+1}, \sigma_{p+2}, \sigma_{p+3})$
3. $\sigma_m = \#$

Wher f_T is the function that determines the $(p+l+1)$-st symbol of a partial computation from the p-th, $(p+1)$-st, $(p+2)$-nd, and $(p+3)$-rd symbols of the partial computation. The first requires that the first configuration is correct, the second requires that each pair of consecutive configurations are legal transitions according to T, and the third requires that the last configuration is complete.

We must now impose the above three properties on a computation string with d-ary numerical representation n using the language of $\mathcal{N}$. Toward this goal, we define the following helper functions:

- "n divided by m = q", $Div(n, m, q) = \exists r < m(m \cdot q + r = n)$
- "n divided by m yields the remainder r", $mod(n, m, r) = \exists q \leq n(m \cdot q + r = n \text{ and } r < m)$
- "the i^{th} dit of n is s", $D(i, n, s) = \exists q \leq n(Div(n, d^\wedge i, q) \wedge Mod(q, d, s))$

Note that the quantifiers in all three helper functions are *bounded* within a range of possible values

Furthermore, each d-ary numerical value i can be represented in the language of $\mathcal{N}$ as $s^i(0)$, denoted $\underline{i}$, where s is the successor function of $\mathcal{N}$.

Now define the formula $pcomp_{x,T}(n)$ in the language of $\mathcal{N}$ which forces the 3 properties listed above onto the partial computation string of T with d-ary numerical representation n:

$pcomp_{x,T}(n) = (\exists m \leq n)(\exists l \leq n)(\psi_1 \wedge \psi_2 \wedge \psi 3)$, where terms ψ_1, ψ_2, ψ_3 impose the above properties 1,2,and 3, respectively:

- $\psi_1 = Div(n, d^\wedge m, \underline{\#}) \wedge D(0, n, \underline{\#}) \wedge D(\underline{1}, n, \underline{s}) \wedge D(\underline{2}, n, \underline{x}_1) \wedge \cdots \wedge D(\underline{t}+\underline{1}, n, \underline{x}_t) \wedge (\forall i \leq l - \underline{1})(\underline{t} + \underline{1} < i \Rightarrow D(i, n, \underline{\sqcup})) \wedge D(l, n, \underline{\#})$
- $\psi_2 = (\forall p \leq m - l - \underline{1})(\bigwedge_{\sigma_0\sigma_1\sigma_2\sigma_3}(D(p, n, \underline{\sigma}_0) \wedge D(p+\underline{1}, n, \underline{\sigma}_1) \wedge D(p+\underline{2}, n, \underline{\sigma}_2) \wedge D(p+\underline{3}, n, \underline{\sigma}_3) \Rightarrow D(p + l + \underline{1}, n, \underline{f_T(\sigma_0, \sigma_1, \sigma_2, \sigma_3)})))$
- $\psi_3 = Div(n, d^\wedge m, \underline{\#})$

Finally, define the related formula $comp_{x,T}$ which additionally requires that the computation string for T with d-ary numerical representation n have an accepting state, A:

$comp_{x,T}(n) = pcomp_{x,T}(n) \wedge (\exists p \leq n) D(p, n, \underline{A})$

Claim: $x \in \overline{L(T)} \Leftrightarrow \mathcal{N} \models \forall n\ \neg comp_{x,T}(n)$

The above is a reduction from a non-recursive Co-r.e. language to $\mathcal{N}$ model satisfiability:

$\overline{L(T)} \leq_m \{\alpha : \mathcal{N} \models \alpha\}$

Since $\overline{L(T)}$ is not r.e., neither is $\{\alpha : \mathcal{N} \models \alpha\}$.

□

15.2. Natural Axioms for NumberTheory

The set NT include the following 14 axioms:

1. $\forall x(s(x) \neq 0)$

2. $\forall x \forall y (s(x) = s(y) \Rightarrow x = y)$
3. $\forall x (x = 0 \vee \exists y s(y) = x)$
4. $\forall x (x + 0 = x)$
5. $\forall x \forall y (x + s(y) = s(x + y))$
6. $\forall x (x \cdot 0 = 0)$
7. $\forall x \forall y (x \cdot s(y) = (x \cdot y) + x)$
8. $\forall x (x^\wedge 0 = s(0))$
9. $\forall x \forall y (x^\wedge s(y) = (x^\wedge y) \cdot x)$
10. $\forall x (x < s(x))$
11. $\forall x \forall y (x < y \Rightarrow (s(x) \leq y))$
12. $\forall x \forall y (\neg (x < y) \Leftrightarrow y \leq x)$
13. $\forall x \forall y \forall z (((x < y) \wedge (y < z)) \Rightarrow x < z)$
14. $\forall x \forall y \forall z \forall z' (mod(x, y, z) \wedge mod(x, y, z') \Rightarrow z = z')$

Define a *Bounded Formula* as either of the following:

- a quantifier-free formula
- formulas $(\exists x_n < t)\alpha(x_1 \cdots x_n)$ and $(\forall x_n < t)\alpha(x_1 \cdots x_n)$, given that $\alpha(x_1 \cdots x_n)$ is known to be bounded and t is a term over the variables $x_1 \cdots x_{n-1}$.

Theorem 15.2: If α is a bounded sentence (all its variables bounded) and $\mathcal{N} \models \alpha$, then $NT \models \alpha$.

Since alpha is bounded, all its variables have a finite domain, and thus all bounded quantifiers can be expressed by a finite terms. Thus, the sentence is equivalent to a finite length quantifier free formula. Any atomic formula can be verified for its truth or falsity from the axioms NT. Thus, any finite length quantifier free formula can be determined from NT.

Theorem 15.3: If $NT \subseteq \Gamma$ and Γ is consistent, then $CN(\Gamma)$ is *not* computable

On simple consequence is that $CN(NT)$ is not computable
Proof:
Definition: If S and T are 2 sets such that $S \cap T = \emptyset$, then S and T are *recursively inseparable* if there is no computable set R such that $S \subseteq R$ and $T \subseteq \overline{R}$.
For example, the following 2 languages are recursively inseparable:

- $L_1 = \{\hat{T} : T \text{ halts and accepts } \lambda\}$
- $L_2 = \{\hat{T} : T \text{ halts and rejects } \lambda\}$

where $\hat{T}$ is a standard encoding of Turing machines and λ is the empty string. See Papadimitriou, page 63, for a proof that L_1 and L_2 are recursively inseparable.

Claim: $CN(NT)$ and $USAT$ are recursively inseparable, where $USAT = \{\alpha : \models \neg\alpha\}$

Proof: By reduction from L_1 and L_2 above. Given Turing Machine T, construct the formula ϕ_T such that:

1. if λ is accepted by T, then $NT \models \phi_T$
2. if λ is rejected by T, then $\models \neg\phi_T$

To be continued..... $\square$

Lecture 16

Number Theory and Presburger Arithmetic

March 3, 1998
Lecturer: Richard Ladner
Notes: Jason Hartline

16.1. Continuing our look at the recursive inseparability of *CN*(**NT**) and *USAT*

We were in the process of proving the theorem that if $NT \subseteq \Gamma$ and Γ is consistent then CN(NT) is not computable.

Previously we defined two recursively inseparable languages L_1, and L_2 as

$$\begin{aligned} L_1 &= \left\{\hat{T} \mid T \text{ halts and accepts } \lambda\right\} \\ L_2 &= \left\{\hat{T} \mid T \text{ halts and rejects } \lambda\right\} \end{aligned}$$

and made the claim that claim that CN(NT) and $USAT = \{\alpha \mid \alpha\neg\alpha\}$ are recursively inseparable. We will now finish the proof by reduction from L_1 and L_2. We will construct a formula φ_T with the properties:

1. $\text{NT} \models \varphi_T$ if T accepts λ
2. $\models \neg\varphi_T$ if T rejects λ

The number n will be used as a sequence of bits that represents a computation history T on input λ.

$$\begin{aligned} \alpha_T(n) \quad = \quad & pcomp_{\lambda,T}(n) \wedge (\exists p \leq n) D(p, n, \underline{A}) \\ & \wedge \forall (m \leq n) \left[pcomp_{\lambda,T}(m) \rightarrow (\forall p \leq m) \neg D(p, m, \underline{R})\right] \end{aligned}$$

$\alpha_T(n)$ is true if T accepts λ. Here $pcomp_{\lambda,T}(n)$ is true if n represents a valid computation history of T on x (definition given last time). $D(p, m, d)$ is a predicate that means that the pth digit of n is d. In this case we use $\underline{A}$ and $\underline{R}$[1] for the accept and reject digit respectively.

$$\varphi_T = \text{NT} \wedge \exists n \alpha_T(n)$$

[1]The notation $\underline{x}$ represents a constant that is not a free or quantified variable in the formula. $\underline{x} = \overbrace{ss\ldots s}^{x}0$ (the xth successor of 0)

$$\begin{array}{rcll}\hat{T} \in L_1 & \Rightarrow & \mathcal{N} \models \alpha_T(\underline{n}) & \text{for some } n \in \{0,1,\ldots\} \\ & \Rightarrow & \mathrm{NT} \models \alpha_T(\underline{n}) & \alpha_T(n) \text{ is bounded)} \\ & \Rightarrow & \mathrm{NT} \models \exists n \alpha_T(n) & \\ & \Rightarrow & \mathrm{NT} \models \mathrm{NT} \wedge \exists n \alpha_T(n) & \\ & \Rightarrow & \mathrm{NT} \models \varphi_T & \end{array}$$

Now we will define $\beta_T(n)$ as true if n is a rejecting computation history.

$$\begin{aligned}\beta_T(n) \quad = \quad & pcomp_{\lambda,T}(n) \wedge (\exists p \leq n) D(p,n,\underline{R}) \\ & \wedge \forall (m \leq n) \left[pcomp_{\lambda,T}(m) \rightarrow (\forall p \leq m) \neg D(p,m,\underline{A}) \right]\end{aligned}$$

$$\begin{array}{rcll}\hat{T} \in L_2 & \Rightarrow & \mathcal{N} \models \beta_T(\underline{n}) & \text{(for some } n \in \{0,1,\ldots\}) \\ & \Rightarrow & \mathrm{NT} \models \beta_T(\underline{n}) & \\ & \Rightarrow & \mathrm{NT} \models \exists n \beta_T(n) & \\ & \Rightarrow & \models \mathrm{NT} \rightarrow \exists n \beta_T(n) & \text{(deduction theorem)} \end{array}$$

We can show using just the axioms about $<$ that

$$\mathrm{NT} \models \neg \left[\exists n_1 \alpha_T(n_1) \wedge \exists n_2 \beta_T(n_2) \right]$$

The three cases $n_1 < n_2$, $n_1 = n_2$, and $n_1 > n_2$ can be checked.

Let

$$N = \mathrm{NT}, \quad A = \exists n \alpha_T(n), \quad B = \exists n \beta_T(n)$$

From above we have

$$\begin{array}{l} \models N \rightarrow B \\ \models N \rightarrow \neg(A \wedge B) \end{array}$$

Using prepositional logic alone, we can conclude

$$\begin{array}{ll} \models \neg (N \wedge A) & \\ \models \neg (\mathrm{NT} \wedge \exists n \alpha_T(n)) & \\ \models \neg \varphi_T & \text{by the definition of } N \text{ and } A \end{array}$$

16.2. Presburger Arithmetic

We define Presburger Arithmetic model as

$$\mathcal{P} = (\mathcal{N}, s, +, <, 0)$$

where s is the successor function and $+$ and $<$ and 0 have their normal meaning over $\mathcal{N}$.

Theorem 16.1: $\mathit{Th}(\mathcal{P})$ is decidable.

Proof: by elimination of quantifiers

First we will extend our model to include equivalence classes as follows:

$$\mathcal{PE} = (\mathcal{N}, s, +, <, 0, \equiv_2, \equiv_3, \equiv_4, \ldots)$$

Here, $x \equiv_m y$ means that $x - y$ is divisible by m. Note that if $\mathcal{PE}$ is decidable then so is $\mathcal{P}$ because we can have an expression for $\mathcal{PE}$ that does not have any of the equivalence relations and this would also be an expression in $\mathcal{P}$.

Let

$$\varphi = Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \alpha(x_1, \ldots, x_n)$$

where α is quantifier free. We will now show how quantifiers (and variables) can be removed starting with Q_n. We will rewrite φ without x_n. This can then be done repeatedly to eliminate all the quantifiers.

We will show that given an existential formula $\exists x \alpha(x_1, \ldots, x_{n-1}, x)$ we can construct an equivalent formula $\beta(x_1, \ldots, x_{n-1})$. That is, we construct a β such that for all $a_1, ..., a_{n-1} \in \mathcal{N}$

$$\mathcal{PE} \models \exists x \alpha(a_1, \ldots, a_{n-1}, x) \equiv \beta(a_1, \ldots, a_{n-1}).$$

The case of a universal quantifier reduces to the existential case because $\forall x \alpha(x_1, \ldots, x_{n-1}, x) \equiv \neg \exists x \neg \alpha(x_1, \ldots, x_{n-1}, x)$. Thus, we can construct β so that

$$\mathcal{PE} \models \forall x \alpha(x_1, \ldots, a_{n-1}, x) \equiv \neg \beta(a_1, \ldots, a_{n-1}).$$

There are a number of steps in eliminating the quantifier $\exists x$ from $\exists x \alpha(x_1, \ldots, x_{n-1}, x)$.

1. Remove negations:

$$\begin{aligned}
\neg(t_1 = t_2) &\Rightarrow t_1 < t_2 \lor t_2 < t_1 \\
\neg(t_1 < t_2) &\Rightarrow t_2 < t_1 \lor t_2 = t_1 \\
\neg(t_1 \equiv_k t_2) &\Rightarrow \bigvee_{i=1}^{k-1} t_1 \equiv_k t + \underline{i}
\end{aligned}$$

2. Simplify to conjunctions of atomic positive formulas (a.k.a. DNF conversion):

$$\exists x \alpha(x_1, \ldots, x_n, x) \Rightarrow \exists x \bigvee_{i=1}^{m} \bigwedge_{j=1}^{m} \alpha_{i,j}(x_1, \ldots, x_{n-1}, x)$$

Move the $\exists x$ inside the $\bigvee$. We now have:

$$\bigvee_{i=1}^{m} \exists x \bigwedge_{j=1}^{m} \alpha_{i,j}(x_1, \ldots, x_{n-1}, x)$$

Now, we just have to eliminate quantifiers of the form

$$\exists x(\alpha_1 \land \alpha_2 \land \ldots \land \alpha_m)$$

where each α_i has the form where x is alone on one side of the equation, inequality, or congruence:

$$\begin{array}{ccc}
nx + t = u & \xrightarrow{\text{rewrite}} & nx = u - t \\
nx + t \equiv_k u & & nx \equiv_k u - t \\
nx + t < u & & nx < u - t \\
u < nx + t & & u - t < nx
\end{array}$$

Here, nx denotes n additions of x to itself. The terms t and u are linear combinations of the variables $x_1, \ldots, x_{n-1}$. For example, t and u have the form $j + \sum_{i=1}^{n-1} n_i x_i$, where j is a sum of constants and $n_i x_i$ is the sum of x_i with itself n_i times.

3. Solve for x. This means for all expressions α_i containing $n_i x$, find the least common multiple of n_i. Let p be the least common multiple, $p = \operatorname{lcm}(n_1, \ldots, n_{n-1})$, and let p_i be $\frac{p}{n_i}$ Then transform the statement $\exists x(\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_m)$ to:

$$\exists x(\alpha'_1 \wedge \alpha'_2 \wedge \ldots \wedge \alpha'_m)$$

where

$$\begin{array}{rcll}
\alpha'_i & = & px < p_i u - p_i t & \text{if} \quad \alpha_i = n_i x < u - t \\
\alpha'_i & = & p_i u - p_i t < px & \text{if} \quad \alpha_i = u - t < n_i x \\
\alpha'_i & = & px = p_i u - p_i t & \text{if} \quad \alpha_i = n_i x = u - t \\
\alpha'_i & = & px \equiv_{p_i m} p_i u - p_i t & \text{if} \quad \alpha_i = n_i x \equiv_m u - t
\end{array}$$

Now, we need to make sure that the solution to this set of equalities, inequalities, and congruences is divisible by p and remove px from the expression entirely. To do this we will just add the congruence $x \equiv_p 0$ to obtain the equivalent formula

$$\exists x \ (\alpha''_1 \wedge \alpha''_2 \wedge \ldots \wedge \alpha''_m \wedge x \equiv_p 0)$$

where α'' is simply α' with x replacing px.

4. Special case with equalities: $x = u - t$ is in the conjunction. In this case, just substitute $u - t$ for x in the formula and add the clause $u - t \geq 0$.

5. Finally eliminate x: If we have no equalities then our formula looks like the following:

$$\begin{aligned}
\exists x \quad & [r_0 - s_0 < x \wedge \ldots \wedge r_{l-1} - s_{l-1} < x \\
& \wedge x < t_0 - u_0 \wedge \ldots \wedge x < t_{k-1} - u_{k-1} \\
& \wedge x \equiv_{m_0} v_0 - w_0 \wedge \ldots \wedge x \equiv_{m_{h-1}} v_{h-1} - w_{h-1}]
\end{aligned}$$

In this expression, the terms $r_i - s_i$ represent lower bounds on x. Every $r_i - s_i$ term could be negative of there may be no lower bound terms at all, so we add the inequality $r_l - s_l < x$ with $r_l = \underline{0}$ and $s_l = \underline{1}$ making -1 a strict lower bound. The $t_j - u_j$ represent upper bounds on choices for x. The least common multiple $m = \operatorname{lcm}(m_0, \ldots, m_{h-1})$ represents an spacing on which x must lie on. If there is a x on this spacing that is above all the lower bounds and below all the upper bounds. Then this part of the equation is satisfiable. Our final step is to rewrite as:

$$\begin{aligned}\beta \quad = \quad & \bigvee_{i\leq l} \bigvee_{j<k} \bigvee_{1<q\leq m} \left[\bigwedge_{i'\leq l} (r_{i'} - s_{i'} \leq r_i - s_i) \wedge \bigwedge_{j'<k} \left(t_j - u_j \leq t'_j - u'_j\right) \right. \\ & \left. \wedge \left(r_i - s_i + \underline{q} < t_j - u_j\right) \wedge \bigwedge_{p<h} \left(r_i - s_i + \underline{q} \equiv_{m_p} v_p - w_p\right) \right]\end{aligned}$$

Our decision procedure will be to remove all the quantifiers and then evaluate the final quantifier free formula. □

This quantifier elimination method has very high complexity. It can be shown by a different but related method that $\mathit{Th}(\mathcal{P}) \in \mathit{SPACE}(2^{2^{cn}})$.

Lecture 17

Ehrenfeucht Games and Bounded Quantifiers

March 5, 1998
Lecturer: Richard Ladner
Notes: Jayram S. Thathachar

The Ehrenfeucht game techniques shown in this section are found in the book of Ferrante and Rackoff [6]

In this lecture, we will consider the problem of deciding validity of first-order sentences in theories by bounding the range of the quantifiers in finite subsets. Suppose $Qx_1 \ldots Qx_n \alpha(x_1, x_2, \ldots x_n)$ is a first order sentence whose validity in needs to determined in some first-order theory. If we can bound the range of each of the variables x_i in some finite set, and still obtain an equivalent formula, then this problem becomes decidable. A technique that has proved useful for this approach is to analyze the validity of the formula using *Ehrenfeucht games*. As an illustration of this technique, we consider the first-order theory of arithmetic with only the less-than predicate, denoted by $\mathcal{N}_< = (N, <)$, and the decidability problem for sentences in $\mathrm{Th}(\mathcal{N}_<)$.

17.1. Ehrenfeucht Games

An Ehrenfeucht game is a 2-person game with players P_1 and P_2, and sets $S_1, S_2, \ldots S_n$, and $T_1, T_2, \ldots T_n$, each a subset of N. The game is played in n stages by P_1 and P_2 taking alternate turns with P_1 playing first. In the i-th stage, P_1 chooses to pick either a point $a_i \in S_i$ or a point $b_i \in T_i$. P_2 has to pick a point $b_i \in T_i$ in the former case, and a point $a_i \in S_i$ in the latter case. P_2 wins at the end of n stages if and only if for all $1 \leq i < i' \leq n$, $a_i < a_{i'}$ if and only if $b_i < b_{i'}$. The Ehrenfeucht game is good if P_2 has a winning strategy irrespective of the strategy used by P_1. For the problem of bounding the quantifiers, $S_i = N$ and $T_i = [0, B_i]$, for all i, where the B_i's are integers representing the bounds. For the rest of this section fix the sets S_i's and T_i's.

We formalize the notion that P_2 always has a winning strategy in the following way. For each k, we recursively define relations $\equiv_{k,n-k}$ between the sets $S_1 \times \cdots \times S_{n-k}$ and $T_1 \times \cdots \times T_{n-k}$. (These are not necessarily equivalence relations.)

For $k = 0$, $(a_1, \ldots, a_n) \equiv_{0,n} (b_1, \ldots, b_n)$ is true whenever $a_i < a_{i'}$ holds if and only if $b_i < b_{i'}$ holds for all $i < i'$. (This corresponds to the fact that at the end of the game P_2 has chosen the right set of points.)

For $k > 0$, $(a_1, \ldots, a_{n-k}) \equiv_{k,n-k} (b_1, \ldots, b_{n-k})$ is true whenever

$$(\forall a_{n-k+1} \in S_{n-k+1})(\exists b_{n-k+1} \in T_{n-k+1})(a_1, \ldots, a_{n-k+1}) \equiv_{k-1,n-k+1} (b_1, \ldots, b_{n-k+1}), \text{ and },$$

$$(\forall b_{n-k+1} \in T_{n-k+1})(\exists a_{n-k+1} \in S_{n-k+1})(a_1, \ldots, a_{n-k+1}) \equiv_{k-1,n-k+1} (b_1, \ldots, b_{n-k+1}).$$

This corresponds to the fact that P_2 must always have a move no matter what P_1 chooses to do. Note that when $k = n$, the relation $() \equiv_{n,0} ()$ implies that P_2 always has a winning strategy.

The simple but remarkable fact is that any formula with n quantifiers ranging over the S_i's is equivalent to the same formula with the quantifiers ranging over the T_i's. This is expressed in the more general lemma below by substituting $k = n$.

Notation: Denote

$$\mathcal{N}_<, x_1 = a_1, x_2 = a_2, \ldots, x_{n-k} = a_{n-k} \models Q_1 x_1 \ldots Q_n x_n \alpha(x_1, x_2, \ldots x_n)$$

by the simpler form

$$\mathcal{N}_< \models Q_{n-k+1} x_{n-k+1} \ldots Q_n x_n \alpha(a_1, \ldots, a_{n-k}, x_{n-k+1}, \ldots, x_n)$$

Lemma 17.1: Fix any k, $a_1 \in S_1, \ldots, a_{n-k} \in S_{n-k}$, and $b_1 \in T_1, \ldots, b_{n-k} \in T_{n-k}$. Then, $(a_1, \ldots, a_{n-k}) \equiv_{k,n-k} (b_1, \ldots, b_{n-k})$ if and only if for all quantifier-free formulas $\alpha(x_1, x_2, \ldots, x_n)$, and all quantifiers $Q_{n-k+1}, \ldots, Q_n$,

$$\begin{aligned}\mathcal{N}_< \ \models\ & (Q_{n-k+1} x_{n-k+1} \in S_{n-k+1}) \ldots (Q_n x_n \in S_n) \alpha(a_1, \ldots, a_{n-k}, x_{n-k+1}, \ldots, x_n) \\ & \leftrightarrow (Q_{n-k+1} x_{n-k+1} \in T_{n-k+1}) \ldots (Q x_n \in T_n) \alpha(b_1, \ldots, b_{n-k}, x_{n-k+1}, \ldots, x_n).\end{aligned}$$

Proof: Let $P(k)$ denote the statement of the lemma. The proof is by induction on k. When $k = 0$, there are no quantifiers. By the definition of $\equiv_{0,n}$, for all $1 \leq i, j \leq n$, $a_i < a_j$ if and only if $b_i < b_j$. Since $\alpha(x_1, \ldots, x_n)$ is equivalent to a Boolean combination of atomic formulas of the form $x_i < x_j$, then

$$\mathcal{N}_< \models \alpha(a_1, \ldots, a_n) \leftrightarrow \alpha(b_1, \ldots, b_n).$$

For the induction step, we assume that $P(k)$ holds and show below that $P(k+1)$ is true.

("only if"): Here we are assuming that $(a_1, \ldots, a_{n-k-1}) \equiv_{k+1,n-k-1} (b_1, \ldots, b_{n-k-1})$. By the definition of $\equiv_{k+1,n-k-1}$ and by the induction hypothesis $P(k)$, we have that for any $t \in T_{n-k}$, there exists a $s \in S_{n-k}$ such that

$$\begin{aligned}\mathcal{N}_< \ \models\ & (Q_{n-k+1} x_{n-k+1} \in S_{n-k+1}) \ldots (Q_n x_n \in S_n) \alpha(a_1, \ldots, a_{n-k-1}, s, x_{n-k+1}, \ldots, x_n) \\ \leftrightarrow\ & (Q_{n-k+1} x_{n-k+1} \in T_{n-k+1}) \ldots (Q_n x_n \in T_n) \alpha(b_1, \ldots, b_{n-k-1}, t, x_{n-k+1}, \ldots, x_n)\end{aligned} \tag{17.1}$$

Fix any $\alpha(x_1, x_2, \ldots x_n)$ and we are required to prove that

$$\begin{aligned}\mathcal{N}_< \ \models\ & (Q_{n-k} x_{n-k} \in S_{n-k}) \ldots (Q_n x_n \in S_n) \alpha(a_1, \ldots, a_{n-k-1}, x_{n-k}, \ldots, x_n) \\ \leftrightarrow\ & (Q_{n-k} x_{n-k} \in T_{n-k}) \ldots (Q x_n \in T_n) \alpha(b_1, \ldots, b_{n-k-1}, x_{n-k}, \ldots, x_n)\end{aligned} \tag{17.2}$$

It suffices to prove Equation 17.2 when Q_{n-k} is a universal quantifier since we can always negate both sides of the equivalence. So, assume that $Q_{n-k} = \forall$. We prove the forward direction of Equation 17.2; the other direction has a similar proof because of symmetry.

To prove this direction, fix any $t \in T_{n-k}$. By our earlier observation, there is an s satisfying Equation 17.1. Because we are assuming that the forward direction of Equation 17.2 is true, we can put $x_{n-k} = s$ to conclude that the premise of Equation 17.1 is holds. Therefore, the consequence of Equation 17.1 also holds. Since t was picked arbitrarily, the forward direction of Equation 17.2 follows.

("if"): Here we are assuming that for all $\alpha(x_1, x_2, \ldots x_n)$

$$\begin{aligned}\mathcal{N}_< \quad \models \quad & (Q_{n-k}x_{n-k} \in S_{n-k}) \ldots (Q_n x_n \in S_n)\alpha(a_1, \ldots, a_{n-k-1}, x_{n-k}, \ldots, x_n) \\ & \leftrightarrow (Q_{n-k}x_{n-k} \in T_{n-k}) \ldots (Qx_n \in T_n)\alpha(b_1, \ldots, b_{n-k-1}, x_{n-k}, \ldots, x_n)\end{aligned}$$

and we are required to prove $(a_1, \ldots, a_{n-k-1}) \equiv_{k+1,n-k-1} (b_1, \ldots, b_{n-k-1})$. The proof is left as an exercise. □

17.2. Bounding Quantifiers in $\mathcal{N}_<$

We will now give explicit bounds for the B_i's and prove that P_2 always has a winning strategy with respect to these bounds. Define

$$B_1 = 2^{n-1}, B_2 = 2^{n-1} + 2^{n-2}, \ldots, B_n = 2^{n-1} + \cdots + 2^0.$$

Let $T_i = N$ and $S_i = [0, B_i]$ for all i.

Lemma 17.2: $() \equiv_{n,0} ()$, i.e. P_2 always has a winning strategy.

Proof: We will show at in the stage with k moves left, whether P_1 picks a_{n-k+1} or b_{n-k+1}, P_2 can always choose some b_{n-k+1} or a_{n-k+1}, respectively, such that for all $1 \le i < j \le n-k$, $a_i < a_j$ if and only if $b_i < b_j$. We will actually maintain the following stronger set of invariants $I(k)$ (with $a_0 = b_0 = 0$): for all $0 \le i < j \le n-k$,

I1: $a_i < a_j$ if and only if $b_i < b_j$.

I2: $|a_j - a_i| < 2^k \Rightarrow a_j - a_i = b_j - b_i$.

I3: $|a_j - a_i| \ge 2^k \Rightarrow |b_j - b_i| \ge 2^k$.

We prove that $I(k)$ holds by downward induction on k. For $k = n$, $I(k)$ holds vacuously. Assume that $I(k)$ holds for $k \le n$ and we want to now show that $I(k-1)$ holds. First, it can be verified that $I(k-1)$ holds whenever $i, j \le n-k$ so we only need to verify the invariant when one of i or j is $n-k+1$. Suppose P_1 picks $a_{n-k+1} \in N$. (An analogous proof shows that $I(k-1)$ can be satisfied if P_1 picks $b_{n-k+1} \in B_{n-k+1}$.) We have the following cases of choosing b_{n-k+1} which depends on the range from which a_{n-k+1} was chosen.

Case 1: Suppose that $a_i < a_{n-k+1} < a_j$, for some i and j and for $\ell \neq i, j$, $a_\ell \notin [a_i, a_j]$ By the induction hypothesis no other b_ℓ lies in the interval, $[b_i, b_j]$ for $\ell \neq i, j$. Choose

$$b_{n-k+1} = \begin{cases} b_j - (a_j - a_{n-k+1}) & \text{if } a_j - a_{n-k+1} < 2^{k-1} \\ b_i + (a_{n-k+1} - a_i) & \text{if } a_{n-k+1} - a_i < 2^{k-1} \\ \left\lceil \frac{b_i + b_j}{2} \right\rceil & \text{otherwise} \end{cases}$$

(Observe that when $a_j - a_i < 2^k$, it is possible that both $a_j - a_{n-k+1} < 2^{k-1}$ and $a_{n-k+1} - a_i < 2^{k-1}$ are true but because of Invariant I2, b_{n-k+1} is still well-defined.

Case 2: Suppose that $a_i < a_{n-k+1}$ for all $i \leq n-k$. Let i be such that a_i is the maximum of all a_j, $j \leq n-k$. In this case, choose

$$b_{n-k+1} = \begin{cases} b_i + (a_{n-k+1} - a_i) & \text{if } a_{n-k+1} - a_i < 2^{k-1} \\ B_{n-k+1} & \text{otherwise} \end{cases}$$

We will not prove it here but it can be verified in each case that $I(k-1)$ holds. $\square$

Corollary 17.3: $\mathrm{Th}(\mathcal{N}_<) \in \mathrm{PSPACE}(n^2)$. Moreover, it can be accepted by an alternating TM with n alternations and quadratic-time. To see this, the alternating machine follows the alternation of quantifiers using existenial moves for the existential quantifiers and universial moves for the universal quantifiers. In alternation i, the machine produces a number b_i in the range $[0, B_i]$. This number is of linear length in the formula. In the end the quantifier-free formula is checked for truth by substituting the b_i's for the variables x_i.

Lecture 18

Super-Exponential Complexity of Presburger Arithmetic

March 10, 1998
Lecturer: Richard Ladner
Notes:

The results in this section can be found in the paper by Fischer and Rabin [7].

Recall that Presburger Arithmetic is the theory of the natural numbers with addition. More specifically, let $\mathcal{P} = (N, s, +, <, 0)$. Presburger arithmetic is $\mathrm{Th}(\mathcal{P})$.

We have the following theorem proved by Fischer and Rabin in 1974.

Theorem 18.1: $\mathrm{Th}(\mathcal{P}) \notin \mathrm{NTIME}(2^{2^{cn}})$ for some $c > 0$.

Proof: The idea is to build a formula of linear size that describes the computation of a nondeterministic Turing machine that runs in double exponential time. There are a number of building blocks needed in order to build our formula.

We begin by defining a formula $M_n(x, y, z)$ if length $O(n)$ with the property for all nonnegative integers a, b, and c.

$$\mathcal{P} \models M_n(a, b, c) \text{ if and only if } a < 2^{2^n} \text{ and } ab = c.$$

As can be seen the formula M_n defines a limited form of multiplication. We build M_n inductively on n.

$$M_0(x, y, z) = (x = 0 \wedge z = 0) \vee (x = 1 \wedge z = y).$$

In order to understand the formula M_n for $n > 0$ we use the following fact about nonnegative integers:

$$x < 2^{2^{n+1}} \Leftrightarrow \exists x_1, x_2, x_3, x_4 < 2^{2^n} (x = x_1 x_2 + x_3 + x_4).$$

Hence, $xy = x_1(x_2 y) + x_3 y + x_4 y$. If we let $u_1 = x_2 y$, $u_2 = x_1 u_1$, $u_3 = x_3 y$, $u_4 = x_4 y$ then we can define M_{n+1} as:

$$\begin{aligned} M_{n+1}(x, y, z) \quad = \quad & (\exists u_1 u_2 u_3 u_4 x_1 x_2 x_3 x_4)(\forall w_1 w_2 w_3)[(((w_1 = x_2 \wedge w_2 = y \wedge w_3 = u_1) \\ & \vee (w_1 = x_1 \wedge w_2 = u_1 \wedge w_3 = u_2) \vee (w_1 = x_3 \wedge w_2 = y \wedge w_3 = u_3) \\ & \vee (w_1 = x_4 \wedge w_2 = y \wedge w_3 = u_4)) \Rightarrow M_n(w_1, w_2, w_3)) \wedge z = u_2 + u_3 + u_4] \end{aligned}$$

The key thing to observe is that the length of M_n is linear in n.

Now, M_n is a good formula defining multiplication by numbers less than 2^{2^n}. Unfortunately, this would only let us describe computations of exponential length, not double exponential length. So we have to go a bit further.

We will now describe a formula M'_n with the property

$$\mathcal{P} \models M'_n(a,b,c) \text{ if and only if } a,b,c < g(n) \text{ and } ab = c$$

where g is a function such that $g(n) \geq 2^{2^{2^{n+1}}}$.

Here we need a little number theory. Define $m(n) = 2^{2^{n+2}}$ and

$$g(n) = \prod_{\substack{p<m(n) \\ p \text{ prime}}} p.$$

We define three auxiliary formulas P_n, G_n, and C_n, each of length $O(n)$. We begin with P_n that has the following meaning.

$$\mathcal{P} \models P_n(a) \text{ if and only if } a < m(n) \text{ and } a \text{ is prime.}$$

Recall is that $\mathcal{P} \models M_{n+2}(a,0,0)$ if and only if $a < 2^{2^{n+2}} = m(n)$. Thus, we can define P_n as follows:

$$P_n(x) = M_{n+2}(x,0,0) \wedge \forall x \forall y (M_{n+2}(y,z,x) \Rightarrow (y = 1 \vee y = x)).$$

Next, G_n has the following meaning.

$$\mathcal{P} \models G_n(a) \text{ if and only if } a = g(n).$$

Using the fact that $g(n)$ is the smallest number divisible by all primes less than $m(n)$, we define G_n as follows:

$$G_n(x) = \forall p(P_n(p) \Rightarrow \exists z M_{n+2}(p,z,x)) \wedge \forall y(\forall p(P_n(p) \Rightarrow \exists z M_{n+2}(p,z,y)) \Rightarrow x \leq y)$$

Next, C_n has the following meaning.

$$\mathcal{P} \models C_n(a,b,c,p) \text{ if and only if } p < m(n) \text{ and } ab \equiv c \bmod p.$$

Notice that $xy \equiv z \bmod p$ if and only if there are q_1, q_2, q_3, $r_1 < p$, and $r_2 < p$ such that

$$x = pq_1 + r_1, \quad r_1 y = pq_2 + r_2, \quad z = pq_3 + r_2$$

Each of the multiplications above has its first multiplicand $< m(n)$ so that we can use our previously defined multiplication predicate. Thus we can define C_n as follows using u_1, u_2, u_3, and u_4 to hold the various products.

$$\begin{aligned} C_n(x,y,z,p) \quad = \quad & M_{n+2}(p,0,0) \wedge \exists u_1 u_2 u_3 u_4 q_1 q_2 q_3 r_1 r_2 [r_1 < p \wedge r_2 < p \\ & \wedge M_{n+2}(p,q_1,u_1) \wedge x = u_1 + r_1 \\ & \wedge M_{n+2}(r_1,y,u_2) \wedge M_{n+2}(p,q_2,u_3) \wedge u_2 = u_3 + r_2 \\ & \wedge M_{n+2}(p,q_3,u_4) \wedge z = u_4 + r_2] \end{aligned}$$

Given G_n and C_n we can now define M'_n as follows.

$$\begin{aligned} M'_n(x,y,z) &= \exists w(G_n(w) \wedge x < w \wedge y < w \wedge z < w \\ &\quad \forall p(p < w \Rightarrow C_n(x,y,z,p)). \end{aligned}$$

So we can define multiplication for numbers less than $2^{2^{2^n}}$. In order to simulate a Turing machine like we did in Gödel's incompleteness theorm we need bounded exponentiation also. That is, we need a formula Pow_n of length $O(n)$ with the property that

$$\mathcal{P} \models Pow_n(a,b,c) \text{ if and only if } a, b^a, , c < g(n) \text{ and } b^a = c.$$

To do this in the required length we need to define bounded powering (in conjuction with multiplication) in a single formula E_k with the property If $a < 2^{2^k}, b^a, c < 2^{2^n}$

$$\mathcal{P} \models E_k(a,b,c,A,B,C) \text{ if and only if } A < 2^{2^n} AB = C, \text{ and } b^a = c.$$

We can begin with

$$E_0(x,y,z,u,v,w) = [(x = 0 \wedge z = 1) \vee (x = 1 \wedge z = y)] \wedge M_n(u,v,w).$$

To define $E_{k+1}(x,y,z,u,v,w)$ note that $x < 2^{2^{k+1}}$ if and only if $x = x_1 x_2 + x_3 + x_4$ for some $x_1, x_2, x_3, x_4 < 2^{2^k}$. Thus $y^x = (y^{x_1})^{x_2} \cdot y^{x_3} \cdot y^{x_4}$. Thus, E_{k+1} can be defined recursively from E_k.

Now that we have powering defined for numbers less than 2^{2^n} we can using the modular arithmetic technique define powering for numbers less than $g(n)$.

Using Gödel's proof we can, given any nondeterministic Turing machine T that runs in time $2^{2^{cn}}$ and input x of length n, construct a formula F of length $O(n)$ with the property that $\mathcal{P} \models F$ if and only if x is accepted by T. This is sufficient to prove the theorem. $\square$

Lecture 19

Categoricity and Second Order Predicate Logic

March 12, 1998
Lecturer: Richard Ladner
Notes: Nicholas Bone

In this lecture we cover some techniques to prove decidability that don't immediately give us algorithms. These techniques are covered nicely in the survey by Rabin [13]. In this lecture we will introduce two new concepts and briefly review the relationship between complexity and logic. We begin with a theorem.

Theorem 19.1: Let Γ be a complete, recursively ennumerable theory. Then Γ is decidable.

Proof: If Γ is inconsistent then it must contain everything (because it is a theory), and thus it is decidable. If Γ is consistent then $\forall \alpha, \alpha \in \Gamma \Leftrightarrow \neg\alpha \notin \Gamma$. Thus, any Turing machine that enumerates Γ will eventually enumerates either α or $\neg\alpha$, and this gives us a decision procedure for Γ. $\square$

19.1. Categoricity

Definition: A consistent theory Γ is λ-categorical if and only if all models of Γ of cardinality λ are isomorphic.

Theorem 19.2: Let Γ be a consistent λ-categorical theory (for some infinite λ) with no finite model. Then Γ is complete.

Proof: In order to prove this theorem we need another theorem called the "upward Lowenheim-Skolem theorem." It states that if Γ is a theory with an infinite model then it has a model of cardinality λ. Naturally, we are assuming that Γ only uses countably many symbols. Consider the sentences

$$\Delta = \Gamma \cup \{c_x \neq c_y : x \neq y \text{ and } x, y \in C\}$$

where C is a set of cardinality λ. Every finite subset of Δ is satisfiable because Γ has an infinite model that can be extended to include interpretations for any finite subset of the new constant symbols. By the compactness theorem Δ has a model. Now, use the Herbrand construction to build a model of Δ. The number of terms used in the construction has cardinality λ. Furthermore, the number of equivalence classes of terms in the model has the same cardinality because the equivalence class of each c_x is distinct from all the others. Thus, the Herbrand model has cardinality λ.

Now to show the Γ is complete, let α be a sentence. Let $\mathcal{M}$ be *the* model of Γ of cardinality λ. Either α is true or false in $\mathcal{M}$. Without loss of generality assume $\mathcal{M} \models \alpha$. We will now show that $\alpha \in \Gamma$. Suppose not. Then there is a model $\mathcal{M}'$ of $\Gamma \cup \{\neg\alpha\}$. $\mathcal{M}'$ is not finite by the hypothesis of the theorem. Thus, $\Gamma \cup \{\neg\alpha\}$ must have a model of cardinality λ by the upward Lowenheim-Skolem theorem. That model is isomorphic to $\mathcal{M}$. Thus, $\mathcal{M} \models \neg\alpha$, a contradiction. □

19.1.1. Dense Linear Order without endpoints (DLO)

$CN(DLO) = Th(\mathcal{Q}, <)$ is recursively enumerable, and $\aleph_0$-categorical (from homework 3)**. There are various methods of proving this.

(**Footnote: $\aleph_0 = |\mathcal{N}|$ is the "smallest" infinity. $\aleph_1$ is defined as the "next smallest", and so on. The Continuum Hypothesis states that $\aleph_1 = 2^{\aleph_0}$. This is unprovable from any standard axioms of set theory.)

19.1.2. Algebraically closed fields of characteristic zero

The theory of algebraically closed fields is $2^{\aleph_0}$-categorical ($2^{\aleph_0} = |\mathcal{R}|$). In general, any algebraically closed field of characteristic zero may be described by three sets of axioms: the field axioms, the closure axioms, and the axioms that define the characteristic. We call these axioms ACF.

The Field Axioms: The classic field axioms hold for the rationals, reals, complex numbers, integers mod a prime, and many other important mathematical structures.

1.	$\forall x(x + 0 = x \wedge 0 + x = x)$	(additive identity)
2.	$\forall x \forall y(x + y = x \Rightarrow y = 0)$	(0 is unique)
3.	$\forall x \exists y(x + y = 0 \wedge y + x = 0)$	(additive inverse)
4.	$\forall x \forall y \forall z((x + y = 0 \wedge x + z = 0) \Rightarrow y = z)$	(inverse is unique)
5.	$\forall x(1 \neq 0 \wedge x * 1 = x \wedge 1 * x = x)$	(multiplicative identity)
6.	$\forall x \forall y(x * y = x \Rightarrow (y = 1 \vee x = 0))$	(1 is unique)
7.	$\forall x \forall y(x * y = 0 \Rightarrow (x = 0 \vee y = 0))$	(no zero divisors)
8.	$\forall x \exists y(x = 0 \vee (x * y = 1 \wedge y * x = 1))$	(multiplicative inverse)
9.	$\forall x \forall y \forall z((x * y = 1 \wedge x * z = 1) \Rightarrow y = z)$	(inverse is unique)

Axioms 1-4 describe a group, 1-6 a ring with unit, 1-7 an integral domain, and the nine together describe a field.

The Closure Axioms: A field is algebraically closed if and only if it contains the roots of all its polynomials. The closure axioms are the following countable set:

1. $\forall y_0 \forall y_1 \exists x(y_0 + y_1 * x = 0)$
2. $\forall y_0 \forall y_1 \forall y_2 \exists x(y_0 + y_1 * x + y_2 * x * x = 0)$
3. $\forall y_0 \forall y_1 \forall y_2 \forall y_3 \exists x(y_0 + y_1 * x + y_2 * x * x + y_3 * x * x * x = 0)$
4. $\ldots$.

Characteristic Zero: If there is a finite positive integer p such that $\overbrace{1 + 1 + \cdots + 1}^{p} = 0$ and for all $0 < p' < p$ $\overbrace{1 + 1 + \cdots + 1}^{p'} \neq 0$, then we say the field has characteristic p. If there is no such p

the field is of characteristic 0. (Note that the only finite fields are of characteristic that is a prime power. We can define this as follows:

1. $1+1 \neq 0$
2. $1+1+1 \neq 0$
3. $1+1+1+1+1 \neq 0$
4. ... etcetera.

It can be shown that every model of ACF of cardinality is isomorphic to $(\mathcal{C},+,*,0,1)$ where $\mathcal{C}$ is the complex numbers. So that $CN(ACF)$ is $2^{\aleph_0}$-categorical. $CN(ACF)$ has no finite models because of the "characteristic zero" axioms. Thus by theorem 19.2 $CN(ACF)$ is complete and equal to $Th((\mathcal{C},+,*,0,1))$. Finally ACF is recursively enumerable and so is $CN(ACF)$. Hence, by theorem 19.1 $CN(ACF)$ is decidable. Hence $Th((\mathcal{C},+,*,0,1))$ is decidable. This is an interesting proof because it gives no idea of the complexity of any algorithm to decide $Th((\mathcal{C},+,*,0,1))$.

Corollary 19.3: There is no way to define $\mathcal{Z}$ (the integers) in $Th((\mathcal{C},+,*,0,1))$.

Proof: Suppose that we could define $\mathcal{Z}$ in $Th((\mathcal{C},+,*,0,1))$. Then $Th((\mathcal{Z},+,*,0,1))$ would be reducible to $Th((\mathcal{C},+,*,0,1))$, but this is impossible because $Th((\mathcal{Z},+,*,0,1))$ is undecidable. □

19.1.3. Real ordered fields

Consider $Th((\mathcal{R},+,*,<,0,1))$. This is not categorical for any λ. However, it is decidable. Decision procedures can be arrived at using a variety of techniques including elimination of quantifiers and Ehrenfeucht games.

Note: $Th(\mathcal{C},+,*,0,1)$ has a countable model call the algebraic numbers. The algebraic numbers are formed my including all the roots to polynomials over the rationals and closing under addition, substraction, multiplication, and division. The algebraic numbers do not contain *trancendental numbers*, such as e and π.

19.2. Second Order Predicate Logic

Second order predicate logic is stronger than first order predicate logic in that we can quantify over sets, relations, and functions as well as the elements in the universe. First a definition:

$CN_2(\Gamma)$ = the second order consequences of Γ.

Note that $CN_2(\emptyset)$ (the second order validities) is not recursively enumerable. In fact, $CN_2(\emptyset)$ is *non-arithmetic*, which means that it cannot be reduced to any member of the infinite chain:

$$\phi_0 <_T \phi_1 <_T \phi_2 <_T \cdots,$$

where $\phi_0 = \emptyset$ and $\phi_{i+1} = K^{\phi_i}$.

Define $S1S = Th_2^m(\mathcal{N},s,0)$ = the monadic second order theory of successor. That is, we allow quantification over sets. Note that this is essentially the same as quantifying over unary relations, because any set A may be defined by the relation $R_A(x) \Leftrightarrow x \in A$. The following is an example of a valid formula in $S1S$:

$$\forall X \forall y \forall z((0 \in X \wedge (y \in X \Leftrightarrow s(y) \notin X)) \Rightarrow (z \in X \Leftrightarrow s(s(z)) \in X))$$

Theorem 19.4: $S1S$ is decidable.

This result was proved in 1960 by Büchi using a reduction to the emptiness problem for ω-automata (automata on infinite strings).

Theorem 19.5: $S1S$ is *non-elementary.* That is, $S1S \notin \mathrm{SPACE}(2^{2^{\cdot^{\cdot^{2^n}}}})$ for any finite stack of 2's.

This result was proved in 1975 by Albert Meyer.

19.3. Summary of Complexity of Logical Theories

We conclude the lecture with a brief review of the complexity of logical problems.

	Prop. Logic	Modal Logic	Temp. Logic	Knowledge & Time	1^{st} order Logic	2^{nd} order Logic
NA				|	$Th(\mathcal{N},+,*)$	validities
RE				|	validities	|
NE				|	|	$S1S$, $S2S$
2-EXP			CTL^*	|	$Th(\mathcal{N},+)$	
EXP		common-knowledge	CTL	|	$Th(\mathcal{R},+)$	
PSPACE	QBF	K-modal, 2-knowledge	LTL		$Th(=)$, $Th(<)$, $Th(s)$	
NP	$PROP$	1-knowledge				
P	$HORN$					

Complexity classes are shown on the vertical axis (NA = non-arithmetic, RE = recursively enumerable, NE = non-elementary). Logic systems are shown on the horizontal axis. A vertical line (|) represents the existence of natural problems in that logic system and complexity class that we wish to leave nameless.

As can be seen, very few logical theories have worst case complexity even in NP and many are not even arithmetic. This appears to put a crimp on trying to directly uses classical logic as a basis for automatic verification of programs. Nonetheless, there are fragments of varies theories that appear to be effective tools in verification. Certainly, model checking has been proven to a success in hardware verification.

Hopefully, this course has provided insight into the power and complexity of logic.

References

[1] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, Vol. 8. pp 244-263. 1986.

[2] E.A. Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science, Vol. B, Formal Models and Semantics*. Ed. J. van Leeuwen. pp. 995-1072. 1990.

[3] E.A. Emerson and C. Jutla. The Complexity of Tree Automata and Logic of Programs. Proceedings of the 29th IEEE Symposium on Foundations of Computer Science. pp. 328-337. 1988.

[4] H.B. Enderton. *A Mathematical Introduction to Logic.* Academic Press. New York. 1972.

[5] R. Fagin, J.Y. Halpern, Y. Moses, and M.Y. Vardi. *Reasoning about Knowledge.* The MIT Press. Cambridge, MA. 1995.

[6] J. Ferrante and C.W. Rackoff. *The Computational Complexity of Logical Theories.* Springer-Verlag Lecture Notes in Mathematics. New York. 1979.

[7] M.J. Fischer and M.O. Rabin. Super-Exponential Complexity of Presburger Arithmetic. In *Complexity of Computation.* Ed. R.M. Karp. pp. 27-41. 1974.

[8] M.J. Fischer and R.E. Ladner. Propositional Dynamic Logic of Programs. *Journal of Computer and System Sciences*, Vol. 18. pp. 194-211. 1979.

[9] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley Publishing Company. Reading, MA. 1979.

[10] S.A. Kripke. Semantical Analysis of Modal Logic, I Normal Modal Propositional Calculi. *Z. Math. Logik Grunlagen Math.* Vol. 9. pp. 67-96. 1963.

[11] R.E. Ladner. The Computational Complexity of Provability in Systems of Modal Propositional Logic. *SIAM Journal on Computing* Vol. 6, No. 3. pp. 467-480. 1977.

[12] C.H. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company. Reading, MA. 1994.

[13] M.O. Rabin. Decidable Theories. In *Handbook of Mathematical Logic.* Ed. J. Barwise. pp. 596-629. 1977.

[14] A.P. Sistla and E.M. Clarke. The Complexity of Propositional Linear Temporal Logic. *Journal of the ACM*, Vol. 32. pp. 733-749. 1985.

[15] M.Y. Vardi and L. Stockmeyer. Improved Upper and Lower Bounds for Modal Logics of Programs. Proceedins of the Seventeenth Annual ACM Symposium on Logic in Computer Science. pp. 240-251. 1985.