185.A09 Advanced Mathematical Logic

www.volny.cz/behounek/logic/teaching/mathlog13

Libor Běhounek, behounek@cs.cas.cz

Lecture #6, January 7, 2014

This and the next lecture give a brief sketch of topics related to (un)decidability and (in)completeness of theories. Some simplifications had to be made in this exposition; for a thorough and rigorous exposition consult the literature on classical mathematical logic.

Models of computation

There are several widely known mathematical models of computation, or algorithmic manipulation with symbols. Let us briefly describe just three of them.

Turing machine. Informally, a Turing machine consists of:

- An infinite *tape* divided into cells, each of which can contain a symbol of a given finite alphabet; the alphabet is supposed to contain a special *blank* symbol.
- A *head* that can read and write the symbols on a tape, and move over the tape to the right or to the left by one cell in each step.
- A *state register* registering one of the finitely many states in which the machine can be; one of the states is designated as the *start state,* and a subset of states is designated as the *final states.*
- And a *program,* or a finite set of instructions of the form

$$\langle state_1, symbol_1, symbol_2, direction, state_2\rangle.$$

The machine works in *steps*. In each step, the instruction matching the state of the machine and the symbol under its head is executed, interpreted as:

> "If the machine is in $state_1$ and the head reads $symbol_1$, overwrite the cell to $symbol_2$, move the head in the *direction* (left, right, or stay) and assume $state_2$".

Initially, the machine is in the initial state, and the tape is supposed to contain only finitely many non-blank symbols. The computation terminates as soon as the the machine reaches one of the final states; the contents of the tape is then the output of the computation of the machine for the input given by the initial state of the tape.

This informal description can be formalized as a mathematical object in a straightforward way (as a 7-tuple of suitable sets of symbols, states, and instructions).

Register machine with a structured programming language. The machine has infinitely many *registers* for natural numbers (only finitely many of which are initialized by non-zero numbers) and a *program* consisting of *instructions* of the form INC(X) (increment by 1 of the register X), DEC(X) (decrement), CLR(X) (setting to 0), and $X := Y$ (assignment); concatenations **begin** $P_1; \ldots; P_n$ **end** of programs; and constructions **for** X **do** P (perform the program P X-times) and **while** $X \neq 0$ **do** P (with the usual meaning).

Exercise. *Program addition of natural numbers on the register machine.*

Recursive functions. *Arithmetic functions* are all functions $f\colon X \to N$, where $X \subseteq N^k$. If $X = N^k$, the arithmetic function is *total.*

Basic recursive functions are the functions $o(x) = 0$ (zero function), $s(x) = x+1$ (successor), and $p_i^n(x_1, \ldots, x_n) = x_i$ (i-th projection of n).

An $(n+1)$-ary function is derived by *primitive recursion* from the n-ary function g and the $(n+2)$-ary function h if

$$f(0, y_1, \ldots, y_n) = g(y_1, \ldots, y_n)$$
$$f(x+1, y_1, \ldots, y_n) = h(f(x, y_1, \ldots, y_n), x, y_1, \ldots, y_n)$$

An n-ary function f is derived by *minimization* from the $(n+1)$-ary function g if:
$f(x_1, \ldots, x_n) = y$ iff $[g(y, x_1, \ldots, x_n) = 0$ and $(\forall v < y)(g(v, x_1, \ldots, x_n) \neq 0)]$

The set of *primitive recursive functions* is the smallest set of arithmetic functions containing the basic recursive functions and closed under functional composition and primitive recursion.

The set of *partial recursive functions* is the smallest set of arithmetic functions containing basic recursive functions and closed under functional composition, primitive recursion, and minimization.

An arithmetic function is *recursive* if it is partially recursive and total.

A set of natural numbers A is *(primitive) recursive* if its characteristic function χ_A is (primitive) recursive; it is *recursively enumerable* if there is a partial recursive function f such that $A = \mathrm{Dom}(f)$.

Church thesis

By a suitable encoding of the alphabet, any finite input and output can be encoded by (tuples of) natural numbers, and vice versa, any (tuple of) natural numbers can be encoded by a sequence of symbols of any (at least two-element) alphabet. Any function computed by one of the above computational models can thus be regarded as an arithmetical function.

By mutual simulation (under a suitable encoding) one can show that all of the aforementioned models of computation are equivalent. In particular, the (arithmetical) functions computable by a Turing machine or a structured program are exactly the partial recursive functions. (Recursive functions are those computable by a structured program not using **while**.)

Since Turing machines or structured programs capture our intuitive idea of algorithmic computation, partial recursive functions represent the arithmetical model of computable functions.

The identification of algorithmically computable functions with partial recursive functions (or equivalently, functions computable by any model of computation equivalent to Turing

machines) is called the *Church* (or *Church–Turing*) *thesis.* N.B.: The Church thesis is not a mathematical statement (it refers to the intuitive idea of algorithm). Caveat: Not every 'computation' corresponds to a Turing machine—e.g., if the inputs are dynamically changing during the computation (e.g., the results of measurement of a real-world process).

Not all arithmetic functions are computable. The computability of functions is studied by *recursion theory,* an important branch of mathematical logic (and the foundations of computer science). An example of non-computable function is the *halting problem,* i.e., the problem whether a program terminates for a given input: there is no program that would decide this for all pairs *program–input* (as proved by Turing in 1936). Examples of non-computable functions (incl. the halting problem) can be constructed by *diagonalization*, i.e., by showing that a program computing the function would not work when applied to itself (i.e., to its own formalization) on input.

Recursively enumerable sets are those which can be algorithmically enumerated, i.e., all of its elements be algorithmically generated one by one so that each element of the set will sooner or later occur on the list. However, the membership in a recursively enumerable set cannot in general be *decided* in a finite time, since for a non-element we will never know during the enumeration whether it is going to be listed in the future or not. On the other hand, the membership in *recursive* sets is algorithmically decidable, since by definition of recursive function, the decision algorithm for its membership always terminates (recall, e.g., that recursive functions are computable by structured programs without **while**-cycles, which always terminate). If both the membership and non-membership lists of a set X can be algorithmically generated, then generating them in parallel gives a decision procedure for X. This fact is formalized by *Post's Theorem:* if both X and the complement of X are recursively enumerable, then X is recursive.

Decidability of theories in logic

The question of algorithmic computability can be applied to sets of formulae (such as the set of axioms of a given theory, the set of theorems provable in a given theory, or the set of formulae true in a given model).

Gödel's Completeness Theorem showed that a 'non-finitistic' problem of validity in a (usually infinite) class of models of a given theory can be reduced to a 'finitistic' problem of provability in the theory (recall that a proof is a finite sequence of finite formulae). However, this reduction does not address the algorithmic feasibility of the task: e.g., if the set of the axioms of the theory is not recursively enumerable (i.e., by Church thesis, algorithmically computable), then we have no algorithmic method how to recognize a valid proof in the theory.

Taking the algorithmic aspects into account, the problem of the computability of the sets of formulae occurring in mathematical logic becomes prominent. Thus in mathematical logic, we are predominantly interested in *recursive* theories, as only those have algorithmically decidable set of axioms.

Let $\mathrm{Thm}(T)$ denote the set of sentences provable in a (first-order classical) theory T and $\mathrm{Rft}(T)$ the set of sentences refuted by T, i.e., $\mathrm{Thm}(T) = \{\varphi \text{ a sentence} \mid T \vdash \varphi\}$ and $\mathrm{Rft}(T) = \{\varphi \text{ a sentence} \mid T \vdash \neg\varphi\}$.

It can be shown that if T is recursively enumerable, then so are $\mathrm{Thm}(T)$ and $\mathrm{Rft}(T)$. Moreover, if T is recursively enumerable, then there is a recursive T' such that $\mathrm{Thm}(T) = \mathrm{Thm}(T')$ (and consequently $\mathrm{Rft}(T) = \mathrm{Rft}(T')$). (Thus we can restrict our interest to *recursively* axiomatized

theories, even though recursively enumerable sets of axioms are computable, too.)

Since a recursive theory T has a recursively enumerable set $\mathrm{Thm}(T)$ of theorems, the theorems of T can be algorithmically generated. However, in order to have an algorithmic decidability of theoremhood, $\mathrm{Thm}(T)$ need be recursive. Therefore we define:

A theory is *decidable* if the set $\mathrm{Thm}(T)$ (so also $\mathrm{Rft}(T)$) is recursive; otherwise it is *undecidable.*

Trivially, every inconsistent theory is decidable (as the set of all formulae is recursive). The decidability of some important theories occurring in mathematical practice has been proved; for instance, the theory RCF of *real closed fields* or *Presburger arithmetic* PrA (i.e., roughly speaking, Robinson's arithmetic without multiplication).

Note, however, that recursive decidability only reflects theoretical algorithmic computability, and neglects such features as the feasibility of computation. Such features are treated in the subject area of *computational complexity,* another branch of modern mathematical logic, which refines the class of decidable theories into a hierarchy of subclasses according to their demands on computational resources (e.g., computation time, memory space, etc.).

Quantifier elimination

A method for showing the decidability of a theory is *quantifier elimination.* A theory has quantifier elimination if every formula φ there is a quantifier-free formula φ' such that $T \vdash \varphi \leftrightarrow \varphi'$. (This is usually proved by recursively eliminating quantifiers from a formula by equivalences provable in the theory.) If a theory has quantifier elimination by a recursively computable method, then the validity of formulae in the theory is algorithmically reduced to the validity of quantifier-free sentences in the theory, which can often be easily shown to be decidable.

The method will be illustrated on the theory of successors. Consider the following three theories:

- $\mathrm{Th}(\langle \omega, 0, S\rangle)$, where $S\colon n \mapsto n+1$ is the successor function on natural numbers; abbreviate the theory by $\mathrm{Th}(S)$.
- SUCC = Q1–Q3 plus the axiom schema S4, for each k:

$$\text{S4:} \qquad s(x_1) = x_2 \wedge s(x_2) = x_3 \wedge \ldots \wedge s(x_{k-1}) = x_k \rightarrow x_1 \neq x_k$$

- SUCC^+ = SUCC plus functions s^n defined as follows: $s^0(x) = x$ and $s^{n+1}(x) = s(s^n(x))$, for each metamathematical number n.

Lemma. *Observe the following facts:*

1. $\mathrm{SUCC}^+ \vdash s^k(x) \neq x$ *for any* $k > 0$.
2. *Terms of* SUCC^+ *contain at most one variable. Moreover,* $\mathrm{SUCC}^+ \vdash s^n(s^m(x)) = s^{m+n}(x)$*; consequently, for any term* t, $\mathrm{SUCC}^+ \vdash t = s^n(x)$ *for some* n *and some variable* x.
3. $\mathrm{SUCC}^+ \vdash s(x) = s(y) \leftrightarrow x = y$. (Hint: $\Rightarrow$ by Q1, $\Leftarrow$ by equality axioms.)
4. $\mathrm{SUCC}^+ \vdash (\exists x)\big(y = s^k(x)\big) \leftrightarrow \big(y \neq 0 \wedge \ldots \wedge y \neq s^{k-1}(0)\big)$. (Hint: demonstrate the provability by induction on k, using Lemma 3 and Q3.)

It can be shown that SUCC^+ extends SUCC conservatively. Obviously the three theories are consistent, since $\langle \omega, 0, S\rangle$ is a model. Furthermore, $\mathrm{Th}(S)$ is complete and extends SUCC. We shall prove by quantifier elimination that actually $\mathrm{Th}(S) = \mathrm{Thm}(\mathrm{SUCC})$ and the three theories are complete and decidable.

Let us call the following formulae *simple:* (i) all formulae of the form $s^n(x) = t$ where the term t does not contain x and (ii) the formula $0 = 0$. Let any simple formula or a negation of a simple formula be called a *simple literal,* and any disjunction of conjunctions of simple literals a *simple DNF* (disjunctive normal form).

Theorem. *For any formula φ in the language of* SUCC^+ *with* $\mathrm{Free}(\varphi) \subseteq \{x_1, \ldots, x_n\}$ *there is an algorithmically constructible formula φ' in the same language with* $\mathrm{Free}(\varphi') \subseteq \{x_1, \ldots, x_n\}$ *such that φ' is a simple DNF and* $\mathrm{SUCC}^+ \vdash \varphi \leftrightarrow \varphi'$.

Proof. By induction on φ:

- If φ is *atomic,* then it is a formula $s = t$ for some terms s, t.
  - (a) If neither t nor s contains a variable (i.e., are closed), then φ is $s^n(0) = s^m(0)$ for some m, n.
    - (a1) If $n = m$ then $\mathrm{SUCC}^+ \vdash s^n(0) = s^m(0)$ (proof: exercise!), so $\mathrm{SUCC}^+ \vdash \varphi \leftrightarrow 0 = 0$, a simple formula.
    - (a2) If $n \neq m$ then $\mathrm{SUCC}^+ \vdash s^n(0) \neq s^m(0)$ (exercise!), so $\mathrm{SUCC}^+ \vdash \varphi \leftrightarrow 0 \neq 0$, a simple literal.
  - (b) If t, s contain different variables or one of t, s is closed, then φ is already simple.
  - (c) If t, s contain both the same variable (say, x), then φ is $s^n(x) = s^m(x)$. Apply Lemma 3 above $\min(n, m)$ times:
    - (c1) If $n = m$, obtain $\mathrm{SUCC}^+ \vdash \varphi \leftrightarrow x = x$, thus $\mathrm{SUCC}^+ \vdash \varphi \leftrightarrow 0 = 0$ (exercise: why?), a simple formula.
    - (c2) If $n \neq m$, obtain $\mathrm{SUCC}^+ \vdash \varphi \leftrightarrow s^k(x) = x$ for some $k > 0$; thus by Lemma 1, $\mathrm{SUCC}^+ \vdash \varphi \leftrightarrow 0 \neq 0$, a simple literal.
- If φ is $\neg\psi$ or $\psi \to \chi$, where ψ, χ are already simple DNFs, then apply (De Morgan and double negation) propositional laws on $\neg\psi$ or $\neg\psi \vee \chi$ to obtain φ in simple DNF.
- If φ is $(\exists x)\psi$, where ψ is already a simple DNF:
  1. Apply $\vdash (\exists x)(\psi_1 \vee \psi_2) \leftrightarrow (\exists x)\psi_1 \vee (\exists x)\psi_2$: thus we can treat each disjunct separately and assume without loss of generality that ψ is just a conjunction of simple literals.
  2. Rearrange the conjuncts in ψ so that ψ is $\psi_1 \wedge \psi_2$, where:
     - ψ_1 is conjunction of all conjuncts of ψ containing x
     - ψ_2 is conjunction of all conjuncts of ψ *not* containing x

     Then apply $\vdash (\exists x)(\psi_1 \wedge \psi_2) \leftrightarrow ((\exists x)\psi_1) \wedge \psi_2$, a theorem of first-order classical logic if ψ_2 does not contain free x. Thus without loss of generality we can assume that all conjuncts of ψ *do* contain x, i.e., that φ is:

     $$(\exists x)\big(s^{k_1}(x) = t_1 \wedge \ldots \wedge s^{k_n}(x) = t_n \wedge s^{k'_1}(x) \neq t'_1 \wedge \ldots \wedge s^{k'_{n'}}(x) \neq t'_{n'}\big),$$

     where t_i, t'_i do not contain x. (Note that conjuncts $0 = 0$ are irrelevant and conjuncts $0 \neq 0$ trivialize φ, so this is the only non-trivial form of φ). If $n = n' = 0$, we are done; thus assume $n > 0$ or $n' > 0$.

3. Let $k = \max(k_1, \ldots, k_n, k'_1, \ldots, k'_{n'})$. Apply Lemma 3 sufficiently many times to obtain:

   $$\mathrm{SUCC}^+ \vdash \varphi \leftrightarrow (\exists x)\big(s^k(x) = \hat{t}_1 \wedge \ldots \wedge s^k(x) = \hat{t}_n \wedge s^k(x) \neq \hat{t}'_1 \wedge \ldots \wedge s^k(x) \neq \hat{t}'_{n'}\big)$$

   Thus by $\vdash (\exists x)\big(\varphi(t(x)\big) \leftrightarrow (\exists y)\big(y = t(x) \wedge \varphi(y)\big)$ for any term t,

   $$\mathrm{SUCC}^+ \vdash \varphi \leftrightarrow (\exists y)\big(y = s^k(x) \wedge y = \hat{t}_1 \wedge \ldots \wedge y = \hat{t}_n \wedge y \neq \hat{t}'_1 \wedge \ldots \wedge y \neq \hat{t}'_{n'}\big)$$

   whence by Lemma 4,

   $$\mathrm{SUCC}^+ \vdash \varphi \leftrightarrow (\exists y)\big(y \neq 0 \wedge \ldots \wedge y \neq s^{k-1}(0) \wedge y = \hat{t}_1 \wedge \ldots \wedge y = \hat{t}_n \wedge y \neq \hat{t}'_1 \wedge \ldots \wedge y \neq \hat{t}'_{n'}\big)$$

   (a) If $n = 0$, then $\mathrm{SUCC}^+ \vdash \varphi \leftrightarrow 0 = 0$, since $0, s(0), \ldots, s^{k+n'}(0)$ are $k + n' + 1$ terms provably non-equal in SUCC^+ (so we can instantiate y among them to prove φ)

   (b) If $n > 0$, then (by $\vdash (\exists y)(y = \hat{t}_1 \wedge \chi(y)) \leftrightarrow \chi(\hat{t}_1)$; notice that it is here where we eliminate the quantifier):

   $$\mathrm{SUCC}^+ \vdash \varphi \leftrightarrow \big(\hat{t}_1 \neq 0 \wedge \ldots \wedge \hat{t}_1 \neq s^{k-1}(0) \wedge \hat{t}_1 = \hat{t}_1 \wedge \ldots \wedge \hat{t}_1 = \hat{t}_n \wedge \hat{t}_1 \neq \hat{t}'_1 \wedge \ldots \wedge \hat{t}_1 \neq \hat{t}'_{n'}\big)$$

   (b1) The conjuncts which are closed or contain the same variable on both sides of the equality predicate will be replaced by $0 = 0$ or $0 \neq 0$ as in the step for atomic formulae.

   (b2) The other conjuncts are already simple literals.

Observe that the construction of φ' was algorithmic and that no new variables have been introduced in the induction, therefore $\mathrm{Free}(\varphi') \subseteq \{x_1, \ldots, x_n\}$ as required. □

Corollary. *If φ is closed, then φ' is closed as well.*

Corollary. SUCC^+ *(so* SUCC *as well) is complete and decidable.*

Proof. For every *closed* φ we have algorithmically found a *closed* SUCC^+-equivalent φ' in simple DNF. Since the only closed simple literals are $0 = 0$ and $0 \neq 0$, we can algorithmically decide the provability of φ' by evaluating the DNF as in classical propositional logic. □

Corollary. $\mathrm{Thm}(\mathrm{SUCC}) = \mathrm{Th}(S)$*, as* $\mathrm{Th}(S)$ *is a complete extension of* SUCC*, and* SUCC *is already complete. Thus* $\mathrm{Th}(S)$ *is decidable, too.*