Lesson 12: Gödel’s incompleteness theorem, part. I

Theme: Robinson arithmetic and its arithmetization.

In this lesson and the next, we are only dealing with logic over vocabulary \{\tilde{0}, \text{Succ}, +, \cdot\}, where \tilde{0} is a constant symbol intended to represent the number zero; \text{Succ} is a unary function intended to represent +1, i.e., \text{Succ}(x) = x + 1; and finally, + and \cdot are intended to represent the standard addition and multiplication operands.

1 Robinson arithmetic

Robinson’s arithmetic is a theory \(Q\) derived from the following axioms.

(Q1) \(\forall x \ (\text{Succ}(x) \neq 0)\).

(Q2) \(\forall x \forall y \ (\text{Succ}(x) \approx \text{Succ}(y) \rightarrow x \approx y)\).

(Q3) \(\forall x \ (x \neq \tilde{0} \rightarrow \exists y \ x \approx \text{Succ}(y))\).

(Q4) \(\forall x \ (x + \tilde{0} \approx x)\).

(Q5) \(\forall x \forall y \ (x + \text{Succ}(y) \approx \text{Succ}(x + y))\).

(Q6) \(\forall x \ (x \cdot \tilde{0} \approx \tilde{0})\).

(Q7) \(\forall x \forall y \ (x \cdot \text{Succ}(y) \approx (x \cdot y) + x)\).

Note that by its definition, \(Q\) is a finitely axiomatizable theory, and that \(Q\) is a proper subtheory of \(\text{Th}(\mathcal{N})\), where \(\mathcal{N}\) is the standard structure \(\mathcal{N} = (\mathbb{N}, 0, \text{Succ}, +, \cdot)\). What we call number theory usually refers to \(\text{Th}(\mathcal{N})\). Note that \(\text{Th}(\mathcal{N})\) is much stronger than \(Q\). For example, \(\forall x \ x \neq \text{Succ}(x)\) is not provable in \(Q\).

In the following, we will often write \(x \leq y\) as an abbreviation for \(\exists z \ x + z \approx y\), and \(x < y\) for \(x \leq y \land x \neq y\).

Remark 12.1 For the rest of this lesson and the next, the proof system will always be in a theory \(T \supseteq Q\), with the sentences (Q1)–(Q7) above being included as axioms of \(T\).

2 Arithmetization

We denote the set \(\text{Symb} = \{\neg, \land, \lor, (\ ), \approx, \tilde{0}, \text{Succ}, +, \cdot, x_0, x_1, x_2, \ldots\}\) In principle, we can assume that every formula is a string with symbols from \(\text{Symb}\), and every proof is a sequence of formulas with a comma in between two formulas.

In this section we are going to see how to encode a formula \(\varphi\) as a number, and hence, a proof as a number too. For this purpose, we assign each symbol \(s \in \text{Symb} \cup \{\}\) a number \#s as follows.

\[
\begin{array}{cccccccccccc}
\text{s} & \neg & \land & \lor & ( ) & \approx & \tilde{0} & \text{Succ} & + & \cdot & x_0 & x_1 & x_2 & \ldots \\
\#s & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & \ldots
\end{array}
\]

Let \(\{p_0, p_1, \ldots\}\) be the set of all prime numbers with \(p_0 < p_1 < \cdots\).
For a string $str = s_0 \cdots s_n$ with each symbol $s_i$ coming from $\text{Symb} \cup \{\|\}$, the \textit{Gödel number} of $str$, denoted by $\sharp str$ is the number:

$$\sharp str := p_{\sharp s_0}^0 p_{\sharp s_1}^1 \cdots p_{\sharp s_n}^n$$

The Gödel numbers of a formula $\varphi$ and a proof $\xi$ are defined as $\sharp \varphi$ and $\sharp \xi$, respectively, where $\varphi$ and $\xi$ are viewed as a string of symbols coming from $\text{Symb} \cup \{\|\}$.

\textbf{Remark 12.2}

- We can write a computer program \textsc{IsFormula} for the following task.
  - \textbf{Input:} A positive number $N$.
  - \textbf{Output:} Output \textsc{True}, if $N$ “represents” a formula, i.e., $N$ is the Gödel number of a formula. Otherwise, output \textsc{False}.

Likewise, we can write a program \textsc{IsSentence} that checks whether an input number $N$ represents a sentence.

- We can write a computer program \textsc{IsProof}_Q for the following task.
  - \textbf{Input:} A positive number $N$.
  - \textbf{Output:} Output \textsc{True}, if $N$ represents a proof in $Q$. Otherwise, output \textsc{False}.

- We can write a computer program \textsc{IsProofOf}_Q for the following task.
  - \textbf{Input:} Two positive numbers $N$ and $M$.
  - \textbf{Output:} Output \textsc{True}, if $N$ represents a proof, $M$ represents a formula, and $N$ is a proof of $M$ in $Q$. Otherwise, output \textsc{False}.

\textbf{Definition 12.3} Let $T$ be a theory such that $T = \text{Cn}(\Sigma)$. We say that $T$ is \textit{recursively axiomatizable}, if there is a computer program \textsc{IsAxiom}_T for the following task.

- \textbf{Input:} A positive number $N$.
- \textbf{Output:} Output \textsc{True}, if $N$ represents an axiom in $T$, i.e., $N$ represents a sentence $\Sigma$. Otherwise, output \textsc{False}.

\textbf{Remark 12.4}

- We can write a computer program \textsc{IsProofOf}_T for the following task.
  - \textbf{Input:} Two positive numbers $N$ and $M$.
  - \textbf{Output:} Output \textsc{True}, if $N$ represents a proof in $T$, $M$ represents a formula, and $N$ is a proof of $M$ in $T$. Otherwise, output \textsc{False}.
3 A sketch proof of the incompleteness theorem

Gödel’s incompleteness theorem states that for every consistent and recursively axiomatizable theory $T \supseteq Q$, there is a sentence $\Psi$ such that neither $\Psi$ nor $\neg \Psi$ are provable in $T$.

For an integer $N \geq 0$, let $\bar{N}$ denote the following term:

$$
\bar{N} := \text{Succ} \cdots \text{Succ}(\bar{0}) \text{ N times}
$$

Now, suppose that instead of being a computer program, the boolean function $\text{IsProofOf}_T(y, x)$ is a first-order formula that indicates $y$ is a proof of $x$ in $T$. So, for every sentence $\varphi$,

$$
T \vdash \varphi \iff T \vdash \exists y \text{IsProofOf}_T(y, \#\varphi). \quad (1)
$$

Consider a sentence $\Psi$ such that

$$
T \vdash \Psi \iff \left( \forall y \neg \text{IsProofOf}_T(y, \#\Psi) \right) \quad (2)
$$

which is an abbreviation for:

$$
T \vdash \Psi \rightarrow \left( \forall y \neg \text{IsProofOf}_T(y, \#\Psi) \right) \quad (3)
$$

$$
T \vdash \left( \forall y \neg \text{IsProofOf}_T(y, \#\Psi) \right) \rightarrow \Psi \quad (4)
$$

From Equation (4), we can derive

$$
T \vdash \neg \Psi \rightarrow \neg \left( \forall y \neg \text{IsProofOf}_T(y, \#\Psi) \right) \quad (5)
$$

We now argue that neither $T \vdash \Psi$ nor $T \vdash \neg \Psi$.

- Suppose $T \vdash \Psi$.
  Applying modus ponens on $T \vdash \Psi$ and Equation (3), we have

$$
T \vdash \forall y \neg \text{IsProofOf}_T(y, \#\Psi)
$$

which by Equation (1), means $\Psi$ is not provable in $T$, contradicting supposition $T \vdash \Psi$.

- Suppose $T \vdash \neg \Psi$.
  Applying modus ponens on $T \vdash \neg \Psi$ and Equation (5),

$$
T \vdash \neg \forall y \neg \text{IsProofOf}_T(y, \#\Psi)
$$

which is equivalent to

$$
T \vdash \exists y \text{IsProofOf}_T(y, \#\Psi).
$$

By Equation (1), it means $T \vdash \Psi$, contradicting the consistency of $T$.

Therefore, neither $\Psi$ nor $\neg \Psi$ are provable in $T$, hence the incompleteness of $T$.

In this lesson and the next, we focus on the following two tasks in order to complete our proof above.

(a) Find the formula for $\text{IsProofOf}_T(y, x)$ using the vocabulary $\{\bar{0}, \text{Succ}, +, \cdot\}$.

(b) Find the statement $\Psi$.

---

*Recall that in Lesson 4 we show if $X \vdash \alpha \rightarrow \beta$, then $X \vdash \neg \beta \rightarrow \neg \alpha$, which is called contrapositive.

†Recall that in Lesson 4 we show if $X \vdash \alpha \rightarrow \beta$ and $X \vdash \alpha$, then $X \vdash \beta$, which is called modus ponens.
Appendix

We will formalize the notion of *recursive functions*, which are equivalent to the notion of computable functions. Recall that $\mathbb{N} = \{0, 1, 2, \ldots \}$. Let $F_n$ be the set of all functions from $\mathbb{N}^n$ to $\mathbb{N}$, and let $F := \bigcup_{n \geq 1} F_n$.

$\mu$-recursive functions, or shortly, recursive functions, are functions that are built inductively as follows.

- **Base case:** All three kinds of functions below are recursive.

  - **Constant function:** $f(v_1, \ldots, v_n) = 0$.
  - **Successor function (on the $i$-component):** $f(v_1, \ldots, v_n) = \text{Succ}(v_i)$.
  - **Projection function (to the $i$-component):** $f(v_1, \ldots, v_n) = v_i$.

- **Induction step:** All the functions built up from recursive functions using one of the rules below are recursive functions.

  - **Composition (Oc).** If $h \in F_m$ and $g_1, \ldots, g_m \in F_n$ are recursive, then the following function $f$ is also recursive. For every $\bar{a} = (a_1, \ldots, a_n) \in \mathbb{N}^n$,

    \[
    f(\bar{a}) := h(g_1(\bar{a}), \ldots, g_m(\bar{a})).
    \]

  We usually write $h[g_1, \ldots, g_m]$ to denote the function $f$ constructed above.

  - **Primitive recursion (Op).** If $g \in F_n$ and $h \in F_{n+2}$ are recursive functions, then so is $f \in F_{n+1}$, defined as follows. For every $\bar{a} = (a_1, \ldots, a_n) \in \mathbb{N}^n$,

    \[
    f(\bar{a}, 0) := g(\bar{a})
    \]

    \[
    f(\bar{a}, \text{Succ}(b)) := h(\bar{a}, b, f(\bar{a}, b))
    \]

  - **$\mu$ operation (O$\mu$).** Let $g \in F_{n+1}$ be such that for every $\bar{a} \in \mathbb{N}^n$, there is $b \in \mathbb{N}$, where $g(\bar{a}, b) = 0$. If $g$ is computable, then so is the following function $f$. For every $\bar{a} = (a_1, \ldots, a_n) \in \mathbb{N}^n$,

    \[
    f(\bar{a}) := \text{the smallest } b \text{ such that } g(\bar{a}, b) = 0
    \]

  We write $f(\bar{a}) := \mu b[g(\bar{a}, b) = 0]$ to denote the function $f$ as constructed above.

A recursive function obtained without using the O$\mu$ rule is called a *primitive recursive* function.

**Example 12.5**

- The function $f_{\text{add}}(a, b) = a + b$ is recursive by an application of Op rule.

  \[
  f_{\text{add}}(a, 0) := a \quad \text{and} \quad f_{\text{add}}(a, \text{Succ}(b)) := \text{Succ}(f_{\text{add}}(a, b))
  \]

- The functions $f_{\text{mul}}(a, b) = a \cdot b$ and $f_{\text{exp}}(a, b) = a^b$ are recursive.

  \[
  f_{\text{mul}}(a, 0) := 0 \quad \text{and} \quad f_{\text{mul}}(a, \text{Succ}(b)) := f_{\text{add}}(b, f_{\text{mul}}(a, b))
  \]

  \[
  f_{\text{exp}}(a, 0) := \text{Succ}(0) \quad \text{and} \quad f_{\text{exp}}(a, \text{Succ}(b)) := f_{\text{mul}}(a, f_{\text{exp}}(a, b))
  \]

- The function $f_{\text{abs}}(a, b) := |a - b|$ is recursive.

- The function $f_{\text{div}}(a, b) := 0$, if $b$ divides $a$, and 1, otherwise, is recursive.
• The function $f_{\text{prime}}(n) := p_n$, where $p_n$ is the $n^{th}$ prime number, is recursive.

**Theorem 12.6 (Church-Turing thesis)** If a function $f$ is computable (by a “computer program”), then it is also (i) computable in $\lambda$-calculus; (ii) computable by a Turing machine; (iii) $\mu$-recursive.

In fact, the notions of $\lambda$-calculus, Turing machines, and $\mu$-recursive are all equivalent. That is, a function is computable in $\lambda$-calculus if and only if it is computable by a Turing machine if and only if it is $\mu$-recursive.

In his original paper, Gödel showed the following.

• An explicit construction of the primitive recursive function for $\text{IsProofOf}(x, y)$ as specified in Remark 12.4.
• For every primitive recursive function $f : \mathbb{N}^n \to \mathbb{N}$, there is a formula $\alpha(x_1, \ldots, x_n, y)$ over vocabulary $\{\text{Succ}, +, \cdot, \tilde{0}\}$ such that
  \[ f(a_1, \ldots, a_n) = b \quad \text{if and only if} \quad T \vdash \alpha(a_1, \ldots, a_n, b) \]  

An explicit formula for $\text{IsProofOf}$ is conceptually not difficult, but long and tedious. In this class, having convinced ourselves that we can write a computer program for $\text{IsProofOf}(x, y)$, we can invoke Church-Turing thesis to arrive at the conclusion that $\text{IsProofOf}(x, y)$ is recursive. On the other hand, converting a recursive function $f$ to a formula $\alpha$ as specified in Equation (6) involves a very nice piece of mathematics, and this will be our focus in our next lesson.

---


§To be exact, expressing $\text{Oc}$ and $\text{O}_\mu$ rules in formulas over $\{\tilde{0}, \text{Succ}, +, \cdot\}$ is not difficult. The main difficulty is in expressing the $\text{Op}$. 

---