Лекция 6: Теорема за еквивалентност. Теорема за универсалната функция

Основни теореми в Теория на изчислимостта

3.1 Теорема за еквивалентност

В този раздел ще покажем, че двата подхода към изчислимостта, които въведохме дотук — подходът на Клини с частично рекурсивните функции и подходът с изчислителния модел, базиран на МНР, са еквивалентни, т.е. определят един и същ клас от функции. Твърдения от този тип обикновено се наричат *теорема за еквивалентност*.

3.1.1 От частична рекурсивност към изчислимост

Първо ще се заемем с по-лесната посока на теоремата за еквивалентност, а именно:

Твърдение 3.1. Всяка частично рекурсивна функция е изчислима.

Доказателство. Нека f е произволна частично рекурсивна функция. Ще разсъждаваме с индукция по дефиницията на f, за да покажем, че за нея съществува програма за МНР, която я пресмята.

Ако f е базисна функция, нещата са ясни:

– ако f е функцията $\mathcal{S}(x) = x + 1$, то f се пресмята от програмата $P\colon\ S(1)$;

– ако f е функцията $\mathcal{O}(x) = 0$, то f се пресмята от програмата $P\colon\ Z(1)$;

– ако f е проектиращата функция I^n_k (където $I^n_k(x_1, \dots, x_n) \stackrel{\text{деф}}{=} x_k$), то f се пресмята от програмата P: $T(1, k)$.

Нека сега

$$f = g(h_1, \dots, h_k),$$

като за g и $h_1, \dots, h_k$, съгласно индукционната хипотеза, съществуват програми P и $Q_1, \dots, Q_k$, които ги пресмятат. С тяхна помощ ще построим нова програма R, която пресмята f.

Тук ще направим една уговорка, чийто смисъл ще стане ясен след малко. Ще предполагаме, че горните програми $P, Q_1, \dots, Q_k$ са от т. нар. "стандартен тип". По определение една програма $I_0, \dots, I_k$ е *стандартна*, ако за всеки адрес q от оператор за преход $J(m, n, q)$ е вярно, че ако $q > k$, то $q = k + 1$. Ясно е, че по всяка програма за МНР алгоритмично можем да получим еквивалентна на нея стандартна програма.

Нека горната функция f е на n аргумента. Тогава за всяко $\bar{x} \in \mathbb{N}^n$:

$$\begin{aligned} f(\bar{x}) \simeq y \stackrel{\text{деф}}{=} \exists z_1 \dots \exists z_k \ (h_1(\bar{x}) \simeq z_1 \ \& \ \dots h_k(\bar{x}) \simeq z_k \ \& \\ h_k(\bar{x}) \simeq z_k \ \& \ g(z_1, \dots, z_k) \simeq y) \end{aligned}$$

Ясно е, че програмата R първо трябва да извика $Q_1, \dots, Q_k$ върху $\bar{x}$, за да се пресметнат (ако съществуват) стойностите $z_1, \dots, z_k$, и после да извика P с вход $(z_1, \dots, z_k)$.

В общи линии, алгоритъмът за f е този (трудно би могъл да бъде друг $\ddot\smile$), но тук възникват няколко технически въпроса. Единият е, че трябва да се погрижим да запазим входните стойности $x_1, \dots, x_n$, защото те ще бъдат загубени още при извикването на Q_1. Освен това, получените междинни резултати $z_1, \dots, z_k$ също трябва да бъдат съхранени в достатъчно далечни регистри, за да не бъдат загубени, докато дойде време да бъдат включени в изчислението. Тук под "далечен регистър" имаме предвид регистър, който няма да бъде засегнат при работата на програмите $Q_1, \dots, Q_k$. Лесно е вижда, че един такъв регистър е X_{m+1} за

$$m \ = \ max\{n, m_1, \dots, m_k\},$$

където m_i е най-големият индекс на регистър, който се среща в програмата Q_i, $1 \leq i \leq k$.

Тогава програмата R трябва да започва с инструкциите:

$$X_{m+1} := X_1, \ \dots \ , X_{m+n} := X_n.$$

Искаме веднага след тях да сложим инструкциите на първата програма Q_1. Само че те не могат да останат същите, защото *преди* тях вече сме сложили горните n на брой трансфери. Значи сега всеки адрес q на оператор за преход $J(m,n,q)$ трябва да бъде увеличен с n. Освен това, след като Q_1 приключи работата си върху $\bar{x}$, трябва да се погрижим да прехвърлим резултата от първия регистър в някой по-далечен — да кажем X_{m+n+1}.

След това преминаваме към пресмятане на $h_2(\bar{x})$ от Q_2. За целта трябва да възстановим входните стойности $(x_1, \ldots, x_n)$ в първите n регистъра, а останалите регистри — от X_{n+1} до X_m да инициализираме с 0. Така след инструкциите на Q_1 ще имаме допълнително следните $m+1$ инструкции:

$$X_{m+n+1} := X_1, X_1 := X_{m+1}, \ldots, X_n := X_{m+n}, X_{n+1} := 0, \ldots, X_m := 0,$$

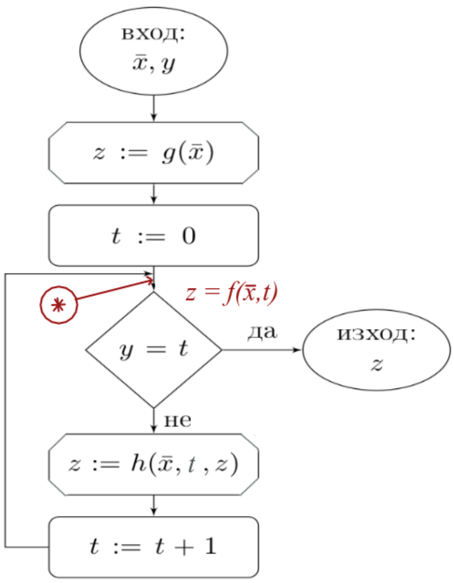
Следователно адресите в операторите за преход на Q_2 трябва да бъдат увеличени с общо $n + (m+1) + k_1$, където k_1 е броят на инструкциите на Q_1. По същия начин процедираме и със следващите програми $Q_3, \ldots, Q_k$ и P, като за P, разбира се, инициализираме първите k регистъра $X_1, \ldots, X_k$ със стойностите на последните регистри $X_{m+n+1}, \ldots, X_{m+n+k}$. Ясно е, че новата програма R ще пресмята f.

Нека сега f се получава с примитивна рекурсия от g и h, като за тези две функции по индуктивната хипотеза има програми, които ги пресмятат.

Нека по-конкретно за f е изпълнено:

$$\left\| \begin{array}{l} f(\bar{x},0) \simeq g(\bar{x}) \\ f(\bar{x},y+1) \simeq h(\bar{x},y,f(\bar{x},y)). \end{array} \right. \tag{3.1}$$

Да съобразим, че f се пресмята от следния алгоритъм:



За целта да видим, че когато, образно казано, изчислението "преминава" през контролната точка $(*)$, е изпълнено равенството

$$z = f(\bar{x}, t)$$

за текущите стойности на регистрите z и t (тези стойности също ще означаваме със z и t).

Това ще направим с индукция по броя на преминаванията през тази контролна точка (което в случая означава индукция по t). Когато изчислението за първи път премине през нея, ще имаме

$$z = g(\bar{x}) \stackrel{(3.1)}{=} f(\bar{x}, 0) = f(\bar{x}, t).$$

Да приемем, че при някакво преминаване през $(*)$ за текущите стойности на z и t е било вярно, че $z = f(\bar{x}, t)$. Тогава при следващото преминаване през тази контролна точка за новите стойности z_{new} и t_{new} ще имаме $z_{new} = h(\bar{x}, t, z)$ и $t_{new} = t + 1$, откъдето

$$z_{new} = h(\bar{x}, t, z) \stackrel{\text{и.х.}}{=} h(\bar{x}, t, f(\bar{x}, t)) \stackrel{(3.1)}{=} f(\bar{x}, t+1) = f(\bar{x}, t_{new}).$$

С това индукцията е приключена. Разбира се, при излизане от цикъла условието $z = f(\bar{x}, t)$ ще продължи да е в сила. Но тогава вече $t = y$, и значи накрая $z = f(\bar{x}, y)$.

Получихме, че ако горната програма завърши при вход $(\bar{x}, y)$, то резултатът ще е $f(\bar{x}, y)$. Ако пък за някой вход $(\bar{x}, y)$ тя не върне резултат, лесно се съобразява, че в такъв случай f няма как да е дефинирана в $(\bar{x}, y)$.

За да обобщим, да означим с F функцията, която се пресмята от горната програма. Тогава това, което показахме по-горе, може да се запише така:

$$F(\bar{x}, y) \simeq z \implies f(\bar{x}, y) \simeq z,$$

за произволни $\bar{x}, y$ и z. Но това по дефиниция означава, че $F \subseteq f$.

Освен това се съгласихме, че

$$(\bar{x}, y) \notin Dom(F) \implies (\bar{x}, y) \notin Dom(f),$$

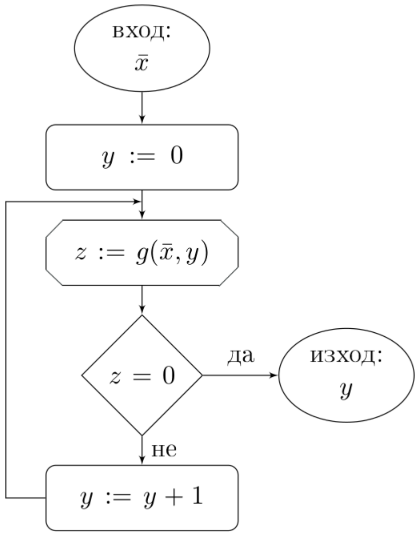
или все едно, $Dom(f) \subseteq Dom(F)$.

Сега вече можем да приложим *Задача* 1.1, която казва, че в такъв случай двете функции f и F са равни. Това означава, че горната блок схема всъщност пресмята функцията f. Това, което остава, е да препишем тази блок схема във вид на програма за МНР ☺.

Остана да разгледаме и последния случай, когато f е получена с минимизация от някоя функция g:

$$f(x_1, \ldots, x_n) \simeq \mu y[g(x_1, \ldots, x_n, y) \simeq 0].$$

Съгласно индуктивната хипотеза, за g съществува програма за МНР, която я пресмята. Да съобразим, че f ще се пресметне от алгоритъм, който записан на блок схемен език изглежда така:



За целта трябва да вземем под внимание една особеност на μ-операцията, която обсъдихме след нейната дефиниция 1.8. Става въпрос за това, че ако $f(\bar{x}) \simeq y$, то y не само е първото естествено число, за което $g(\bar{x}, y) \simeq 0$, но за това y е вярно още, че за всички z, по-малки от него, $g(\bar{x}, z)$ *има стойност* (която, разбира се, трябва да е ненулева). Лесно се вижда, че ако има такова y, то ще бъде намерено от горната блок схема и обратно, ако тя върне резултат при вход $(\bar{x})$, той ще е същият като $f(\bar{x})$. $\square$

3.1.2 От изчислимост към частична рекурсивност

За да покажем, че всяка изчислима функция е частично рекурсивна, ще трябва предварително да свършим известно количество техническа работа. За начало да си припомним най-важните понятия и означения, свързани с работата на една машина с неограничени регистри. *Конфигурация* за машината е безкрайна редица от естествени числа

$$(l, (x_1, x_2, \dots)),$$

която ще съкращаваме до $(l, \widetilde{x})$ и ще отъждествяваме с редицата $(l, x_1, x_2, \dots)$. В нея l е адресът на инструкцията, която предстои да се изпълни, а $(x_1, x_2, \dots)$ е текущото състояние на паметта в момента на изпълнение на тази изнструкция. *Начална конфигурация* (за входа $(x_1, \ldots, x_n)$) е редицата $(0, x_1, \ldots x_n, 0, 0, \ldots)$.

По дефиниция n-местната функция f се пресмята от програмата $P\colon I_0, \ldots, I_k$, ако за всички естествени $x_1, \ldots, x_n, y$ е изпълнено:

$$f(x_1, \ldots, x_n) \simeq y \iff P(x_1, \ldots, x_n, 0, 0, \ldots) \downarrow y.$$

Тук записът $P(\widetilde{x}) \downarrow y$, който въведохме в раздел 2.1.2, означава, че P спира върху паметта $\widetilde{x}$ с резултат y. Този резултат се получава след итериране на функцията *step* (едностъпковото преобразование за програмата P) дотогава, докато се достигне до финално състояние.

Стойността на $step(l, \widetilde{x})$ дефинирахме в зависимост от вида на l-тия оператор на програмата P. За нашите цели се оказва удобно да препишем *step* като използваме една спомагателна функция, която не зависи от конкретната програма. Тази функция по дадени оператор I и конфигурация $(l, \widetilde{x})$ ще връща *следващата* конфигурация, към която се преминава, когато се приложи оператора I към конфигурацията $(l, \widetilde{x})$. Да наречем тази функция *next*.

Формалната дефиниция на *next* е с разглеждане на четирите възможности за оператора I:

$$\begin{aligned}
next(S(n), (l, \widetilde{x})) &= (l{+}1, x_1, \ldots, x_{n-1}, x_n{+}1, x_{n+1}, \ldots)\\
next(Z(n), (l, \widetilde{x})) &= (l{+}1, x_1, \ldots, x_{n-1}, 0, x_{n+1}, \ldots)\\
next(T(m,n), (l, \widetilde{x})) &= (l{+}1, x_1, \ldots, x_{m-1}, x_n, x_{m+1}, \ldots)\\
next(J(m,n,q), (l, \widetilde{x})) &= \begin{cases} (q, \widetilde{x}), & \text{ако } x_m = x_n\\ (l{+}1, \widetilde{x}), & \text{ако } x_m \neq x_n. \end{cases}
\end{aligned}$$

Сега функцията *step* за програмата $P:\ I_0, \ldots, I_k$, изразена чрез *next*, изглежда така:

$$step(l, \widetilde{x}) = \begin{cases} next(I_0, (l, \widetilde{x})), & \text{ако } l = 0\\ \ldots\ldots\ldots\ldots & \\ next(I_k, (l, \widetilde{x})), & \text{ако } l = k\\ (l, \widetilde{x}), & \text{ако } l > k. \end{cases} \tag{3.2}$$

Идеята ни е да покажем, че *step* е примитивно рекурсивна. Тя, обаче, е изображение от вида

$$step\colon\ \mathbb{N}^{\mathbb{N}} \longrightarrow \mathbb{N}^{\mathbb{N}}$$

и въобще не е числова функция! За да стане такава, е ясно, че трябва да преминем от конфигурации към някакви техни *кодове*. Проблемът е, че конфигурациите са *безкрайни* редици от естествени числа, които очевидно няма как да бъдат "запомнени" с едно естествено число.

За щастие, конфигурациите, които участват в едно реално изчисление, са от по-специален вид. Нашите програми тръгват от начални конфигурации $(0, x_1, \dots, x_n, 0, 0, \dots)$, в които само първите n регистъра могат да са различни от 0. Освен това, тъй като всяка програма е *краен* текст, в хода на изчисленията тя може да променя само *краен брой* от регистрите — само тези, които участват в нея. Следователно във всеки момент от едно изчисление само краен брой регистри ще имат съдържание, различно от 0.

Конфигурации $(l, x_1, \dots, x_n, 0, 0, \dots)$, в които само за краен брой i е вярно, че $x_i \neq 0$, ще наричаме *финитни*. Такива конфигурации вече можем да кодираме с естествени числа.

Код на финитната конфигурация $(l, \widetilde{x}) = (l, x_1, x_2, \dots)$ ще наричаме числото

$$\delta(l, \widetilde{x}) \stackrel{\text{деф}}{=} 2^l . p_1^{x_1} . p_2^{x_2} . \dots .$$

Ясно е, че всяко естествено число $z > 0$ е код на единствена конфигурация $(l, x_1, x_2, \dots)$ — тази, за която

$$l = (z)_0, \quad x_1 = (z)_1, \quad x_2 = (z)_2, \dots .$$

Когато казваме конфигурация, оттук нататък ще имаме предвид финитна конфигурация.

Да означим със $Step$ функцията, която действа върху кодовете на конфигурациите така, както $step$ действа върху самите конфигурации, т.е.

$$Step(\delta(l, \widetilde{x})) = \delta(step(l, \widetilde{x})),$$

или все едно

$$Step(z) = \delta(step((z)_0, (z)_1, (z)_2, \dots)).$$

Разбира се, горното равенство е в сила при $z > 0$. За да бъде тотална функцията $Step$, можем да положим $Step(0) \stackrel{\text{деф}}{=} 0$.

Да дефинираме с подобна идея и функция $Next$ чрез $next$. Тя ще преработва кодовете на конфигурациите точно както $next$ преработва самите конфигурации. По-точно, при фиксиран оператор I полагаме

$$Next(I, z) = \begin{cases} \delta(next(I, ((z)_0, (z)_1, (z)_2, \dots))), & \text{ако } z > 0 \\ 0, & \text{ако } z = 0. \end{cases} \tag{3.3}$$

Да разпишем как изглежда $Next(I, z)$ във всеки от четирите случая за оператора I. Навсякъде по-долу ще предполагаме, че $z > 0$ (т.е. z е код на някаква конфигурация).

Имаме, че $next(S(n),(l,\widetilde{x})) = (l+1, x_1, \dots, x_{n-1}, x_n+1, x_{n+1}, \dots)$, следователно

$$Next(S(n),z) = \delta((z)_0{+}1, (z)_1, ..., \underbrace{(z)_n+1}_{(n)}, ...) = 2^{(z)_0+1}.p_1^{(z)_1} ... \, p_n^{(z)_n+1} ... = 2zp_n.$$

Аналогично, от $next(Z(n),(l,\widetilde{x})) = (l+1, x_1, \dots, x_{n-1}, 0, x_{n+1}, \dots)$ получаваме

$$Next(Z(n),z) = \delta((z)_0{+}1, (z)_1, ..., \underbrace{0}_{(n)}, ...) = 2^{(z)_0+1}.p_1^{(z)_1} ... \, p_n^0 \, ... = \frac{2z}{p_n^{(z)_n}}.$$

За оператора $T(m,n)$ имаме по определение

$next(T(m,n),(l,\widetilde{x})) = (l+1, x_1, \dots, x_{m-1}, x_n, x_{m+1}, \dots)$, и значи

$$Next(T(m,n),z) = \delta((z)_0{+}1, (z)_1, ..., \underbrace{(z)_n}_{(m)}, ...) = 2^{(z)_0+1}.p_1^{(z)_1} ... \, p_m^{(z)_n} ... = \frac{2zp_m^{(z)_n}}{p_m^{(z)_m}}.$$

При оператор за преход $J(m,n,q)$ имаме, че

$$next(J(m,n,q),(l,\widetilde{x})) = \begin{cases} (q,\widetilde{x}), & \text{ако } x_m = x_n \\ (l+1,\widetilde{x}), & \text{ако } x_m \neq x_n. \end{cases}$$

Следователно $Next(J(m,n,q),z)$ можем да представим като:

$$Next(J(m,n,q),z) = \begin{cases} \frac{z.2^q}{2^{(z)_0}}, & \text{ако } (z)_m = (z)_n \\ 2z, & \text{ако } (z)_m \neq (z)_n. \end{cases}$$

Така проверихме верността на следното

Твърдение 3.2. За всеки фиксиран оператор I, функцията $\lambda z.Next(I,z)$ е примитивно рекурсивна.

Оттук ще следва, че и функцията *Step* ще е примитивно рекурсивна. Да видим:

Твърдение 3.3. За всяка МНР програма P функцията $Step\colon \mathbb{N} \longrightarrow \mathbb{N}$ е примитивно рекурсивна.

Доказателство. Да фиксираме програмата $P\colon\ I_0, \dots, I_k$ и да препишем представянето (3.2) за *step* посредством $Next$. Ще получим

$$Step(z) = \begin{cases} Next(I_0,z), & \text{ако } z > 0 \ \&\ (z)_0 = 0 \\ \dots\dots\dots\dots & \\ Next(I_k,z), & \text{ако } z > 0 \ \&\ (z)_0 = k \\ z, & \text{ако } z > 0 \ \&\ (z)_0 > k \\ 0, & \text{ако } z = 0. \end{cases}$$

Виждаме, че $Step$ се дефинира с разглеждане на случаи и следователно е примитивно рекурсивна, съгласно *Твърдение* 1.10 и горното *Твърдение* 3.2. $\square$

Вече сме всъстояние да покажем твърдението, което беше основна цел на настоящия раздел, а именно:

Твърдение 3.4. Всяка изчислима функция е частично рекурсивна.

Доказателство. Да вземем произволна n-местна изчислима функция f, която се пресмята от някаква МНР програма $P\colon\ I_0, \dots, I_k$. Тогава за всички естествени $x_1, \dots, x_n, y$ е изпълнено:

$$f(x_1, \dots, x_n) \simeq y \iff \exists t \exists l \exists \tilde{z}\ (step^t(0, \bar{x}, 0, 0, \dots) = (l, y, \tilde{z})\ \&\ l > k).$$

Да означим с $t(\bar{x})$ функцията, която връща първия такт, на който програмата P спира върху $(\bar{x}, 0, 0, \dots)$ (ако въобще спре). Ако P не спира върху $(\bar{x}, 0, 0, \dots)$ $t(\bar{x})$, по дефиниция $\neg!t(\bar{x})$. Тогава за $t(x_1, \dots, x_n)$ ще имаме:

$$\begin{aligned} t(x_1, \dots, x_n) &\simeq \mu t[(Step^t(\delta(0, x_1, \dots, x_n, 0, 0, \dots)))_0 > k] \\ &\simeq \mu t[(Step^*(t, \delta(0, x_1, \dots, x_n, 0, 0, \dots)))_0 > k]. \end{aligned}$$

Кодът на началната конфигурация $\delta(0, x_1, \dots, x_n, 0, 0, \dots)$ е $p_1^{x_1}. \dots .p_n^{x_n}$, така че функцията $d(\bar{x}) = \delta(0, \bar{x}, 0, 0, \dots)$ е примитивно рекурсивна. Освен това итерацията $Step^*$ на функцията $Step$ също е примитивно рекурсивна, съгласно *Твърдение* 1.17 и *Твърдение* 3.3. Тогава функцията $t(\bar{x})$, която можем да препишем като

$$t(\bar{x}) \simeq \mu t[(Step^*(t, d(\bar{x})))_0 > k]. \tag{3.4}$$

ще е частично рекурсивна. Финално за f ще имаме

$$f(\bar{x}) \simeq (Step^*(t(\bar{x}), d(\bar{x})))_1 \tag{3.5}$$

и следователно f също е частично рекурсивна. $\square$

Сега събираме заедно *Твърдение* 3.1 и *Твърдение* 3.4, за да получим следния важен резултат.

Теорема 3.1. (Теорема за еквивалентност) Една функция е частично рекурсивна точно тогава, когато е изчислима.

Като непосредствено следствие от представянето (3.5) получаваме следното любопитно наблюдение:

Следствие 3.1. Всяка частично рекурсивна функция може да бъде получена с прилагане на една единствена минимизация.

Доказателство. Ако f е частично рекурсивна, то съгласно горната теорема тя е изчислима, и значи може да се представи във вида (3.5). В това представяне всички функции, с изключение на $t(\bar{x})$, са примитивно рекурсивни, а самата $t(\bar{x})$ се получава от примитивно рекурсивни с една единствена минимизация, както е видно от равенството (3.4).

$\square$

3.1.3 Тезис на Чърч-Тюринг

Теоремата за еквивалентност има и важно методологическо значение. Фактът, че съвпадат два класа от функции, определени посредством два съвсем различни изчислителни модела означава, че тези класове не са случайни. Тази теорема е аргумент в подкрепа на *Тезиса на Тюринг*, който в първоначалния си вариант, изказан през 1936 г. от Тюринг, гласи: една функция е алгоритмично изчислима (в широк, неформален смисъл) точно когато е изчислима с машина на Тюринг. По същото време, независимо от Тюринг, и Чърч формулира свой *Тезис на Чърч*, който твърди подобно нещо, само че за λ-определимите функции, въведени от него.

Впоследствие възникват и много други изчислителни модели, за които се доказва, че функциите, определими във всеки един от тях съвпадат с функциите, изчислими с машини на Тюринг. С други думи, всеки независим опит да се въведе формално понятие за алгоритмично изчислима функция води до изчислимостта по Тюринг. Затова всички тези модели се наричат *тюрингово пълни.*

До ден днешен никой не е посочил функция, която да е изчислима в някакъв приемлив смисъл, но да не е изчислима с машина на Тюринг. Този факт, заедно с казаното по-горе, дава основание на хората, които се занимават с Теоретична информатика, да се обединят около следното твърдение, което е прието да се нарича

Тезис на Чърч-Тюринг. Една функция е алгоритмично изчислима тогава и само тогава, когато е изчислима с машина на Тюринг.

Разбира се, Тезисът на Чърч-Тюринг е твърдение, която никога няма да бъде доказано, защото в неговата формулировка участва нематематическото понятие "алгоритмично изчислима функция".

3.2 Теорема за универсалната функция

Да си припомним дефиницията (2.7) за универсална функция на произволен клас $\mathcal{K}$. Нашата цел ще бъде да построим универсалната функция за класа $\mathcal{C}_n$ на всички n-местни изчислими функции.

Функцията $U(a, x_1, \ldots, x_n)$ е *универсална* за класа $\mathcal{K} \subseteq \mathcal{F}_n$, ако:

0) U е изчислима;

1) за всяка $f \in \mathcal{K}$ съществува $a \in \mathbb{N}$: $f = \lambda\bar{x}.U(a, \bar{x})$;

2) за всяко $a \in \mathbb{N}$ функцията $\lambda\bar{x}.U(a, \bar{x}) \in \mathcal{K}$.

3.2.1 Теорема за универсалната функция

При фиксирано $n \geq 1$ да означим с Φ_n следната $n+1$-местна функция:

$$\Phi_n(a, x_1, \ldots, x_n) \simeq \varphi_a^{(n)}(x_1, \ldots, x_n) \tag{3.6}$$

за всяко $a \in \mathbb{N}$ и $\bar{x} \in \mathbb{N}^n$.

Ще покажем, че Φ_n се явява универсална за класа $\mathcal{C}_n$. Най-напред, от *Твърдение* 2.4 имаме, че за всяка функция f:

$$f \in \mathcal{C}_n \implies \exists a\ f = \varphi_a^{(n)} \implies \exists a\ f = \lambda\bar{x}.\Phi_n(a, \bar{x}).$$

Следователно за Φ_n е в сила условието 1) от дефиницията за УФ за класа $\mathcal{C}_n$. Следващото условие 2) е изпълнено автоматично, защото

$$\lambda\bar{x}.\Phi_n(a, \bar{x}) \stackrel{\text{деф}}{=} \varphi_a^{(n)}$$

принадлежи на класа $\mathcal{C}_n$ по смисъла на определението си. Никак не е автоматична, обаче, проверката, че Φ_n удовлетворява и условието 0). Фактът, че Φ_n е изчислима, е един от най-важните резултати в Теория на изчислимостта и обикновено се нарича *теорема за универсалната функция*.

Теорема 3.2. (Теорема за универсалната функция) За всяко $n \geq 1$ функцията Φ_n е изчислима.

Доказателство. Да фиксираме някакво $n \geq 1$. Ще покажем, че Φ_n е частично рекурсивна функция, което ще означава, че Φ_n е изчислима, съгласно теоремата за еквивалентност.

Да отбележим, че директното доказателство на изчислимостта на Φ_n на практика означава да се построи *универсалната програма* за МНР — нещо, което технически е доста по-трудно за реализиране, защото трябва да програмираме на езика за МНР, а той далеч не е най-удобният за тази цел.

Да препишем определението на Φ_n чрез дефиницията (2.5) на $\varphi_a^{(n)}$:

$$\Phi_n(a, \bar{x}) \simeq y \ = \ \varphi_a^{(n)}(\bar{x}) \simeq y \iff P_a \text{ спира върху } \bar{x} \text{ с резултат } y.$$

С други думи, $\Phi_n(a, \bar{x})$ връща резултата от работата на програмата P_a върху входа $\bar{x}$. За да опишем формално как работи P_a върху $\bar{x}$, въвеждаме следната спомагателна функция Q_n:

$$Q_n(a, \bar{x}, t) \stackrel{\text{деф}}{=} \text{кода на конфигурацията, която се получава след } t \text{ такта от работата на } P_a \text{ върху } \bar{x}.$$

Ще покажем, че Q_n е примитивно рекурсивна, като напишем за нея примитивно рекурсивна схема по последния ѝ аргумент t. Но как да изразим $Q_n(a, \bar{x}, t+1)$ чрез $Q_n(a, \bar{x}, t)$? Тъй като конфигурацията $Q_n(a, \bar{x}, t+1)$ е "следващата конфигурация" след $Q_n(a, \bar{x}, t)$ на помощ ни идва функцията $next$ от по-горе, и по-точно, нейната "цифровизираната" версия $Next$, която въведохме с равенството (3.3). $Next(I, z)$ дава кода на конфигурацията, която се получава, когато приложим инструкцията I към конфигурацията с код z. В нашия случай искаме да извикаме $Next$ върху конфигурацията на стъпка t, т.е. при $z = Q_n(a, \bar{x}, t)$. Инструкцията, която ще прилагаме към тази конфигурация, е текущата инструкция на стъпка t. Нейният адрес е точно $(Q_n(a, \bar{x}, t))_0$, а (кодът на) самата инструкция с този адрес няма проблем да възстановим от кода a на програмата P_a.

За да направим нещата точни, ще трябва да въведем още една — този път изцяло числова функция — от серията "следващи", която ще наречем $NEXT$. Тази функция, извикана върху *кода* $\beta(I)$ на инструкцията I и кода на конфигурацията $(l, \widetilde{x})$, ще връща $Next(I, \delta(l, \widetilde{x}))$. Ако си представяме $\delta(l, \widetilde{x}))$ като z, ще имаме:

$$\underline{NEXT(\beta(I), z) = Next(I, z).}$$

Тъй като 0 не е код на конфигурация, можем да положим $NEXT(\beta(I), 0) \stackrel{\text{деф}}{=} 0$ за всяка инструкция I.

В доказателството на *Твърдение* 3.2 видяхме, че за всяка от четирите вида инструкции I имаме следните представяния за $Next(I, z)$ при $z > 0$:

$$Next(S(n), z) = 2zp_n$$

$$Next(Z(n), z) = \frac{2z}{p_n^{(z)_n}}$$

$$Next(T(m, n), z) = \frac{2zp_m^{(z)_n}}{p_m^{(z)_m}}$$

$$Next(J(m, n, q), z) = \begin{cases} \frac{z.2^q}{2^{(z)_0}}, & \text{ако } (z)_m = (z)_n \\ 2z, & \text{ако } (z)_m \neq (z)_n. \end{cases}$$

Следователно при $z > 0$ за функцията $NEXT$ ще имаме:

$$NEXT(\beta(I), z) = \begin{cases} 2z.p_n, & \text{ако } \ I = S(n) \\ \frac{2z}{p_n^{(z)_n}}, & \text{ако } \ I = Z(n) \\ \frac{2z.p_m^{(z)_n}}{p_m^{(z)_m}}, & \text{ако } I = T(m, n) \\ \frac{z.2^q}{2^{(z)_0}}, & \text{ако } \ I = J(m, n, q) \ \& \ (z)_m = (z)_n \\ 2z, & \text{ако } \ I = J(m, n, q) \ \& \ (z)_m \neq (z)_n. \end{cases}$$

Разбира се, добре е да заменим с някаква променлива — примерно i, първия аргумент $\beta(I)$. Правим го веднага:

$$NEXT(i,z) = \begin{cases} 2z.p_n, & \text{ако } \ i = \beta(S(n)) \\ \frac{2z}{p_n^{(z)n}}, & \text{ако } \ i = \beta(Z(n)) \\ \frac{2z.p_m^{(z)n}}{p_m^{(z)m}}, & \text{ако } \ i = \beta(T(m,n)) \\ \frac{z.2^q}{2^{(z)0}}, & \text{ако } \ i = \beta(J(m,n,q)) \ \& \ (z)_m = (z)_n \\ 2z, & \text{ако } \ i = \beta(J(m,n,q)) \ \& \ (z)_m \neq (z)_n. \end{cases}$$

Остана една последна стъпка — да премахнем всички променливи, различни от i и z в дясната част на горното равенство. За целта трябва да имаме пред себе си определението (2.1) на код на инструкция $\beta(I)$, който дефинирахме с различен остатък по модул 4, в зависимост от вида на I:

$$\begin{aligned} \beta(S(n)) &= 4(n-1) \\ \beta(Z(n)) &= 4(n-1)+1 \\ \beta(T(m,n)) &= 4\Pi(m-1,n-1)+2 \\ \beta(J(m,n,q)) &= 4\Pi_3(m-1,n-1,q)+3. \end{aligned}$$

Ясно е, че ще ни е нужно да изразим параметрите в инструкцията I (числата m, n или q) чрез нейния код $\beta(I)$. Когато доказвахме *Твърдение* 2.2, ние всъщност вече го направихме. Да си припомним как ставаше това във всеки от четирите случая за I:

— ако $i = \beta(S(n))$, т.е. $i = 4(n-1)$, то $n = [\frac{i}{4}]+1$

— ако $i = \beta(Z(n))$, т.е. $i = 4(n-1)+1$, то $n = [\frac{i}{4}]+1$

— ако $i = \beta(T(m,n))$, т.е. $i = 4\Pi(m-1,n-1)+2$, то $m = L([\frac{i}{4}])+1$ и $n = R([\frac{i}{4}])+1$

— ако $i = \beta(J(m,n,q))$, т.е. $i = 4\Pi_3(m-1,n-1,q)+3$, то $m = J_1^3([\frac{i}{4}])+1$, $n = J_2^3([\frac{i}{4}])+1$ и $q = J_3^3([\frac{i}{4}])$.

Сега преписваме финално дефиницията на $NEXT(i,z)$, като си представяме, че навсякъде в нея буквите m, n и q са заместени със съответните изрази от по-горе:

$$NEXT(i,z) = \begin{cases} 2z.p_n, & \text{ако } \ z > 0 \ \& \ i \equiv 0 \ (mod\ 4) \\ \frac{2z}{p_n^{(z)n}}, & \text{ако } \ i \equiv 1 \ (mod\ 4) \\ \frac{2z.p_m^{(z)n}}{p_m^{(z)m}}, & \text{ако } \ i \equiv 2 \ (mod\ 4) \\ \frac{z.2^q}{2^{(z)0}}, & \text{ако } \ i \equiv 3 \ (mod\ 4) \ \& \ (z)_m = (z)_n \\ 2z, & \text{ако } \ i \equiv 3 \ (mod\ 4) \ \& \ (z)_m \neq (z)_n \\ 0, & \text{ако } \ z = 0. \end{cases}$$

Всички участващи в дефиницията на $NEXT$ функции и предикати са примитивно рекурсивни (да си дадем сметка също, че всички деления са целочислени). Следователно $NEXT$ е примитивно рекурсивна.

Сега да се върнем на функцията $Q_n(a, \bar{x}, t)$, която дефинирахме с равенството (3.7). Тази функция даваше кода на конфигурацията на стъпка t от работата на P_a върху $\bar{x}$. Ще покажем, че тя е примитивно рекурсивна, като напишем примитивно рекурсивна схема за нея. Както вече съобразихме, рекурсията ще бъде по последната променлива t. Базовият случай е ясен:

$$Q_n(a, \bar{x}, 0) = \delta(0, x_1, \dots, x_n, 0, \dots) \stackrel{\text{деф}}{=} p_1^{x_1} \ \dots \ p_n^{x_n}.$$

Нека конфигурацията на стъпка t е $(l, \widetilde{x})$, т.е. $Q_n(a, \bar{x}, t) = \delta(l, \widetilde{x})$. Нека още $l \leq lh(a)$. Адресът l на инструкцията, която трябва да се изпълни на тази стъпка, е $(Q_n(a, \bar{x}, t))_0$, а самата инструкция I_l възстановяваме от кода a на програмата P_a. Ако P_a е програмата $I_0, \dots, I_k$ (тук $k = lh(a)$), то от определението (2.2) за нейния код $\gamma(P_a)$ имаме:

$$\gamma(P_a) \stackrel{\text{деф}}{=} a = \tau(\langle \beta(I_0), \dots, \beta(I_k) \rangle).$$

Тогава $\beta(I_l)$ просто ще е l-тият елемент от редицата с код a, който се даваше от функцията $mem(a, l)$, дефинирана с (1.10).

Значи имаме следното изразяване на $Q_n(a, \bar{x}, t+1)$ чрез $Q_n(a, \bar{x}, t)$:

$$Q_n(a, \bar{x}, t+1) = \begin{cases} NEXT(mem(a, l), Q_n(a, \bar{x}, t)), & \text{ако } l \leq lh(a) \\ Q_n(a, \bar{x}, t), & \text{ако } l > lh(a) \end{cases}$$

$$= \begin{cases} NEXT(mem(a, (Q_n(a, \bar{x}, t))_0), Q_n(a, \bar{x}, t)), & \text{ако } (Q_n(a, \bar{x}, t))_0 \leq lh(a) \\ Q_n(a, \bar{x}, t), & \text{ако } (Q_n(a, \bar{x}, t))_0 > lh(a). \end{cases}$$

Така получаваме следната примитивно рекурсивната схема за Q_n:

$$\left| \begin{array}{l} Q_n(a, \bar{x}, 0) = p_1^{x_1} \ \dots \ p_n^{x_n} \\ Q_n(a, \bar{x}, t+1) = G(a, \bar{x}, t, Q_n(a, \bar{x}, t)). \end{array} \right.$$

В нея с G сме означили функцията

$$G(a, \bar{x}, t, z) = \begin{cases} NEXT(mem(a, (z)_0), z), & \text{ако } (z)_0 \leq lh(a) \\ z, & \text{ако } (z)_0 > lh(a). \end{cases}$$

Понеже g и G са примитивно рекурсивни, то и Q_n ще е примитивно рекурсивна.

От дефиницията на Q_n се вижда, че ако програмата P_a спре върху вход $\bar{x}$, то това ще стане за брой стъпки $t_n(a, \bar{x})$, където

$$t_n(a, \bar{x}) \simeq \mu t[(Q_n(a, \bar{x}, t))_0 > lh(a)].$$

Тогава очевидно t_n е частично рекурсивна функция.

Резултатът y (ако го има) от работата на P_a върху $\bar{x}$ по дефиниция е съдържанието на първия регистър на заключителната конфигурация, или все едно

$$y \simeq (Q_n(a, \bar{x}, t_n(a, \bar{x})))_1.$$

Следователно функцията $\varphi_a^{(n)}$, която P_a пресмята, се изразява чрез Q_n по следния начин:

$$\varphi_a^{(n)}(\bar{x}) \simeq (Q_n(a, \bar{x}, t_n(a, \bar{x})))_1.$$

Тук отчитаме, че ако P_a не спре върху $\bar{x}$, то $t_n(a, \bar{x})$ и $\varphi_a^{(n)}(\bar{x})$ няма да имат стойност. Следователно и двете страни на горното условно равенство ще са недефинирани, и значи то отново ще е в сила.

От това равенство и от дефиницията (3.6) на Φ_n получаваме финално

$$\Phi_n(a, \bar{x}) \simeq (Q_n(a, \bar{x}, t_n(a, \bar{x})))_1.$$

Следователно Φ_n е частично рекурсивна, което значи и изчислима, съгласно *Твърдение* 3.1. □

Забележка. От това, че $\Phi_n(a, \bar{x})$ е частично рекурсивна, можем да заключим, че такива ще са и функциите $\varphi_a^{(n)} = \lambda\bar{x}.\Phi_n(a, \bar{x})$ за всяко фиксирано a. Но това означава, че всички изчислими функции са частично рекурсивни, което е точно *Твърдение* 3.4. Излиза, че можеше да го получим и като следствие от горната теорема. Така е, но ние предпочетохме да докажем теоремата за еквивалентност независимо от теоремата за универсалната функция, за да не смесваме двете явления — еквивалентността на двата подхода и съществуването на универсална функция. Освен това цялата техническа работа, която свършихме при доказателството на първата теорема, беше използвана в доказателството на втората, така че трудът ни не беше напразен.