Vyčíslitelnost I

TeXed by Martin Všetička

1 O poznámkách

Toto jsou poznámky, které vznikly kompilací materiálů:

- wiki-skriptíčka na http://wiki.matfyz.cz,
- poznámky Martina Trčky ze Studnice,
- moje poznámky a
- poznámky Honzy Bíma.

Budu rád za jakékoli návrhy na úpravu, připomínky a konstruktivní kritiku. Rád rozšířím seznam autorů (Pokud zde chceš být, napiš mi!), zatím však nevím, jak se k tomu postavit. Snad tedy tyto poznámky pomůžou v úspěšném složení zkoušky z Vyčíslitelnosti I :-).

2 Historie

- 1931 . . . Godel - důkaz věty o neúplnosti (existují sekvence, které nejsou dokazatelné ani vyvratitelné)
- 1936 . . . Vznik vyčíslitelnosti: Godel, Kleene, Herbrand, Alonso Church, Turing, . . . Snaha o přesné zadefinování algoritmu:
  - částečně rekurzivní funkce (Godel, Kleene,. . .)
  - Turingovy stroje (Turing)
  - λ-definované funkce (Church, Herbrant)

  - Postovy systémy

  Ukázalo se, že tyto věci jsou vzájemně ekvivalentní.

- Začátek 20. století:
  - *finitismus* - "nekonečné množiny neexistují"
  - *formalismus* (od Hilberta) - formální systém (teorie 1. řádu, axiomy)
    * Lze algoritmicky rozhodovat?
    * Finitně dokázat, že systém je bezesporný?
    * Godel odpověděl: NELZE, stačí mít trochu silnější teorii (+, *) a efektivní axiomy a nedokáži bezespornost.

3 Úvod

Některé pojmy, o kterých se píše v této kapitole (zvlášť ke konci), budou podrobně vysvětleny v následujících kapitolách. Myslím si ale, že je užitečné mít už předem aspoň nějakou mlhavou představu, ono to pak snadněji do sebe zapadá. Představu o tom, k čemu ta vyčíslitelnost vlastně je, si můžete udělat také z Johančiny pohádky první.

3.1 Čísla, funkce, programy

3.1.1 $0 \in \mathbb{N}$. Číslo = přirozené číslo

Ve vyčíslitelnosti (stejně jako například v teorii množin) se jako přirozená čísla označuje množina $\mathbb{N} = \{0, 1, 2, ...\}$. Je to jednodušší, než vždy psát $\mathbb{N} \cup \{0\}$. S jinými čísly (celá, reálná, komplexní,...) se ve vyčíslitelnosti nepracuje (pokud ano, tak se zakódují do přirozeného čísla), takže dál v tomto textu se pojmem *číslo* rozumí vždy *přirozené číslo* včetně nuly.

3.1.2 Co to je funkce ve vyčíslitelnosti?

Funkcí se myslí vždy tzv. aritmetická funkce, která k-tici čísel přiřazuje jedno číslo, tedy $f : \mathbb{N}^k \to \mathbb{N}$. Tato funkce navíc nemusí být *totální* (říkáme také *úplná*), tedy může být *parciální* (říkáme také *částečná*), což znamená, že nemusí být "všude definovaná". V matematice se zápisem $f : A \to B$ obvykle značí, že definiční obor f je A (zapisujeme $dom(f) = A$). Ve vyčíslitelnosti, ale platí $dom(f) \subseteq A$. Ještě jinými slovy: pro některé k-tice nemusí být hodnota f definována a tyto k-tice pak nepatří do definičního oboru f.

3.1.3 Funkce lze chápat jako programy a naopak

Funkci $f : \mathbb{N}^k \to \mathbb{N}$ si můžeme představit jako program, který na vstupu dostane k čísel a vrátí jedno číslo. Tedy schematicky:

```
program_pro_f(x1,...,xk){

  ...
  return(y);
}
```

Každý konečný matematický objekt (např. celé číslo, slovo nad nějakou abecedou, dvojice čísel,...) lze zakódovat jako jedno (přirozené) číslo (konkrétní metody kódování budou popsány dále). Vezmeme-li si tedy program (který nemá žádné side-efekty (např. nic netiskne, nepřiřazuje do globálních proměnných, ...) a jeho návratová hodnota závisí jen na vstupních parametrech), tak tento program můžeme chápat jako funkci $\mathbb{N}^k \to \mathbb{N}$.

3.1.4 Nulární funkce

U programů je celkem běžné, že nemusí mít vstup. U funkcí by tomu odpovídaly nulární funkce, které se občas zavádějí (např. v algebře), aby se nemuselo zvlášť ošetřovat konstanty. Mohli bychom si je zavést i ve vyčíslitelnosti, ale na přednášce se to tak nedělá, protože se bez nich obejdeme. Nic nám totiž nebrání v tom, abychom si vyrobili funkci, která svůj parametr vůbec nepoužije (funkce je konstantní; např. $f(x) = 5$).

3.1.5 Co znamená, že program či funkce konverguje či diverguje?

Jistě znáte nějaký program, který se pro nějaký vstup zacyklí a nikdy se nezastaví.

Příklad:

```
if (x == 42) {
   while (true) do {}
}
```

Pokud se program zacyklí pro všechny vstupy, říkáme, že je to *prázdný program*, kterému odpovídá *prázdná funkce*, což jest funkce f taková, že $dom(f) = \emptyset$. Aby to znělo učeněji, tak místo *program se zacyklí* (resp. *funkce není definována*) říkáme *program diverguje* (resp. *funkce diverguje*) a píšeme $f(x_1, ..., x_n) \uparrow$. Obdobně místo *program se zastaví (po konečném počtu kroků)* (resp. *funkce je definována*) říkáme, že *program konverguje* (resp. *funkce konverguje*) a píšeme $f(x_1, ..., x_n) \downarrow$. Například pro funkci f jedné proměnné tedy máme $dom(f) = \{x | f(x) \downarrow\}$.

3.1.6 Co znamená zkratka „$\downarrow=$“ ?

Zápis $f(x) \downarrow= y$ je zkratkou za $f(x)\downarrow \ \& \ \ f(x) = y$, tedy $f(x)$ konverguje a rovná se y. Exaktně vzato by stačilo psát $f(x) = y$, protože z toho už vyplývá, že funkce konverguje. Zápis $f(x) \downarrow= y$ se ale používá v situacích, kdy nás chce jeho autor upozornit, že to není tak samozřejmé, že $f(x)$ konverguje. Případně to může znamenat, že se v důkazu nejprve musí dokázat konvergence a až poté výsledná hodnota.

3.2 Funkce, predikát, množina, relace, problém

Je dobré si uvědomit, že tyto pojmy jsou na sebe jednoduše převeditelné, jde jen o úhel pohledu. V důkazech se obvykle tyto převody ani nezmiňují.

3.2.1 Predikát jako funkce

Predikát k proměnných $P(x_1, ..., x_k)$ je nějaké tvrzení o těchto proměnných, které může být pravdivé či nepravdivé.

Příklad: $P(x_1, x_2) := (x_1 \text{ je prvočíslo}) \ \& \ (x_2 > x_1)$.

Ve vyčíslitelnosti budou proměnnými vždy čísla a pravdu reprezentujeme jako 1, nepravdu jako 0. Predikáty lze tedy chápat jako funkce, jejichž obor hodnot je 0,1 (zapisujeme $rng(f) = \{0, 1\}$).

3.2.2 Množina jako predikát

Množinu lze chápat jako soubor prvků, které mají nějakou vlastnost P, tedy množina $M = \{x \mid P(x) \text{ platí}\}$. Pokud nebude uvedeno jinak, bude v následujícím textu pojem *množina* znamenat vždy *množina (nějakých) přirozených čísel*. Každý predikát jedné proměnné odpovídá nějaké množině a naopak. Jelikož už víme, že predikáty odpovídají funkcím, tak nás nepřekvapí, že množiny odpovídají funkcím jedné proměnné s oborem hodnot 0,1. Takovouto funkci nazýváme *charakteristická funkce množiny* a pro množinu M píšeme $C_M : \mathbb{N} \to \{0, 1\}$ a $C_M(x) = \begin{cases} 1 & \text{pro } x \in M \\ 0 & \text{pro } x \notin M \end{cases}$.

Všimněte si, že charakteristická funkce je z definice totální.

3.2.3 Relace jako predikát

Relace na množinách $A_1, ..., A_n$ je podmnožina jejich kartézského součinu. Ve vyčíslitelnosti jako množiny A_i bereme (kupodivu opět:-)) $\mathbb{N}$. Relace na n množinách je v důsledku to samé jako predikát n proměnných. Nenechte se tedy zmást, že se občas místo predikátu používá relace.

Poznámka: Zajímavé je, že zatímco v teorii množin se odvozuje funkce z relace, zde bereme funkci jako základní pojem a relace převádíme na funkce.

3.2.4 Problém jako množina

V teorii složitosti se *instance problému* definuje jako slovo nad abecedou 0,1 (tedy pro nás číslo) a *problém* Q je pak definován jako úloha rozhodnout, zda daná instance problému patří do jazyka $L(Q)_Y$, tedy do množiny kladných instancí. Zjednodušeně řečeno: problémy jsou zase jen množiny.

- Problém je *rozhodnutelný* (též se říká *efektivně rozhodnutelný* či *rekurzivní*), jestliže existuje TS, který se zastaví na libovolném vstupu a skončí v přijímacím stavu právě pro ty vstupy, které jsou řešením daného problému. O množině kladných instancí pak říkáme, že je to *rekurzivní množina*, případně že je efektivně rozhodnutelná (o každém prvku můžeme efektivně zjistit, zda do ní patří, nebo nepatří).
- Problém je *částečně rozhodnutelný* (též se říká *vyčíslitelný* či *rekurzivně spočetný*), jestliže existuje TS, který se zastaví (v přijímacím stavu) na vstupech, pro které je odpověď ANO (na vstupech, pro které je odpověď NE, se zastavit nemusí). O množině kladných instancí pak říkáme, že je to *rekurzivně spočetná množina* - existuje pro ni program, který se na vstupu x zastaví, pokud prvek x patří do té množiny. Dokud se nezastaví, nevíme nic. *Příklad ze života: když se hraje hokej, prodloužení, je to nerozhodně, vy to nesledujete a jen slyšíte případný řev zvenku, tak pokud se tento řev ozve, víte, že jsme vyhráli, ale pokud je ticho, nevíte nic. Takže hokejový řev je rekurzivně spočetný.*
- Problém je *nerozhodnutelný*, jestliže není rozhodnutelný. Existují nerozhodnutelné problémy, které jsou částečně rozhodnutelné (Halting problem), ale existují také problémy, které nejsou ani částečně rozhodnutelné.

3.3 ČRF a další trojpísmenné zkratky

Jak jsem již předeslal, exaktní definice ČRF, ORF a spol. se nacházejí v následujících kapitolách, ale přijde mi šikovné, když si člověk napřed udělá nějaký rámcový přehled, případně si nové termíny propojí se znalostmi, které už má z jiných přednášek (hlavně z Automatů a gramatik).

3.4 ČRF = částečně rekurzivní funkce

Lze dokázat (viz dále), že ke každému programu (v libovolném programovacím jazyce, či abstraktním modelu jakým je např. Turingův stroj – dále jen TS) jde zkonstruovat ekvivalentní ČRF. Slovem *ekvivalentní* se zde myslí to, že program i funkce dojdou pro libovolný vstup ke stejnému výsledku, délka výpočtu (počet kroků, ať už to znamená cokoli) ovšem může být řádově delší. To je ale ve vyčíslitelnosti běžné: zajímá nás jen co (ne)jde spočítat a jakými prostředky, nikoli, jak dlouho to bude trvat, od toho tu je teorie složitosti. Obecný program se může pro nějaký vstup i zacyklit, tudíž i ČRF nemusí být totální.

Téměř jako synonymum k termínu *částečně rekurzivní* (ČR) se používá termín *rekurzivně spočetné* (RS), každý z těchto dvou přívlastků se ovšem pro zmatení nepřítele (jak jinak než z řad studentstva) tradičně spojuje s jinými podstatnými jmény. O funkci se říká, že je to ČRF, kdežto o predikátu, že je to RSP (rekurzivně spočetný predikát), a o množině, že je to RSM (rekurzivně spočetná množina).

Možná už znáte z Automatů a gramatik rekurzivně spočetné jazyky definované jako třídu jazyků přijímaných nějakým TS. Tedy jazyk L je RS, existuje-li TS, jenž se zastaví v koncovém přijímacím stavu právě na všech slovech daného jazyka. Pokud nějaké slovo v daném jazyce není, tak se buď TS zastaví v nekoncovém stavu nebo se zacyklí (tj. nikdy se nezastaví). Chceme-li zjistit, zda nějaké slovo patří do L, pak pustíme TS a čekáme, ale dokud se stroj nezastaví, tak nic nevíme.

TS přijímající L lze jednoduše upravit tak, aby se zastavil právě na slovech z L: pokud by se mohl zastavit v nepřijímacím stavu, tak přidáme instrukce, které výpočet zacyklí. Máme-li ČRF f, která je ekvivalentní s takto upraveným TS, tak platí, že $L = dom(f)$. Vysvětlení: každému slovu lze přiřadit jeho kód, tedy číslo; f bude funkce jedné proměnné; $dom(f)$ je množina kódů slov jazyka L. Množina $dom(f)$ je RSM.

Z částečně rekurzivních funkcí lze odvodit rekurzivně spočetné množiny a rekurzivně spočetné predikáty.

3.5 ORF = obecně rekurzivní funkce

OR funkce jsou ty ČR funkce, které jsou totální. Z OR funkcí lze odvodit *obecně rekurzivní predikáty* (ORP) a *(obecně) rekurzivní množiny* – u množin se slovo *obecně* vynechává.

3.6 PRF = primitivně rekurzivní funkce

Jak si později ukážeme, PR funkce jsou vlastní podmnožinou OR funkcí. Jsou to takové hezké funkce, které vždy konvergují. Lze pomocí nich vyjádřit většina známých operací jako například: sčítání, násobení, negace, mocnění, test na prvočíselnost,.... Naopak se těžko hledá úkol, který by nezvládly (tím je např. Ackermannova funkce). Zatím naprosto neformálně můžeme říct, že PRF odpovídají programům, které mohou používat jen for-cykly, ale nikoli while-cykly.

Příklad: To jest v PRF jde naprogramovat:

```
for i := 1 to x do ...
```

Nikoli však ty céčkovské for-cykly, které simulují while-cyklus!

Programy odpovídající ČRF mohou navíc používat právě onen while-cyklus (který ale může běžet do nekonečna a tak vznikne divergence).

3.6.1 Přehled zkratek a terminologie

	Funkce	Predikát	Množina	Problém
Primitivně rek.	PRF	PRP	nezavádí se	nezavádí se
Obecně rek.	ORF	ORP	rek. množina	rozhodnut.
Částečně rek.	ČRF	RSP	RSM	část. rozhodnut.
Rekurzivně spoč.	ČRF	RSP	RSM	část. rozhodnut.

3.7 Efektivní vyčíslitelnost

Efektivní vyčíslitelnost se definuje jako to, co je vyčíslitelné (řešitelné) pomocí ČRF. Vzhledem k již zmíněné ekvivalenci TS a ČRF (kterou ale dokážeme

až později[1]), jsou následující výrazy synonyma:

efektivně vyčíslitelné = turingovsky vyčíslitelné = algoritmicky vyčíslitelné = algoritmicky řešitelné atd.

Když jsem poprvé slyšel o efektivní vyčíslitelnosti, tak jsem si vzpomněl na různé efektivní algoritmy s $n \log n$ složitostí a podobně. Ne, ne, to je něco kapku jiného. Ve vyčíslitelnosti slovem *efektivní* myslíme, že to vůbec jde. (Jestli někomu přijde efektivní způsob, kterým pracuje univerzální TS, když pobíhá po pásce sem tam a přenáší čárky... no, já nevím.) Zdrojem tohoto nedorozumění je překlad dvou různých anglických přídavných jmen, *efficient* a *effective*, pomocí jednoho českého *efektivní*.

4 Prerekvizity

4.1 Spočetnost a kardinality množin

Neformálně:

- Množiny jsou stejně velké (mají stejnou kardinalitu), pokud mezi nimi existuje bijekce.
- Množina je spočetná, má-li stejnou kardinalitu jako $\mathbb{N}$.
- Není těžké dokázat, že množina $\mathbb{N} \cup \{a\}$ (přirozená čísla a ještě jeden nový prvek k tomu) je spočetná. Bijekcí na $\mathbb{N}$ je zobrazení $f(a) = 0$ a $f(x) = x + 1$ pro $x \in \mathbb{N}$.
- Obdobně snadné je najít bijekci mezi $\mathbb{Z}$ a $\mathbb{N}$.
- O něco těžší je dokázat, že i množina $\mathbb{N} \times \mathbb{N}$ je spočetná. Jde tedy o to, umět zakódovat dvojici čísel jako jedno číslo, tedy najít funkci $f(a, b) = \langle a, b \rangle$, kde $\langle a, b \rangle$ značí kód dvojice čísel. Kdyby stačilo jen najít prosté zobrazení do $\mathbb{N}$, tak můžeme použít $f(a, b) = 2^a 3^b$ (místo 2 a 3 mohou být i jiná prvočísla). Pokud požadujeme zobrazení bijektivní, můžeme použít například Cantorovo diagonální uspořádání.

[1] TODO: add link

4.2 Cantorovo diagonální uspořádání

Následující funkce nám říká, jakým způsobem můžeme zakódovat dvojici čísel do jednoho:

$$f(a,b) = \langle a,b \rangle = \frac{(a+b)(a+b+1)}{2} + a$$

Pár hodnot si můžeme prohlédnout v tabulce:

a/b	0	1	2	3
0	0	1	3	3
1	2	4	7	...
2	5	8	...	...

Můžeme sestrojit funkce *left*(n), resp. *right*(n), které dostanou kód dvojice n a vrátí její levý (resp. pravý) člen. To lze udělat různě chytře (rychle), ale ve vyčíslitelnosti stačí, že to jde. Jednou možností je:

- Můžeme vyzkoušet všechny dvojice čísel menších než zadaný kód, což jde realizovat dvěma for-cykly. Výpočet kódu může být pro obrovská čísla nutno realizovat také for-cyklem. Ve výsledku máme algoritmus složený pouze z for-cyklů, což ve vyčíslitelnosti stačí. Proto není potřeba vymýšlet sofistikovanější algoritmus.

4.3 Kódování n-tic

Právě jsme si ukázali, že pomocí Cantorova diagonálního uspořádání můžeme zakódovat dvojici čísel do jednoho čísla.

Když už umíme kódovat dvojice, není těžké vymyslet způsob, jak kódovat n-tice:

$$\langle x_1, ..., x_{n+1} \rangle_{n+1} = \langle x_1, \langle x_2, ..., x_{n+1} \rangle_n \rangle$$

Poznámka Index n u právé závorky značí, že jde o kód n-tice. Závorky bez indexu značí kód dvojice, jak jsme si ho zavedli v předchozím odstavci.

Poznámka Můžeme si též zavést funkci pro výběr j-tého z n prvků zakódovaných do jednoho čísla.

Příklady

- Chceme získat z dvojice první složku:

$$left(\langle 42, 3\rangle) = left(1038) = (\langle 42, 3\rangle)_{2,1} = (1038)_{2,1} = 42$$

- Separace čtvrté složky z pětice:

$$\langle 1, 2, 3, 4, 5\rangle_{5,4} = 4$$

4.4 O nespočetnosti

Věta: $\mathcal{P}(\mathbb{N})$ je nespočetná.

Důkaz. (sporem) Předpokládejme, že $\{A_n\}_{n\in\mathbb{N}}$ je posloupnost *všech* podmnožin $\mathbb{N}$. Zkonstruujme množinu

$$B = \{n | n \in \mathbb{N} \ \ \& \ \ n \notin A_n\}.$$

Množina B je tedy složená z indexů takových, že daný index (číslo) i neleží v množině A_i. Platí, že B[2] je podmnožinou N (a jako taková by měla být logicky zařazena v posloupnosti $\{A_n\}_{n\in\mathbb{N}}$). Zároveň však vidíme, že pro $\forall n \in \mathbb{N} : A_n \neq B$ (index n je totiž obsažen v právě jedné z množin B a A_n), a tedy $\{A_n\}_{n\in\mathbb{N}}$ nebyla posloupnost všech podmnožin $\mathbb{N}$, což je spor s predpokladem, že byla. □

5 Turingovy stroje

5.1 Definice

Turingův stroj je šestice $M = (Q, \Gamma, s, b, F, \delta)$ kde :

[2]Dobrým vodítkem pro představu o B je uvědomit si, v jak velkém počtu množin je jedno přirozené číslo.

- Q je konečná množina stavů
- Γ je konečná abeceda symbolů.
- $s \in Q$ je počáteční stav
- b (*blank*, jinak také λ) je symbol reprezentující prázdný symbol
- $F \subseteq Q$ je množina koncových stavů
- $\delta : (Q \setminus F) \times (\Gamma \cup \{\lambda\}) \to Q \times (\Gamma \cup \{\lambda\}) \times \{L, N, R\}$ přechodová funkce, kde:
  - L znamená posun čtecí hlavy vlevo
  - N znamená, že čtecí hlava zůstane na místě
  - R znamená posun čtecí hlavy vpravo

Turingův stroj Viděli jsme matematickou definici. Tato definice však vznikla z velmi názorné představy: Mějme nekonečnou pásku složenou z políček, ve kterých může být znak z Γ nebo symbol *blank*. Mějme hlavu (dokáže číst a zapisovat), která ukazuje na právě jedno políčko pásky, která si pamatuje právě jeden stav z množiny Q. Tato hlava se pomocí přechodové funkce posunuje po pásce a přitom provádí výpočet tím, že na pásku může zapisovat. Turingovu stroji tedy předložíme pásku s nějakým vstupem, TS pracuje, dokud se nedostane do koncového stavu a my si můžeme prohlédnout výsledek.

Poznámka: Turingův stroj vznikl jako reakce na otázku, kterou si Turing položil: Co je efektivní výpočet?

Poznámka: Stavy dělíme je na *aktivní* (nekoncové) a *pasivní* (koncové). V definici přechodové funkce vidíme, že stroj se v pasivním stavu zastaví, protože pro něj není přechodová funkce definována.

5.1.1 1 takt Turingova stroje

Turingův stroj může v jednom taktu provést nejvýše tři změny:

- přepsat symbol pod hlavou
- změnit stav

- provést pohyb doleva, doprava nebo setrvat na místě

5.1.2 Konfigurace Turingova stroje

Obsah nejmenší souvislé části pásky, která obsahuje:

- všechny symboly abecedy kromě λ,
- čtené pole s,
- stav stroje q

Konfiguraci můžeme zapsat takto: $UqsV$, kde U, resp. V, jsou posloupnosti symbolů abecedy nalevo, resp. napravo, od s na pásce.

5.1.3 Pojmy

Def: TS se *zastaví*, pokud z daného stavu není definována přechodová funkce.

Def: TS *přijme*, jestliže se po konečném výpočtu zastaví v přijímacím stavu.

Def: *Výpočet* TS je posloupnost konfigurací, které "jdou za sebou".

5.1.4 Zastavení Turingova stroje

Skutečnost, že Turingův stroj se zastaví nad vstupem K označujeme formálně takto: $T(K) \downarrow$. Říká se také, že stroj *konverguje* nad vstupem K. Opakem konvergence je *divergence*, která se značí $T(K) \uparrow$.

Def: *Podmíněná rovnost* dvou programů nad vstupy K_1 a K_2 je definována:

$T(K_1) \simeq T(K_2) \Leftrightarrow$ Je-li definována jedna strana, je i druhá a platí rovnost.

5.1.5 Operace nad Turingovými stroji

1. Skládání: Nechť A, B jsou TS, jejich spojením vznikne stroj AB. Předpoklady pro skládání:

- Q_A, Q_B jsou disjunktní

  [Případně přejmenujeme.]

- A má úplnou sadu instrukcí (pokud se zastaví, tak v koncovém stavu)

  [Případně zúplníme přechodovou funkci pomocí přechodů do F_A.]

Nyní pouze nahradíme všechny výskyty koncových stavů F_A stavem q_{0B} (počáteční stav TS B) a získáme tak TS AB.

Poznámka: U operace skládání nás nezajímá, zda TS A skončil v přijímajícím nebo nepřijímajícím stavu (z toho důvodu můžeme dodefinovat přechody z F_A do q_{0B}.

2. Větvení: Mějme Turingovy stroje A, B_0, B_1, schématicky se nám jedná o toto:

   $$A, B_0, B_1 \to A \begin{cases} B_0 \\ B_1 \end{cases}$$

   Předpoklady pro větvení:

   - A má dva koncové stavy q_0 a q_1.
   - A je úplný[3].
   - Q_A, Q_{B0}, Q_{B1} jsou disjunktní množiny.

Příklad: Implementace while cyklu:

- Pomocí skládání a větvení TS:

  $$T_1 T_2 \begin{cases} T_3 \\ T_4 T_5 \end{cases}$$

  V obrázku chybí šípka od T_5 na T_2!

- Totožná implementace v assembleru:

```
T1
condition:
```

[3] Nelze zúplnit!

```
    # jump to 'end' label, if T2 is not true
    JNE T2 end
while:
    T4
    # T5 represents goto statement
    goto condition
end:
    T3
```

5.2 Varianty TS

Modifikace buďto mohou být omezující (jakoby se "ubírá síla"):

- "Neposedný TS": hlava smí jen doleva či doprava, nesmí zůstat na místě
- V každém taktu lze provést jen dvě činnosti; dokonce i omezení, že provedu v jednom taktu pouze jednu činnost!
- Abeceda se zredukuje na dva symboly (lambda a nelambda) (zvýší se počet stavů)
- Jednostranná páska - lze řešit jednoduše (naivně) tak, že pokud jsem úplně nalevo pásky a stroj chce jít opět doleva, tak všechny symboly na pásce odsunu o jedno políčko doprava; lepší je řešení pomocí dvoustopé pásky.
- ...

Nebo mohou modifikace jakoby "přidávat sílu":

- Více pásek, více hlav
- Nedeterministický TS (na rozdíl od zásobníkových automatů se nedeterministický TS dá simulovat pomocí deterministického)
- ...

Modifikace, které skutečně redukují sílu Turingova stroje:

- Povolím pouze pohyby N, R – úroveň konečných automatů.
- ...

5.2.1 TS s okraji

TODO

5.2.2 Vícepáskové TS

TODO

5.3 Univerzální TS

Myšlenka: *Univerzální Turingův stroj U* umí simulovat libovolný jiný TS T nad libovolným vstupem V. Univerzální TS U dostane na vstupu kód stroje T a vstup V a vrátí stejný výsledek, jako by vrátil T na vstupu V.

Formálně: Píšeme: $U(kód(T), V) \simeq T(V)$.

Poznámka: Pokud se T na vstupu V zacyklí, musí se zacyklit i *U(kód(T), V)*.

5.3.1 Kódování

Pokud uvažujeme kanonický TS s jedinou páskou a chceme mu předat na vstup dva parametry, můžeme je oddělit na pásce nějakým speciálním symbolem.

Poznámka: Na přednášce se k tomuto účelu používá symbol "hvězdička", tedy *U(kód(T), V)* je totéž co *U(kód(T) * V)*.

Druhým technickým detailem je: *Jak zakódovat stroj T jako slovo nad nějakou abecedou?* Abychom si zjednodušili práci, předpokládáme, že stroj T:

- pracuje nad abecedou s pouze dvěma symboly (λ a |), což jak bylo popsáno v odstavci o variantách TS lze udělat bez újmy na obecnosti,
- má pouze jeden koncový stav a
- má úplnou (až na ten jeden koncový stav) přechodovou funkci.

Kódování přechodové funkce: Mějme stavy $Q_0, Q_1, ..., Q_n$ stroje T, přičemž Q_0 je onen jediný koncový stav. Přechodovou funkci δ stroje T pak můžeme

zakódovat jako posloupnost:

$$\delta(Q_1, \lambda), \delta(Q_1, |), ..., \delta(Q_n, \lambda), \delta(Q_n, |)$$

na pásku univerzálního TS. Jednotlivé členy této posloupnosti samozřejmě oddělujeme vhodnými symboly.

Každý člen posloupnosti je trojice:

- kód symbolu, který se má zapsat na pásku (λ nebo |),
- kód pohybu (L,R,N (doleva, doprava, stát)) a
- kód nového stavu.

Kódy stavů můžeme kódovat různě, ale pro popis univerzálního TS je nejjednodušší použít unární kódování: Q_j se zapíše jako j+1 čárek.

Poznámka: Přičítáme +1 proto, aby i kód stavu Q_0 měl aspoň jednu čárku, jde pouze o detail.

5.3.2 Páska univerzálního stroje

Páska UTS pak vypadá následovně (může obsahovat i jiné symboly, než vstupní λ a | ze simulovaného stroje, takže má navíc $Y, \Delta, M, \times$ a možná další):

$$Y[blok1]Y[blok2]\Delta[blok3] \times \underbrace{[O_1 \times O_2 \times O_3 \ldots O_m]}_{\text{blok4}} Y$$

Kde:

- *blok1* je obsah pásky stroje T s drobnou úpravou: právě čtený symbol (tedy ten nad nímž má simulovaný stroj hlavu) je nahrazen pomocným symbolem M.
- *blok2* kóduje aktuální stav stroje T - tedy je zde zaznamenán příslušný počet čárek (např. [||||$\lambda\lambda$] odpovídá stavu Q_4).
- *blok3* je jen jedna buňka, v níž je uložen obsah právě čteného pole (λ nebo |). Tedy přesně ten symbol, který je nahrazen symbolem M v *bloku*1.

- *blok4* je složen ze symbolů O_i, což jsou kódy přechodových funkcí z jednotlivých stavů (pro oba dva znaky na vstupu), tedy $O_i \equiv f(Q_i, \lambda)f(Q_i, |)$.

5.3.3 Jak funguje simulace T

Práce UTS pak sestává z:

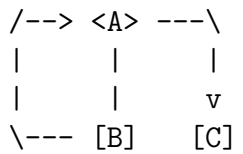
- odsimulování jednoho kroku práce původního stroje,
- otestování, zda *blok*2 obsahuje koncový stav a
- procedury vyčištění pásky a vydání výsledku.

Takže se vždycky odkrokuje jeden stav původního TS a otestuje se, zda se v *bloku*2 neobjevil koncový stav, a pokud ano, spustí se čištění a konec, jinak se pokračuje simulací dalšího kroku.

Krok původního TS se simuluje následovně:

- najde se relevantní blok O_i – stav i si nelze pamatovat přímo, proto musím z *bloku*2 postupně umazávat čárky a posouvat nějaký speciální symbol "zarážku"doprava v *bloku*4 (zřejme místo symbolu "×"si můžu dát nějaký speciální pro zarážku a ten posunovat),
- zarážka se posune na konkrétní instrukci podle *bloku*3 (čteného znaku) a pak
- se provede instrukce (po kouscích se nový stav přenese do *bloku*2, mám 6 možností co má instrukce provádět $\{\lambda, |\} \times \{L, N, R\}$; při mazání a pohybu se musí dávat pozor na okraje pásky původního stroje – pokud někde něco chybí či přebývá, celá páska původního stroje se po pásce UTS musí posouvat).

5.3.4 Schéma práce UTS



a: test: obsahuje *blok*2 pouze 1 čáru?
ANO . . . konec a vyčisti pásku
NE . . . simuluj další krok T.

5.4 Halting problem

"Lze algoritmicky rozhodnout, zda se Turingův stroj T na vstupu V zastaví, nebo zacyklí?"

Toto je asi nejznámější příklad nerozhodnutelného problému (čímž naznačuji, že odpověď na otázku výše je *ne*).

Halting problem (HP) lze formulovat různě, například i pro libovolný programovací jazyk (kód(T) pak může být např. zdrojový kód v Pascalu). My se ale budeme držet Turingových strojů. Jistě vás už nepřekvapí, že algoritmickou rozhodnutelností budeme rozumět to, zda existuje nějaký TS, který by daný problém řešil (přesněji řečeno *rozhodoval*, tedy zastavil se po konečném počtu kroků a odpověděl ANO, nebo NE).

V důkazu nerozhodnutelnosti HP se obvykle využívá univerzální Turingův stroj (UTS). Osobně si myslím, že je to spíš matoucí, protože důkaz funguje i bez UTS. Jediné co z UTS využijeme, je fakt, že každý TS (přesněji jeho přechodová funkce) lze zakódovat jako slovo nad nějakou abecedou. To je ale celkem zřejmé: jak jinak bychom asi dostali na vstup stroj T?

Věta: Halting problém není algoritmicky rozhodnutelný.

Důkaz. Pro spor předpokládejme, že existuje univerzální TS HALT takový že:

HALT(kód(T),V)=ANO právě tehdy, když se T(V) zastaví, a

HALT(kód(T),V)=NE právě tehdy, když se T(V) nezastaví.

Poznámka: Pro Matematiky: Pokud vynecháme z předpokladů existenci univerzálního TS, je potřeba říct, co je to ta funkce kód. Pokud předpokládáme UTS je funkce *kód* součástí jeho definice. Pro důkaz sporu stačí

předpokládat, že existuje TS HALT a k němu (libovolná) funkce kód z množiny všech TS do abecedy užívané TS HALT, taková, že .... a dále je již důkaz stejný.

To, že nějaký TS stroj odpoví ANO, můžeme chápat buď tak, že skončí v koncovém stavu Q_{ANO}, nebo že zapíše na pásku slovo ANO (nebo pro jednoduchost jen znak 1 jakožto kód kladné odpovědi) a zastaví se. Tyto varianty jsou na sebe snadno převeditelné, my budeme zde používat tu první.

V právě uvedeném popisu stroje HALT jaksi implicitně předpokládáme, že první parametr bude kódem nějakého stroje T. Možná přemýšlíte nad tím, co má stroj HALT udělat, když jako první parametr dostane něco, co nelze interpretovat jako TS. Něco udělat samozřejmě musí, ale pro náš důkaz je to celkem nezajímavé, protože to nebudeme nikde potřebovat. Klidně tedy můžeme předpokládat, že odpoví NE.

Protože jsme zlomyslní, tak vyrobíme TS PODRAZ tak, aby:

$$PODRAZ(V)\downarrow \iff HALT(V,V) = NE. \tag{1}$$

Co to znamená?

- Pokud HALT(V,V)=NE, tak se PODRAZ(V) zastaví.
- Pokud HALT(V,V)=ANO, tak PODRAZ(V) se (uměle) zacyklí.

Turingův stroj PODRAZ tedy vyrobíme jen malou úpravou stroje HALT: stav Q_{ANO} odebereme z koncových stavů a přidáme instrukci, která bude v tomto stavu do nekonečna cyklit.

Pointa důkazu je v tom, rozmyslet si, co by měl udělat HALT(kód(PODRAZ), kód(PODRAZ)). Platí totiž:

$$\begin{aligned} &HALT(kód(PODRAZ), kód(PODRAZ)) = ANO \\ &\iff PODRAZ(kód(PODRAZ))\downarrow \\ &\underbrace{\iff}_{(1)} HALT(kód(PODRAZ), kód(PODRAZ)) = NE. \end{aligned}$$

První ekvivalence plyne jednoduše z významu stroje HALT: je to univerzální stroj, který odpoví ANO, pokud se TS PODRAZ zastaví na vstupu $kód(PODRAZ)$, což přesně opovídá pravé straně ekvivalence.

A kýžený spor je na světě.

□

5.5 Další pojmy

- Aritmetická funkce $f : \mathbb{N}^k \to \mathbb{N}$ je *turingovsky vyčíslitelná* (TS-vyčíslitelná), pokud existuje Turingův stroj, který ji vyčísluje.
- Turingův stroj *vyčísluje* funkci f, jestliže má na vstupu:

  $\lambda \underbrace{||||||}_{n_1+1} \lambda \underbrace{||||}_{n_2+1} \lambda \ldots \lambda \underbrace{|||}_{n_k+1} \lambda$

  a na výstupu

  $\lambda \quad \underbrace{||||||}_{f(n_1,n_2,\ldots,n_k)} \quad \lambda.$

6 Primitivně a částečně rekurzivní funkce

Před definováním pojmů primitivní a částečně rekurzivní funkce je potřeba si nadefinovat pojmy, které tvoří jejich základ.

6.1 Induktivní výstavba funkcí

Funkce budeme konstruovat induktivně: zavedeme si tři základní funkce

- *nulovou* funkci: $\mathbf{o}(x) = 0 \quad (\forall x)$
- funkci *následník*: $\mathbf{s}(x) = x + 1 \quad (\forall x)$
- funkci *vydělení j-té složky*: $\mathbf{I^j_n}(x_1, ..., x_n) = x_j \quad (\forall j, n : 1 \leq j \leq n)$

a tři operátory:

- Operátor *substituce*: $S^m_n(\mathbf{f}, \mathbf{g_1}, ..., \mathbf{g_m}) = \mathbf{h}$, kde

  $\mathbf{h}(x_1, ..., x_n) \simeq \mathbf{f}(\mathbf{g_1}(x_1, ..., x_n), ..., \mathbf{g_m}(x_1, ..., x_n))$.

  Poznámka:

  - Z uvedené definice je zřejmé, že $\mathbf{f}$ je funkce m proměnných a všechny $\mathbf{g_i}$ jsou funkce n proměnných.
  - Pokud zavedeme konvenci, že $\vec{x} = (x_1, ..., x_n)$, tak můžeme zjednodušeně psát $\mathbf{h}(\vec{x}) \simeq \mathbf{f}(\mathbf{g_1}(\vec{x}), ..., \mathbf{g_m}(\vec{x}))$.
  - Operátor substituce je základní prvek programování (snad ve všech známých programovacích jazycích) – jednoduše úlohu, kterou mám řešit, vyřeším pomocí funkcí, které už naprogramoval někdo dřív.

- operátor *primitivní rekurze*: $R_n(\mathbf{f}, \mathbf{g}) = \mathbf{h}$, kde:
  - $\mathbf{h}(0, x_2, ..., x_n) \simeq \mathbf{f}(x_2, ..., x_n)$
  - $\mathbf{h}(i+1, x_2, ..., x_n) \simeq \mathbf{g}(i, \mathbf{h}(i, x_2, ..., x_n), x_2, ..., x_n)$

  Poznámka:
  - Z uvedené definice je zřejmé, že $\mathbf{f}$ je funkce n-1 proměnných a $\mathbf{g}$ je funkce n+1 proměnných.
  - Je vidět, že operátor primitivní rekurze má sílu for-cyklu.
  - Proměnná x_1 má zvláštní význam – slouží jako čítač.
  - Na přednášce se uváděl vysvětlující příklad na primitivní rekurzi pro n=1. Pak h(0)=konstanta a h(i+1)=g(i,h(i)). Malý problém je v tom, že jsme si nedefinovali nulární funkce a f by v tomto případě musela mít n-1=0 proměnných.

- operátor *minimalizace*: $M_n(\mathbf{f}) = \mathbf{h}$, kde

  $$\mathbf{h}(x_1, ..., x_n) \downarrow = z \iff \Big(\mathbf{f}(x_1, ..., x_n, z) \downarrow = 0 \ \ \& \ \ \forall j < z : \mathbf{f}(x_1, ..., x_n, j) \downarrow \neq 0\Big)$$

  Poznámka:
  - Z uvedené definice je zřejmé, že $\mathbf{f}$ je funkce n+1 proměnných.
  - Je vidět, že operátor minimalizace má sílu while-cyklu.
  - Poslední proměnná ve funkci f (v pseudokódu označená jako i) má zvláštní význam. Snažíme se ji minimalizovat, to jest najít nejmenší i takové, aby f vracela nulu.

– Pokud se jako f předá funkce, která nikdy nevrací nulu, tak operátor minimalizace vytvoří funkci, která nebude nikde definovaná (výše uvedený pseudokód se zacyklí). To se u předchozích operátorů stát nemohlo: pokud dostaly na vstupu totální funkce, vrátily také totální funkci.

– Pokud nechceme zavádět nulární funkce, je nutné definici operátoru minimalizace doplnit o podmínku, že f je funkce alespoň dvou proměnných.

– Místo $\downarrow=$ (zkratka za konverguje a rovná se) bychom v definici klidně mohli psát jen $=$. Použití $\downarrow\neq$ (zkratka za konverguje a nerovná se) je ale důležité, například pokud $\mathbf{f}(x_1, ..., x_n, 0) \uparrow$, tak $\mathbf{h}(x_1, ..., x_n)$ také není definováno, i kdyby třeba $\mathbf{f}(x_1, ..., x_n, 1) = 0$.

Značení:

– Pomocí symbolu μ_y značíme while cyklus přes iterační proměnnou y a myslíme tím:

$$M_n(f) = h(\dots) \simeq \mu_y(f(\dots, y) \simeq 0).$$

– Symbol μ_y je možné použít i zobecněně takto:

$$M_n(f) = h(\dots) \simeq \mu_y(< \text{pomínka} >),$$

což odpovídá nalezení takového (nejmenšího) y, že <podmínka> je splněna a $\forall j < y$ je <podmínka> definováno, ale není splněno.

Poznámka: Operátor je jakási *"metafunkce"* – na vstupu dostane jednu či více funkcí a na výstupu vrátí nějakou novou funkci. Všechny tři operátory mají v dolním indexu n, což značí, že výsledná funkce $\mathbf{h}$ bude funkcí n proměnných.

Def: *Odvození funkce f* je posloupnost funkcí $f_0 \dots f_m = f$, kde každá funkce je buď základní nebo vznikne použitím operátoru na předchozí funkce.

Pozorování: Základní funkce jsou totální funkce (= jsou všude definované) a jsou také efektivně vyčíslitelné[4].

[4] O efektivitu ve skutečnosti nejde, podstatné je, že funkce jdou vůbec vyčíslit a to ve Vyčíslitelnosti stačí.

Poznámka: Pokud vám induktivní výstavba funkcí připomíná induktivní výstavbu důkazů v logice, tak si můžete připsat malé nevýznamné plus. Ano, základní funkce jsou paralelou k axiomům, operátory jsou paralelou k odvozovacím pravidlům (modus ponens,...) a odvození funkce je paralelou k odvození důkazu.

Poznámka: Pokud vám induktivní výstavba funkcí připomíná funkcionální programování (nebo dokonce lambda kalkulus), tak si můžete připsat větší (leč stále nevýznamné) plus. Ano, principem je funkcionální programování (ale bez určené implementace).

Formálně: Ve vyčíslitelnosti se často volně zaměňují termíny *základní funkce* a *schéma základní funkce* (podobně jako v logice se zaměňuje axiom a schéma axiomu). Nulová funkce i následník jsou základní funkce. Ale vydělení je schéma, které definuje nekonečně mnoho základních funkcí.

6.2 Definice rekurzivních funkcí

Def: Funkce je *částečně rekurzivní funkce* (ČRF), pokud ji lze získat pomocí uvedených tří základních funkcí a tří operátorů.

Def: Funkce je *obecně rekurzivní funkce* (ORF), pokud je ČRF a totální (všude definovaná).

Def: Funkce je *primitivně rekurzivní funkce* (PRF), pokud ji lze získat pomocí uvedených tří základních funkcí a operátorů substituce a primitivní rekurze. Nesmí se tedy použít operátor minimalizace.

V uvedených definicích chápeme ČRF (resp. ORF a PRF) jako vlastnost. Můžeme se na věc dívat algebraicky a ekvivalentně definovat ČRF (resp. ORF a PRF) jako třídu funkcí, které mají tuto vlastnost:

- *ČRF* je nejmenší třída funkcí, která obsahuje tři základní funkce a je uzavřená na S_n^m, R_n, M_n.
- *ORF* je nejmenší třída totálních funkcí, která obsahuje tři základní funkce a je uzavřená na S_n^m, R_n, M_n.
- *PRF* je nejmenší třída funkcí, která obsahuje tři základní funkce a je uzavřená na S_n^m, R_n.

Def: Množina M je *rekurzivní* jestliže její charakteristická funkce je ORF.

$$C_M(x) = \begin{cases} 1 & x \in M \\ 0 & x \notin M \end{cases}$$

Poznámka: Množina M je rekurzivní, jestliže je algoritmicky rozhodnutelná, to znamená, že existuje "program", který pro libovolný vstup x se zastaví a rozhodne $x \in M$, nebo $x \notin M$.

Def: Predikát P je *primitivně rekurzivní* (PR), resp. *obecně rekurzivní* (OR), jestliže jeho charakteristická funkce je PRF (resp. ORF).

Poznámka: PR (resp. OR) predikát P je algoritmicky rozhodnutelný primitivně rekurzivní funkcí (resp. obecně rekurzivní funkcí).

Def: Množina M je *rekurzivně spočetná*, jestliže M je definičním oborem nějaké ČRF.

Poznámka: M je rekurzivně spočetná, jestliže existuje „program“ Pr takový, že:

$$x \in M \Leftrightarrow Pr(x) \downarrow$$

Def: Predikát P je *rekurzivně spočetný*, jestliže P je definičním oborem nějaké ČRF.

👁 **Pozorování:** Rekurzivita $\subseteq$ rekurzivní spočetnosti.

Důkaz. If $\underbrace{x \in M}_{\text{procedure}}$ then HALT else NEHALT. □

6.3 [*] Vlastnosti rekurzivních funkcí

- Ke každé ČRF (i ORF a PRF) funkci existuje její odvození. Toto odvození lze chápat jako program vyčíslující danou funkci. Například funkce přičtení dvojky má odvození: $S_1^1(\mathbf{s}, \mathbf{s})$. Je dobré si uvědomit, že jedna funkce může mít více programů. To znamená, že ty programy jsou různé (mají jiný zdroják = odvození), ale dělají to samé. Ve skutečnosti má každá ČRF nekonečně mnoho odvození (čili programů). Například funkci přičtení jedničky můžeme snadno odvodit jako $\mathbf{s}$, ale můžeme si odvození (uměle) prodloužit: $S_1^1(\mathbf{s}, \mathbf{I_1^1})$, nebo dokonce $S_1^1\Big(\mathbf{s}, S_1^1(\mathbf{I_1^1}, \mathbf{I_1^1})\Big)$ atd.
- Částečně rekurzivních funkcí je spočetně mnoho (jak si za chvíli ukážeme), a klidně by se místo *třída* částečně rekurzivních funkcí mohlo

říkat *množina* částečně rekurzivních funkcí. Jenže ve vyčíslitelnosti je zvykem, že množinou se vždy myslí nějaká podmnožina přirozených čísel, a tak když chtějí mluvit o množině něčeho jiného, tak to radši i nazvou jinak (třída, systém,...). Obdobně pro ORF a PRF.

- Pro zápisky z vyčíslitelnosti se mi osvědčila následující konvence. Místo "Nechť f je částečně rekurzivní funkce 3 proměnných"psát $f \in \text{ČRF}_3$. Na přednášce se sice nepoužívá, ale je to praktické.

6.4 Programování v ČRF

Z "programování"v ČRF se sice nezkouší, ale přesto doporučuji si následující příklady vyzkoušet odvodit. Člověk pak získá konkrétnější představu o tom, jak je možné s tak příšerným aparátem jako jsou ČRF něco naprogramovat, a to dokonce cokoli, co lze naprogramovat v libovolném jiném programovacím jazyce.

6.4.1 Sčítání

Odvození funkce, která sečte dvě čísla: $R_2(I_1^1, S_3^1(s, I_3^2))$

Příklad: Pro $x_1 = 2 \;\; x_2 = 3$

$$\begin{aligned} h(0, x_2) &= I_1^1(x_2) = I_1^1(3) = 3 \\ h(1, x_2) &= s(I_3^2(0, h(0, x_2), x_2)) = s(I_3^2(0, 3, 3)) = s(3) = 4 \\ h(2, x_2) &= s(I_3^2(1, h(1, x_2), x_2)) = s(I_3^2(1, 4, 3)) = s(4) = 5 \end{aligned}$$

6.4.2 Odčítání

Ve vyčíslitelnosti nemáme záporná čísla, a proto si musíme definovat *aritmetické mínus* (značí se symbolem „$\dot{-}$“):

$$x \dot{-} y = \begin{cases} x - y & x \geq y \\ 0 & x < y \end{cases}$$

$$sgn(x) = \begin{cases} 0 & x = 0 \\ 1 & x \geq 1 \end{cases}, \quad \bar{sgn}(x) = \begin{cases} 0 & x \geq 1 \\ 1 & x = 0 \end{cases}$$

Operace $1 \dot{-} x$ se často používá v podstatě ve významu negace x: pokud totiž bereme 0=false a nenula=true, tak je to totéž.

Odvození:

$$S_2^2(R_2(I_1^1, S_3^1(S_1^2(R_2(o, I_3^1), I_1^1, I_1^1), I_3^2)), I_2^2, I_2^1)$$

6.4.3 Násobení

Odvození:

$$R_2(o, S_3^2(R_2(I_1^1, S_3^1(s, I_3^2)), I_3^2, I_3^3))$$

Příklad: Ukázka výpočtu pro $x_1 = 2$ a $x_2 = 3$

```
R_2(o,S^2_3(R_2(I^1_1,S^1_3(s,I^2_3)),I^2_3,I^3_3))(3,2)
..R_2(o,S^2_3(R_2(I^1_1,S^1_3(s,I^2_3)),I^2_3,I^3_3))(2,2)
....R_2(o,S^2_3(R_2(I^1_1,S^1_3(s,I^2_3)),I^2_3,I^3_3))(1,2)
......R_2(o,S^2_3(R_2(I^1_1,S^1_3(s,I^2_3)),I^2_3,I^3_3))(0,2)
........o(2)
........= 0
......= 0
......S^2_3(R_2(I^1_1,S^1_3(s,I^2_3)),I^2_3,I^3_3)(0,0,2)
........I^2_3(0,0,2)
........= 0
........I^3_3(0,0,2)
........= 2
........R_2(I^1_1,S^1_3(s,I^2_3))(0,2)
..........I^1_1(2)
..........= 2
........= 2
......= 2
....= 2
....S^2_3(R_2(I^1_1,S^1_3(s,I^2_3)),I^2_3,I^3_3)(1,2,2)
......I^2_3(1,2,2)
......= 2
......I^3_3(1,2,2)
......= 2
......R_2(I^1_1,S^1_3(s,I^2_3))(2,2)
........R_2(I^1_1,S^1_3(s,I^2_3))(1,2)
..........R_2(I^1_1,S^1_3(s,I^2_3))(0,2)
```

```
............I^1_1(2)
............= 2
..........= 2
..........S^1_3(s,I^2_3)(0,2,2)
............I^2_3(0,2,2)
............= 2
............s(2)
............= 3
..........= 3
........= 3
........S^1_3(s,I^2_3)(1,3,2)
..........I^2_3(1,3,2)
..........= 3
..........s(3)
..........= 4
........= 4
......= 4
....= 4
..= 4
..S^2_3(R_2(I^1_1,S^1_3(s,I^2_3)),I^2_3,I^3_3)(2,4,2)
....I^2_3(2,4,2)
....= 4
....I^3_3(2,4,2)
....= 2
....R_2(I^1_1,S^1_3(s,I^2_3))(4,2)
......R_2(I^1_1,S^1_3(s,I^2_3))(3,2)
........R_2(I^1_1,S^1_3(s,I^2_3))(2,2)
..........R_2(I^1_1,S^1_3(s,I^2_3))(1,2)
............R_2(I^1_1,S^1_3(s,I^2_3))(0,2)
..............I^1_1(2)
..............= 2
............= 2
............S^1_3(s,I^2_3)(0,2,2)
..............I^2_3(0,2,2)
..............= 2
..............s(2)
..............= 3
............= 3
```

```
..........= 3
..........S^1_3(s,I^2_3)(1,3,2)
............I^2_3(1,3,2)
............= 3
............s(3)
............= 4
..........= 4
........= 4
........S^1_3(s,I^2_3)(2,4,2)
..........I^2_3(2,4,2)
..........= 4
..........s(4)
..........= 5
........= 5
......= 5
......S^1_3(s,I^2_3)(3,5,2)
........I^2_3(3,5,2)
........= 5
........s(5)
........= 6
......= 6
....= 6
..= 6
= 6
```

6.5 Vlastnosti

Fakt: PRF jsou všude definované.

Důkaz. Indukcí. □

👁 **Pozorování:** Všechny ČRF jsou "efektivně vyčíslitelné"[5] [6].

Důkaz. Indukcí (potřebujeme všechny programy pro ČRF (odvození, ČR-term)):

[5] Efektivně vyčíslitelné je to samé jako Turingovsky vyčíslitelné, Pascal-like, ...
[6] Později uvidíme, že: Každá "efektivně vyčíslitelná"funkce je ČRF

- $o(x), s(x), I_n^j$ ... ČR-termy, které jsou efektivně vyčíslitelné.
- Operátor substituce S_n^m ... triviální, jde o volání funkcí
- Operátor primitivní rekurze R_n ... $h(0) \simeq$ konst., $h(y+1) \simeq g(y, h(y))$ ... odpovídá FOR cyklu.
- Operátor minimalizace M_n: $\mu_y(h(\ldots, y) \simeq 0)$ odpovídá WHILE cyklu.

□

Lemma. *Spojky výrokového počtu (& $, \rightarrow, \neg, \vee$) zachovávají u predikátů primitivní rekurzivitu, resp. obecnou rekurzivitu (totéž u množin s operacemi: doplněk,$\cup$, $\cap$).*

Důkaz. Závisí na námi zvoleném programovacím jazyku - triviální:

$$\begin{array}{rr} \neg P & \bar{sg}(C_P) \\ P \ \& \ Q & C_P \dot{C}_Q \\ P \vee Q & sg(C_P + C_Q) \end{array}$$

□

Věta: PRF $\subsetneq$ ORF $\subsetneq$ ČRF

Důkaz.

1. Je zřejmé, že každá ORF je ČRF a že i každá PRF je ČRF. Základní funkce jsou totální a operátory substituce a primitivní rekurze totálnost zachovávají, PRF tedy musí být totální, tedy ORF. Tím jsme dokázali, že platí:
$$PRF \subseteq ORF \subseteq \text{ČRF}.$$

2. Nyní dokazujeme ORF $\subsetneq$ ČRF, hledáme tedy ČRF, která není totální.

   Zkonstruujeme tedy například funkci f, která není nikde definovaná (tzv. prázdná funkce). K tomu budeme muset zjevně použít operátor minimalizace:
$$f := M(g),$$

kde g je nějaká funkce, která nikdy nevrací nulu. Nejjednodušší by bylo vzít jako g rovnou funkci následníka, tedy

$$f := M_0(\mathbf{s}).$$

Problém je, že jsme si nezavedli nulární funkce (a tou by f byla). Zvolme tedy

$$g := S_2^1(\mathbf{s}, \mathbf{I_2^2}),$$

což odpovídá funkci $g(x, y) = y + 1$, u které tedy na první proměnné nezáleží. Pak máme

$$f := M_1(g) \simeq \mu_y(g(x, y) \simeq 0),$$

která zřejmě je ČRF a není ORF.

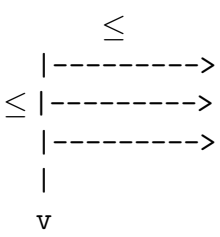
3. Nyní se snažíme dokázat, že PRF $\subsetneq$ ORF. Příkladem takové funkce je *Ackermannova funkce*[7]:

$$\begin{aligned} A(0, x) &= \begin{cases} 1 \ldots x = 0 \\ 2 \ldots x = 1 \\ x + 2 \ldots x >= 2 \end{cases} \\ A(x, 0) &= 1 \\ A(x + 1, y + 1) &= A(x, A(x + 1, y)) \end{aligned}$$

- A je ORF: *(myšlenka)*

  (a) Je A totální (všude definovaná)?

  Předpis určuje jednoznačně definovanou funkci podle transfinitní indukce ω^2 (uspořádání dvojic).



[7] Případně také univerzální funkce pro třídu PRF - TODO!!

(b) Je A je efektivně vyčíslitelná?

Zastaví se program na vstupních datech?[8] Zobrazíme vstupní data na dobré uspořádání (tj. z každé podmnožiny lze vybrat nejmenší prvek) a v průběhu programu klesáme, v dobrém uspořádání neexistuje nekonečný klesající řetězec a tedy program vždy skončí.

(c) Je A ČRF?

Stačí najít μ_y, kde y je je kód souvislé oblasti v matici s hodnotami funkce A, která stačí na výpočet požadovaného vstupu. TODO

- A není PRF:

Myšlenka: Základní myšlenkou je měření hloubky do sebe zanořených for-cyklů.

Def: *Hloubka PR-termů* je definována jako maximální počet do sebe zanořených for-cyklů (operátorů rekurze) takto:

(a) Pro základní funkce definujeme:

$$hl(s) = hl(\sigma) = hl(I_n^d) = 0.$$

(b) Pro operátor substituce definujeme[9]:

$$hl(S_n^m(P, Q_1, ..., Q_n)) = max(hl(P), hl(Q_1), ..., hl(Q_n))..$$

(c) Pro operátor primitivní rekurze definujeme:

$$hl(R_n(P, Q)) = max(hl(P), hl(Q) + 1)\,.$$

Def: Definujeme třídu funkcí R_i jako třídu obsahující všechny PRF, pro které existuje odvození hloubky nejvýše i.

Poznámka: Platí, že PRF $= \bigcup_i R_i$, tedy můžeme klasifikovat PRF funkci podle složitosti PR-programu.

Def: Označme

$$f_y(x) = A(y, x) \ldots \text{y-tý řádek},$$

[8] TODO

[9] Nejspíše se jedná skutečně o syntaktickou strukturu termu a ne o význam - kdyby šlo o skutečný způsob výpočtu, pak by bylo intuitivnější $max(hl(Q_1)+hl(P), ..., hl(Q_n)+hl(P))$

pak

$$
\begin{aligned}
f_0(x) &= x + 2 && x \geq 2 \\
f_1(x) &= 2 \cdot x && x \geq 1 \\
f_2(x) &= 2^x && x \geq 0 \\
f_3(x) &= \underbrace{2^{2^{2^{\cdots}}}}_{\text{x-krát umocňuji na druhou}}
\end{aligned}
$$

Fakt: Existuje $f_i \in R_i \setminus R_{i-1}$ pro $i \leq 1$. Tedy existuje program, který dokážeme naprogramovat i cykly, ale už to nelze pomocí $i - 1$ cyklů.

Věta: $A(x, y)$ a $A(x, x)$ nejsou PRF, dokonce pro každou PRF funkci α existuje $x_0 : \forall x \geq x_0 : \alpha(x) < A(x, x)$.[10]

Důkaz. Náznak, bez detailů

(a) A roste v obou proměnných kromě konečně mnoha hodnot ($f_x(y)$ roste ...)

(b) *(Bez důkazu)* $f_{i+1}(x) \overset{\text{def}}{=} A(i + 1, x)$ skoro všude majorizuje libovolnou funkci[11] z R_i, tedy platí:

$$\forall g \in R_i \, \exists x_0 : \forall x > x_0 : g(x) < f_{i+1}(x).$$

Sporem (méně formálně):

$$
\begin{aligned}
A \in PRF \Rightarrow A(x, x) \in PRF && \text{Předpoklad pro spor} \\
A \in PRF \Rightarrow f_x(x) \in PRF && \text{Ekvivalentní zápis}
\end{aligned}
$$

Dle faktu nad touto větou platí $f_i \in R_i \setminus R_{i-1}$, $f_x(x)$ tedy f_i můžeme vyčíslit pouze pomocí x do sebe zanořených for-cyklů, takže potřebujeme hloubku $1, 2, 3, \ldots, \infty$ - SPOR, předpokládali jsme, že A může mít jen konečný počet for-cyklů.

Sporem (formálněji):

[10] To tedy znamená, že $A(x, x)$ roste rychleji než libovolná PRF.

[11] Pouze formálně říkáme, že: „Síla $(i + 1)$ do sebe vnořených for-cyklů je větší než síla i for-cyklů."

Pro spor předpokládejme:

$$A(x,x) \in PRF \Rightarrow \exists i : A(x,x) \in R_i.$$

Pak platí:

$$A(x,x) \overset{(b)}{<} f_{i+1}(x) \overset{(a)}{<} f_x(x)$$

pro $x > i+1$ (rostoucí) a pro $x > x_0$ (až na konečně mnoho bodů)

□

□

7 Výpočetní síla TS a ČRF je ekvivalentní

7.1 ČRF => TS

Chceme ukázat, že každá ČRF je turingovsky vyčíslitelná[12]. Jelikož ČRF jsou definované induktivně, tak i důkaz provedeme indukcí podle konstrukce ČRF:

- Základní funkce (↑) jsou jistě Turingovsky vyčíslitelné.
- Operátory zachovávají Turingovskou vyčíslitelnost:
  - Operátor substituce ($S_n^m(\mathbf{f}, \mathbf{g_1}, ..., \mathbf{g_m})$): Díky indukčnímu předpokladu vyčíslím podfunkce g_i (pomocí m pásek), dám výsledky dohromady a vyčíslím na tomto vstupu f.
  - Primitivní rekurze ($R_n(\mathbf{f}, \mathbf{g})$): Vyhodnotím f a pak y-krát „otočím“ g za použití vstupních parametrů, počítadla cyklů a výsledku z predcházejícího výpočtu.
  - Minimalizace ($M_n(\mathbf{f})$): Opakovaně vyčísluji f na vstupních parametrech a zvětšujícím se počítadle cyklů. Když f poprvé vrátí 0 skončím a vrátím hodnotu počítadla.

[12] Fixujeme kódování přirozených čísel jako k-tic na pásku. *Turingovská vyčíslitelnost* funkce $\varphi : \mathbb{N}^k \to \mathbb{N}$ znamená, že existuje Turingův stroj T, který při zvoleném kódování vyčísluje φ.

7.2 TS => ČRF

1. *Základním trikem je, že 1 krok práce TS je PR-záležitost.*

   Výpočet TS lze chápat jako posloupnost konfigurací (obsah pásky, stav řídící jednotky, pozice hlavy). Tyto konfigurace jde zakódovat[13], kódujeme tzv. „Postova slova“ $UqsV$ do $\mathbb{N}$:

   $$UqsV \to n \in \mathbb{N}$$

   a výpočet TS lze chápat jako posloupnost těchto kódů.

   Jeden krok TS představuje pouze lokální změnu v Postově slově. Jistě tedy existuje funkce na kódech, která ji charakterizuje. Tato funkce vlastně obsahuje jenom rozhodovací strom odpovídající přechodové funkci našeho TS a na to nám stačí prostředky PRF (na if podmínku použijeme operátor primitivní rekurze).

2. *TS pracuje, dokud neskončí ... while cyklus*

   Pomocí for cyklu (primitivní rekurze) pak můžeme sestavit funkci $Comp(x, s)$, která vypočte kód stavu stroje startujícího ze stavu odpovídajícímu kódu x po s krocích. Pak už jen pomocí while cyklu přes to celé nalezneme posloupnost kroků do konečného stavu, tj.

   $$\mu_s(comp(x, s) \ \ \& \quad \text{za s kroků skončím}).$$

   Formálně bychom mohli psát, že pro každý turingův stroj TS existuje ČRF g taková, že pro libovolnou konfiguraci k platí $kod(TS(k)) \simeq g(kod(k))$.

8 Numerace ČRF

Každá ČRF má z definice nějaké své odvození a každé takové odvození lze převést na přirozené číslo (a zpět, tedy převod je injektivní = prostý). Způsobů, jak tento převod počítat je mnoho, například bychom mohli vzít po-

[13] Nemůžeme použít induktivní přístup podobně jako v předchozí implikaci.

sloupnost ASCII kódů odvození a chápat to jako jedno číslo, nejčastěji se však používá standardní numerace definovaná v Kleenově větě.

- Převodu se říká *numerace* a výslednému číslu *kód funkce* nebo *číslo programu* a vetšinou ho budeme značit jako e.
- Jen tak na okraj: Podobně jako ČRF, jdou i logické formule (RSP) převádět na čísla, převodu se říká gödelizace a výsledku gödelova čísla.
- Jedna funkce může mít víc odvození a tedy i víc kódů. Různé funkce mají ale zaručeně různá odvození a tedy i kódy.
- Důsledkem je, že množina ČRF je spočetná.

9 Kleeneova věta

Toto je jedna z nejdůležitějších vět vyčíslitelnosti, protože většina ostatních je na ní založená.

Věta: *Kleeneova věta o normální formě a o univerzální funkci:*

Pro každé $k \geq 1$ (libovolný počet proměnných):

- *Univerzální ČR funkce k+1 proměnných pro ČR funkce k proměnných:*

  Existuje ČRF k+1 proměnných $\Psi_k(e, x_1, ..., x_k)$ taková, že Ψ_k je *univerzální funkcí* pro třídu funkcí k proměnných, tedy *vyčísluje* e-tou takovou funkci.

- *Normalizace:*

  Jde efektivně získat numerace e z odvození ČRF a naopak z e lze získat efektivně odvození ČRF.

- *Souvislost s Turingovým strojem:*

  Existuje PRF jedné proměnné U a PRP $k+2$ proměnných T_k[14] takové, že

[14] Kleeneho T predikát nebo *Turingův predikát*

$$\Psi_k(e, x_1, ..., x_k) \simeq U(\mu_y(T_k(e, x_1, x_2, ..., x_k, y))).$$

Interpretace U a dalších symbolů je umístěna až v důkaze.

- *s-m-n věta:*

  Existuje PRF $m+1$ proměnných s_m prostá a rostoucí ve všech proměnných (*fixace prvních m proměnných*) taková, že

$$\Psi_{m+n}(e, y_1, ..., y_m, x_1, ..., x_n) \simeq \Psi_n(s_m(e, y_1, ..., y_m), x_1, ..., x_n)$$

- *(Univerzální funkce je standardní:*

$$T_k(e, x_1, ..., x_k, y) \quad \& \quad T_k(e, x_1, ..., x_k, z) \Rightarrow y = z$$

Důkaz. (Náznak)

Cože zase mám něco dokazovat? A on už někdo dokázal existenci univerzálního Turingova stroje, že? A on už někdo dokázal, že TS má stejnou sílu jako ČRF, že? Tak to by pro ty dva už neměl být takový problém dokázat tohle, ne?[15]

Co znamená univerzální funkce?

Obecně mějme třídu $\mathcal{A}$ funkcí n proměnných, které mají nějakou vlastnost (např. jsou to PRF, ČRF,...). *Univerzální funkce* pro třídu $\mathcal{A}$ je funkce *univ*, která dostane jako vstup číslo e a parametry $x_1, ..., x_n$ a vrátí to samé, co by vrátila e-tá funkce z třídy $\mathcal{A}$[16]. Důležité je, že takto dostaneme všechny funkce z té třídy, tedy $\forall f \in \mathcal{A} \quad \exists e \in \mathbb{N} : f(x_1, ..., x_n) \simeq univ(e, x_1, ..., x_n)$. To mimochodem znamená, že třída funkcí $\mathcal{A}$ musí být spočetná.

Kleenova věta říká důležitou informaci: Existuje *univerzální funkce pro ČRF k-proměnných*, kterou budeme nazývat Ψ_k. Tato univerzální funkce je navíc sama také ČRF, z čehož plynou zajímavé důsledky.

Co znamená predikát T?

[15] Jestli čekáte lepší důkaz než tento monolog, tak hledejte, ale o moc exaktnější to asi nebude a rozhodně to nebude elegantnější. Celý důkaz se nedělá ani na přednášce. Důležitější než přesný důkaz je ale pochopit, co znamenají jednotlivé funkce a predikát T.

[16] Pokud $f(x_1, ..., x_n) \uparrow$, pak i $univ(e, x_1, ..., x_n) \uparrow$. To, která funkce je vlastně ta e-tá, je definováno právě pomocí univerzální funkce.

Kleeneho predikát $T_1(e, x, y)$ platí, právě když $y =< y_0, y_1 >$ (dvojice zakódovaná jako jedno číslo) a e je kód funkce jedné proměnné, která se při simulaci na TS nad vstupem x zastaví (přesně) po y_0 krocích s výsledkem y_1. Obecně můžeme zavést predikát $T_k(e, x_1, ..., x_k, y)$ pro funkce k proměnných.

- Funkce odpovídající predikátu T_k je primitivně rekurzivní. Přesný důkaz je pracný, ale základní myšlenkou je simulace výpočtu TS, přičemž víme, kolik nejvýše kroků je třeba simulovat, takže nám stačí for-cyklus a nepotřebujeme while.
- Pro konkrétní TS a zadaný vstup x buďto existuje právě jedna dvojice $< y_0, y_1 >$ taková, že po y_0 krocích se stroj zastaví s výsledkem y_1, a nebo neexistuje žádná taková dvojice, pokud se stroj nikdy nezastaví. (Tím jsme dokázali, že je univerzální funkce standardní.)

(To ovšem není součástí znění věty, ani to z ní nějak neplyne, pouze to může posloužit jejímu lepšímu pochopení. Jiná možná definice tohoto predikátu je, že y kóduje posloupnost všech konfigurací výpočtu TS nad vstupem x až do jeho zastavení. Pak se odpovídajícím způsobem změní i definice U.)

Co znamená μ_y?

$\mu_y T_1(e, x, y)$ je funkce dvou proměnných (e a x), která vrátí nejmenší y takové, aby platil predikát $T_1(e, x, y)$. Když už máme sestrojen predikát T (včetně odvození), tak tuto funkci vytvoříme jednoduše pomocí operátoru minimalizace.

Co znamená funkce U?

U je pouze funkce, která vydělí druhou složku z kódu dvojice[17]. Speciálně funkce $U\Big(\mu_y T_1(e, x, y)\Big)$ vrátí výsledek e-tého programu (e-té ČRFunkce) jedné proměnné spuštěného na vstupu x. Protože to je hodně významná funkce, tzv. *univerzální pro ČRF 1 proměnné*, zavedeme si pro ni značení $\Psi_1(e, x)$, obdobně $\Psi_k(e, x_1, ..., x_k)$ pro k proměnných. A aby každý věděl, jak moc významná ta funkce je, tak pro ni zavedeme ještě další značení, φ_e, ve kterém se pro stručnost explicitně neuvádí k, ale pozná se to z počtu parametrů, protože $\Psi_k(e, x_1, ..., x_k) \equiv \varphi_e(x_1, ..., x_k)$.

Co říká s-m-n věta?

[17] Mimochodem při zavádění kódování dvojic (↑) jsme si přesně tuto funkci nazvali *right*, ale dnes je v módě U

Máme číslo e, tedy e-tý program pro $m+n$ proměnných. Rozhodnu se prvních m proměnných zafixovat na hodnotách $y_1, ..., y_m$ a vytvořit tak novou funkci (program) n proměnných[18]. Schematicky zapsáno:

```
nova_funkce(x_1,...,x_n){
  return program_s_cislem_e(y_1, ..., y_m, x_1, ..., x_n);
}
```

S-m-n věta říká, že můžu snadno zjistit číslo této nové funkce (programu). Existuje totiž sada PRFunkcí $s_1, s_2, s_3, ...$ a my použijeme konkrétně $s_m(e, y_1, ..., y_m)$ a získáme tak číslo té nové funkce.

□

[18] Ve funkcionálním programování se toto označuje jako curryfikace