Kapitola: Primitívna rekurzia

1.1 Motivácia

Matematických funkcií je nespočítateľne veľa. Dokonca už unárnych funkcií na $\mathbb{N}$ je nespočítateľne veľa. (Ľahký dôkaz úpravou Cantorovej diagonalizácie.) Tým pádom ale väčšina z nich pre nás nie je zaujímavá – nikdy nikde takéto funkcie nestretneme, lebo ich nevieme nijak rozumne popísať. (Konečných popisov je nutne len spočítateľne veľa.) Istá zaujímavosť začína až u funkcií, ktoré aspoň vieme exaktne definovať.

Samotná definícia však ešte nemusí byť všetko. Uvažujme napríklad funkciu:

$$\mathit{fives}(x) = \begin{cases} 1 & \leftarrow \text{v desatinnom rozvoji } \pi \text{ sa niekde nachádza } x \text{ pätiek za sebou} \\ 0 & \leftarrow \text{inak} \end{cases}$$

Toto je síce exaktná definícia, jasne popisujúca práve jednu funkciu – je nám ale nanič. Nehovorí nám v podstate nič o tom, ako túto funkciu vypočítať. Je dokonca otvorený problém či je funkcia *fives* identicky rovná jednej.

Všimnite si tiež rozdiel medzi „(na základe definície) nevieme funkciu vyhodnotiť“ a „neexistuje algoritmus ktorý by túto funkciu vyhodnocoval“. To prvé je v tomto prípade (aspoň zatiaľ) pravda, to druhé však nie. Totiž určite existuje triviálny program, ktorý funkciu *fives* ráta. My len nevieme, či je to program:

```
return 1;
```

alebo jeden z programov tvaru:

```
if (vstup < K) then return 1 else return 0;
```

Pekné funkcie by teda boli také, u ktorých je definícia priamo „návodom“ na ich počítanie.

Ako takéto definície robiť systematicky? Pre úvod: Ako ich robiť systematicky pre funkcie na $\mathbb{N}$?

Pekná funkcia je napríklad $f(x, y) = x^y$. Ako to vyzerá, keď ju vyhodnocujeme? Hodnotu x^y dostaneme tak, že zoberieme 1ku a y-krát ju vynásobíme hodnotou x.

Čo znamená „vynásobíme hodnotu a hodnotou b“? Zoberieme nulu a b-krát k nej pripočítame a.

A čo znamená „pripočítame“?

Takto sa môžeme na zložité procesy vyhodnocovania funkcií dívať ako na veľa malých, jednoduchých krokov. Otázka znie, kde už sme nútení prestať, čo sa už nedá ďalej zjednodušiť?

Taktiež nás tento príklad môže inšpirovať k voľbe nástroja na veľmi jednoduchý popis mnohých funkcií: rekurziu. Napr. „x umocnené na y“ môžeme definovať pomocou predpisu „x umocnené na $n+1$“ je „x umocnené na n“ krát x. (Plus samozrejme potrebujeme začiatočné podmienky, v tomto prípade: „x umocnené na 0“ je 1.)

1.2 Konvencie

Pred samotnou definíciou jednej konkrétnej triedy takto popísateľných funkcií si ešte pripomenieme nejaké skutočnosti a zadefinujeme nejaké nové značenie.

Ako všade na tomto predmete, tak aj tu nulu považujeme za prirodzené číslo: $0 \in \mathbb{N}$.

Keďže pri konštrukciách v tejto kapitole často silno záleží na arite použitých funkcií (teda počte vstupov, ktoré majú), zavedieme si pre aritu funkcií špeciálne značenie: horný index v obdĺžničku. Napr. $f^{\boxed{3}}$ bude teda označovať funkciu f, o ktorej čitateľa explicitne informujeme, že má tri vstupy (je ternárna).

Aby bolo pre čitateľa jednoduchšie rozlíšiť, kedy sa pri rovnosti objektov jedná o rovnosť čísel a kedy o rovnosť celých funkcií, budeme pre rovnosť funkcií používať symbol $\equiv$. (Na funkciu sa môžeme dívať ako na množinu dvojíc (vstup, výstup); značenie $\equiv$ je potom rovnosť takýchto množín.)

1.3 Definícia

Trieda PREC primitívne rekurzívnych funkcií je definovaná nasledovne:

1. Nulárna funkcia z taká, že $z() = 0$, je v PREC.
   (Formálnejšie: z je funkcia s 0 vstupmi a 1 výstupom, ktorý je rovný nule.)
2. Unárna funkcia s definovaná predpisom $\forall x : s(x) = x + 1$ je v PREC.
   Túto funkciu voláme nasledovník (successor).
3. Pre každé $1 \leq k \leq n$ je v PREC n-árna funkcia $proj_{n,k}$ definovaná $\forall x_1, \ldots, x_n : P_{n,k}(x_1, \ldots, x_n) = x_k$.
   Tieto funkcie voláme projekcie. Špeciálne funkciu $proj_{n,k}$ voláme „k-ta projekcia z n“.
4. Ak $g^{\boxed{k}}$, $f_1^{\boxed{m}}, \ldots, f_k^{\boxed{m}} \in$ PREC, tak aj $h^{\boxed{m}} \in$ PREC, kde h je funkcia daná nasledujúcim predpisom:
   $\forall x_1, \ldots, x_m : \quad h(x_1, \ldots, x_m) = g(\ f_1(x_1, \ldots, x_m),\ \ldots,\ f_k(x_1, \ldots, x_m)\ )$.
   Hovoríme, že h vzniká kompozíciou pôvodných funkcií. Zapisujeme to $h \equiv COMP[g, f_1, \ldots, f_k]$. Tento zápis môžeme tiež podrobnejšie čítať nasledovne: „Funkcia h sa počíta tak, že do funkcie g dosadíme výstupy funkcií $f_1, \ldots, f_k$.“

   (Všimnite si, že g uvádzame ako prvé, hoci sa počíta posledné. Tento zdanlivo nelogický spôsob zápisu bude mať svoj zmysel, keď budeme niekedy neskôr primitívne rekurzívne funkcie kódovať do čísel.)
5. Ak $f^{\boxed{k}}, g^{\boxed{k+2}} \in$ PREC, tak aj $h^{\boxed{k+1}} \in$ PREC, kde h je funkcia definovaná nasledovne:

$$\forall x_1, \ldots, x_k : h(0, x_1, \ldots, x_k) = f(x_1, \ldots, x_k)$$
$$\forall n, x_1, \ldots, x_k : h(n+1, x_1, \ldots, x_k) = g(\ h(n, x_1, \ldots, x_k),\ n,\ x_1, \ldots, x_k\ )$$

   Hovoríme, že h vzniká z f a g primitívnou rekurziou. Zapisujeme to $h \equiv PR[f, g]$.

   Funkcia f nám popisuje, ako počítať základný prípad rekurzie (funkčnú hodnotu v nule) a funkcia g zase ako počítať posledný krok rekurzie (funkčnú hodnotu v $n+1$ z funkčnej hodnoty pre n, čísla n a hodnôt ostatných parametrov).
6. V PREC nie sú žiadne iné funkcie.

Ekvivalentne, PREC je najmenšia trieda funkcií, ktorá obsahuje funkcie z bodov 1-3 a je uzavretá vzhľadom na kompozíciu a na primitívnu rekurziu.

Všetky funkcie v PREC sú totálne.

Dôkaz spravíme štrukturálnou indukciou podľa definície. Jediné netriviálne miesto je použitie primitívnej rekurzie. Tam si ale môžeme všimnúť, že hodnota prvého parametra odovzdávaného funkcii h pri každom kroku klesá, preto náš výpočet určite skončí po vyhodnotení h pre konečne veľa vstupov. No a to znamená len konečne veľa volaní funkcií f a g, a o tých z indukčného predpokladu vieme, že všetky skončia.

Pre neformalistov

Operácia kompozície je vlastne naše štandardné skladanie funkcií, až na to, že tu máme prísne obmedzené arity funkcií, ktoré vieme skladať. Neskôr si ale ukážeme, že trieda PREC je uzavretá na kompozíciu aj v zmysle, na ktorý sme zvyknutí z bežnej matematiky.

(Aj) na toto budú slúžiť práve projekcie. Pomocou nich si budeme vedieť upravovať to, aké parametre berú funkcie, s ktorými pracujeme.

Na primitívnu rekurziu sa ekvivalentne dá dívať ako na jednoduchý for-cyklus s pevným počtom iterácií. Funkciu h, ktorá vznikne primitívnou rekurziou z f a g, si môžeme ekvivalentne predstaviť ako funkciu počítanú nasledovným programom:

```
def h(y, x1, ..., xk):
    tmp = f(x1, ..., xk)
    for i = 0 .. y-1:
        tmp = g(tmp, i, x1, ..., xk)
    return tmp
```

Funkcia f teda predstavuje inicializáciu a funkcia g výpočet jednej iterácie cyklu.

Pre formalistov

Pre zarytých formalistov tu máme poznámku k nulárnym funkciám: Formálne si môžeme $\mathbb{N}^k$ definovať ako množinu všetkých k-tic prirodzených čísel. Špeciálne teda $\mathbb{N}^0$ **nie je** prázdna množina – je to množina obsahujúca všetky 0-tice prirodzených čísel. Teda množina obsahujúca jeden prvok: prázdnu 0-ticu prirodzených čísel. Ak si tú označíme napr. symbolom ε, tak je $\mathbb{N}^0 = \{\varepsilon\}$.

Nulárne funkcie, ako napr. funkcia z (zero), sú potom priamočiaro funkcie z $\mathbb{N}^0$ do $\mathbb{N}$; konkrétne funkcia z vstupu ε priradí výstup 0.

Toto nás však nijak extra netrápi, na tento level formalizmu sa nepotrebujeme znížiť. Spokojne môžeme s nulárnymi funkciami pracovať ako s funkciami, ktoré nemajú žiadne vstupy a vracajú konštantu.

Pri definícii kompozície sme pre lepšiu čitateľnosť vynechali jeden technický detail: požadujeme $k > 0$. Funkcia g, ktorá počíta posledný krok výpočtu novej funkcie h, musí mať aspoň jeden vstup, aby existovala aspoň funkcia f_1 a tým aby bola jednoznačne určená arita funkcie h, ktorú vyrobíme.

Na *COMP* a *PR* sa môžeme dívať ako na funkcionály – teda funkcie, ktoré dostávajú na vstupe funkcie a na výstupe tiež vracajú funkciu. To, že pre ich aplikáciu používame hranaté zátvorky namiesto obyčajných má rovnaký dôvod ako zavedenie značenia $\equiv$.

A ešte vás ubezpečíme, že bolo v poriadku písať $z^{\boxed{0}}$, $s^{\boxed{1}}$ a $proj^{\boxed{n}}_{n,k}$, len sme to vzhľadom na kontext nepovažovali za potrebné.

1.4 Budovanie základných primitívne rekurzívnych funkcií

V tejto kapitole budeme spolu postupne objavovať, aké všetky funkcie musia byť primitívne rekurzívne. V priebehu kapitoly si postupne dokážeme aj niekoľko tvrdení, ktoré nám umožnia pohodlnejšie vyrábať nové primitívne rekurzívne funkcie.

1.4.1 Konštanty s aritou nula

Konštanty, presnejšie nulárne funkcie, vieme vyrobiť postupnou kompozíciou veľa successorov a nuly. Formálne môžeme napríklad funkciu $j^{\boxed{0}}$ takú, že $j() = 1$, vyrobiť nasledovne: $j \equiv COMP[s, z]$. (Slovne: j vzniká kompozíciou, pri ktorej do funkcie s dosadíme funkciu z.)

1.4.2 Unárna nula a ostatné konštanty s aritou 1

Unárna nula z_1 je funkcia, ktorú si neformálne vieme definovať nasledovným rekurzívnym vzťahom: na vstupe 0 vracia 0, a pre každé n platí, že na vstupe $n+1$ vracia to isté ako na vstupe n.

My si teraz chceme takúto funkciu „vyrobiť“ pomocou primitívnej rekurzie. Jej „pseudokód“ by teda vyzeral nasledovne:

```
def z1(y):
    tmp = fz1()
    for i = 0 .. y-1:
        tmp = fg1(tmp, i)
    return tmp
```

Na výrobu unárnej funkcie z_1 teda potrebujeme funkciu $fz1^{\boxed{0}}$ vracajúcu nulu a funkciu $gz1^{\boxed{2}}$ vracajúcu svoj prvý parameter. Takéto funkcie už máme: sú to funkcie z a $proj_{2,1}$. Platí teda $z_1 \equiv PR[z, proj_{2,1}]$.

Ostatné unárne konštanty dostaneme postupnou kompozíciou successorov a unárnej nuly.

1.4.3 Konštanty vyššej arity

Konštantné nuly s vyššou aritou sa dá vyrobiť viacerými spôsobmi, napríklad:

- Rovnako, ako sme vyrobili z_1 zo z, vieme následne vyrobiť ďalšou primitívnou rekurziou binárnu nulu z_2 ako $PR[z_1, proj_{3,1}]$, a tak ďalej.
- Vieme priamo vyrobiť k-árnu nulu (pre $k > 1$) primitívnou rekurziou zo z_1 a $proj_{k+1,1}$.
- Ide to aj bez ďalšieho použitia primitívnej rekurzie: na výrobu z_k stačí kompozíciou do z_1 dosadiť $proj_{k,1}$.

Iné konštanty s vyššou aritou opäť dostaneme postupnou kompozíciou successorov a nuly správnej arity.

1.4.4 Identita

Identitu máme samozrejme v našej množine projekcií, je to $i \equiv proj_{1,1}$.

1.4.5 Sčítanie

Intuitívne: sčítanie môžeme definovať nasledovne:

$$add(0, y) = y$$
$$add(x + 1, y) = add(x, y) + 1$$

Podľa tejto intuitívnej definície vyrobíme formálnu, pomocou operácie primitívnej rekurzie. Na jej použitie potrebujem mať dve funkcie: *fadd* a *gadd*. Funkcia *fadd* popisujúca základný prípad má v našom prípade pre ľubovoľný vstup y vrátiť výstup y, je to teda identita.

Funkcia *gadd*, ktorá počíta rekurzívny krok, dostane vstupy $add(x, y)$, x a y a má vrátiť výstup $add(x, y)+1$. Teda ide o funkciu $g(u, v, w) = u + 1$. Takáto funkcia je zjavne primitívne rekurzívna, lebo vzniká kompozíciou, pri ktorej do s dosadíme $proj_{3,1}$.

Formálne teda $add \equiv PR[proj_{1,1}, COMP[s, proj_{3,1}]]$.

1.4.6 Násobenie

Intuitívne: násobenie môžeme definovať nasledovne:

$$mul(0, y) = 0$$
$$mul(x + 1, y) = mul(x, y) + y$$

Formálne teda $mul \equiv PR[z_1, COMP[add, proj_{3,1}, proj_{3,3}]]$.

Iná definícia násobenia

Keďže sčítanie je komutatívne, fungovalo by aj nasledovné: $mul \equiv PR[z_1, COMP[add, proj_{3,3}, proj_{3,1}]]$.

Pri tejto príležitosti upozorňujeme, že ako v každom inom programovacom jazyku, aj tu platí, že môže (a dokonca vždy bude) existovať viacero syntakticky rôznych programov, ktoré počítajú tú istú funkciu. V našom príklade je touto funkciou (v matematickom slova zmysle) násobenie a rôzne programy sú vlastne rôzne spôsoby, ako „poskladať“ túto funkciu dohodnutým spôsobom z dohodnutých „stavebných kameňov“.

1.4.7 Mocniny, tetrácia a ďalej

Pomocou sčítania sme práve definovali násobenie. Pomocou opakovaného násobenia teraz môžeme rovnako definovať ďalším použitím primitívnej rekurzie umocňovanie: x umocnené na nultú je 1, a x umocnené na $n+1$ je súčinom x umocneného na n a obyčajného x.

Pomocou umocnenia môžeme ďalej definovať mocninové veže (tetráciu): mocninová veža so základom x výšky 0 je 1, a mocninová veža výšky $n + 1$ je x umocnené na mocninovú vežu výšky n.

Tetrácia rastie vcelku rýchlo. Napríklad by ste ju asi nechceli mať v časovej zložitosti: už program, ktorý spraví 3^{3^3} krokov, pobeží niekoľko hodín. Číslo $4^{4^{4^4}}$ je tak obrovské, že už len jeho *počet cifier* je 154-ciferný.

Inými slovami, táto mocninová veža má viac ako 10^{153} cifier. Pre porovnanie, celá globálna datasféra Zeme sa v roku 2020 odhaduje na niečo v okolí 10^{23} bajtov, takže celá Zem *ani rádovo* nemá šancu si uložiť desiatkový zápis tohto čísla.

No a my pomocou tetrácie môžeme ďalej definovať... ozaj rýchlo rastúcu funkciu. A z nej ešte rýchlejšie rastúcu. A tak ďalej. Nech to letí!

1.4.8 Polynómy

Funkcia x^2 je primitívne rekurzívna. Vyrobiť ju vieme napríklad kompozíciou: do funkcie násobenia dosadíme identitu a identitu. Formálne $sqr \equiv COMP[mul, proj_{1,1}, proj_{1,1}]$.

Funkcia $2x$ je primitívne rekurzívna, vyrobiť ju vieme rovnako, len namiesto *mul* použijeme *add*.

Funkciu $2x$ vieme vyrobiť aj tak, že pri kompozícii do násobenia dosadíme (v ľubovoľnom poradí) identitu a unárnu konštantu 2. Podobne sme vedeli vyrobiť aj x^2 tak, že do umocnenia dosadíme (tentokrát na poradí záleží) najskôr identitu a potom unárnu konštantu 2.

Vhodnou kombináciou vyššie uvedených postupov si vieme vyrobiť funkciu ax^b pre ľubovoľné a, b. No a postupným sčítaním takýchto funkcií si vieme vyrobiť ľubovoľný polynóm, ktorého koeficienty sú prirodzené čísla. Všetky tieto polynómy sú teda primitívne rekurzívne funkcie.

1.4.9 Predecessor

Pri definícii predecessora (predchádzajúcej hodnoty) máme drobný problém: čo spraviť so vstupom 0? Keďže sa hráme len s prirodzenými číslami, nemáme hodnotu -1. Mohlo by nás pokúšať nechať túto funkciu v nule nedefinovanú, avšak všetky primitívne rekurzívne funkcie sú totálne, takže to spraviť nevieme. Dohodneme sa preto na najmenšom zle: predecessorom nuly bude opäť nula. Použitie predecessora môžeme teda čítať „zmenši, ak sa to dá“.

Intuitívna rekurzívna definícia:

$$p(0) = 0$$
$$p(x+1) = x$$

Podľa nej ľahko napíšeme definíciu formálnu: $p \equiv PR[z, proj_{2,2}]$.

Využili sme pri tom technický trik, že v definícii primitívnej rekurzie je do funkcie g odovzdávaná o jedno menšia hodnota „riadiacej premennej“ ako tá, pre ktorú počítame výstup. Pre názornosť, tu je pseudokód zodpovedajúci našej definícii funkcie p:

```
def p(x):
    tmp = 0
    for i = 0 .. x-1:
        tmp = i
    return tmp
```

1.4.10 Odčítanie

Pri výrobe odčítania budeme mať dva problémy. Prvý: čo so zápornými číslami, čomu sa má rovnať $3-7$? Tu sa dohodneme rovnako ako u predecessora: odčítanie nebude vedieť podtiecť pod nulu. Formálne si teda vyrobíme funkciu $\max(0, x-y)$.

Druhý problém si pozorný čitateľ už možno všimol, hoci my sme mu to vtedy naschvál zamlčali. Miesto, kde sa tento problém skrýval, bol náš neformálny popis toho, ako vyrobiť funkciu umocnenia. Ten mal jeden syntaktický problém: rekurziu (cyklus) potreboval robiť podľa *druhého* parametra funkcie. Na tento istý problém narážame aj tu. Vyššie popísanú funkciu si totiž neformálne vieme definovať nasledovne: „$x-y$ sa počíta tak, že zoberieš x a y-krát ho zmenšíš o 1“.

Druhý probém vyriešime tak, že výrobu odčítania spravíme na dva kroky. Najskôr spravíme funkciu, ktorá bude mať parametre v opačnom poradí: funkciu *bus* („*sub*“ odzadu), ktorá od svojho druhého parametra odčíta prvý.

Funkcia bus spĺňa nasledovnú rekurzívnu definíciu (pričom p je predecessor):

$$bus(0, x) = x$$
$$bus(y + 1, x) = p(bus(y, x))$$

Podľa nej teda zostrojíme bus v našom formalizme: $bus \equiv PR[proj_{1,1}, COMP[p, proj_{3,1}]]$.

No a teraz z bus zostrojíme funkciu sub, ktorá už má parametre v očakávanom poradí. Na prehodenie parametrov použijeme kompozíciu s vhodnými projekciami: $sub \equiv COMP[bus, proj_{2,2}, proj_{2,1}]$.

1.5 Veta o kompozícii

Ako už naznačuje konštrukcia funkcie sub, projekcie sa dajú používať na prehadzovanie poradia parametrov a menenie ich počtu.

Keď sme definovali kompozíciu, spravili sme definíciu veľmi striktne: všetky funkcie f_i museli mať presne rovnakú aritu a dostávali presne tie isté vstupy. V súčasnej dobe sme však zvyknutí pojmom „kompozícia“ označovať všeobecnejšie skladanie funkcií. Napríklad aj o takejto funkcii φ hovoríme, že vzniká kompozíciou (zložením) funkcií použitých na pravej strane:

$$\forall x, y, z : \varphi(x, y, z) = \psi(\ \alpha(x),\ \beta(y, y, z),\ \gamma(z, x)\ )$$

Veta o kompozícii (slabšia verzia): Každá funkcia, ktorú vieme zapísať ako takúto kompozíciu primitívne rekurzívnych funkcií, je tiež primitívne rekurzívna.

Dôkaz (príkladom): Na vyššie uvedenej funkcii φ predvedieme, ako takúto kompozíciu „preložiť“ do povolenej podoby. Jediné, čo spravíme, je úprava funkcií α, β a γ do podoby, v ktorej bude každá z nich brať na vstupe všetky tri parametre funkcie φ v správnom poradí.

V našom prípade teda vyrobíme nové funkcie:

$$\alpha' \equiv COMP[\ \alpha,\ proj_{3,1}\ ]$$
$$\beta' \equiv COMP[\ \beta,\ proj_{3,2},\ proj_{3,2},\ proj_{3,3}\ ]$$
$$\gamma' \equiv COMP[\ \gamma,\ proj_{3,3},\ proj_{3,1}\ ]$$

Keďže funkcie α', β' a γ' vznikli z primitívne rekurzívnych funkcií kompozíciou, sú tiež primitívne rekurzívne. A následne je teda primitívne rekurzívna aj funkcia $\varphi \equiv COMP[\psi, \alpha', \beta', \gamma']$.

Poďme ďalej. Občas môžeme mať pri definícii novej funkcie aj viac úrovní vnorenia:

$$\forall x, y, z : \varphi_2(x, y, z) = \psi(\ \alpha(\alpha(x)),\ \beta(y, y, z),\ \gamma(\gamma(\alpha(z), z), x)\ )$$

Veta o kompozícii (silnejšia verzia): Furt to platí. Dôkaz štrukturálnou indukciou: robíme to isté čo vyššie, postupne od najhlbšej úrovne vnorenia smerom von.

Dôsledok: Ak je zjavné, že nejakú novú funkciu vieme vyrobiť kompozíciou už existujúcich, nemusíme to odteraz rozpisovať.

1.6 Predikáty a ďalšie užitočné funkcie

„Predikát“ je matematický pojem označujúci funkciu, ktorá vracia boolovskú hodnotu. V našom formalizme primitívnej rekurzie môžeme jednoducho za predikáty považovať funkcie, ktoré vracajú len hodnoty 0 (tú interpretujeme ako false) a 1 (true).

1.6.1 Signum a negácia

Inou možnosťou, ako definovať predikáty, by bolo interpretovať celé čísla ako pravdivostné hodnoty v štýle jazyka C: 0 je false, čokoľvek kladné je true. Oba spôsoby sú samozrejme ekvivalentné čo do vyjadrovacej sily. Formálne si to vieme zdôvodniť tak, že ukážeme primitívnu rekurzívnosť funkcie signum:

$$sgn(x) = \begin{cases} 0 & \leftarrow x = 0 \\ 1 & \leftarrow \text{inak} \end{cases}$$

Signum vieme zostrojiť primitívnou rekurziou z funkcie z a funkcie j_2 (binárnej konštanty 1): $sgn \equiv PR[z, j_2]$.

Ešte sa nám bude hodiť funkcia $\overline{sgn}$ určená predpisom $\overline{sgn}(x) = 1 - sgn(x)$. Tá z nuly vyrobí jednotku a z nie-nuly nulu, budeme ju teda pri predikátoch používať ako negáciu. Vyrobiť $\overline{sgn}$ si môžeme buď podobne ako sgn, teda primitívnou rekurziou, alebo napríklad kompozíciou unárnej jednotky, sgn a odčítania: $\overline{sgn} \equiv COMP[sub, j_1, sgn]$.

1.6.2 Logické spojky

Ak máme primitívne rekurzívne predikáty p a q, ľahko zostrojíme predikáty pre ich logický and a logický or: na logický and stačí použiť súčin, na logický or súčet ich hodnôt. Presnejšie, $COMP[mul, p, q]$ je nový predikát, ktorý je pravdivý len vtedy, keď je pravdivé aj p, aj q. A podobne, $COMP[sgn, COMP[add, p, q]]$ je predikát pravdivý len vtedy, keď je pravdivý aspoň jeden z p a q. (Všimnite si technický detail: pri sčítaní sme ešte pridali kompozíciu so sgn, aby sme vracali hodnotu 1 a nie hodnotu 2, ak sú pravdivé oba predikáty.)

No a pomocou logického and, or a not (pripomíname $\overline{sgn}$) si už vieme naskladať ľubovoľnú logickú spojku ľubovoľnej arity.

1.6.3 Porovnania čísel

Testy na rovnosť a nerovnosť vieme ľahko spraviť pomocou nášho odčítania. Napr. test, či $x > y$, vieme spraviť tak, že sa pozrieme, či je hodnota $sub(x, y)$ kladná. Teda $gt \equiv COMP[sgn, sub]$ a analogicky $leq \equiv COMP[\overline{sgn}, sub]$. Na porovnanie opačným smerom potrebujeme odčítanie s opačným poradím argumentov, teda použijeme pomocnú funkciu bus z definície sub: bude $lt \equiv COMP[sgn, bus]$ a analogicky $geq \equiv COMP[\overline{sgn}, bus]$.

Test na rovnosť dvoch čísel: $eq(x, y)$ ak $geq(x, y)$ a zároveň $leq(x, y)$.

1.7 Veta o if-e

Ako ďalší dobrý nástroj na výrobu nových primitívne rekurzívnych funkcií si teraz môžeme vysloviť a dokázať vetu o if-e. Vyslovíme ju najskôr v jednoduchšej, binárnej verzii:

1.7.1 Znenie vety (binárna verzia)

Nech $f_1^{\boxed{m}}, f_0^{\boxed{m}}, g^{\boxed{m}} \in$ PREC. Potom aj $h \in$ PREC, kde h je funkcia definovaná nasledovne:

Pre ľubovoľné $\overline{x} = x_1, \ldots, x_m$ platí: ak $g(\overline{x}) > 0$, tak $h(\overline{x}) = f_1(\overline{x})$, inak $h(\overline{x}) = f_0(\overline{x})$.

1.7.2 Značenie

Všimnite si nové značenie $\overline{x}$ (čítaj „postupnosť x-ov“ alebo „vektor x-ov“), ktoré sme v definícii zaviedli na jej zostručnenie. Toto značenie budeme odteraz používať všade, kde budeme mať nejakú postupnosť argumentov funkcie, ktorých počet ani hodnoty sa nemenia. Konkrétny počet argumentov v takejto postupnosti bude vždy jasný z kontextu – zväčša teda priamo z arity dotyčnej funkcie.

1.7.3 Pseudokód

Na funkciu g sa dívajme ako na predikát. Vyššie definovaná funkcia h sa potom dá v pseudokóde zapísať nasledovne:

```
def h(x1, ..., xm):
    if g1(x1, ..., xm):
        return f1(x1, ..., xm)
    else:
        return f0(x1, ..., xm)
```

Odtiaľ teda názov vety o if-e. Táto veta nám hovorí, že ak sa pomocou primitívne rekurzívnej podmienky vieme rozhodnúť, ktorú primitívne rekurzívnu funkciu počítať, aj nová funkcia bude primitívne rekurzívna. Inými slovami, do nášho „programovacieho jazyka“ dostávame podmienku if.

1.7.4 Dôkaz (binárnej verzie)

Funkcia $sgn(g(\overline{x})) \cdot f_1(\overline{x})$ je buď $f_1(\overline{x})$ alebo 0, podľa toho, či $g(\overline{x}) > 0$.
Funkcia $\overline{sgn}(g(\overline{x})) \cdot f_0(\overline{x})$ je buď 0 alebo $f_0(\overline{x})$, podľa toho, či $g(\overline{x}) > 0$.
Obe vyššie uvedené funkcie sú primitívne rekurzívne. Funkcia h je ich súčtom.

1.7.5 Znenie vety (všeobecná verzia)

Pre zaujímavosť ešte uvedieme znenie jednej možnej všeobecnej verzie, kde sa rozhodujeme podľa viacerých podmienok. Dôkaz je priamočiarym zovšeobecnením toho uvedeného vyššie, detaily prenechávame na čitateľa (ale aspoň mu pripomenieme, že logický and viacerých podmienok vieme robiť násobením príslušných predikátov).

- Nech $f_1^{\boxed{\mathrm{m}}}, \ldots, f_{k+1}^{\boxed{\mathrm{m}}} \in \textsc{Prec}$
- Nech $g_1^{\boxed{\mathrm{m}}}, \ldots, g_k^{\boxed{\mathrm{m}}} \in \textsc{Prec}$ a nech $g_{k+1}^{\boxed{\mathrm{m}}}$ je m-árna konštanta 1.
- Nech h je funkcia definovaná nasledovne: Pre ľubovoľné $\overline{x} = x_1, \ldots, x_m$ platí, že $h(\overline{x})$ je rovné $f_i(\overline{x})$, kde i je najmenšie prirodzené číslo, pre ktoré $g_i(\overline{x}) > 0$.
- Potom $h \in \textsc{Prec}$.

1.7.6 Jednoduché aplikácie

Maximum je primitívne rekurzívna funkcia, lebo vzniká vetou o if-e z funkcií $proj_{2,1}$, $proj_{2,2}$ a predikátu gt (väčší). Slovne: ak je prvý vstup väčší ako druhý, vráť prvý vstup, inak vráť druhý vstup.

Analogicky je primitívne rekurzívne aj binárne minimum a následne kompozíciou aj maximum a minimum ľubovoľnej vyššej arity.

1.8 Veta o prefixových súčtoch

V tejto časti si vyrobíme funkciu počítajúcu prefixové súčty a následne pomocou nej zadefinujeme niekoľko ďalších pomerne elementárnych funkcií, ktoré nám ešte chýbajú – špeciálne bude zaujímavé delenie so zvyškom.

1.8.1 Základné znenie vety

Pre ľubovoľnú primitívne rekurzívnu funkciu $f^{\boxed{\mathrm{k+1}}}$ je primitívne rekurzívna aj funkcia $g^{\boxed{\mathrm{k+1}}}$ definovaná nasledovne:

$$\forall x, \overline{y} : g(x, \overline{y}) = \sum_{i<x} f(i, \overline{y})$$

Slovne: pre ľubovoľné fixne zvolené $\overline{y}$ je $g(x, \overline{y})$ súčtom prvých x hodnôt $f(\cdot, \overline{y})$.

Dôkaz: Funkciu g vyrobíme primitívnou rekurziou. Pomôžeme si opäť potrebným pseudokódom:

```
def g(x, y1, ..., yk):
    tmp = 0
    for i = 0 .. x-1:
        tmp = tmp + f(i, y1, ..., yk)
    return tmp
```

Vidíme teda, že g vieme vyrobiť primitívnou rekurziou. Základný prípad nám počíta k-árna konštantná nula: prázdny súčet je nula. Jednu iteráciu cyklu nám počíta funkcia, ktorú vieme vyrobiť vhodnou kompozíciou primitívne rekurzívnej funkcie f, projekcií a sčítania.

1.8.2 Zovšeobecnenie

Vyššie uvedená veta má veľa možných zovšeobecnení. Uvedieme jedno: Pre ľubovoľné primitívne rekurzívne funkcie $lo^{\boxed{k}}$, $hi^{\boxed{k}}$ a $f^{\boxed{k+1}}$ je primitívne rekurzívna aj funkcia $g^{\boxed{k}}$ definovaná nasledovne:

$$g(\overline{x}) = \sum_{lo(\overline{x}) \leq i \leq hi(\overline{x})} f(i, \overline{x})$$

Myšlienka dôkazu: Pomocou základnej verzie vety si vieme vyrobiť funkciu, ktorá nám spraví prefixové súčty f. Súčet úseku od $lo(\overline{x})$ po $hi(\overline{x})$ je rovný prefixovému súčtu prvých $hi(\overline{x}) + 1$ prvkov, od ktorého odčítame prefixový súčet prvých $lo(\overline{x})$ prvkov.

1.8.3 Dolná celá časť podielu

Nasledovať bude pomerne technická konštrukcia, pomocou ktorej si vyrobíme delenie.

Pri delení sa samozrejme opäť musíme najskôr vysporiadať s technickým problémom: na jednej strane sú všetky primitívne rekurzívne funkcie totálne, na druhej strane však nulou deliť nevieme. Čo s tým? Zadefinujeme si funkciu delenia tak, aby fungovala, keď nedelíme nulou, a nebude nás trápiť, aké hodnoty vracia pri delení nulou, lebo na delenie nulou ju nebudeme používať.

Komu by sa toto nepáčilo, môže si následne použiť vetu o if-e a pomocou nej definovať poriadnejšie delenie, ktoré napríklad pri delení čohokoľvek nulou explicitne vráti nulu.

Majme teda x a y, pričom $y \geq 1$. Ako by sme mohli nejakým cyklom spočítať $\lfloor x/y \rfloor$? Pokúšalo by nás zrejme robiť niečo ako „dokola odčítať y a počítať si kroky“, ako ale niečo takéto v našom formalizme zapísať? Pomôžeme si nasledovne:

Predstavme si nasledovnú postupnosť:

$$1y, 2y, \ldots, (x-1)y, xy$$

Všimnime si, že posledný člen tejto postupnosti je aspoň x. Keby sme v nej pokračovali, ďalšie členy už budú väčšie ako x.

Vyrobíme si teraz novú postupnosť. Každý člen tej pôvodnej postupnosti porovnáme s x. Do novej postupnosti napíšeme 1 ak bol menší alebo rovný, respektíve 0 ak už bol väčší. Napr. pre $x = 11$ a $y = 3$ by to vyzeralo nasledovne:

stará	3	6	9	12	15	18	21	24	27	30	33
nová	1	1	1	0	0	0	0	0	0	0	0

Nech $d = \lfloor x/y \rfloor$. Potom je zjavné, že nová postupnosť obsahuje presne d jednotiek: posledná jednotka zodpovedá číslu dy, keďže platí $dy \leq x < (d+1)y$.

No a už sme skoro vyhrali. Našu „novú postupnosť“ počíta ternárny predikát $f(i, x, y) = leq(mul(s(i), y), x)$. Nech teraz funkcia g počíta prefixové súčty funkcie f. Potom $div(x, y) = g(x, x, y)$.

1.8.4 Ďalšie funkcie súvisiace s delením

- Zvyšok po delení: $mod(x, y) = x - mul(y, div(x, y))$.
- Predikát deliteľnosti: $divisible(x, y) = \overline{sgn}(mod(x, y))$.

- Horná celá časť podielu: $ceildiv(x,y) = div(x,y) + \overline{sgn}(divisible(x,y))$.
- Test prvočíselnosti $isprime(x)$: x je prvočíslo práve vtedy, keď $\sum_{1 \le d \le x} divisible(x,d) = 2$. (Použijeme teda všeobecnejšiu verziu vety o prefixových súčtoch a predikát pre rovnosť takto definovanej funkcie a konštantnej dvojky so správnou aritou.)
- Počet prvočísel menších ako x: $primesbelow(x)$ je prefixový súčet prvých x hodnôt predikátu $isprime$.

1.9 Ohraničená minimalizácia

V tejto časti sa prvýkrát stretneme s minimalizáciou. Tak, ako je primitívna rekurzia akýmsi predchodcom for-cyklu, minimalizácia bude predchodcom while-cyklu. Aby sme ho vedeli simulovať for-cyklom, budeme musieť prijať jedno obmedzenie: vopred budeme musieť povedať maximálny počet iterácií. (Presnejšie, tento počet si musíme vypočítať nejakou primitívne rekurzívnou funkciou.)

Ukážeme si, že na takto ohraničenú minimalizáciu je trieda PREC tiež uzavretá, čím dostaneme ďalší nástroj na výrobu nových funkcií.

1.9.1 Veta o ohraničenej minimalizácii

Pre ľubovoľné primitívne rekurzívne funkcie $f^{\boxed{k+1}}(y,\overline{x})$ a $g^{\boxed{k}}(\overline{x})$ je primitívne rekurzívna aj funkcia $h^{\boxed{k}}$ definovaná nasledovne:

$$\forall \overline{x} : h(\overline{x}) = \begin{cases} \min\{i \mid i < g(\overline{x}) \ \wedge \ f(i,\overline{x}) > 0\} & \leftarrow \text{ak také } i \text{ existuje} \\ g(\overline{x}) & \leftarrow \text{inak} \end{cases}$$

Slovne, $h(\overline{x})$ si môžeme predstaviť tak, že postupne počíta $f(0,\overline{x})$, $f(1,\overline{x})$, ..., až kým buď prvýkrát nedostane pre nejaké i nenulovú hodnotu, alebo nedosiahne vopred určenú hranicu $g(\overline{x})$. Na výstup vráti buď index i, ak našla, alebo hornú hranicu cyklu, ak nenašla.

Myšlienka dôkazu: Uvažujme postupnosť hodnôt $\overline{sgn}(f(i,\overline{x}))$ pre $0 \le i < g(\overline{x})$. Návratová hodnota h je zjavne rovná počtu jednotiek na začiatku tejto postupnosti.

Rovnako, ako sme robili prefixové súčty, si teraz spravíme funkciu počítajúcu prefixové *súčiny* tejto funkcie. To spôsobí, že jednotky na začiatku ostanú jednotkami, ale od prvej nuly ďalej budeme mať nuly.

No a teraz už použijeme klasické prefixové súčty: sčítame prvých $g(\overline{x})$ funkčných hodnôt tejto novej funkcie a dostaneme $h(\overline{x})$.

1.9.2 Aplikácie

- Hornú celú časť podielu vieme teraz zostrojiť aj nasledovne:
  - Vieme, že hľadáme najmenšie i také, že $iy \ge x$.
  - Vieme, že takéto i stačí hľadať v intervale $[0,x]$, čiže medzi prvými $x+1$ číslami.

  Hornú celú časť podielu teda vieme vyrobiť ohraničenou minimalizáciou predikátu $iy \ge x$ podľa premennej i a s hornou hranicou $x+1$. (Pre ľubovoľné $y > 0$ vieme, že táto minimalizácia nájde odpoveď skôr, než narazí na nami zvolenú hornú hranicu. No a vonkoncom nás netrápi, čo sa stane pre $y = 0$.)
- Horná celá časť odmocniny z x je najmenšie y také, že $y^2 \ge x$. Aj toto y zjavne vieme nájsť ohraničenou minimalizáciou vhodného predikátu.
- Aj funkcia $prime(n)$ vracajúca n-té prvočíslo (číslované od nuly) je primitívne rekurzívna.

  Indukciou z Euklidovho dôkazu nekonečného počtu prvočísel vieme dokázať veľmi voľný horný odhad: n-té prvočíslo (číslujúc od nuly) je menšie alebo rovné 2^{2^n}.

  Zjavne platí, že $prime(n)$ je najmenšie také x, pre ktoré $primesbelow(x+1) = n$. Toto x vieme nájsť minimalizáciou tohto primitívne rekurzívneho predikátu, pričom ako hornú hranicu použijeme primitívne rekurzívnu funkciu $2^{2^n} + 1$.

1.10 Číslovacie funkcie a kódovanie postupností do čísel

Čo už vieme: Vo formalizme primitívnej rekurzie vieme definovať práve tie programy, ktoré vieme počítať programami s jednoduchými for-cyklami. Ide teda, aspoň zatiaľ, o programy bez všeobecných while-cyklov, rekurzie a podobných zveriniek – o týchto zatiaľ netušíme, či vôbec zachovávajú primitívnu rekurzívnosť.

Otvorené otázky: Existujú nejaké ďalšie veci ekvivalentné primitívnej rekurzii? Iné pohľady, cez ktoré vieme povedať čo je a čo nie je primitívne rekurzívne? Sú vlastne na niečo dobré while-cykly a všeobecná rekurzia? Samozrejme, umožnia nám napísať program, ktorý pre niektoré vstupy neskončí, ale je to všetko? Aký je súvis medzi primitívnou rekurziou a napr. tým, čo poznáme z teórie formálnych jazykov ako rekurzívne jazyky? Ak nám niekto zaručí, že program vždy zastaví, musí nutne počítať primitívne rekurzívnu funkciu?

Aby sme sa do týchto otázok vedeli pustiť, potrebujeme sa postupne dopracovať k tomu, že budeme vedieť jedným programom simulovať iný. No a keďže naše programy pracujú len s prirodzenými číslami, budeme musieť nejaké zložitejšie veci kódovať do čísel.

1.10.1 Konkrétna trojica číslovacích funkcií

Našim prvým cieľom na tejto ceste bude pomocou primitívne rekurzívnych funkcií vedieť zakódovať dvojicu prirodzených čísel do jedného. Chceme teda také primitívne rekurzívne funkcie $c(x, y)$, $l(x)$ a $r(x)$, aby platilo:

- c je prostá
- pre každé x, y: $l(c(x, y)) = x$
- pre každé x, y: $r(c(x, y)) = y$

Ako na to? Zoraďme si všetky dvojice prirodzených čísel do postupnosti. Systematicky to môžeme spraviť napr. tak, že ich zoradíme primárne podľa súčtu a sekundárne podľa prvého z nich:

$$(0,0),\ (0,1),\ (1,0),\ (0,2),\ (1,1),\ (2,0),\ (0,3),\ (1,2),\ \ldots$$

Teraz $c(x, y)$ definujeme ako zero-based index dvojice (x, y) v tomto poradí.

Pre lepšiu predstavu si všetky dvojice čísel môžeme predstaviť ako súradnice políčok v prvom kvadrante. Každé políčko dostane svoje číslo. Táto konkrétna funkcia c ich čísluje postupne po diagonálach:

```
  |
y |
  | ..
4 | 10
3 |  6  11
2 |  3   7  12
1 |  1   4   8  ..
0 |  0   2   5   9  ..
--+-------------------------
  |  0   1   2   3   4    x
```

Zjavne $c(x, y) = (1 + \cdots + (x + y)) + x = \frac{(x+y)(x+y+1)}{2} + x$, preto c je primitívne rekurzívna. (Prvý sčítanec je celkový počet čísel na predchádzajúcich diagonálach, druhý je poradové číslo nášho políčka na tej správnej.)

Ako je to s primitívnou rekurzívnosťou l a r? Mohli by sme skúsiť ich vyjadriť v uzavretom tvare (pomocou nejakých celých častí odmocnín a podobných chuťoviek) a potom sa o niečo snažiť, ide to ale aj ľahšie – len sa treba zbaviť hlúpej predispozície o snahu funkcie implementovať *efektívne*.

Pre naše funkcie l a r zjavne platí, že $\forall x : l(x) \leq x$ a tiež $\forall x : r(x) \leq x$. Ak teda chceme vypočítať napríklad $l(x)$, stačí vyskúšať všetky (a, b) z rozsahu od $(0, 0)$ po (x, x), nájsť tú jedinú dvojicu, ktorej kód je naozaj x, a vrátiť na výstup a.

Formálnejšie, zoberme si ternárnu funkciu $f(x, y, z) = mul(x, eq(c(x, y), z))$. Keď si zvolíme fixné z (kód dvojice čísel), tak pre všetky nesprávne dvojice x, y je $f(x, y, z) = 0$, len pre tú jedinú správnu dvojicu je to x.

Funkciu l teraz z f vyrobíme tak, že spravíme prefixové súčty podľa prvej premennej a z nich prefixové súčty podľa druhej premennej a potom sa pozrieme na konkrétnu funkčnú hodnotu, ktorá zodpovedá prvým $z+1$ premenným v každom rozmere.

1.10.2 Fibonacciho čísla

Dokážeme si, že Fibonacciho postupnosť je primitívne rekurzívna. Spravíme si pomocnú funkciu $fibhelper(n)$, ktorá bude vracať kód dvojice (F_n, F_{n+1}).

V Pythone by to vyzeralo takto:

```
def fibhelper(n):
    if n==0:
        return (0, 1)
    else:
        x, y = fibhelper(n-1)
        return (y, x+y)
```

Neformálna rekurzívna definícia:

$$\begin{aligned} fibhelper(0) &= c(0,1) \\ fibhelper(n+1) &= c(\ r(fibhelper(n)),\ l(fibhelper(n)) + r(fibhelper(n))\ ) \end{aligned}$$

$$fib(x) = l(fibhelper(x))$$

A ešte pre názornosť úplne formálne (až na vynechanie definícií c, l a r):

$$\begin{aligned} add &\equiv PR[proj_{1,1}, COMP[s, proj_{3,1}]] \\ j &\equiv COMP[s, z] \\ addpair &\equiv COMP[add, l, r] \\ almost &\equiv COMP[c, r, addpair] \\ g &\equiv COMP[almost, proj_{2,1}] \\ helper &\equiv PR[j, g] \\ fib &\equiv COMP[l, helper] \end{aligned}$$

1.10.3 Kódovanie konštantne veľa čísel do jedného

Samozrejme, akonáhle máme nejakú primitívne rekurzívnu trojicu číslovacích funkcií, zjavne vieme pomocou primitívne rekurzívnych funkcií zakódovať do jedného celého čísla nielen dvojicu, ale aj ľubovoľnú konštantne veľkú n-ticu čísel.

Napríklad jedna možnosť, ako to spraviť pre $n = 3$ je zobrať konkrétnu trojicu číslovacích funkcií c, l, r, zakódovať trojicu (x, y, z) funkciou $c(x, c(y, z))$ a dekódovať kód x na jednotlivé zložky funkciami $l(x)$, $l(r(x))$ a $r(r(x))$. Zovšeobecnenie pre väčšie n je zjavné.

1.10.4 Kódovanie postupností ľubovoľnej dĺžky

Pomocou primitívne rekurzívnych funkcií vieme počítať aj injektívne zobrazenie zo všetkých konečných postupností čísel do čísel – a teda kódovať „polia čísel“ ľubovoľnej dĺžky a pracovať s nimi. Keďže ale toto ešte pre záver tejto kapitoly nepotrebujeme, vrátime sa k tejto konštrukcii detailnejšie neskôr.

1.11 Simulácia Turingovsky úplných modelov

V tejto sekcii sa najskôr pozrieme na simuláciu Minského registrových strojov (tá je dostačujúca pre záver, ku ktorému smerujeme) a potom sa aspoň v náznakoch zmienime o simulácii iných modelov (pre názornejšiu predstavu).

1.11.1 Simulácia Minského registrových strojov

Uvažujme konkrétny Minského registrový stroj M s r registrami a p inštrukciami v programe. Konfigurácia takéhoto stroja je $(r+1)$-tica prirodzených čísel: aktuálny instruction pointer a hodnoty vo všetkých registroch.

Zjavne teraz vieme pomocou primitívne rekurzívnych číslovacích funkcií vyrobiť nasledujúce primitívne rekurzívne funkcie:

- Funkcia $kod^{\boxed{r+1}}$ zakóduje $r+1$ prirodzených čísel do jedného.
- Funkcia $ip^{\boxed{1}}$: $ip(x)$ interpretuje x ako kód $r+1$ čísel a vráti prvé z nich (instruction pointer).
- Funkcia $reg^{\boxed{2}}$: $reg(x,i)$ interpretuje x ako kód $r+1$ čísel a pre $i < r$ vráti $(i+1)$. z nich (hodnotu v registri i), inak nulu. (Keďže r je fixné, túto funkciu vieme vyrobiť použitím vety o if-e.)

Veta: K ľubovoľnej jednej inštrukcii v programe stroja M existuje primitívne rekurzívna funkcia, ktorá ju simuluje. (Teda funkcia, ktorá pre ľubovoľný platný kód konfigurácie, v ktorej je instruction pointer na tejto inštrukcii, vráti kód nasledovnej konfigurácie.)

Dôkaz (príkladmi):

Príklad 1: Nech $r = 3$ a inštrukcia číslo 7 je „INC 1“ (teda inkrementuj register 1). Funkcia, ktorá robí príslušnú zmenu konfigurácie je funkcia $step_7(x) = kod(\ 8,\ reg(x,0),\ s(reg(x,1)),\ reg(x,2)\ )$.

Príklad 2: Nech $r = 3$ a inštrukcia číslo 13 je „ZERO 0 7“ (teda otestuj register 0 na nulu, a ak je v ňom nula skoč na inštrukciu 7). Z vety o if-e si vieme vyrobiť funkciu $next_{13}(x)$, ktorá vráti 7 ak $reg(x,1) = 0$ a 14 inak. Následne funkcia, ktorá robí príslušnú zmenu konfigurácie, je funkcia $step_{13}$ definovaná nasledovne:

$step_{13}(x) = kod(\ next_{13}(x),\ reg(x,0),\ reg(x,1),\ reg(x,2)\ )$.

Veta: Ku Minského registrovému stroju M existuje primitívne rekurzívna funkcia $krok_M^{\boxed{1}}$, ktorá odsimuluje jeden krok jeho výpočtu – teda z danej konfigurácie vypočíta nasledujúcu.

Dôkaz: Cez vetu o if-e pospájame všetky vyššie definované funkcie $step_i$ (je ich len konštantný počet – presne p) a na záver identitu. Program teda postupne rozoberie prípady $ip(x) = 0, \ldots, ip(x) = p-1$ a ak ani jeden nenastal (máme konfiguráciu v ktorej už výpočet skončil) tak nespraví nič.

Veta: Ku stroju M existuje primitívne rekurzívna funkcia $simuluj_M^{\boxed{r+1}}$, ktorá pre dané k a začiatočné hodnoty v registroch odsimuluje prvých k krokov výpočtu M.

Dôkaz: Zo zadaných registrov vyrobíme začiatočnú konfiguráciu M a následne vo for-cykle k-krát odsimulujeme jeden krok výpočtu.

1.11.2 Veta o primitívne rekurzívnej časovej zložitosti

Zdalo by sa, že už sme vyhrali: primitívnou rekurziou vieme simulovať Minského registrové stroje. Nie je tomu úplne tak. Kde je pes zakopaný? V tom, že nás zaujíma, čo program vlastne vypočítal, keď skončil, a my vo všeobecnosti nevieme, *ako veľa krokov* potrebujeme odsimulovať, aby sme sa na ten koniec výpočtu dostali.

Zatiaľ si preto aspoň vyslovíme o čosi slabšie tvrdenie.

Veta: Nech existuje Minského registrový stroj M počítajúci nejakú funkciu f. Ďalej nech existuje *primitívne rekurzívna* funkcia t, taká, že pre ľubovoľné $\overline{x}$ platí, že M na vstupe $\overline{x}$ spraví nanajvýš $t(\overline{x})$ krokov. (Teda t zhora odhaduje časovú zložitosť M.) Potom aj funkcia f je primitívne rekurzívna.

Dôkaz: Zjavné. Vypočítame $f(\overline{x})$ a následne odsimulujeme daný počet krokov výpočtu M.

1.11.3 Dôsledky vety o primitívne rekurzívnej časovej zložitosti

Ľubovoľný bežný program vieme odsimulovať na Turingovom stroji len s polynomiálnym spomalením. Ľubovoľný Turingov stroj vieme odsimulovať na Minského registrovom stroji s nanajvýš troj-exponenciálnym spomalením (konštrukciu sme mali). Umocnenie je primitívne rekurzívne. Ak teda máme nejaký primitívne rekurzívny horný odhad časovej zložitosti programu v našom bežnom programovacom jazyku, vieme z neho primitívne rekurzívnou funkciou vypočítať aj horný odhad časovej zložitosti jeho simulácie na registrovom stroji.

Analogicky by sa dalo postupovať aj pre ostatné Turingovsky úplné modely. Vetu o primitívne rekurzívnej časovej zložitosti teda môžeme vysloviť aj pre ne.

V trochu zjednodušenej podobe táto veta teda znie: „Ak nejakú funkciu vieš naprogramovať a časovú zložitosť programu zhora odhadnúť nejakou primitívne rekurzívnou funkciou, tak aj funkcia, ktorú tvoj program počíta, je nutne primitívne rekurzívna.“

Spomeňme si teraz na mocninové veže a z nich zostrojiteľné ešte rýchlejšie rastúce funkcie. Všetky tieto funkcie sú primitívne rekurzívne, a teda ich môžeme používať ako odhady časovej zložitosti. No a čokoľvek aspoň trochu prakticky použiteľné má časovú zložitosť lepšiu ako mocninové veže. Ergo honosný záver tejto kapitoly:

*V*šetko rozumne efektívne naprogramovateľné je primitívne rekurzívne.

A ešte úplne posledný dôsledok: Ak vôbec existujú totálne Turingovsky vypočítateľné funkcie, ktoré nie sú primitívne rekurzívne, musí ísť o funkcie, ktoré sa počítajú fakt neuveriteľne dlho.