VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DISERTAČNÍ PRÁCE

k získání akademického titulu Doktor (Ph.D.)

ve studijním programu
INFORMAČNÍ TECHNOLOGIE

Luboš Lorenc

ALTERNATIVNÍ ZPŮSOB PŘEKLADU
FORMÁLNÍ MODELY A OPTIMALIZAČNÍ TECHNIKY

Školitel: prof. RNDr. Alexander Meduna, CSc.

Datum státní doktorské zkoušky: 15. 6. 2004

Datum odevzdání práce: 15. 6. 2006

Práce je k dispozici: .....

Poděkování

Děkuji svému vedoucímu, profesoru Alexandru Medunovi, za jeho ochotu, otevřenost, cenné rady a přátelský přístup, s nímž mě provázel mým doktorským studiem.

Klíčová slova

formální modely, zásobníkové automaty, zásobníkové převodníky, sebereprodukující zásobníkové převodníky, kompilátory, přidělování registrů

Abstrakt

Tato disertační práce se věnuje alternativnímu pohledu na překlad překladu. Sestává z teoretické a praktické části. V teoretické části jsou diskutovány dva formální modely související s překladači. Konkrétně se jedná o sebereprodukující zásobníkové převodníky a omezené dvouzásobníkové automaty. Sebereprodukující zásobníkové převodníky představují zcela nový formální model schopný přijímat i generovat celou třídu rekurzivně spočetných jazyků. Studium dvouzásobníkových automatů je soustředěno na případ, kdy je na běžný dvouzásobníkový automat kladena podmínka, aby rozdíl délek mezi jeho zásobníky v průběhu výpočtu nepřekročil jistou hodnotu k. Při splnění této podmínky klesne síla dvouzásobníkových automatů na sílu ekvivalentní síle běžných zásobníkových automatů.

Ačkoliv je těžištěm práce teoretická část, obsahuje i alternativní pohled na využití optimalizací v kompilátorech. Optimalizační technika původně navržená pro paralelizující kompilátory je použita v sekvenčním kompilátoru. Díky tomu je nutné provést jisté změny v generátoru vnitřního kódu a především vypracovat nový způsob přidělování registrů. Řešení zajímavých oblastí této problematiky včetně popisu použitých algoritmů je součástí praktické části této práce.

Obsah

Část I

Úvod a definice

Kapitola 1

Úvod

V životě se setkáváme s různými problémy spadajícími do různých oblastí. Všechny problémy mají jeden společný rys. Každý z nich lze popsat nějakým jazykem. Některé problémy lze popsat přirozeným jazykem, jiné i jazykem mnohem jednodušším, snadno formálně popsatelným. Pokud se nám daný problém podaří popsat formálním jazykem, je možné pro jeho řešení použít různé algoritmické prostředky či přesněji formální modely.

Formálních modelů byla vytvořena celá řada a liší se rozsahem problémů, které dokáží popsat. Rozsah problémů, jež lze daným formálním modelem popsat přesně odpovídá tomu, s jakou třídou jazyků daný model pracuje. Nejmocnější třídu formálních jazyků tvoří takzvané rekurzivně spočetné jazyky. Tuto třídu jazyků lze přijímat pomocí Turingova stroje. Žádný jiný algoritmicky pracující prostředek není schopen jeho sílu přesáhnout. Většina prakticky používaných formálních modelů má však podstatně nižší sílu. Je to dáno tím, že obtížnost návrhu a vytvoření potřebného modelu značně roste spolu se třídou jazyků, kterou je tento model schopen definovat.

V praxi se tak často setkáváme s modely schopnými definovat třídu bezkontextových jazyků podle Chomského klasifikace. Někdy je tato síla více než dostačující. Jako příklad můžeme uvést lexikální analýzu programovacího jazyka, kdy obvykle vystačíme s regulárními jazyky. O něco obtížnější situace nastává, pokud chceme popsat syntaxi tohoto jazyka. Tento problém je obvykle řešitelný v rámci bezkontextových jazyků. Chceme-li ale obsáhnout i aspekty jako jsou definice dříve dekla-

rovaných proměnných, dostáváme se do oblasti kontextových jazyků a tudíž tento problém nelze řešit žádným modelem definujícím třídu bezkontextových jazyků.

Protože tvorba modelu schopného zvládnout tyto kontextové závislosti je velmi obtížná, řeší se tento problém v praxi použitím slabšího modelu a kontextové závislosti se řeší externě jiným (opět poměrně slabým) formálním modelem. Tento způsob jistě není zcela ideální, proto se neustále objevují snahy o modifikaci současných formálních modelů tak, aby zůstala zachována jejich jednoduchost a současně vzrostla jejich síla, v ideálním případě až na sílu Turingova stroje. Dvěma různými modely, které při zachování jednoduché vnitřní struktury dokáží definovat celou třídu rekurzivně spočetných jazyků, se zabývá druhá část této práce.

Prvním z těchto modelů je sebereprodukující zásobníkový převodník. Tento model vznikl velmi malou modifikací běžného zásobníkového převodníku. Prakticky téměř nemění jeho vnitřní strukturu, zato velmi výrazně zvyšuje jeho sílu — až na úroveň Turingova stroje.

Ne vždy však platí, že model s jednoduchou vnitřní strukturou lze pokaždé jednoduše navrhnout tak, aby přijímal, generoval, či překládal požadovaný jazyk. Bývá obvyklé, že se vzrůstající složitostí požadovaného jazyka neúměrně stoupá rozsáhlost tohoto modelu či složitost jeho specifikace. Pomineme-li nyní složitost specifikace a budeme předpokládat, že dokážeme jistý model specifikovat tak, aby dokázal zpracovávat potřebný jazyk, stále nám zůstává ještě jeden problém. A tím je časová, případně prostorová, náročnost výpočtu. Často se tedy objevují snahy některé modely různým způsobem omezovat tak, aby v průběhu výpočtu neustále splňovaly jisté požadavky.

Jeden takový požadavek položíme na druhý formální model diskutovaný v druhé části práce. Tímto modelem jsou dvouzásobníkové automaty. Jedná se opět o velmi silný model se sílou Turingova stroje. Nás bude zajímat, co se stane s jeho sílou, pokud budeme požadovat, aby v průběhu celého výpočtu nepřesáhl rozdíl délek jeho zásobníků jistou hodnotu.

Ačkoliv těžiště této práce spočívá především v teoretické oblasti, ve třetí části přechází od formálních modelů souvisejících s překladači k jejich praktickému využití, tedy ke skutečné implementaci kompilátoru. Ukážeme si zde některé alternativní přístupy k optimalizaci a jejich využití v jednoduchém kompilátoru. Budeme se zabývat použitím optimalizační metody původně navržené pro paralelizující kompilátory

v běžném jednoduchém sekvenčním kompilátoru. Nejdříve si přiblížíme princip činnosti této metody a poté si popíšeme způsob generování vnitřního kódu v tomto kompilátoru. Protože generovaný kód nesplňuje podmínky nutné pro možnost dělení na základní bloky, budeme se nakonec věnovat i dvěma možným způsobům řešení problémů spojených s přidělováním registrů v tomto kompilátoru. Oba popsané způsoby byly zároveň implementovány a jako dvě samostatné verze překladače jsou přiloženy na doprovodném CD.

Kapitola 2

Definice

Formální modely pro popis jazyků lze zhruba rozdělit na dvě základní oblasti, a to na modely generující jazyky a na modely akceptující jazyky. Co se týče vyjadřovací schopnosti, jsou obě tyto skupiny rovnocenné. Příkladem formálního modelu patřícího do skupiny generativních modelů mohou být gramatiky, které svoji činnost obvykle začínají ve výchozím bodě (obvykle nazývaném startovací nonterminál) a postupnou aplikací přepisovacích pravidel generují řetězec daného jazyka. Svoji činnost končí většinou v okamžiku, kdy již žádný symbol vygenerované věty nelze dále přepisovat.

Typickým zástupcem modelů pro akceptování jazyků jsou automaty. Narozdíl od gramatik, automaty svoji činnost začínají s již existujícím řetězcem na vstupní pásce, postupně jej načítají a na základě úspěšnosti či neúspěšnosti načítání rozhodnou o příslušnosti či nepříslušnosti tohoto řetězce do daného jazyka.

Pro většinu těchto modelů je typické, že každá gramatika či automat definují právě jeden jazyk. Za jistých okolností je však možné oba modely doplnit o možnost generování druhého, takzvaného výstupního, jazyka a tím vytvořit model pro popis vztahu mezi jazyky, tedy pro popis překladu.

Nejpoužívanějším a zřejmě nejpropracovanějším modelem pro popis překladu jsou v současné době *překladové gramatiky*. Jedná se o dvě spojené, většinou bezkontextové, gramatiky, přičemž jedna gramatika generuje vstupní jazyk a druhá gramatika generuje výstupní jazyk. Ekvivalentním modelem k překladové gramatice je *převodník*. Převodník si můžeme představit jako automat, který je rozšířen o schopnost zápisu na výstupní pásku. Zatímco automaty specifikují pouze jazyk,

převodníky, stejně jako překladové gramatiky, specifikují *překlad* — tedy dva jazyky a jednoznačný vztah mezi nimi. V praxi se nejčastěji používají převodníky konečné a zásobníkové. Konečné převodníky jsou ekvivalentní k regulárním překladovým gramatikám a zásobníkové jsou ekvivalentní k bezkontextovým překladovým gramatikám. Především zásobníkové převodníky mají obrovský význam v kompilátorech, kde téměř vždy tvoří základ syntaktického analyzátoru.

V kapitolách 3 a 4 se budeme zabývat zásobníkovými automaty a zásobníkovými převodníky a zejména některými jejich modifikovanými verzemi. Většina modifikací formálních systémů je motivována snahou co nejvíce rozšířit oblast jazyků, které je tento formální model schopen definovat při současném zachování přijatelné složitosti modifikovaného modelu. Nejinak je tomu i u modifikací, kterým se budeme věnovat v těchto kapitolách.

2.1 Pomocné definice

Předpokládá se, že čtenář této práce se orientuje v teoretické informatice a jsou mu známa základní fakta z oblasti formálních jazyků a částečně též překladu a kompilátorů. Nicméně si nyní pro úplnost uvedeme několik definic základních pojmů používaných v této práci. Podrobnější výklad různých pojmů souvisejících s touto oblastí lze nalézt například v [7, 8, 10, 11, 13, 16, 20, 27, 28, 29, 30, 5, 4, 9, 12, 6].

Definice 2.1. Nechť Q je množina, potom $Card(Q)$ označuje kardinalitu (počet prvků) množiny Q.

Definice 2.2. Nechť V je abeceda, potom zápis V^* reprezentuje volný monoid generovaný V nad operací konkatenace. Nulový prvek V^* budeme označovat jako ε.

Jednotlivé prvky množiny V^* budeme dále nazývat řetězce nad abecedou V.

Definice 2.3. Nechť $V^+ = V^* - \{\varepsilon\}$, potom V^+ je volná pologrupa generovaná V nad operací konkatenace.

Definice 2.4. Nechť $w \in V^*$ je řetězec, potom zápisy $|w|$, případně $Len(w)$, označují délku tohoto řetězce. Pro každé $i \in \{0, 1, \ldots, |w|\}$ označuje $Suffix(w, i)$ sufix řetězce w délky i a analogicky $Prefix(w, i)$ označuje prefix řetězce w délky i.

Definice 2.5. Nechť m, n jsou libovolná celá čísla. Funkce $Min(m, n)$ je definována takto: Je-li $m \leq n$, potom $Min(m, n) = m$, je-li $m > n$, potom $Min(m, n) = n$.

Definice 2.6. Nechť m, n jsou libovolná celá čísla. Funkce $Max(m, n)$ je definována takto: Je-li $m \leq n$, potom $Max(m, n) = n$, je-li $m > n$, potom $Max(m, n) = m$.

2.2 Automaty

Tato kapitola podává krátký přehled základních definic z oblasti konečných a zásobníkových automatů jako teoretický základ pro další kapitoly práce. Podrobnější rozbor této rozsáhlé problematiky lze najít například v [20].

2.2.1 Konečné automaty

Konečné automaty definují třídu regulárních jazyků. Konečný automat je formální model sestávající z konečného stavového řízení a vstupní pásky se čtecí hlavou. Automat začíná zpracování vstupního řetězce v počáteční konfiguraci, kdy se čtecí hlava nachází na počátku vstupní pásky a stavové řízení je v počátečním stavu. Automat postupně čte symboly ze vstupní pásky a mění svůj stav. V případě úspěšného přečtení vstupního řetězce ukončí svoji činnost v koncové konfiguraci, kdy vnitřní stav patří do množiny koncových stavů.

Definice 2.7 (Konečný automat). Konečný automat je pětice

$$M = (Q, \Sigma, R, q_0, F)$$

kde

- Q je konečná množina stavů,
- Σ je konečná vstupní abeceda,
- $q_0 \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina koncových stavů,
- R je konečná množina pravidel tvaru $pa \to q$, kde $p, q \in Q$ a $a \in (\Sigma \cup \{\varepsilon\})$.

Definice 2.8 (Konfigurace konečného automatu). Nechť $M = (Q, \Sigma, R, q_0, F)$ je konečný automat. Potom konfigurace M je řetězec qx, kde $q \in Q,\ x \in \Sigma^*$.

Definice 2.9 (Relace přechodu konečného automatu). Nechť $M = (Q, \Sigma, R, q_0, F)$ je konečný automat a pax, qx jsou dvě konfigurace M, kde $p, q \in Q;\ a \in \Sigma;\ x \in \Sigma^*$ a $r = pa \rightarrow q$, $r \in R$ je pravidlo. Potom M provede přechod z konfigurace pax do konfigurace qx podle pravidla r, což zapíšeme jako $pax \Rightarrow qx\ [r]$. Tento zápis můžeme za předpokladu, že tím nevznikne nejednoznačnost, zkrátit na $pax \Rightarrow qx$. Relaci $\Rightarrow$ můžeme ve standardním významu rozšířit na $\Rightarrow^n$, kde $n \geq 0$. Potom na základě $\Rightarrow^n$ definujeme relace $\Rightarrow^+$ jako tranzitivní a $\Rightarrow^*$ jako tranzitivní a reflexivní uzávěr relace $\Rightarrow$.

Definice 2.10 (Jazyk přijímaný konečným automatem). Jazyk přijímaný konečným automatem M označujeme $L(M)$ a je definován $L(M) = \{x \in \Sigma^* \mid q_0 x \Rightarrow^* q_F;\ q_F \in F\}$.

2.2.2 Zásobníkové automaty

Řadu problémů není možné definovat, případně popsat, pomocí regulárních jazyků. To vede k nutnosti použití složitějších jazyků. Vlastní nadtřídou regulárních jazyků jsou jazyky bezkontextové. Tyto jazyky jsou definovány bezkontextovými gramatikami či zásobníkovými automaty. Zásobníkový automat, podobně jako konečný automat, obsahuje konečné stavové řízení a vstupní pásku se čtecí hlavou, ale oproti konečnému automatu je rozšířen o zásobník. Zásobníkový automat začíná zpracování vstupního řetězce v počáteční konfiguraci, kdy se čtecí hlava nachází na začátku vstupní pásky, stavové řízení je v počátečním stavu a na vrcholu zásobníku se nachází počáteční symbol. Automat postupně čte symboly ze vstupní pásky a mění svůj vnitřní stav a vrchol zásobníku. Úspěšné přijetí vstupního řetězce je indikováno buď přechodem stavového řízení do koncového stavu nebo vyprázdněním zásobníku případně přechodem do koncového stavu při současném vyprázdnění zásobníku. Všechny tři uvedené způsoby ukončení činnosti jsou ekvivalentní, co se týče síly automatu. Důkaz ekvivalence způsobů ukončení je poměrně jednoduchý, přesahuje však rámec tohoto úvodu do problematiky.

Definice 2.11 (Zásobníkový automat). Zásobníkový automat je sedmice

$$M = (Q, \Sigma, \Gamma, R, S, q_0, F)$$

kde

- Q je konečná množina stavů,
- Σ je konečná vstupní abeceda,
- Γ je konečná zásobníková abeceda,
- S je počáteční symbol zásobníku,
- $q_0 \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina koncových stavů,
- R je konečná množina pravidel tvaru $upa \to vq$, kde $u \in \Gamma,\ v \in \Gamma^*,\ p, q \in Q$ a $a \in (\Sigma \cup \{\varepsilon\})$.

Definice 2.12 (Konfigurace zásobníkového automatu). Nechť $M = (Q, \Sigma, \Gamma, R, S, q_0, F)$ je zásobníkový automat. Potom konfigurace M je řetězec zqa, kde $z \in \Gamma^*;\ q \in Q$ a $a \in \Sigma^*$.

Definice 2.13 (Relace přechodu zásobníkového automatu). Nechť $M = (Q, \Sigma, \Gamma, R, S, q_0, F)$ je zásobníkový automat a $uvpax$ a $uwqx$ jsou dvě konfigurace M, kde $u \in \Gamma;\ v, w \in \Gamma^*;\ p, q \in Q;\ a \in \Sigma;\ x \in \Sigma^*$ a $r = vpa \to wq,\ r \in R$ je pravidlo. Potom M provede přechod z konfigurace $uvpax$ do konfigurace $uwqx$ podle pravidla r, což zapíšeme jako $uvpax \Rightarrow uwqx\ [r]$. Tento zápis můžeme za předpokladu, že tím nevznikne nejednoznačnost, zkrátit na $uvpax \Rightarrow uwqx$. Relaci $\Rightarrow$ můžeme ve standardním významu rozšířit na $\Rightarrow^n$, kde $n \geq 0$. Potom na základě $\Rightarrow^n$ definujeme relace $\Rightarrow^+$ jako tranzitivní a $\Rightarrow^*$ jako tranzitivní a reflexivní uzávěr relace $\Rightarrow$.

Definice 2.14 (Jazyk přijímaný zásobníkovým automatem). Nechť $M = (Q, \Sigma, \Gamma, R, S, q_0, F)$ je zásobníkový automat. Jazyk přijímaný M označujeme $L(M)$ a je definován $L(M) = \{x \in \Sigma^* \mid Sq_0x \Rightarrow^* vq_F;\ v \in \Gamma^*;\ q_F \in F\}$ pro automat ukončující přechodem do koncového stavu, $L(M) = \{x \in \Sigma^* \mid Sq_0x \Rightarrow^* q;\ q \in Q\}$

pro automat ukončující vyprázdněním zásobníku a $L(M) = \{x \in \Sigma^* \mid Sq_0x \Rightarrow^* q_F;\ q_F \in F\}$ pro automat ukončující přechodem do koncového stavu se současným vyprázdněním zásobníku.

2.3 Převodníky

Tato kapitola, podobně jako předchozí, podává krátký přehled základních definic z oblasti konečných a zásobníkových převodníků jako teoretický základ pro další kapitoly práce. Podrobnější rozbor této rozsáhlé problematiky lze najít například v [20].

2.3.1 Konečné převodníky

Pokud konečný automat rozšíříme o možnost zápisu řetězce na výstupní pásku při každém přechodu, získáme konečný převodník. Konečný převodník, podobně jako konečný automat, začíná zpracování vstupního řetězce v počáteční konfiguraci, kdy se čtecí a zápisová hlava nacházejí na počátcích pásek a stavové řízení je v počátečním stavu. Převodník postupně čte symboly ze vstupní pásky, mění svůj stav a zapisuje řetězce na výstupní pásku. Po úspěšném překladu ukončí svoji činnost po přečtení celé vstupní pásky přechodem do koncové konfigurace, kdy vnitřní stav patří do množiny koncových stavů. Výstupní páska poté obsahuje překlad vstupního řetězce.

Definice 2.15 (Konečný převodník). Konečný převodník je šestice

$$M = (Q, \Sigma, \Delta, R, q_0, F)$$

kde

- Q je konečná množina stavů,
- Σ je konečná vstupní abeceda,
- Δ je konečná výstupní abeceda,
- $q_0 \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina koncových stavů,

- R je konečná množina pravidel tvaru $pa \to qb$, kde $p, q \in Q,\ a \in (\Sigma \cup \{\varepsilon\})$ a $b \in \Delta^*$.

Definice 2.16 (Konfigurace konečného převodníku). Nechť $M = (Q, \Sigma, \Delta, R, q_0, F)$ je konečný převodník a $\$ \notin (\Sigma \cup \Delta)$ je speciální oddělovací symbol. Potom konfigurace M je řetězec $qa\$x$, kde $q \in Q,\ a \in \Sigma^*$ a $x \in \Delta^*$.

Definice 2.17 (Relace přechodu konečného převodníku). Nechť $M = (Q, \Sigma, \Delta, R, q_0, F)$ je konečný převodník a $pax\$z$, $qx\$zb$ jsou dvě konfigurace M, kde $p, q \in Q;\ a \in \Sigma;\ x \in \Sigma^*;\ b \in \Delta;\ z \in \Delta^*$ a $r = pa \to qb$, $r \in R$ je pravidlo. Potom M provede přechod z konfigurace $pax\$z$ do konfigurace $qx\$zb$ podle pravidla r, což zapíšeme jako $pax\$z \Rightarrow qx\$zb\ [r]$. Tento zápis můžeme za předpokladu, že tím nevznikne nejednoznačnost, zkrátit na $pax\$z \Rightarrow qx\$zb$. Relaci $\Rightarrow$ můžeme ve standardním významu rozšířit na $\Rightarrow^n$, kde $n \geq 0$. Potom na základě $\Rightarrow^n$ definujeme relace $\Rightarrow^+$ jako tranzitivní a $\Rightarrow^*$ jako tranzitivní a reflexivní uzávěr relace $\Rightarrow$.

Definice 2.18 (Vstupní a výstupní jazyk konečného převodníku). Vstupní jazyk konečného převodníku M označujeme jako $L_i(M)$ a je definován $L_i(M) = \{x \in \Sigma^* \mid q_0x\$ \Rightarrow^* q_F\$y;\ q_F \in F;\ y \in \Delta^*\}$.
Výstupní jazyk konečného převodníku M označujeme jako $L_o(M)$ a je definován $L_o(M) = \{y \in \Delta^* \mid q_0x\$ \Rightarrow^* q_F\$y;\ q_F \in F;\ x \in \Sigma^*\}$.

Definice 2.19 (Překlad definovaný konečným převodníkem). Překlad definovaný konečným převodníkem M označujeme $\tau(M)$ a je definován $\tau(M) = \{(x, y) \mid x \in \Sigma^*;\ y \in \Delta^*;\ q_0x\$ \Rightarrow^* q_F\$y;\ q_F \in F\}$.

2.3.2 Zásobníkové převodníky

Podobně, jako v případě konečných převodníků, lze o možnost zápisu na výstupní pásku rozšířit i zásobníkový automat. Získáme tak zásobníkový převodník. Podobně jako zásobníkový automat, i zásobníkový převodník začíná zpracování vstupního řetězce v počáteční konfiguraci, kdy se čtecí a zápisová hlava nacházejí na počátcích pásek, stavové řízení je v počátečním stavu a na vrcholu zásobníku se nachází počáteční symbol. Převodník postupně čte symboly ze vstupní pásky, mění svůj vnitřní stav a vrchol zásobníku a současně zapisuje řetězce na výstupní pásku. Při úspěšném

překladu ukončí svoji činnost po přečtení celé vstupní pásky buď přechodem stavového řízení do koncového stavu nebo vyprázdněním zásobníku případně přechodem do koncového stavu při současném vyprázdnění zásobníku. Výstupní páska poté obsahuje překlad vstupního řetězce. Všechny tři uvedené způsoby ukončení činnosti jsou opět ekvivalentní, co se týče síly převodníku.

Definice 2.20 (Zásobníkový převodník). Zásobníkový převodník je osmice

$$M = (Q, \Sigma, \Gamma, \Delta, R, S, q_0, F)$$

kde

- Q je konečná množina stavů,
- Σ je konečná vstupní abeceda,
- Γ je konečná zásobníková abeceda,
- Δ je konečná výstupní abeceda,
- S je počáteční symbol zásobníku,
- $q_0 \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina koncových stavů,
- R je konečná množina pravidel tvaru $upa \rightarrow vqb$, kde $u \in \Gamma,\ v \in \Gamma^*,\ p, q \in Q,\ a \in (\Sigma \cup \{\varepsilon\})$ a $b \in \Delta^*$.

Definice 2.21 (Konfigurace zásobníkového převodníku). Nechť $M = (Q, \Sigma, \Gamma, \Delta, R, S, q_0, F)$ je zásobníkový převodník a $\$ \notin (\Sigma \cup \Gamma \cup \Delta)$ je speciální oddělovací symbol. Potom konfigurace M je řetězec $zqa\$x$, kde $z \in \Gamma^*;\ q \in Q;\ a \in \Sigma^*$ a $x \in \Delta^*$.

Definice 2.22 (Relace přechodu zásobníkového převodníku). Nechť $M = (Q, \Sigma, \Gamma, \Delta, R, S, q_0, F)$ je zásobníkový převodník a $uvpax\$z$ a $uwqx\$zb$ jsou dvě konfigurace M, kde $u \in \Gamma;\ v, w \in \Gamma^*;\ p, q \in Q;\ a \in \Sigma;\ x \in \Sigma^*;\ b \in \Delta;\ z \in \Delta^*$ a $r = vpa \rightarrow wqb, r \in R$ je pravidlo. Potom M provede přechod z konfigurace $uvpax\$z$ do konfigurace $uwqx\$zb$ podle pravidla r, což zapíšeme jako $uvpax\$z \Rightarrow uwqx\$zb\ [r]$.

Tento zápis můžeme za předpokladu, že tím nevznikne nejednoznačnost, zkrátit na $uvpax\$z \Rightarrow uwqx\$zb$. Relaci $\Rightarrow$ můžeme ve standardním významu rozšířit na $\Rightarrow^n$, kde $n \geq 0$. Potom na základě $\Rightarrow^n$ definujeme relace $\Rightarrow^+$ jako tranzitivní a $\Rightarrow^*$ jako tranzitivní a reflexivní uzávěr relace $\Rightarrow$.

Definice 2.23 (Vstupní a výstupní jazyk zásobníkového převodníku). Vstupní jazyk zásobníkového převodníku M označujeme jako $L_i(M)$ a je definován $L_i(M) = \{x \in \Sigma^* \mid Sq_0x\$ \Rightarrow^* vq_F\$y;\ v \in \Gamma^*;\ q_F \in F;\ y \in \Delta^*\}$.
Výstupní jazyk zásobníkového převodníku M označujeme jako $L_o(M)$ a je definován $L_o(M) = \{y \in \Delta^* \mid Sq_0x\$ \Rightarrow^* vq_F\$y;\ v \in \Gamma^*;\ q_F \in F;\ x \in \Sigma^*\}$.

Definice 2.24 (Překlad definovaný zásobníkovým převodníkem). Překlad definovaný zásobníkovým převodníkem M označujeme $\tau(M)$ a je definován $\tau(M) = \{(x, y) \mid x \in \Sigma^*;\ y \in \Delta^*;\ Sq_0x\$ \Rightarrow^* vq_F\$y;\ v \in \Gamma^*;\ q_F \in F\}$

2.4 Frontové gramatiky

Frontové gramatiky byly zavedeny v roce 1983 (viz [14]) a představují speciální druh bezkontextových gramatik, které zpracovávají symboly ze začátku pracovního řetězce a nové symboly generují na jeho konec. S tímto řetězcem tedy pracují jako s frontou. Ke své činnosti navíc využívají i vnitřní stav, který se v každém kroku může měnit. Na rozdíl od běžných bezkontextových gramatik jsou však schopny generovat všechny rekurzivně spočetné jazyky (viz [14]).

Definice 2.25 (Frontová gramatika). Frontová gramatika je šestice

$$Q = (V, T, W, F, s, P)$$

kde

- V je konečná totální abeceda,
- W je konečná množina stavů taková, že $V \cap W = \varnothing$,
- $T \subseteq V$ je abeceda terminálních symbolů,
- $F \subseteq W$ je množina koncových stavů,

- $s \in (V - T)(W - F)$ je počáteční axiom a
- $P \subseteq (V \times (W - F)) \times (V^* \times W)$ je konečná množina přepisovacích pravidel taková, že pro každé $a \in V$ existuje element $(a, b, x, c) \in P$.

Definice 2.26 (Relace derivace frontové gramatiky). Nechť $Q = (V, T, W, F, s, P)$ je frontová gramatika a $u, v \in V^*W$ jsou takové řetězce, že platí $u = arb$; $v = rxc$; $a \in V$; $r, x \in V^*$; $b, c \in W$ a $(a, b, x, c) \in P$. Potom u přímo derivuje v v Q, což zapíšeme $u \Rightarrow v$ $[(a, b, x, c)]$ v G nebo jednoduše, pokud tím nevznikne nejednoznačnost, $u \Rightarrow v$. Relaci $\Rightarrow$ můžeme ve standardním významu rozšířit na $\Rightarrow^n$, kde $n \geq 0$. Potom na základě $\Rightarrow^n$ definujeme $\Rightarrow^+$ jako tranzitivní a $\Rightarrow^*$ jako tranzitivní a reflexivní uzávěr relace $\Rightarrow$.

Definice 2.27 (Jazyk generovaný frontovou gramatikou). Nechť $Q = (V, T, W, F, s, P)$ je frontová gramatika. Jazyk $L(Q)$ generovaný Q je definován $L(Q) = \{w \in T^* : s \Rightarrow^* wf$, kde $f \in F\}$.

Jak již bylo řečeno v úvodu této kapitoly, frontová gramatika pracuje s jedním řetězcem. V každém kroku podle vhodného pravidla zpracuje jeho prefix a (to znamená, že jej z řetězce odstraní) a vygeneruje nový sufix x, který vloží na konec pracovního řetězce. Zároveň při tom provede přechod ze stavu b do stavu c. Derivace řetězce je ukončena v okamžiku, kdy pracovní řetězec neobsahuje žádné nonterminální symboly po dosažení koncového stavu.

Frontová gramatika je formální model, který se výborně hodí pro konstrukci důkazů v oblasti automatů a převodníků. Její nevýhodou pro některé důkazy však je skutečnost, že v průběhu derivování řetězce nikde neuchovává zpracovanou část tohoto řetězce. Proto byla v článku Regulated Pushdown Automata (viz [22]) zavedena *levě rozšířená frontová gramatika*, která uchovává zpracovavaný řetězec.

Definice 2.28 (Levě rozšířená frontová gramatika). Levě rozšířená frontová gramatika je šestice

$$Q = (V, T, W, F, s, P)$$

kde

- V je konečná totální abeceda,
- $T \subseteq V$ je abeceda terminálních symbolů,

- W je konečná množina stavů taková, že $V \cap W = \varnothing$,
- $F \subseteq W$ je množina koncových stavů,
- $s \in (V - T)(W - F)$ je počáteční axiom a
- $P \subseteq (V \times (W - F)) \times (V^* \times W)$ je konečná množina přepisovacích pravidel.

Všimněme si, že na rozdíl od definice frontové gramatiky tato definice nevyžaduje, aby pro každé $a \in V$ existoval element $(a, b, x, c) \in P$.

Definice 2.29 (Relace derivace levě rozšířené frontové gramatiky). Nechť $Q = (V, T, W, F, s, P)$ je levě rozšířená frontová gramatika a $\#$ je speciální oddělovací symbol takový, že $\# \notin V \cup W$. Nechť $u, v \in V^*\{\#\}V^*W$ jsou takové řetězce, že platí $u = w\#arb$; $v = wa\#rxc$; $a \in V$; $r, x, w \in V^*$; $b, c \in W$ a $(a, b, x, c) \in P$, potom u přímo derivuje v v Q, což zapíšeme $u \Rightarrow v\ [(a, b, x, c)]$ nebo jednoduše $u \Rightarrow v$. Relaci $\Rightarrow$ můžeme ve standardním významu rozšířit na $\Rightarrow^n$, kde $n \geq 0$. Potom na základě $\Rightarrow^n$ definujeme $\Rightarrow^+$ jako tranzitivní a $\Rightarrow^*$ jako tranzitivní a reflexivní uzávěr relace $\Rightarrow$.

Definice 2.30 (Jazyk generovaný levě rozšířenou frontovou gramatikou). Nechť $Q = (V, T, W, F, s, P)$ je levě rozšířená frontová gramatika. Jazyk $L(Q)$ generovaný Q je definován $L(Q) = \{v \in T^* : \#s \Rightarrow^* w\#vf$ pro nějaké $w \in V^*$ a $f \in F\}$.

Levě rozšířené frontové gramatiky, podobně jako frontové gramatiky dokáží generovat všechny rekurzivně spočetné jazyky (důkaz viz [22]). Této vlastnosti je využito v některých důkazech v této práci.

Část II

Alternativní formální modely

Kapitola 3

Sebereprodukující zásobníkové převodníky

3.1 Úvod do problematiky

V kapitole 2 jsme si nadefinovali konečné a zásobníkové automaty a převodníky. To jsou běžně používané formální modely pro specifikaci jazyků a překladu. Důležitým faktem však je, že pomocí těchto modelů nejsme schopni přijímat či překládat jazyky ležící mimo množinu regulárních, případně bezkontextových, jazyků. Pro zpracování jazyků ležících mimo tyto množiny sice můžeme použít i silnější formální modely (například Turingovy stroje), ale vytvoření množiny pravidel pro specifikaci konkrétního jazyka je u takovéhoto modelu obecně velmi obtížné.

Motivací pro práci na této kapitole byla snaha o nalezení vhodného formálního modelu, který by dokázal při zachování jednoduchosti zásobníkového převodníku zpracovávat i jazyky ležící mimo třídu bezkontextových jazyků. Z podobně modifikovaných modelů v oblasti zásobníkových automatů, jsou dobře známé dvouzásobníkové automaty či řízené zásobníkové automaty [22, 23]. V oblasti zásobníkových převodníků takovým modelem mohou být například *sebereprodukující zásobníkové převodníky*, které představují pouze malou modifikaci zásobníkových převodníků a dokáží přijímat i generovat celou třídu rekurzivně spočetných jazyků, tedy mají sílu Turingova stroje (viz [24, 25, 19, 18]).

Sebereprodukující zásobníkový převodník vychází ze zásobníkového převodníku a rozšiřuje jej „pouze“ o možnost vícenásobného překladu vstupního řetězce. Až na

několik drobných úprav nezasahuje do vnitřní struktury zásobníkového převodníku a zachovává tak snadnost jeho praktické implementace.

Sebereprodukující zásobníkový převodník překládá vstupní řetězec stejným způsobem jako běžný zásobníkový převodník. Překlad začíná v počáteční konfiguraci, kdy se čtecí a zápisová hlava nacházejí na počátcích pásek, stavové řízení je v počátečním stavu a na vrcholu zásobníku se nachází počáteční symbol. Dále překlad probíhá obvyklým způsobem. Převodník postupně čte symboly ze vstupní pásky, mění svůj vnitřní stav a vrchol zásobníku a zapisuje symboly na výstupní pásku.

Po úspěšném překladu však narozdíl od zásobníkového převodníku nemusí ukončit svoji činnost, ale může obsah výstupní pásky přesunout na vstupní pásku a znovu pokračovat v činnosti. Tento přesun budeme dále nazývat *sebereprodukující krok*. Provedení sebereprodukujícího kroku je podmíněno tím, že stav převodníku v okamžiku dokončení překladu musí patřit do množiny takzvaných *sebereprodukujících stavů*. Dojde-li k úspěšnému zpracování vstupního řetězce (vyprázdnění vstupní pásky) a převodník se nenachází v sebereprodukujícím stavu, je překlad ukončen a na výstupní pásce je zapsán překlad vstupního řetězce.

Definice 3.1 (Sebereprodukující zásobníkový převodník). Sebereprodukující zásobníkový převodník je osmice

$$M = (Q, \Gamma, \Sigma, \Omega, R, s, S, O)$$

kde

- Q je konečná množina stavů,
- Γ je totální abeceda taková, že $Q \cap \Gamma = \varnothing$,
- $\Sigma \subseteq \Gamma$ je vstupní abeceda,
- $\Omega \subseteq \Gamma$ je výstupní abeceda,
- R je konečná množina pravidel tvaru $u_1 q w \to u_2 p v$, kde $u_1, u_2, w, v \in \Gamma^*$ a $q, p \in Q$,
- $s \in Q$ je počáteční stav,
- $S \in \Gamma$ je počáteční symbol zásobníku,

- $O \subseteq Q$ je množina sebereprodukujících stavů.

V definici lze snadno nalézt velkou podobnost se zásobníkovými převodníky definovanými v kapitole 2. Jediný podstatný rozdíl spočívá v zavedení množiny sebereprodukujících stavů, odstranění množiny koncových stavů a sloučení jednotlivých abeced běžného zásobníkového převodníku do jediné totální abecedy. Jistým rozšířením je i možnost čtení řetězce namísto jediného symbolu během jednoho kroku. Jak bude ukázáno později, tato změna je zavedena pouze pro zjednodušení a zpřehlednění důkazů a nemá vliv na vyjadřovací sílu převodníku.

Na obsah zásobníku během provádění sebereprodukujících kroků či po dokončení překladu nejsou kladeny žádné speciální požadavky a může být libovolný. Stejně tak není požadováno ukončení překladu v koncovém stavu. Lze však dokázat, že ke každému sebereprodukujícímu zásobníkovému převodníku lze vytvořit ekvivalentní sebereprodukující zásobníkový převodník, který skončí překlad přechodem do koncového stavu a/nebo vyprázdněním zásobníku.

Věta 3.1. *Nechť $M = (Q, \Gamma, \Sigma, \Omega, R, s, S, O)$ je sebereprodukující zásobníkový převodník. Potom existuje ekvivalentní sebereprodukující zásobníkový převodník $N = (Q, \Gamma, \Sigma, \Omega, R', s, S, O)$, který překlad ukončuje s vyprázdněním zásobníku.*

Důkaz věty 3.1 (nástin). Převodník N vytvoříme jako identickou kopii převodníku M. Poté do množiny stavů R' převodníku N přidáme nový stav a do množiny pravidel přidáme pravidlo umožňující přechod do tohoto nového stavu po dokončení překladu. Následně přidáme do množiny pravidel pravidla umožňující vyprázdnění zásobníku v tomto novém stavu. □

Ukončení překladu přechodem do koncového stavu lze provést obdobně s tím, že je nutné modifikovat definici 3.1 přidáním množiny koncových stavů a nově přidávaný stav poté přidat do množiny koncových stavů.

Definice 3.2 (Konfigurace sebereprodukujícího zásobníkového převodníku). Nechť $\$ \notin (Q \cup \Gamma)$ je speciální oddělovací symbol. Konfigurací sebereprodukujícího zásobníkového převodníku M nazveme libovolný řetězec tvaru $\$zqy\$x$, kde $x, y, z \in \Gamma^*$ a $q \in Q$.

Definice 3.3 (Krok sebereprodukujícího zásobníkového převodníku). Nechť $M = (Q, \Gamma, \Sigma, \Omega, R, s, S, O)$ je sebereprodukující zásobníkový převodník a $u_1qw \rightarrow$

$u_2pv \in R,\ y = \$hu_1qwz\$t$ a $x = \$hu_2pz\$tv$, kde $h, u_1, u_2, w, t, v, z \in \Gamma^*;\ q, p \in Q$. Potom M provádí *překladový krok* z y do x v M, což symbolicky zapíšeme $y\ {}_t\Rightarrow\ x\ [u_1qw \rightarrow u_2pv]$ nebo jednoduše $y\ {}_t\Rightarrow\ x$ v M.

Pokud $y = \$hq\$t$ a $x = \$hqt\$$, kde $t, h \in \Gamma^*,\ q \in O$, potom M provádí *sebereprodukující krok* z y do x v M, což symbolicky zapíšeme $y\ {}_r\Rightarrow\ x$.

Budeme psát $y \Rightarrow x$, pokud $y\ {}_t\Rightarrow\ x$ nebo $y\ {}_r\Rightarrow\ x$. Ve standardním významu rozšíříme $\Rightarrow$ na $\Rightarrow^n$. Potom na základě $\Rightarrow^n$ definujeme pro $n \geq 0$ relace $\Rightarrow^+$ jako tranzitivní a $\Rightarrow^*$ jako tranzitivní a reflexivní uzávěr relace $\Rightarrow^n$.

Definice 3.4 (Překlad definovaný sebereprodukujícím zásobníkovým převodníkem). Nechť $M = (Q, \Gamma, \Sigma, \Omega, R, s, S, O)$ je sebereprodukující zásobníkový převodník a $w, v \in \Gamma^*$. Potom M překládá w na v, pokud $\$Ssw\$ \Rightarrow^* \$q\$v$ v M. Překlad definovaný M označíme $T(M)$ a je definován $T(M) = \{(w, v) \mid \$Ssw\$ \Rightarrow^* \$q\$v$, kde $w \in \Sigma^*;\ v \in \Omega^*;\ q \in Q\}$. Řekneme, že $Domain(T(M)) = \{w \mid (w, x) \in T(M)\}$ a $Range(T(M)) = \{x \mid (w, x) \in T(M)\}$.

Během překladu může sebereprodukující zásobníkový převodník obecně provést libovolný počet sebereprodukujících kroků. Velmi často nás ale bude zajímat, především z hlediska složitosti a efektivity překladu, kolik sebereprodukujících kroků bude během výpočtu maximálně provedeno. Proto zavedeme takzvaný *n-sebereprodukující zásobníkový převodník*, kde n bude nezáporné celé číslo.

Definice 3.5 (*n*-sebereprodukující zásobníkový převodník). Nechť n je nezáporné celé číslo a $M = (Q, \Gamma, \Sigma, \Omega, R, s, S, O)$ je sebereprodukující zásobníkový převodník. Pokud během každého překladu M neprovede více než n sebereprodukujících kroků, potom M nazveme *n-sebereprodukující zásobníkový převodník*.

Velmi často nás též bude zajímat, zda jsou dva sebereprodukující zásobníkové převodníky ekvivalentní, či nikoliv.

Definice 3.6 (Ekvivalence sebereprodukujících zásobníkových převodníků). Dva sebereprodukující zásobníkové převodníky jsou ekvivalentní, pokud oba definují stejný překlad.

V literatuře se můžeme často setkat s požadavkem, aby zásobníkový převodník v každém kroku nahradil maximálně jeden symbol na vrcholu zásobníku a přečetl

maximálně jeden symbol ze vstupní pásky. Nyní si ukážeme, že každý sebereprodukující zásobníkový převodník lze převést na ekvivalentní sebereprodukující zásobníkový převodník splňující tento požadavek.

Věta 3.2. *Nechť $M = (Q, \Gamma, \Sigma, \Omega, R, s, S, O)$ je sebereprodukující zásobníkový převodník. Potom existuje ekvivalentní sebereprodukující zásobníkový převodník $N = (Q, \Gamma, \Sigma, \Omega, R, s, S, O)$, ve kterém každé překladové pravidlo $u_1qw \to u_2pv \in R$, kde $u_1, u_2, w, v \in \Gamma^*$ a $q, p \in Q$, splňuje $|u_1| \leq 1$ a $|w| \leq 1$.*

Důkaz věty 3.2 (nástin). Předpokládejme, že pro každé pravidlo $u_1qw \to u_2pv$ v M platí $|u_1| \geq 2$ nebo $|w| \geq 2$. N simuluje krok podle tohoto pravidla následovně. Nejdříve přejde do nového stavu, potom provede posloupnost $|w|$ kroků, během kterých přečte symbol po symbolu řetězec w. Nyní má přečten celý řetězec w a nachází se v novém stavu $\langle qw \rangle$. Z tohoto stavu provede sekvenci $|u_1|$ kroků, během kterých vyjme ze zásobníku symbol po symbolu řetězec u_1. Nyní má řetězce u_1 a w zaznamenány v novém stavu $\langle u_1qw \rangle$. Poté provede krok $\langle u_1qw \rangle \to u_2pv$, a tím je simulace dokončena. N tedy pracuje stejně jako M. Detailní verze tohoto důkazu je ponechána čtenáři. □

Nyní si ukážeme jednoduchý příklad sebereprodukujícího zásobníkového převodníku.

Příklad 3.1. Nechť $M = (Q, \Gamma, \Sigma, \Omega, R, s, S, O)$ je sebereprodukující zásobníkový převodník takový, že:

- $Q = \{s, q_1, q_2, q_3, q_4\}$,
- $O = \{q_2, q_4\}$,
- $\Sigma = \{a, b, c, d\}$,
- $\Gamma = \{S, a, b, c, d\}$,
- $\Omega = \varnothing$,
- $R = \{Ssx \to Sxs, \quad s \to q_1, \quad xSq_1x \to Sq_1x, \quad Sq_1 \to Sq_2S,$
  $q_2x \to xq_2, \quad q_2S \to q_3, \quad xq_3 \to q_3x, \quad Sq_3 \to q_4,$
  $xq_4x \to q_4\}$, kde $x \in \Sigma$.

Poznámka: Všimněme si, že množina pravidel byla zkrácena použitím zástupného symbolu x pro všechny prvky abecedy Σ. Ve skutečnosti tedy musí být pravidla obsahující symbol x postupně vytvořena se všemi symboly abecedy Σ.

Jazyk přijímaný sebereprodukujícím zásobníkovým převodníkem M je $L(M) = \{yy^R y | y \in \Sigma^*\}$, přičemž y^R představuje reverzi řetězce y. Na první pohled je vidět, že tento jazyk není bezkontextový a tudíž nemůže být přijímán běžným zásobníkovým automatem.

Výpočet pro vstupní řetězec *abcd dcba abcd* probíhá takto:
$\$\ S\ s\ abcd\,dcba\,abcd\ \$\ {}_t\Rightarrow^8\ \$\ abcd\,dcba\,S\ s\ abcd\ \$\ {}_t\Rightarrow\ \$\ abcd\,dcba\,S\ q_1\ abcd\ \$\ {}_t\Rightarrow^4$
$\$\ abcd\,S\ q_1\ \$\ abcd\ {}_t\Rightarrow\ \$\ abcd\,S\ q_2\ \$\ abcd\,S\ {}_r\Rightarrow\ \$\ abcd\,S\ q_2\ abcd\,S\ \$\ {}_t\Rightarrow^4$
$\$\ abcd\,S\,abcd\ q_2\ S\ \$\ {}_t\Rightarrow\ \$\ abcd\,S\,abcd\ q_3\ \$\ {}_t\Rightarrow^4\ \$\ abcd\,S\ q_3\ \$\ dcba\ {}_t\Rightarrow\ \$\ abcd\ q_4\ \$\ dcba$
${}_r\Rightarrow\ \$\ abc\ q_4\,dcba\ \$\ {}_t\Rightarrow^4\ \$\ q_4\ \$$

Z předchozího příkladu je jasně patrné, že pro některé jazyky ležící mimo třídu bezkontextových jazyků je poměrně snadné vytvořit specifikaci sebereprodukujícího zásbníkového převodníku tak, aby tento jazyk přijímal či překládal. Ovšem toto neplatí v obecné rovině. Kromě sebereprodukujících zásobníkových převodníků je v současné době k dispozici řada formálních modelů, které dokáží definovat různé nadtřídy bezkontextových jazyků. Jejich praktickému rozšíření však stále brání to, že tvůrci programů, které by tyto modely mohly využívat, nemají k dispozici obecný postup, jak tyto modely konstruovat. Typickým příkladem mohou být kompilátory. Zde se přímo nabízí využití různých modifikovaných zásobníkových převodníků, které by dokázaly specifikovat alespoň kontextové jazyky. Jelikož však nedokážeme obecným způsobem vytvářet jejich množiny pravidel, jsme nuceni zdrojové texty stále analyzovat jako věty bezkontextových jazyků a kontextové závislosti řešit jinými způsoby, například pomocí tabulek symbolů. Nalezení vhodného obecného způsobu konstrukce takovýchto modelů by jistě znamenalo průlom v konstrukci nejen kompilátorů, ale i v řadě dalších oblastí využívajících zpracování alespoň kontextových jazyků.

3.2 Síla sebereprodukujících zásobníkových převodníků

V této kapitole se budeme věnovat třídě jazyků definované sebereprodukujícími zásobníkovými převodníky. Postupně provedeme důkazy jejich akceptivní a generativní síly. Oba důkazy jsou založeny na využití speciální verze levě rozšířené frontové gramatiky. Pro úplnost byl tedy do této práce začleněn i důkaz, že tyto gramatiky generují všechny rekurzivně spočetné jazyky, i když byl kompletně převzat z [21].

Levě rozšířená frontová gramatika pracuje ve dvou fázích. V první fázi generuje pouze nonterminální symboly, takže pracovní řetězec neobsahuje žádný terminální symbol. V okamžiku, kdy projde přes význačný stav v lemmatu 3.2 označený 1, začne generovat pouze terminální symboly a pokračuje v derivování až do vygenerování řetězce nad abecedou terminálních symbolů.

Lemma 3.1. *Pro každý rekurzivně spočetný jazyk L existuje levě rozšířená frontová gramtika Q taková, že $L(Q) = L$.*

Důkaz lemmatu 3.1. Každý rekurzivně spočetný jazyk je generován frontovou gramatikou (viz [14]). Zároveň ke každé frontové gramatice existuje ekvivalentní levě rozšířená frontová gramatika (viz [22]). Toto lemma tedy platí. □

Lemma 3.2. *Nechť Q' je levě rozšířená frontová gramatika. Potom existuje levě rozšířená frontová gramatika $Q = (V, T, W, F, s, R)$ taková, že $L(Q') = L(Q), W = X \cup Y \cup \{1\}$, kde $X, Y, \{1\}$ jsou po dvojicích disjunktní a každá čtveřice $(a, b, x, c) \in R$ splňuje buď $a \in V - T,\ b \in X,\ x \in (V - T)^*,\ c \in X \cup \{1\}$ nebo $a \in V - T,\ b \in Y \cup \{1\},\ x \in T^*,\ c \in Y$. Q potom generuje každý řetězec $h \in L(Q)$ tímto způsobem:*

$$
\begin{aligned}
&\#a_0q_0 && \\
&\Rightarrow a_0\#x_0q_1 && [(a_0, q_0, z0, q1)] \\
&\Rightarrow a_0a_1\#x_1q_2 && [(a_1, q_1, z1, q2)] \\
&\vdots && \\
&\Rightarrow a_0a_1 \ldots a_k\#x_kq_{k+1} && [(a_k, q_k, z_k, q_{k+1})] \\
&\Rightarrow a_0a_1 \ldots a_ka_{k+1}\#x_{k+1}y_1q_{k+2} && [(a_{k+1}, q_{k+1}, y_1, q_{k+2})] \\
&\vdots &&
\end{aligned}
$$

$$\Rightarrow a_0a_1\dots a_ka_{k+1}\dots a_{k+m-1}\#x_{k+m-1}y_1\dots y_{m-1}q_{k+m} \qquad [(a_{k+m-1}, q_{k+m-1}, y_{m-1}, q_{k+m})]$$

$$\Rightarrow a_0a_1\dots a_ka_{k+1}\dots a_{k+m}\#y_1\dots y_mq_{k+m+1} \qquad [(a_{k+m}, q_{k+m}, y_m, q_{k+m+1})]$$

kde $k, m \geq 1$; $a_i \in V-T$ *pro* $i = 0, \dots, k+m$; $x_j \in (V-T)^*$ *pro* $j = 1, \dots, k+m-1$; $s = a_0q_0$, $a_jx_j = x_{j-1}z_j$ *pro* $j = 1, \dots, k$; $a_1 \dots a_kx_k = z_0 \dots z_k$; $a_{k+1} \dots a_{k+m} = x_k$; $q_0, q_1, \dots q_{k+m} \in W - F$ *pro* $q_{k+m+1} \in F$; $z_1, \dots, z_k \in (V-T)^*$; $y_1, \dots, y_m \in T^*$; $h = y_1y_2 \dots y_{m-1}y_m$; $q_{k+1} \in \{1\}$.

Důkaz lemmatu 3.2 (převzato z [21]). Připoměňme, že každý rekurzivně spočetný jazyk L je generován frontovou gramatikou (viz [14]) a že ke každé frontové gramatice existuje ekvivalentní levě rozšířená frontová gramatika.

Nechť $H = (\varsigma, T, \Omega, \Phi, \sigma, \Pi)$ je levě rozšířená frontová gramatika generující L. Položme $\Omega' = \{q' : q \in \Omega\}$, $\Omega'' = \{q'' : q \in \Omega\}$ a $\varsigma' = \{a' : a \in \varsigma\}$. Definujme bijekci α z Ω do Ω' jako $\alpha(q) = q'$ pro každé $q \in \Omega$. Analogicky definujme bijekci β z Ω do Ω'' jako $\beta(q) = q''$ pro každé $q \in \Omega$. Nakonec zaveďme bijekci δ z ς do ς' jako $\delta(a) = a'$ pro každé $a \in \varsigma$. Ve standardním významu rozšiřme δ tak, aby bylo definováno z ς^* na $(\varsigma')^*$. Položme $U = \{\langle y, p\rangle : y \in T^*,\ p \in \Omega$ a $(a, q, xy, p) \in \Pi$ pro nějaké $a \in \varsigma,\ q \in \Omega,\ x \in \varsigma^*\}$. Bez ztráty obecnosti předpokládejme, že $(\delta(\varsigma) \cup T \cup \alpha(\Omega) \cup \beta(\Omega) \cup U) \cap \{1, f\} = \emptyset$. Položme $V = \delta(\varsigma) \cup \{1\} \cup T$, $W = \alpha(\Omega) \cup \beta(\Omega) \cup \{f\} \cup U$, $F = \{f\}$ a $s = \delta(a)\alpha(q)$, kde $\sigma = aq$.

Definujme levě rozšířenou frontovou gramatiku

$$Q = (V, T, W, F, s, R)$$

s množinou R konstruovanou tímto způsobem:

1. Pokud $(a, q, xy, p) \in \Pi$, kde $a \in \varsigma$, $q \in \Omega - \Phi$, $x, y \in \varsigma^*$ a $p \in \Omega$, potom přidej $(\delta(a), \alpha(q), \delta(x)\delta(y), \alpha(p))$ a $(\delta(a), \alpha(q), \delta(x)1\delta(y), \alpha(p))$ do R,

2. pokud $(a, q, xy, p) \in \Pi$, kde $a \in \varsigma$, $q \in \Omega - \Phi$, $x \in \varsigma^*$, $y \in T^*$, $p \in \Omega$ a $(\langle y, p\rangle \in U)$, potom přidej $(\delta(a), \alpha(q), \delta(x), \langle y, p\rangle)$ a $(1, \langle y, p\rangle, y, \beta(p))$ do R,

3. pokud $(a, q, x, p) \in \Pi$, kde $a \in \varsigma$, $q \in \Omega - \Phi$, $x \in T^*$ a $p \in \Omega$, potom přidej $(\delta(a), \beta(q), x, \beta(p))$ do R,

4. pokud $(a, q, x, p) \in \Pi$, kde $a \in \varsigma$, $q \in \Omega - \Phi$, $x \in T^*$ a $p \in \Phi$, potom přidej $(\delta(a), \beta(q), x, f)$ do R (připomeňme, že $F = \{f\}$),

pro každé $(a, b, c, d) \in R$, $a \in V - T$, $b \in W - F$ a $x \in ((V - T)^* \cup T^*)$.

Rigorózní důkaz, že $L(H) = L(Q)$ ponecháme čtenáři a uvedeme následující nástin.

Abychom ukázali, že $L(H) \subseteq L(Q)$, předpokládejme nějaké $v \in L(H)$. Jelikož $v \in L(H)$, $\#\sigma \Rightarrow^* w\#vt$ v H, $w \in \varsigma^*$, $v \in T^*$ a $t \in \Phi$. Vyjádřeme $\#\sigma \Rightarrow^* w\#vt$ jako $\#\sigma \Rightarrow^* u\#zq \Rightarrow ua\#xyp \Rightarrow^* w\#vt$, kde $a \in \varsigma$, $u, x \in \varsigma^*$, $y = \mathit{Prefix}(v, |y|)$, $z = ax$, $w = uax$ a během $ua\#xyp \Rightarrow^* w\#vt$ jsou generovány pouze terminály, takže výsledný řetězec terminálů je shodný s v. Q simuluje $\#\sigma \Rightarrow^* u\#zq \Rightarrow ua\#xyp \Rightarrow^* w\#vt$ následovně. Nejdříve Q používá pravidla uvedená v bodu 1 konstrukce k simulaci $\#\sigma \Rightarrow^* u\#zq$. Během této iniciální simulace jednou použije pravidlo, které generuje 1 tak, že následně může simulovat $u\#zq \Rightarrow ua\#xyp$ provedením dvou derivačních kroků pomocí pravidel $(\delta(a), \alpha(q), \delta(x), \langle y, p\rangle)$ a $(1, \langle y, p\rangle, y, \beta(p))$ (viz [10]). Připomeňme, že použitím pravidel $(1, \langle y, p\rangle, y, \beta(p))$ produkuje Q řetězec y, který je prefixem v. Po aplikaci $(1, \langle y, p\rangle, y, \beta(p))$ simuluje Q derivaci $ua\#xyp \Rightarrow^* w\#vt$ užitím pravidel vytvořených v kroku 3 konstrukce následovaným jednou aplikací pravidla vytvořeného v kroku 4 konstrukce, čímž dosáhne stavu f, a tím ukončí generování v. Tedy $L(H) \subseteq L(Q)$.

K ustanovení, že $L(Q) \subseteq L(H)$, předpokládejme nějaké $v \in L(Q)$. Jelikož $v \in L(Q)$, $\#s \Rightarrow^* w\#vf$ v Q, kde $w \in V^*$ a $v \in T^*$. Prověříme kroky 1 až 4 konstrukce. Všimněme si, že Q projde stavy z $\alpha(\Omega)$, U, $\beta(\Omega)$ a $\{f\}$ v tomto pořadí, přesněji, vyskytne se několikrát ve stavech $\alpha(\Omega)$, jednou ve stavu U, několikrát v $\beta(\Omega)$ a jednou v f. Výsledkem je, že Q používá pravidla vytvořená v kroku 1 konstrukce a v průběhu této iniciální části derivace právě jednou použije pravidlo, které generuje 1, takže následně může provést dva po sobě jdoucí derivační kroky užitím pravidel $(\delta(a), \alpha(q), \delta(x), \langle y, p\rangle)$ a $(1, \langle y, p\rangle, y, \beta(p))$ (viz [10]). Použitím $(1, \langle y, p\rangle, y, \beta(p))$ produkuje Q řetězec y, který je prefixem v. Po aplikaci $(1, \langle y, p\rangle, y, \beta(p))$ aplikuje Q pravidla vytvořená v kroku 3 konstrukce, která vždy obsahují stavy z $\beta(\Omega)$. Nakonec jednou aplikuje pravidlo vytvořené v kroku 4 konstrukce, čímž dosáhne stavu f a tím ukončí generování v. Abychom tato pozorování shrnuli, můžeme vyjádřit $\#s \Rightarrow^* w\#vf$ v Q jako $\#s \Rightarrow^* u\#zq \Rightarrow ua\#xyp \Rightarrow^* w\#vf$, kde $a \in V$, $x \in V^*$, $y \in T^*$, $w = uax$ tak, že během $\#s \Rightarrow^* u\#zq$ používá Q pravidla vytvořená v kroku 1 konstrukce. Potom aplikuje $(1, \langle y, p\rangle, y, \beta(p))$ z kroku 2 konstrukce k provedení $u\#zq \Rightarrow ua\#xyp$ a nakonec provede $ua\#xyp \Rightarrow^* w\#vf$ pomocí něko-

lika aplikací pravidel vytvořených v kroku 3 konstrukce a jedné aplikace pravidla z kroku 4 konstrukce. Nyní prověřením kroků 1 až 4 konstrukce vidíme, že H provádí $\#\sigma \Rightarrow^* u\#zq \Rightarrow ua\#xyp \Rightarrow^* w\#vt$, kde $t \in \Phi$, takže $v \in L(H)$. Tedy $L(Q) \subseteq L(H)$.

Z $L(H) \subseteq L(Q)$ a $L(Q) \subseteq L(H)$ vyplývá $L(H) = L(Q)$.

□

Přijímající sebereprodukující zásobníkový převodník je sebereprodukující zásobníkový převodník, který začíná výpočet se vstupním řetězcem na vstupní pásce a končí výpočet s prázdnou výstupní páskou. Nyní dokážeme, že přijímající sebereprodukující zásobníkové převodníky jsou schopny přijímat všechny rekurzivně spočetné jazyky.

Lemma 3.3. *Nechť G je levě rozšířená frontová gramatika splňující podmínky uvedené v lemmatu 3.2. Potom existuje 2-sebereprodukující zásobníkový převodník M takový, že Domain$(T(M)) = L(G)$ a Range$(T(M)) = \{\varepsilon\}$.*

Důkaz lemmatu 3.3. Nechť $G = (V, T, W, F, s, P)$ je levě rozšířená frontová gramatika splňující podmínky uvedené v lemmatu 3.2. Bez ztráty obecnosti můžeme předpokládat, že $\{0,1\} \cap (V \cup W) = \varnothing$. Pro libovolné kladné celé číslo n, definujme injektivní zobrazení ι z P na $(\{0,1\}^n - \{1\}^n)$ tak, že ι je injektivní homomorfizmus, pokud je jeho definiční obor rozšířen na $(VW)^*$. Po tomto rozšíření tedy ι reprezentuje injektivní homomorfizmus z $(VW)^*$ na $(\{0,1\}^n - \{1\}^n)^*$. Důkaz, že toto injektivní zobrazení nutně existuje je jednoduchý a je ponechán čtenáři. Nyní na základě ι definujeme substituci ν z V na $(\{0,1\}^n - \{1\}^n)$ tak, že pro každé $a \in V;\ \nu(a) = \{\iota(p) \mid p \in P;\ p = (a, b, x, c)$ pro nějaké $x \in V^*;\ b, c \in W\}$. Definiční obor ν rozšíříme na V^*. Dále definujeme substituci μ z W na $(\{0,1\}^n - \{1\}^n)$ tak, že pro každé $q \in W;\ \mu(q) = \{\iota(p) \mid p \in P;\ p = (a, b, x, c)$ pro nějaké $a \in V;\ x \in V^*;\ b, c \in W\}$. Definiční obor μ opět rozšíříme na W^*.

Konstrukce 3.3.1 (Konstrukce M). Předpokládejme sebereprodukující zásobníkový převodník

$$M = (Q, T \cup \{0, 1, S\}, T, \varnothing, R, z, S, O)$$

kde $Q = \{o, f, z\} \cup \{\langle p, i\rangle \mid p \in W$ a $i \in \{1, 2\}\};\ O = \{o, f\}$ a množina R je vytvořena pomocí následujících šesti kroků.

1. Pokud $a_0 q_0 = s$, kde $a \in V - T$ a $q \in W - F$,
   potom přidej $Sz \to uS\langle q_0, 1\rangle w$ do R pro všechna $w \in \mu(q_0)$ a všechna $u \in \nu(a_0)$;

2. pokud $(a, q, y, p) \in P$, kde $a \in V - T,\ p, q \in W - F$ a $y \in (V - T)^*$,
   potom přidej $S\langle q, 1\rangle \to uS\langle p, 1\rangle w$ do R pro všechna $w \in \mu(p)$ a všechna $u \in \nu(y)$;

3. pro každé $q \in W - F$ přidej $S\langle q, 1\rangle \to S\langle q, 2\rangle$ do R;

4. pokud $(a, q, y, p) \in P$, kde $a \in V - T,\ p, q \in W - F$ a $y \in T^*$,
   potom přidej $S\langle q, 2\rangle y \to S\langle p, 2\rangle w$ do R, pro všechna $w \in \mu(p)$;

5. pokud $(a, q, y, p) \in P$, kde $a \in V - T,\ q \in W - F,\ y \in T^*$ a $p \in F$,
   potom přidej $S\langle q, 2\rangle y \to SoS$ do R;

6. přidej $o0 \to 0o,\ o1 \to 1o,\ oS \to c,\ 0c \to c0,\ 1c \to c1,\ Sc \to f,\ 0f0 \to f,\ 1f1 \to f$ do R.

Nejdříve si uvedeme neformální popis činnosti sebereprodukujícího zásobníkového převodníku vytvořeného v předchozí konstrukci. Sebereprodukující převodník nejprve během $k + 1$ kroků simuluje činnost frontové gramatiky ve fázi generování řetězce nonterminálních symbolů. Všechny nonterminální symboly a vnitřní stavy gramatiky jsou vhodným způsobem zakódovány (pomocí substitucí ν a μ). Kódy nonterminálních symbolů převodník průběžně vkládá na zásobník a kódy vnitřních stavů zapisuje na výstupní pásku. V jistém okamžiku nedeterministicky přejde do další fáze simulace.

Ve druhé fázi simulace sebereprodukující zásobníkový převodník během m kroků simuluje činnost frontové gramatiky ve fázi generování terminálních symbolů tak, že postupně načítá celý vstupní řetězec a na výstupní pásku stále zapisuje kódy vnitřních stavů frontové gramatiky. Obsah zásobníku se již nemění.

Po načtení celého vstupního řetězce se sebereprodukující zásobníkový převodník nachází v konfiguraci, kdy na výstupní pásce má řetězec zakódovaných vnitřních stavů frontové gramatiky a na zásobníku reverzovaný řetězec zakódovaných vygenerovaných nonterminálů. Pokud celý předchozí výpočet proběhl korektně, musí se tyto řetězce shodovat.

Nyní sebereprodukující zásobníkový převodník provede sebereprodukující krok. Tím se výstupní řetězec přesune na vstupní pásku a převodník jej v ι krocích postupně vloží na zásobník. Poté jej opět v ι krocích ze zásobníku přesune na výstupní pásku. Tím se na výstupní pásce objeví jeho reverzovaná verze. Poté stačí provést druhý sebereprodukující krok a porovnat znak po znaku obsah vstupní pásky s obsahem zásobníku.

Z důvodu stručnosti budou v následujících důkazech vynechány některé zřejmé drobnosti, které si čtenář může snadno doplnit.

Tvrzení 3.3.1. *Sebereprodukující zásobníkový převodník M přijímá každý řetězec $h \in L(M)$ tímto způsobem:*

$$
\begin{aligned}
&\$Szy_1y_2\ldots y_{m-1}y_m\$\\
\Rightarrow\quad &\$g_0\langle q_0,1\rangle y_1y_2\ldots y_{m-1}y_m\$t_0\\
\Rightarrow\quad &\$g_1\langle q_1,1\rangle y_1y_2\ldots y_{m-1}y_m\$t_1\\
&\vdots\\
\Rightarrow\quad &\$g_k\langle q_k,1\rangle y_1y_2\ldots y_{m-1}y_m\$t_k\\
\Rightarrow\quad &\$g_k\langle q_k,2\rangle y_1y_2\ldots y_{m-1}y_m\$t_k\\
\Rightarrow\quad &\$g_k\langle q_{k+1},2\rangle y_2y_3\ldots y_{m-1}y_m\$t_{k+1}\\
\Rightarrow\quad &\$g_k\langle q_{k+2},2\rangle y_3\ldots y_{m-1}y_m\$t_{k+2}\\
&\vdots\\
{}_t\Rightarrow\quad &\$g_k\langle q_{k+m},2\rangle y_m\$t_{k+m-1}\\
{}_t\Rightarrow\quad &\$g_kSo\$t_{k+m}S\\
{}_r\Rightarrow\quad &\$g_kSot_{k+m}S\$\\
{}_t\Rightarrow^{\iota}\quad &\$g_kSt_{k+m}oS\$\\
{}_t\Rightarrow\quad &\$g_kSt_{k+m}c\$\\
{}_t\Rightarrow^{\iota}\quad &\$u_1Sc\$v_1\\
{}_t\Rightarrow\quad &\$u_1f\$v_1\\
{}_r\Rightarrow\quad &\$u_1fv_1\$\\
\Rightarrow\quad &\$u_2fv_2\$\\
&\vdots\\
\Rightarrow\quad &\$u_\varpi fv_\varpi\$\\
\Rightarrow\quad &\$f\$
\end{aligned}
$$

v M, kde $k, m \geq 1$; $h = y_1 y_2 \dots y_{m-1} y_m$; $q_0, q_1, \dots, q_{k+m} \in W - F$; $y_1, \dots, y_m \in T^*$; $t_i \in \mu(q_0 q_1 \dots q_i)$ pro $i = 0, 1, \dots, k+m$; $g_j \in \nu(d_0 d_1 \dots d_j)$, přičemž $d_1, \dots, d_j \in (V - T)^*$ pro $j = 0, 1, \dots, k$; $d_0 d_1 \dots d_k = a_0 a_1 \dots a_{k+m}$, kde $a_1, \dots, a_{k+m} \in V - T$, $d_0 = a_0$ a $s = a_0 q_0$; $g_k = t_{k+m}$ (to znamená, že $\nu(a_0 a_1 \dots a_{k+m})$ a $\mu(q_0 q_1 \dots q_{k+m})$ jsou identické); $v_i \in Prefix(\mu(q_0 q_1 \dots q_{k+m}), |\mu(q_0 q_1 \dots q_{k+m})| - i)$ pro $i = 1, \dots, \upsilon$, přičemž $\upsilon = |\mu(q_0 q_1 \dots q_{k+m})|$; $u_j \in Suffix(\nu(a_0 a_1 \dots a_{k+m}), |\nu(a_0 a_1 \dots a_{k+m})| - j)$ pro $j = 1, \dots, \varpi$, kde $\varpi = |\nu(a_0 a_1 \dots a_{k+m})|$; $h = y_1 y_2 \dots y_{m-1} y_m$.

Důkaz tvrzení 3.3.1. Prověříme kroky 1 až 6 konstrukce R. Je nutné předeslat, že při každém úspěšném výpočtu M vždy používá pravidla uvedená v kroku i před tím, než začne používat pravidla uvedená v kroku $i + 1$ pro každé $i = 1, \dots, 5$. Každý úspěšný výpočet $\$Szh\$ \Rightarrow^* \$f\$$ tedy může být vyjádřen takto:

$$
\begin{aligned}
& \$Szy_1y_2 \dots y_{m-1}y_m\$ \\
\Rightarrow\ & \$g_0\langle q_0, 1\rangle y_1y_2 \dots y_{m-1}y_m\$t_0 \\
\Rightarrow\ & \$g_1\langle q_1, 1\rangle y_1y_2 \dots y_{m-1}y_m\$t_1 \\
& \vdots \\
\Rightarrow\ & \$g_k\langle q_k, 1\rangle y_1y_2 \dots y_{m-1}y_m\$t_k \\
\Rightarrow\ & \$g_k\langle q_k, 2\rangle y_1y_2 \dots y_{m-1}y_m\$t_k \\
\Rightarrow\ & \$g_k\langle q_{k+1}, 2\rangle y_2 \dots y_{m-1}y_m\$t_{k+1} \\
\Rightarrow\ & \$g_k\langle q_{k+2}, 2\rangle y_3 \dots y_{m-1}y_m\$t_{k+2} \\
\Rightarrow\ & \$g_k\langle q_{k+3}, 2\rangle y_4 \dots y_{m-1}y_m\$t_{k+3} \\
& \vdots \\
{}_t\Rightarrow\ & \$g_k\langle q_{k+m}, 2\rangle y_m\$t_{k+m-1} \\
{}_t\Rightarrow\ & \$g_kSo\$t_{k+m}S \\
\Rightarrow^*\ & \$f\$
\end{aligned}
$$

kde $k, m \geq 1$; $h = y_1 y_2 \dots y_{m-1} y_m$; $q_0, q_1, \dots, q_{k+m} \in W - F$; $y_1, \dots, y_m \in T^*$; $t_i \in \mu(q_0 q_1 \dots q_i)$ pro $i = 0, 1, \dots, k+m$; $g_j \in \nu(d_0 d_1 \dots d_j)$, přičemž $d_1, \dots, d_j \in (V-T)^*$ pro $j = 0, 1, \dots, k$; $d_0 d_1 \dots d_k = a_0 a_1 \dots a_{k+m}$, kde $a_1, \dots, a_{k+m} \in V - T$, $d_0 = a_0$ a $s = a_0 q_0$.

Během $\$g_kSo\$t_{k+m}S \Rightarrow^* \$f\$$ jsou používána pouze pravidla z kroku šest, která jsou:

$$o0 \to 0o,\ o1 \to 1o,\ oS \to c,\ 0c \to c0,\ 1c \to c1,\ Sc \to f,\ 0f0 \to f,\ 1f1 \to f$$

Všimněme si, že k získání $\$f\$$ z $\$g_kSo\$t_{k+m}S$ pomocí těchto pravidel M provede $\$g_kSo\$t_{k+m}S \Rightarrow^* \$f\$$ takto:

$\$g_kSo\$t_{k+m}S$
$\underset{r}{\Rightarrow}\ \$g_kSot_{k+m}S\$$
$\underset{t}{\Rightarrow}^{\iota}\ \$g_kSt_{k+m}oS\$$
$\underset{t}{\Rightarrow}\ \$g_kSt_{k+m}c\$$
$\underset{t}{\Rightarrow}^{\iota}\ \$u_1Sc\$v_1$
$\underset{t}{\Rightarrow}\ \$u_1f\$v_1$
$\underset{r}{\Rightarrow}\ \$u_1fv_1\$$
$\Rightarrow\ \$u_2fv_2\$$
$\vdots$
$\Rightarrow\ \$u_\varpi fv_\varpi\$$
$\Rightarrow\ \$f\$$

v M, kde $g_k = t_{k+m}$; $v_i \in \mathit{Prefix}(\mu(q_0q_1 \dots q_{k+m}),\ |\mu(q_0q_1 \dots q_{k+m})| - i)$ pro $i = 1, \dots, \upsilon$, přičemž $\upsilon = |\mu(q_0q_1 \dots q_{k+m})|$; $u_j \in \mathit{Suffix}(\nu(a_0a_1 \dots a_{k+m}),\ |\nu(a_0a_1 \dots \dots a_{k+m})| - j)$ pro $j = 1, \dots, \varpi$, kde $\varpi = |\nu(a_0a_1 \dots a_{k+m})|$. Tento výpočet implikuje $g_k = t_{k+m}$. Tvrzení 3.3.1 tedy platí. □

Nechť M přijímá každý řetězec $h \in L(M)$ způsobem uvedeným v tvrzení 3.3.1. Zaměřme se nyní na konstrukci R a všimněme si, že P obsahuje $(a_0, q_0, z_0, q_1), \dots \dots, (a_k, q_k, z_k, q_{k+1}),\quad (a_{k+1}, q_{k+1}, y_1, q_{k+2}), \dots, (a_{k+m-1}, q_{k+m-1}, y_{m-1}, q_{k+m}), (a_{k+m}, q_{k+m}, y_m, q_{k+m+1})$, kde $z_1, \dots, z_k \in (V - T)^*$. G tedy generuje každý řetězec h způsobem popsaným v lemmatu 3.2. Tedy $h \in L(G)$. Následně tedy $L(M) \subseteq L(G)$.

Nechť G generuje $h \in L(G)$ způsobem popsaným v lemmatu 3.2. Potom M přijímá h způsobem popsaným v tvrzení 3.3.1, tedy $L(G) \subseteq L(M)$ (detailní důkaz je ponechán čtenáři).

Jelikož $L(M) \subseteq L(G)$ a zároveň $L(G) \subseteq L(M)$, tak $L(G) = L(M)$.

Z tvrzení 3.3.1 vyplývá, že M nikdy neprovede více než dva sebereprodukující kroky, tedy M je 2-sebereprodukující zásobníkový převodník. Z uvedeného vyplývá, že lemma 3.3 platí. □

Generující sebereprodukující zásobníkový převodník je sebereprodukující zásobníkový převodník, který začíná výpočet s prázdnou vstupní páskou a končí výpočet s vygenerovaným řetězcem na výstupní pásce. Nyní dokážeme, že generující sebereprodukující zásobníkové převodníky jsou schopny generovat všechny rekurzivně spočetné jazyky.

Lemma 3.4. *Nechť G je levě rozšířená frontová gramatika splňující podmínky uvedené v lemmatu 3.2. Potom existuje 2-sebereprodukující zásobníkový převodník M takový, že $Domain(T(M)) = \{\varepsilon\}$ a $Range(T(M)) = L(G)$.*

Důkaz lemmatu 3.4. Nechť Q je levě rozšířená frontová gramatika splňující podmínky uvedené v lemmatu 3.2.

Konstrukce 3.4.1 (Konstrukce M). Předpokládejme sebereprodukující zásobníkový převodník

$$M = (Q, V \cup \{S\}, \varnothing, T, R, z, S, O)$$

kde $Q = z \cup \{\langle p, i\rangle \mid p \in W, i \in \{1, 11\}\} \cup \{\langle 1, i\rangle \mid i \in \{1, \ldots, 11\}\}$, $O = \{\langle 1, 3\rangle, \langle 1, 8\rangle\}$ a R je vytvořena provedením následujících kroků 1 až 6.

1. Pokud $y_s q_0 = s$, kde $y \in V - T$ a $q \in W - F$,
   potom přidej $Sz \to SS\langle q_0, 1\rangle Sy_s$ do R.

2. Pokud $(a, q, y, p) \in P$, kde $a \in V - T,\ p, q \in W - F$, $y \in (V - T)^*$ a $1 \in W$,
   potom přidej $S\langle q, 1\rangle \to aS\langle p, 1\rangle y$ do R.

3. Přidej následující pravidla pro každé $a \in V - T$ do R:

   $S\langle 1, 1\rangle \to \langle 1, 2\rangle S$, $a\langle 1, 2\rangle \to \langle 1, 2\rangle a$, $S\langle 1, 2\rangle \to \langle 1, 3\rangle S$, $\langle 1, 3\rangle S \to S\langle 1, 4\rangle$,
   $\langle 1, 4\rangle a \to a\langle 1, 4\rangle$, $\langle 1, 4\rangle S \to \langle 1, 5\rangle S$, $a\langle 1, 5\rangle \to \langle 1, 5\rangle a$, $S\langle 1, 5\rangle \to S\langle 1, 6\rangle$,
   $\langle 1, 6\rangle a \to a\langle 1, 6\rangle$, $\langle 1, 6\rangle S \to \langle 1, 7\rangle S$, $a\langle 1, 7\rangle \to \langle 1, 7\rangle a$, $S\langle 1, 7\rangle \to \langle 1, 8\rangle S$,
   $\langle 1, 8\rangle S \to S\langle 1, 9\rangle$, $\langle 1, 9\rangle a \to a\langle 1, 9\rangle$, $\langle 1, 9\rangle S \to \langle 1, 10\rangle$, $a\langle 1, 10\rangle a \to \langle 1, 10\rangle$.

4. Pokud $(a, q, y, p) \in P$, kde $a \in V - T,\ p = 1, q \in W - F$ a $y \in T^*$,
   potom přidej $a\langle 1, 10\rangle S \to \langle p, 11\rangle y$ do R.

5. Pokud $(a, q, y, p) \in P$, kde $a \in V - T,\ p, q \in W - F$ a $y \in T^*$, potom přidej $a\langle q, 11\rangle \rightarrow \langle p, 11\rangle y$ do R.

6. Pokud $(a, q, y, p) \in P$, kde $a \in V - T,\ q \in W - F,\ y \in T^*$ a $p \in F$, potom přidej $Sa\langle q, 11\rangle \rightarrow \langle p, 11\rangle y$ do R.

Před vlastním důkazem si opět nejprve neformálně popíšeme činnost sebereprodukujícího zásobníkového převodníku vytvořeného v předcházející konstrukci. Tato činnost opět sestává z několika fází.

V první fázi seberepodukující zásobníkový převodník opět simuluje generování řetězce nonterminálů ve frontové gramatice. Tentokrát však vygenerované nonterminály nekóduje, ale přímo je zapisuje na výstupní pásku. Zároveň si na zásobník ukládá nonterminály, které levě rozšířená frontová gramatika zpracovává. Vnitřní stavy frontové gramatiky jsou mapovány přímo na stavy převodníku.

V okamžiku, kdy frontová gramatika přejde do fáze generování terminálních symbolů, provede sebereprodukující zásobníkový převodník kontrolu shody řetězce zpracovaných nonterminálů na zásobníku s částí řetězce vygenerovaných nonterminálů na výstupní pásce. Tato kontrola je nutná, protože narozdíl od frontové gramatiky sebereprodukující zásobníkový převodník nemůže průběžně jednoduše zpracovávat svůj výstup, pokud pomineme možnost použití obrovského počtu sebereprodukujících kroků.

Kontrola řetězců využívá dva sebereprodukující kroky, neboť je opět nutné řetězce reverzovat a navíc ještě mezi sebou vyměnit tak, aby po kontrole shodnosti a odstranění řetězce zpracovaných nonterminálů z řetězce vygenerovaných nonterminálů byl zbytek tohoto řetězce uložen na zásobníku ve správném pořadí.

Pokud kontrola proběhla vpořádku, je na zásobníku stejný řetězec nonterminálních symbolů, jaký frontová gramatika použila pro generování řetězce terminálních symbolů. V této fázi tudíž stačí přímou simulací činnosti frontové gramatiky (nonterminály se čtou ze zásobníku a vnitřní stavy frontové gramatiky jsou mapovány na stavy převodníku) vygenerovat řetězec teminálních symbolů.

Z důvodu stručnosti budou v následujících důkazech vynechány některé zřejmé drobnosti, které si čtenář může snadno doplnit.

Tvrzení 3.4.1. *M generuje každý řetězec $h \in L(M)$ takto:*

$$
\begin{aligned}
&\$Sz\$\\
{}_t\Rightarrow\ &\$SS\langle q_0,1\rangle\$Sy_s\\
{}_t\Rightarrow\ &\$Sa_0S\langle q_1,1\rangle\$Sy_sy_0\\
{}_t\Rightarrow\ &\$Sa_0a_1S\langle q_2,1\rangle\$Sy_sy_0y_1\\
&\ \ \vdots\\
{}_t\Rightarrow\ &\$Sa_0a_1\ldots a_{k-1}a_kS\langle q_{k+1},1\rangle\$Sy_sy_0y_1\ldots y_k\\
{}_t\Rightarrow\ &\$Sa_0a_1\ldots a_{k-1}a_k\langle 1,2\rangle\$Sy_sy_0y_1\ldots y_kS\\
{}_t\Rightarrow\ &\$Sa_0a_1\ldots a_{k-1}\langle 1,2\rangle\$Sy_sy_0y_1\ldots y_kSa_k\\
&\ \ \vdots\\
{}_t\Rightarrow\ &\$Sa_0\langle 1,2\rangle\$Sy_sy_0y_1\ldots y_kSa_k\ldots a_1\\
{}_t\Rightarrow\ &\$S\langle 1,2\rangle\$Sy_sy_0y_1\ldots y_kSa_k\ldots a_1a_0\\
{}_t\Rightarrow\ &\$\langle 1,3\rangle\$Sy_sy_0y_1\ldots y_kSa_k\ldots a_1a_0S\\
{}_r\Rightarrow\ &\$\langle 1,3\rangle Sy_sy_0y_1\ldots y_kSa_k\ldots a_1a_0S\$\\
{}_t\Rightarrow\ &\$S\langle 1,4\rangle y_sy_0y_1\ldots y_kSa_k\ldots a_1a_0S\$\\
{}_t\Rightarrow\ &\$Sy_s\langle 1,4\rangle y_0y_1\ldots y_kSa_k\ldots a_1a_0S\$\\
{}_t\Rightarrow\ &\$Sy_sy_0\langle 1,4\rangle y_1\ldots y_kSa_k\ldots a_1a_0S\$\\
&\ \ \vdots\\
{}_t\Rightarrow\ &\$Sy_sy_0\ldots y_{k-1}y_k\langle 1,4\rangle Sa_k\ldots a_1a_0S\$\\
{}_t\Rightarrow\ &\$Sy_sy_0\ldots y_{k-1}y_k\langle 1,5\rangle a_k\ldots a_1a_0S\$S\\
{}_t\Rightarrow\ &\$Sy_sy_0\ldots y_{k-1}\langle 1,5\rangle a_k\ldots a_1a_0S\$Sy_k\\
{}_t\Rightarrow\ &\$Sy_sy_0\ldots y_{k-2}\langle 1,5\rangle a_k\ldots a_1a_0S\$Sy_ky_{k-1}\\
&\ \ \vdots\\
{}_t\Rightarrow\ &\$S\langle 1,5\rangle a_k\ldots a_1a_0S\$Sy_ky_{k-1}\ldots y_0y_s\\
{}_t\Rightarrow\ &\$S\langle 1,6\rangle a_k\ldots a_1a_0S\$Sy_ky_{k-1}\ldots y_0y_s\\
{}_t\Rightarrow\ &\$Sa_k\langle 1,6\rangle a_{k-1}\ldots a_1a_0S\$Sy_ky_{k-1}\ldots y_0y_s\\
{}_t\Rightarrow\ &\$Sa_ka_{k-1}\langle 1,6\rangle a_{k-2}\ldots a_1a_0S\$Sy_ky_{k-1}\ldots y_0y_s\\
&\ \ \vdots\\
{}_t\Rightarrow\ &\$Sa_ka_{k-1}\ldots a_1a_0\langle 1,6\rangle S\$Sy_ky_{k-1}\ldots y_0y_s\\
{}_t\Rightarrow\ &\$Sa_ka_{k-1}\ldots a_1a_0\langle 1,7\rangle\$Sy_ky_{k-1}\ldots y_0y_sS\\
{}_t\Rightarrow\ &\$Sa_ka_{k-1}\ldots a_1\langle 1,7\rangle\$Sy_ky_{k-1}\ldots y_0y_sSa_0\\
{}_t\Rightarrow\ &\$Sa_ka_{k-1}\ldots a_2\langle 1,7\rangle\$Sy_ky_{k-1}\ldots y_0y_sSa_0a_1\\
&\ \ \vdots
\end{aligned}
$$

$$
\begin{aligned}
&{}_t\Rightarrow\ \$S\langle 1,7\rangle\$Sy_ky_{k-1}\dots y_0y_sSa_0a_1\dots a_{k-1}a_k\\
&{}_t\Rightarrow\ \$\langle 1,8\rangle\$Sy_ky_{k-1}\dots y_0y_sSa_0a_1\dots a_{k-1}a_kS\\
&{}_r\Rightarrow\ \$\langle 1,8\rangle Sy_ky_{k-1}\dots y_0y_sSa_0a_1\dots a_{k-1}a_kS\$\\
&{}_t\Rightarrow\ \$S\langle 1,9\rangle y_ky_{k-1}\dots y_0y_sSa_0a_1\dots a_{k-1}a_kS\$\\
&{}_t\Rightarrow\ \$Sy_k\langle 1,9\rangle y_{k-1}\dots y_0y_sSa_0a_1\dots a_{k-1}a_kS\$\\
&{}_t\Rightarrow\ \$Sy_ky_{k-1}\langle 1,9\rangle y_{k-2}\dots y_0y_sSa_0a_1\dots a_{k-1}a_kS\$\\
&\quad\vdots\\
&{}_t\Rightarrow\ \$Sy_ky_{k-1}\dots y_0y_s\langle 1,9\rangle Sa_0a_1\dots a_{k-1}a_kS\$\\
&{}_t\Rightarrow\ \$Sx_{m+k}\dots x_1x_0\langle 1,10\rangle a_0a_1\dots a_{k-1}a_kS\$\\
&{}_t\Rightarrow\ \$Sx_{m+k}\dots x_1\langle 1,10\rangle a_1\dots a_{k-1}a_kS\$\\
&\quad\vdots\\
&{}_t\Rightarrow\ \$Sx_{m+k}\dots x_{k+1}x_k\langle 1,10\rangle a_kS\$\\
&{}_t\Rightarrow\ \$Sx_{m+k}\dots x_{k+1}\langle 1,10\rangle S\$\\
&{}_t\Rightarrow\ \$Sx_{m+k}\dots x_{k+2}\langle q_{k+2},11\rangle\$w_1\\
&{}_t\Rightarrow\ \$Sx_{m+k}\dots x_{k+3}\langle q_{k+3},11\rangle\$w_1w_2\\
&\quad\vdots\\
&{}_t\Rightarrow\ \$Sx_{m+k}\langle q_{k+m},11\rangle\$w_1w_2\dots w_{m-1}\\
&{}_t\Rightarrow\ \$\langle q_{k+m+1},11\rangle\$w_1w_2\dots w_{m-1}w_m
\end{aligned}
$$

v M*, kde* $k,m\geq 1$; $y_s,y_0,\dots,y_k\in(V-T)^*$; $a_0,\dots,a_k\in(V-T)^*$; $s=y_sq_0$; $q_0,\dots,q_k\in X$, $q_{k+1}=1$, $q_{k+2},\dots,q_{k+m}\in Y$, $q_{k+m+1}\in F$; $y_k\dots y_0y_s=x_{m+k}\dots x_0$; $w_1\dots w_m=h$.

Důkaz tvrzení 3.4.1. Prověříme kroky 1 až 6 konstrukce M. Připomeňme, že v průběhu každého úspěšného výpočtu předchází použití pravidel uvedených v kroku i použití pravidel uvedených v kroku $i+1$, pro $i=1,\dots,6$. Tedy detailněji, každý úspěšný výpočet $\$Sz\$\Rightarrow^*\$\langle q_{k+m+1},11\rangle\$h$ může být rozdělen do tří hlavních fází.

Během první fáze výpočtu ($\$Sz\$\Rightarrow^*\$Sa_0a_1\dots a_{k-1}a_kS\langle q_{k+1},1\rangle\$Sy_sy_0y_1\dots y_k$) jsou používána pouze pravidla z kroků 1 a 2 konstrukce.

$$
\begin{aligned}
&\$Sz\$\\
&{}_t\Rightarrow\ \$SS\langle q_0,1\rangle\$Sy_s\\
&{}_t\Rightarrow\ \$Sa_0S\langle q_1,1\rangle\$Sy_sy_0\\
&{}_t\Rightarrow\ \$Sa_0a_1S\langle q_2,1\rangle\$Sy_sy_0y_1
\end{aligned}
$$

$$
\begin{aligned}
&\vdots\\
{}_t\Rightarrow\ & \$Sa_0a_1\dots a_{k-1}a_kS\langle q_{k+1},1\rangle\$Sy_sy_0y_1\dots y_k
\end{aligned}
$$

v M, kde $k \geq 1$; $y_s, y_0, \dots, y_k \in (V-T)^*$; $a_0, \dots, a_k \in (V-T)^*$; $s = y_sq_0$; $q_0, \dots, q_k \in X$, $q_{k+1} = 1$. Připomeňme, že ekvivalence $q_{k+1} = 1$ je základní požadavek plynoucí z lemmatu 3.2.

Ve druhé fázi výpočtu ($\$Sa_0a_1\dots a_{k-1}a_kS\langle 1,1\rangle\$Sy_sy_0y_1\dots y_k \Rightarrow \$Sx_{m+k}\dots x_{k+2}\langle q_{k+2},11\rangle\$w_1$) jsou používána pouze pravidla z kroku 3 konstrukce a nakonec je použito jedno pravidlo z kroku 4 konstrukce.

$$
\begin{aligned}
&\$Sa_0a_1\dots a_{k-1}a_kS\langle 1,1\rangle\$Sy_sy_0y_1\dots y_k\\
{}_t\Rightarrow\ & \$Sa_0a_1\dots a_{k-1}a_k\langle 1,2\rangle\$Sy_sy_0y_1\dots y_kS\\
{}_t\Rightarrow\ & \$Sa_0a_1\dots a_{k-1}\langle 1,2\rangle\$Sy_sy_0y_1\dots y_kSa_k\\
&\vdots\\
{}_t\Rightarrow\ & \$Sa_0\langle 1,2\rangle\$Sy_sy_0y_1\dots y_kSa_k\dots a_1\\
{}_t\Rightarrow\ & \$S\langle 1,2\rangle\$Sy_sy_0y_1\dots y_kSa_k\dots a_1a_0\\
{}_t\Rightarrow\ & \$\langle 1,3\rangle\$Sy_sy_0y_1\dots y_kSa_k\dots a_1a_0S\\
{}_r\Rightarrow\ & \$\langle 1,3\rangle Sy_sy_0y_1\dots y_kSa_k\dots a_1a_0S\$\\
{}_t\Rightarrow\ & \$S\langle 1,4\rangle y_sy_0y_1\dots y_kSa_k\dots a_1a_0S\$\\
{}_t\Rightarrow\ & \$Sy_s\langle 1,4\rangle y_0y_1\dots y_kSa_k\dots a_1a_0S\$\\
{}_t\Rightarrow\ & \$Sy_sy_0\langle 1,4\rangle y_1\dots y_kSa_k\dots a_1a_0S\$\\
&\vdots\\
{}_t\Rightarrow\ & \$Sy_sy_0\dots y_{k-1}y_k\langle 1,4\rangle Sa_k\dots a_1a_0S\$\\
{}_t\Rightarrow\ & \$Sy_sy_0\dots y_{k-1}y_k\langle 1,5\rangle a_k\dots a_1a_0S\$S\\
{}_t\Rightarrow\ & \$Sy_sy_0\dots y_{k-1}\langle 1,5\rangle a_k\dots a_1a_0S\$Sy_k\\
{}_t\Rightarrow\ & \$Sy_sy_0\dots y_{k-2}\langle 1,5\rangle a_k\dots a_1a_0S\$Sy_ky_{k-1}\\
&\vdots\\
{}_t\Rightarrow\ & \$S\langle 1,5\rangle a_k\dots a_1a_0S\$Sy_ky_{k-1}\dots y_0y_s\\
{}_t\Rightarrow\ & \$S\langle 1,6\rangle a_k\dots a_1a_0S\$Sy_ky_{k-1}\dots y_0y_s\\
{}_t\Rightarrow\ & \$Sa_k\langle 1,6\rangle a_{k-1}\dots a_1a_0S\$Sy_ky_{k-1}\dots y_0y_s\\
{}_t\Rightarrow\ & \$Sa_ka_{k-1}\langle 1,6\rangle a_{k-2}\dots a_1a_0S\$Sy_ky_{k-1}\dots y_0y_s\\
&\vdots\\
{}_t\Rightarrow\ & \$Sa_ka_{k-1}\dots a_1a_0\langle 1,6\rangle S\$Sy_ky_{k-1}\dots y_0y_s
\end{aligned}
$$

$$
\begin{aligned}
&{}_t\Rightarrow\ \$Sa_ka_{k-1}\dots a_1a_0\langle 1,7\rangle\$Sy_ky_{k-1}\dots y_0y_sS\\
&{}_t\Rightarrow\ \$Sa_ka_{k-1}\dots a_1\langle 1,7\rangle\$Sy_ky_{k-1}\dots y_0y_sSa_0\\
&{}_t\Rightarrow\ \$Sa_ka_{k-1}\dots a_2\langle 1,7\rangle\$Sy_ky_{k-1}\dots y_0y_sSa_0a_1\\
&\quad\vdots\\
&{}_t\Rightarrow\ \$S\langle 1,7\rangle\$Sy_ky_{k-1}\dots y_0y_sSa_0a_1\dots a_{k-1}a_k\\
&{}_t\Rightarrow\ \$\langle 1,8\rangle\$Sy_ky_{k-1}\dots y_0y_sSa_0a_1\dots a_{k-1}a_kS\\
&{}_r\Rightarrow\ \$\langle 1,8\rangle Sy_ky_{k-1}\dots y_0y_sSa_0a_1\dots a_{k-1}a_kS\$\\
&{}_t\Rightarrow\ \$S\langle 1,9\rangle y_ky_{k-1}\dots y_0y_sSa_0a_1\dots a_{k-1}a_kS\$\\
&{}_t\Rightarrow\ \$Sy_k\langle 1,9\rangle y_{k-1}\dots y_0y_sSa_0a_1\dots a_{k-1}a_kS\$\\
&{}_t\Rightarrow\ \$Sy_ky_{k-1}\langle 1,9\rangle y_{k-2}\dots y_0y_sSa_0a_1\dots a_{k-1}a_kS\$\\
&\quad\vdots\\
&{}_t\Rightarrow\ \$Sy_ky_{k-1}\dots y_0y_s\langle 1,9\rangle Sa_0a_1\dots a_{k-1}a_kS\$\\
&{}_t\Rightarrow\ \$Sx_{m+k}\dots x_1x_0\langle 1,10\rangle a_0a_1\dots a_{k-1}a_kS\$\\
&{}_t\Rightarrow\ \$Sx_{m+k}\dots x_1\langle 1,10\rangle a_1\dots a_{k-1}a_kS\$\\
&\quad\vdots\\
&{}_t\Rightarrow\ \$Sx_{m+k}\dots x_{k+1}x_k\langle 1,10\rangle a_kS\$\\
&{}_t\Rightarrow\ \$Sx_{m+k}\dots x_{k+1}\langle 1,10\rangle S\$\\
&{}_t\Rightarrow\ \$Sx_{m+k}\dots x_{k+2}\langle q_{k+2},11\rangle\$w_1
\end{aligned}
$$

v M, kde $k,m\geq 1$; $y_s,y_0,\dots,y_k\in(V-T)^*$; $a_0,\dots,a_k\in(V-T)^*$; $q_{k+2}\in Y$, $y_k\dots y_0y_s=x_{m+k}\dots x_0$; $w_1\in T^*$.

Připomeňme, že řetězec $y_s,y_0,\dots,y_k$ představuje celý řetězec nonterminálních symbolů vygenerovaný frontovou gramatikou a řetězec $a_0a_1\dots a_{k-1}a_k$ představuje jeho prefix zpracovaný během generování řetězce nonterminálů.

Ve třetí fázi výpočtu ($\$Sx_{m+k}\dots x_{k+2}\langle q_{k+2},11\rangle\$w_1\Rightarrow\$\langle q_{k+m+1},11\rangle\$w_1w_2\dots w_{m-1}w_m$) sebereprodukující zásobníkový převodník simuluje přepisování zbývající části řetězce $y_s,y_0,\dots,y_k$, tedy jeho suffix $x_0\dots x_{m+k}$, frontovou gramatikou na řetězec terminálních symbolů. Jsou používána pouze pravidla z kroků 5 a 6 konstrukce.

$$
\begin{aligned}
&\$Sx_{m+k}\dots x_{k+2}\langle q_{k+2},11\rangle\$w_1\\
&{}_t\Rightarrow\ \$Sx_{m+k}\dots x_{k+3}\langle q_{k+3},11\rangle\$w_1w_2\\
&\quad\vdots\\
&{}_t\Rightarrow\ \$Sx_{m+k}\langle q_{k+m},11\rangle\$w_1w_2\dots w_{m-1}
\end{aligned}
$$

$_t\Rightarrow$ $\$\langle q_{k+m+1}, 11\rangle\$w_1w_2\dots w_{m-1}w_m$

v M, kde $k, m \geq 1$; $q_{k+3}, \dots, q_{k+m} \in Y$; $q_{k+m+1} \in F$; $x_{m+k}\dots x_{k+3} \in V - T$; $w_1\dots w_m = h$.

Z konstrukce pravidel vyplývá, že M musí během výpočtu projít odpovídajícími stavy jako frontová gramatika. Dodržení této podmínky je zaručeno kontrolou zpracovaných nonterminálů ve druhé fázi a konstrukcí pravidel třetí fáze (kroky 4 a 5 konstrukce). Z toho vyplývá, že tvrzení 3.4.1 platí. □

Nechť M generuje každý řetězec $h \in L(M)$ způsobem uvedeným v tvrzení 3.4.1. Zaměřme se nyní na konstrukci R a všimněme si, že P obsahuje $(a_0, q_0, z_0, q_1), \dots$ $\dots, (a_k, q_k, z_k, q_{k+1}), \quad (a_{k+1}, q_{k+1}, y_1, q_{k+2}), \dots, (a_{k+m-1}, q_{k+m-1}, y_{m-1}, q_{k+m}), (a_{k+m},$ $q_{k+m}, y_m, q_{k+m+1})$, kde $z_1, \dots, z_k \in (V - T)^*$. G tedy generuje každý řetězec h způsobem popsaným v lemmatu 3.2. Tedy $h \in L(G)$. Následně tedy $L(M) \subseteq L(G)$.

Nechť G generuje $h \in L(G)$ způsobem popsaným v lemmatu 3.2. Potom M generuje h způsobem popsaným v tvrzení 3.4.1, tedy $L(G) \subseteq L(M)$ (detailní důkaz je ponechán čtenáři).

Jelikož $L(M) \subseteq L(G)$ a zároveň $L(G) \subseteq L(M)$, tak $L(G) = L(M)$.

Z tvrzení 3.4.1 vyplývá, že M nikdy neprovede více než dva sebereprodukující kroky, tedy M je 2-sebereprodukující zásobníkový převodník. Z uvedeného vyplývá, že lemma 3.4 platí. □

Věta 3.3. *Pro každý rekurzivně spočetný jazyk L existuje 2-sebereprodukující zásobníkový převodník M takový, že $Domain(T(M)) = L$ a $Range(T(M)) = \{\varepsilon\}$ nebo $Domain(T(M)) = \{\varepsilon\}$ a $Range(T(M)) = L$.*

Důkaz věty 3.3. Platnost této věty vyplývá z lemmat 3.1, 3.2, 3.3 a 3.4. □

3.3 Shrnutí dosažených výsledků

Sebereprodukující zásobníkové převodníky představují poměrně jednoduchý formální model téměř identický se známými a velmi dobře zvládnutými zásobníkovými převodníky. Narozdíl od zásobníkových převodníků však definují třídu rekurzivně

spočetných jazyků — jinými slovy, jsou schopny přijímat a generovat všechny rekurzivně spočetné jazyky. Této síly dosahují již s provedením maximálně dvou sebereprodukujících kroků v průběhu celého výpočtu.

Kapitola 4

Omezené dvouzásobníkové automaty

Zásobníkové automaty jsou jedním z nejoblíbenějších modelů definujících třídu bezkontextových jazyků. Proto jsou již velmi dlouho intezivně zkoumány a různě modifikovány. Jednu z možností modifikace či přesněji rozšíření představuje i přidání druhého zásobníku do vnitřní struktury automatu. Dostaneme tak dvouzásobníkový automat. Dvouzásobníkové automaty jsou známy již od roku 1965, kdy je zavedl Juris Hartmanis. Přidáním druhého zásobníku se zvýší síla tohoto automatu až na úroveň Turingova stroje. Je zajímavé, že tato síla zůstane zachována i pokud na automat začneme klást jisté, na první pohled poměrně omezující, požadavky. Jedním z těchto požadavků může být například počet obrátek na zásobnících.

Obrátkou se u libovolného zásobníkového automatu rozumí krok výpočtu, během nějž se po předchozím prodlužování poprvé zkrátí délka zásobníku.

Je dokázáno (viz [21]), že současně jednoobrátkové zásobníkové automaty mají sílu Turingova stroje. Tedy jejich síla zůstává zachována, i když požadujeme, že v průběhu celého výpočtu musí platit, že oba zásobníky se musí poprvé zkrátit současně v jediném kroku a poté se již nikdy nesmí prodloužit.

V této kapitole se podíváme, co se stane se sílou dvouzásobníkových automatů v případě jiného požadavku. Zvolený požadavek je jednoduchý. Rozdíl délek zásobníků nesmí v průběhu celého výpočtu přesáhnout libovolné celé číslo k. Na první pohled se tento požadavek v porovnání s požadavkem na jedinou obrátku v obou zásobnících najednou může zdát benevolentní a tudíž snadno splnitelný. My však

dokážeme, že takto jednoduchý požadavek zcela degraduje obrovskou sílu dvouzásobníkových automatů na úroveň jednozásobníkových automatů, tedy takto omezený dvouzásobníkový automat opět definuje pouze třídu bezkontextových jazyků.

Definice 4.1 (Dvouzásobníkový automat). Dvouzásobníkový automat je osmice

$$M = (Q, \Sigma, \Gamma, R, z, Z_1, Z_2, F)$$

kde

- $Q \cap (\Sigma \cup \Gamma) = \emptyset$ je konečná množina stavů,
- Σ je konečná vstupní abeceda,
- Γ je konečná zásobníková abeceda,
- $z \in Q$ je počáteční stav,
- $Z_1 \in \Gamma$ je počáteční symbol zásobníku 1,
- $Z_2 \in \Gamma$ je počáteční symbol zásobníku 2,
- R je konečná množina pravidel tvaru $u_1|u_2qw \to v_1|v_2p$, kde $u_1, u_2 \in \Gamma$, $v_1, v_2 \in \Gamma^*$, $q, p \in Q$ a $w \in \Sigma^*$.

Opět si můžeme všimnout že oproti základní definici zásobníkového automatu je tento automat rozšířen o možnost čtení řetězce ze vstupní pásky. Toto rozšíření nemá vliv na sílu automatu, což lze dokázat obdobným způsobem, jakým byla dokázána věta 3.2.

Definice 4.2 (Konfigurace dvouzásobníkového automatu). Nechť $M = (Q, \Sigma, \Gamma, R, z, Z_1, Z_2, F)$ je dvouzásobníkový automat. Potom konfigurace M je řetězec $\$v_1\$v_2qy$, kde $v_1, v_2 \in \Gamma^*$, $y \in \Sigma^*$, $q \in Q$ a \$ je oddělovací symbol takový, že $\$ \notin Q \cup \Sigma \cup \Gamma$.

Definice 4.3 (Relace přechodu dvouzásobníkového automatu). Nechť $M = (Q, \Sigma, \Gamma, R, z, Z_1, Z_2, F)$ je dvouzásobníkový automat, $\$h_1u_1\$h_2u_2qwz$ a $\$h_1v_1\$h_2v_2pz$, kde $u_1, u_2 \in \Gamma$, $h_1, h_2, v_1, v_2 \in \Gamma^*$, $q, p \in Q$ a $w, z \in \Sigma^*$ jsou dvě konfigurace M a $r = u_1|u_2qw \to v_1|v_2p$, $r \in R$ je pravidlo. Potom M provede přechod

z konfigurace $\$h_1u_1\$h_2u_2qwz$ do konfigurace $\$h_1v_1\$h_2v_2pz$ podle pravidla r, což zapíšeme jako $\$h_1u_1\$h_2u_2qwz \Rightarrow \$h_1v_1\$h_2v_2pz\ [r]$. Tento zápis můžeme za předpokladu, že tím nevznikne nejednoznačnost, zkrátit na $\$h_1u_1\$h_2u_2qwz \Rightarrow \$h_1v_1\$h_2v_2pz$. Relaci $\Rightarrow$ můžeme ve standardním významu rozšířit na $\Rightarrow^n$, kde $n \geq 0$. Potom na základě $\Rightarrow^n$ definujeme relace $\Rightarrow^+$ jako tranzitivní a $\Rightarrow^*$ jako tranzitivní a reflexivní uzávěr relace $\Rightarrow$.

Definice 4.4 (Jazyk přijímaný dvouzásobníkovým automatem). Nechť $M = (Q, \Sigma, \Gamma, R, z, Z_1, Z_2, F)$ je dvouzásobníkový automat. Jazyk přijímaný M označujeme $L(M)$ a je definován $L(M) = \{w \in \Sigma^* : \$Z_1\$Z_2zw \Rightarrow^* \$\$f,\ f \in F\}$.

Jak bylo uvedeno v úvodu této kapitoly, položíme na dvouzásobníkový automat podmínku, aby maximální rozdíl délek jeho zásobníků v průběhu výpočtu nepřekročil jistou hodnotu k. Potom dokážeme, že takto omezený dvouzásobníkový automat má stejnou sílu jako zásobníkový automat rozšířený o možnost čtení řetězců.

Lemma 4.1. *Nechť M je dvouzásobníkový automat a nechť existuje konstanta $k \geq 0$ taková, že pro libovolnou konfiguraci M, $\$u\$vqx$ platí $|\,Len(u) - Len(v)| \leq k$. Potom existuje zásobníkový automat N takový, že $L(M) = L(N)$.*

Důkaz lemmatu 4.1. Nechť $c, d \in \Gamma^*$ jsou řetězce tvaru $c = c_1c_2 \ldots c_nc_{n+1} \ldots c_l$ a $d = d_1d_2 \ldots d_nd_{n+1} \ldots d_m$. Definujme zobrazení ν takové, že:

- Pokud platí $Len(c) = Len(d)$, potom $\nu(c, d) = \langle c_1, d_1\rangle\langle c_2, d_2\rangle \ldots \langle c_l, d_m\rangle$.
- Pokud platí $Len(c) > Len(d)$, potom $\nu(c, d) = \langle c_1, d_1\rangle \ldots \langle c_m, d_m\rangle\langle c_{m+1}, \#\rangle \ldots$ $\ldots \langle c_l, \#\rangle$.
- Pokud platí $Len(c) < Len(d)$, potom $\nu(c, d) = \langle c_1, d_1\rangle \ldots \langle c_l, d_l\rangle\langle \#, d_{l+1}\rangle \ldots$ $\ldots \langle \#, d_m\rangle$.

Nechť $M = (Q, \Sigma, \Gamma, R, z, Z_1, Z_2, F)$ je dvouzásobníkový automat. Vytvoříme zásobníkový automat $M' = (Q', \Sigma, \Gamma', R', z, Z, F)$, kde $Q \subseteq Q'$, $\Gamma' \subset (\Gamma \cup \{\#\})^2$, $Z = \langle Z_1, Z_2\rangle$ a R' a Q' jsou zkonstruovány následujícím způsobem:

Konstrukce 4.1.1. Nechť i je libovolné kladné celé číslo. Pro každé pravidlo $a|bpx \to c|dq \in R$, kde $a, b \in \Gamma,\ c, d \in \Gamma^*,\ p, q \in Q,\ x \in \Sigma^*$, $c = c_1c_2 \ldots c_nc_{n+1} \ldots c_l$ a $d = d_1d_2 \ldots d_nd_{n+1} \ldots d_m$ proveď akci z 1 a všechny akce z 2 a 3:

1. • Přidej $\langle a, b\rangle px \to \nu(c, d)q$ do R'.

2. • Přidej $\langle a, \#\rangle p \to \langle p_{a,b,10}, \varepsilon\rangle$ do R' a $\langle p_{a,b,10}, \varepsilon\rangle$ do Q',
   • přidej $\langle \alpha, \#\rangle\langle p_{a,b,10}, \psi\rangle \to \langle p_{a,b,10}, \psi\alpha\rangle$ do R' a $\langle p_{a,b,10}, \psi\alpha\rangle$ do Q' pro každé $\alpha \in \Sigma$ a pro každé $\psi \in \Delta^*$, kde $Len(\psi) < i$,
   • přidej $\langle \alpha, b\rangle\langle p_{a,b,10}, \psi\rangle \to \langle \alpha, \#\rangle\langle p_{a,b,11}, \psi\rangle$ do R' a $\langle p_{a,b,11}, \psi\rangle$ do Q' pro každé $\alpha \in (\Sigma \cup \varepsilon)$ a pro každé $\psi \in \Delta^*$, kde $Len(\psi) \leq i$,
   • přidej $\langle p_{a,b,11}, \psi\alpha\rangle \to \langle \alpha, \#\rangle\langle p_{a,b,11}, \psi\rangle$ do R' a $\langle p_{a,b,11}, \psi\rangle$ do Q' pro každé $\alpha \in (\Sigma \cup \varepsilon)$ a pro každé $\psi \in \Delta^*$, kde $Len(\psi) < i$,
   • přidej $\langle p_{a,b,11}, \varepsilon\rangle \to \langle a, b\rangle p$ do R'.

3. • Přidej $\langle \#, b\rangle p \to \langle p_{a,b,20}, \varepsilon\rangle$ do R' a $\langle p_{a,b,20}, \varepsilon\rangle$ do Q',
   • přidej $\langle \#, \beta\rangle\langle p_{a,b,20}, \psi\rangle \to \langle p_{a,b,20}, \psi\beta\rangle$ do R' a $\langle p_{a,b,20}, \psi\beta\rangle$ do Q' pro každé $\beta \in \Sigma$ a pro každé $\psi \in \Delta^*$, kde $Len(\psi) < i$,
   • přidej $\langle a, \beta\rangle\langle p_{a,b,20}, \psi\rangle \to \langle \#, \beta\rangle\langle p_{a,b,21}, \psi\rangle$ do R' a $\langle p_{a,b,21}, \psi\rangle$ do Q' pro každé $\beta \in (\Sigma \cup \varepsilon)$ a pro každé $\psi \in \Delta^*$, kde $Len(\psi) \leq i$,
   • přidej $\langle p_{a,b,21}, \psi\beta\rangle \to \langle \#, \beta\rangle\langle p_{a,b,21}, \psi\rangle$ do R' a $\langle p_{a,b,21}, \psi\rangle$ do Q' pro každé $\beta \in (\Sigma \cup \varepsilon)$ a pro každé $\psi \in \Delta^*$, kde $Len(\psi) < i$,
   • přidej $\langle p_{a,b,21}, \varepsilon\rangle \to \langle a, b\rangle p$ do R'.

Následně pro každé $q \in Q$ přidej pravidlo $\langle \#, \#\rangle q \to q$ do R'.

Zásobníkový automat M' vytvořený v konstrukci 4.1.1 simuluje činnost dvouzásobníkového automatu M s využitím jediného zásobníku. Zásobníková abeceda Γ' je proto nyní tvořena dvojicemi symbolů abecedy Γ. Do těchto dvojic jsou vkládány současně jednotlivé znaky obou zásobníků automatu M. V případě, že některé původní pravidlo automatu M zapisuje na jeden zásobník delší řetězec než na druhý, jsou v nových symbolech zapisovaných na zásobník tyto chybějící symboly nahrazeny symbolem #. V každé konfiguraci automatu M', kdy je nutné simulovat čtení ze zásobníku a na vrcholu není k dispozici symbol obsahující oba původní symboly, je postupně načteno maximálně i symbolů ze zásobníku a tyto jsou uchovávány v nově vytvořených stavech. Poté je vyzvednut symbol, který již obsahuje druhý původní symbol a informace uchované ve stavech jsou vráceny zpět na zásobník. Tím se na

vrchol zásobníku dostane symbol odpovídající vrcholům obou zásobníků automatu M v dané konfiguraci. Simulace tedy může pokračovat dalším krokem.

Důkaz korektnosti této konstrukce sestává z několika částí. Nejdříve dokážeme, že automat M' korektně simuluje první krok automatu M. Potom dokážeme, že automat M' správně simuluje libovolný krok automatu M z konfigurace, ve které je rozdíl mezi délkami jeho zásobníků maximálně i symbolů. Nakonec na základě pravdivosti těchto tvrzení poté dokážeme, že automat M' simuluje celý výpočet prováděný automatem M a z vlastností zobrazení ν nakonec dokážeme, že je tato konstrukce korektní.

Z důvodu stručnosti budou v následujících důkazech vynechány některé zřejmé drobnosti, které si čtenář může snadno doplnit.

Tvrzení 4.1.1. *Nechť $\zeta = \$Z_1\$Z_2zax$, kde $a, x \in \Sigma^*$ je počáteční konfigurace M a $r = Z_1|Z_2za \to v_1|v_2q_1$ je pravidlo z R. Potom M provádí krok $\$Z_1\$Z_2zax \Rightarrow \$v_1\$v_2q_1x[r]$, který M' simuluje jako $\langle Z_1, Z_2\rangle zax \Rightarrow \nu(v_1, v_2)q_1x[r']$, kde $r' \in R'$.*

Důkaz tvrzení 4.1.1. Platnost tohoto tvrzení vyplývá z konstrukce 4.1.1. Pro simulaci přechodu z počáteční konfigurace v M může být v M' použito pouze odpovídající pravidlo z bodu 1 konstrukce 4.1.1. □

Tvrzení 4.1.2. *Nechť $\eta = vpx$, kde $p \in Q$ je libovolná nepočáteční konfigurace M'. Potom existuje konfigurace $\eta' = v'px$ taková, že $v' = \varepsilon$ nebo $\mathit{Suffix}(v', 1) = \langle w_0, y_0\rangle$, kde $(w_0 \in \Gamma,\ y_0 \in \Gamma \cup \{\#\})$ nebo $(w_0 \in \Gamma \cup \{\#\},\ y_0 \in \Gamma)$.*

Důkaz tvrzení 4.1.2. Platnost tohoto tvrzení vyplývá z tvaru pravidel vytvořených v konstrukci 4.1.1. Prověřme pravidla, podle kterých může být proveden přechod do konfigurace η.

- Pravidla posledních bodů skupiny 2 a 3 této konstrukce mají tvar $\langle p_{a,b,11}, \varepsilon\rangle \to \langle a, b\rangle p$, případně $\langle p_{a,b,11}, \varepsilon\rangle \to \langle a, b\rangle p$, kde $a, b \in \Gamma$. Tedy konfigurace η po provedení přechodu pomocí těchto pravidel splňuje podmínky kladené na konfiguraci η', tudíž $\eta' = \eta$.

- Pravidla první skupiny této konstrukce mají tvar $\langle a, b\rangle px \to \nu(c, d)q$, přičemž z tvaru pravidla a definice zobrazení ν vyplývá, že $\mathit{Suffix}(\nu(c, d)) = \langle w_0, y_0\rangle$, přičemž platí právě jedna z následujících možností:

  - $(w_0 \in \Gamma,\ y_0 \in \Gamma \cup \{\#\})$ — konfigurace η splňuje podmínky tvrzení pro konfiguraci η', proto $\eta' = \eta$;
  - $(w_0 \in \Gamma \cup \{\#\},\ y_0 \in \Gamma)$ — konfigurace η splňuje podmínky tvrzení pro konfiguraci η', proto $\eta' = \eta$;
  - $w_0 = \varepsilon,\ y_0 = \varepsilon$ — přechodem do konfigurace η došlo k vyjmutí vrcholu zásobníku bez následného zápisu. $\mathit{Suffix}(v, 1) = \langle w_0, y_0 \rangle$, kde $w_0, y_0 \in \Gamma \cup \{\#\}$.

- Aplikací pravidel tvaru $\langle \#, \# \rangle q \to q$ do R' lze provést výpočet $\eta \Rightarrow^n \eta'$, kde $\eta = vpx$, $\eta' = v'px$ a $\mathit{Prefix}(\mathit{Suffix}(v, j), 1) = \langle \#, \# \rangle$ pro každé $1 \leq j \leq n$.

Jiná pravidla umožňující přechod do konfigurace η konstrukce 4.1.1 neobsahuje. Tvrzení 4.1.2 tedy platí. □

Tvrzení 4.1.3. *Nechť $\zeta = \$u_1\$u_2pex$, $u_1 \neq Z_1$, $u_2 \neq Z_2$ je libovolná nepočáteční konfigurace automatu M a $r = a|bpe \to c|dq$ je pravidlo z R. Nechť $\eta = upex$ je konfigurace automatu M', kde $u \in \Gamma'^*$ a $u = \langle w_h, y_h \rangle \langle w_{h-1}, y_{h-1} \rangle \ldots \langle w_1, y_1 \rangle \langle w_0, y_0 \rangle$. Nechť zároveň platí $\mathit{Suffix}(w_h \ldots w_1 w_0, s) = \alpha\#^*$, pro nějaké s, $1 \leq s \leq i$ a současně platí $\mathit{Suffix}(y_h \ldots y_1 y_0, t) = \beta\#^*$, pro nějaké t, $1 \leq t \leq i$, kde i je konstanta zvolená v konstrukci 4.1.1 a $\alpha, \beta \in \Gamma$. Potom M provádí krok $\$u_1\$u_2pex \Rightarrow \$v_1\$v_2qx$, který M' simuluje pomocí sekvence maximálně $\mathit{Max}(s, t) + 4$ kroků $upex \Rightarrow^+ vqx$.*

Důkaz tvrzení 4.1.3. Podle tvrzení 4.1.2 každá konfigurace $\eta = upex$ automatu M', kde $p \in Q$ buď přímo splňuje podmínku, že $u = \varepsilon$ nebo $\mathit{Suffix}(u, 1) = \langle w_0, y_0 \rangle$, kde $(w_0 \in \Gamma,\ y_0 \in \Gamma \cup \{\#\})$ nebo $(w_0 \in \Gamma \cup \{\#\},\ y_0 \in \Gamma)$, případně existuje konfigurace η' splňující tyto podmínky taková, že $\eta \Rightarrow^* \eta'$. Pokud tedy pomineme případ, kdy $u = \varepsilon$, což je nutně koncová konfigurace, mohou s a t nabývat následujících hodnot:

1. $s = t = 1$, tedy $w_0 \in \Gamma,\ y_0 \in \Gamma$: V této konfiguraci lze aplikovat pouze odpovídající pravidla první skupiny konstrukce 4.1.1. Automat M' tak užitím pravidla $\langle a, b \rangle pe \to \nu(c, d)q$ pomocí jediného kroku simuluje jeden krok automatu M.

2. $s = 1,\ t > 1$, tedy $w_0 \in \Gamma,\ y_0 = \#$: V této konfiguraci lze aplikovat pouze pravidla druhé skupiny a následně jedno pravidlo první skupiny konstrukce 4.1.1.

Automat M' simuluje jeden krok $\zeta \Rightarrow \zeta'[r]$ automatu M následujícím způsobem:

$$
\begin{aligned}
&\langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_t, y_t\rangle \ldots \langle w_1, \#\rangle\langle w_0, \#\rangle pex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_t, y_t\rangle \ldots \langle w_1, \#\rangle\langle p_{a,b,10}, \varepsilon\rangle ex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_t, y_t\rangle \ldots \langle w_2, \#\rangle\langle p_{a,b,10}, w_1\rangle ex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_t, y_t\rangle \ldots \langle w_3, \#\rangle\langle p_{a,b,10}, w_1w_2\rangle ex \\
&\quad\vdots \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_t, y_t\rangle\langle w_{t-1}, \#\rangle\langle p_{a,b,10}, w_1w_2 \ldots w_{t-2}\rangle ex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_t, y_t\rangle\langle p_{a,b,10}, w_1w_2 \ldots w_{t-2}w_{t-1}\rangle ex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_t, \#\rangle\langle p_{a,b,11}, w_1w_2 \ldots w_{t-2}w_{t-1}\rangle ex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_t, \#\rangle\langle w_{t-1}, \#\rangle\langle p_{a,b,11}, w_1w_2 \ldots w_{t-2}\rangle ex \\
&\quad\vdots \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_t, \#\rangle\langle w_{t-1}, \#\rangle \ldots \langle w_2, \#\rangle\langle p_{a,b,11}, w_1\rangle ex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_t, \#\rangle \ldots \langle w_2, \#\rangle\langle w_1, \#\rangle\langle p_{a,b,11}\rangle ex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_t, \#\rangle \ldots \langle w_2, \#\rangle\langle w_1, \#\rangle\langle a, b\rangle pex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_t, \#\rangle \ldots \langle w_2, \#\rangle\langle w_1, \#\rangle\nu(c, d)qx
\end{aligned}
$$

kde $w_0 = a$, $y_t = b$, $e \in \Sigma$, $x \in \Sigma^*$, $p, q \in Q$.

3. $s > 1,\ t = 1$, tedy $w_0 = \#,\ y_0 \in \Gamma$: V této konfiguraci lze aplikovat pouze pravidla třetí skupiny a následně jedno pravidlo první skupiny konstrukce 4.1.1. Automat M' simuluje jeden krok $\zeta \Rightarrow \zeta'[r]$ automatu M následujícím způsobem:

$$
\begin{aligned}
&\langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_s, y_s\rangle \ldots \langle \#, y_1\rangle\langle \#, y_0\rangle pex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_s, y_s\rangle \ldots \langle \#, y_1\rangle\langle p_{a,b,20}, \varepsilon\rangle ex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_s, y_s\rangle \ldots \langle \#, y_2\rangle\langle p_{a,b,20}, y_1\rangle ex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_s, y_s\rangle \ldots \langle \#, y_3\rangle\langle p_{a,b,20}, y_1y_2\rangle ex \\
&\quad\vdots \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_s, y_s\rangle\langle \#, y_{s-1}\rangle\langle p_{a,b,20}, y_1y_2 \ldots y_{s-2}\rangle ex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle w_s, y_s\rangle\langle p_{a,b,20}, y_1y_2 \ldots y_{s-2}y_{s-1}\rangle ex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle \#, y_s\rangle\langle p_{a,b,21}, y_1y_2 \ldots y_{s-2}y_{s-1}\rangle ex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle \#, y_s\rangle\langle \#, y_{s-1}\rangle\langle p_{a,b,21}, y_1y_2 \ldots y_{s-2}\rangle ex
\end{aligned}
$$

$$
\begin{aligned}
&\vdots \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle \#, y_s\rangle\langle \#, y_{s-1}\rangle \ldots \langle \#, y_2\rangle\langle p_{a,b,21}, y_1\rangle ex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle \#, y_s\rangle \ldots \langle \#, y_2\rangle\langle \#, y_1\rangle\langle p_{a,b,21}\rangle ex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle \#, y_s\rangle \ldots \langle \#, y_2\rangle\langle \#, y_1\rangle\langle a, b\rangle pex \\
&\Rightarrow \langle w_h, y_h\rangle\langle w_{h-1}, y_{h-1}\rangle \ldots \langle \#, y_s\rangle \ldots \langle \#, y_2\rangle\langle \#, y_1\rangle\nu(c, d)qx
\end{aligned}
$$

kde $w_s = a$, $y_0 = b$, $e \in \Sigma$, $x \in \Sigma^*$, $p, q \in Q$.

Z výše uvedeného vyplývá, že automat M' zakončí simulaci každého kroku automatu M přechodem do stejného stavu, do jakého přechází M, přečtením stejného řetězce ze vstupu a zápisem zobrazení $\nu(c, d)$ na zásobník, přičemž c a d jsou řetězce, které v tomto kroku zapisuje na zásobníky automat M. Z definice zobrazení ν vyplývá, že výsledná konfigurace po dokončení simulace jednoho kroku automatu M musí splňovat podmínky tvrzení 4.1.2. Tvrzení 4.1.3 tedy platí. □

Tvrzení 4.1.4. *Nechť $\zeta = \$u_1\$u_2pex$ a $\zeta' = \$v_1\$v_2qx$ jsou dvě konfigurace automatu M a $r = a|bpe \rightarrow c|dq$ je pravidlo z R takové, že platí $\zeta \Rightarrow \zeta'[r]$ a současně $|\,Len(c) - Len(d)| \leq i$. Potom existují konfigurace η a η' automatu M' a řetězec $\pi \in R'^*$ takové, že platí $\eta \Rightarrow^+ \eta'[\pi]$.*

Důkaz tvrzení 4.1.4. Z platnosti tvrzení 4.1.1 a definice zobrazení ν vyplývá, že pokud η je počáteční konfigurace automatu M' a platí $\eta \Rightarrow \eta'$, potom η' musí nutně splňovat podmínky tvrzení 4.1.2.

Jelikož počáteční konfigurace automatů M a M' jsou ekvivalentní, potom na základě platnosti tvrzení 4.1.3 a definice zobrazení ν musí pro každou následující konfiguraci $\eta = vqx$ automatu M', kde $q \in Q$ platit, že je ekvivalentní konfiguraci $\zeta = \$v_1\$v_2qx$ automatu M. □

Z tvrzení 4.1.3 vyplývá, že limitujícím faktorem schopnosti simulace výpočtu automatu M automatem M' je počet stavů $\langle p_{a,b,10}, \psi\rangle$, $\langle p_{a,b,11}, \psi\rangle$, $\langle p_{a,b,20}, \psi\rangle$ a $\langle p_{a,b,21}, \psi\rangle$ vygenerovaných při konstrukci automatu M'. Počet těchto stavů je limitován maximální délkou řetězce ψ určenou zvolenou konstantou i. Z vlastností zobrazení ν a algoritmu simulace kroku automatu M automatem M' vyplývá, že pokud v každé konfiguraci ζ automatu M bude splněna podmínka $|\,Len(u) - Len(v)| \leq k$, potom pro každou konfiguraci $\eta = vqx$ automatu M', kde $q \in Q$ nutně platí, že

maximální délka sufixu řetězce v mající jednu ze složek tvořenou pouze znaky # je právě k a nebo méně. Zvolíme-li tedy v konstrukci 4.1.1 $i = k$, potom lemma 4.1 platí.

□

Omezené dvouzásobníkové automaty tedy definují třídu bezkontextových jazyků. Toto zjištění je poměrně překvapující, když vezmeme v úvahu, že oproti zásobníkovým automatům mají k dispozici další zásobník a tudíž jsou oproti nim významně rozšířeny.

Část III

Alternativní přístup k optimalizaci

Kapitola 5

Alternativní přístup k optimalizaci

Obsahem této kapitoly je popis části syntaktické analýzy, generování kódu a přidělování registrů v experimentálním kompilátoru využívajícím optimalizační metodu původně navrženou pro paralelizující kompilátory. V kontextu, v jakém je nakonec použita, se nejedná o optimalizaci, ale spíše o alternativní přístup k dané problematice. Vzhledem k tomu, že tím ale vznikly problémy, které u běžných jednoduchých překladačů není nutné řešit, rozhodl jsem se tuto problematiku prozkoumat podrobněji. V dalším textu se předpokládá, že čtenář je alespoň částečně seznámen s problematikou překladu, vnitřní strukturou běžného překladače a metodami syntaktické analýzy. Kompletní popis kompilátoru není zahrnut, protože by již vybočoval mimo zaměření této disertační práce a zbytečně tak zvětšoval její rozsah. Potřebné informace však lze najít na doprovodném CD, případně i v publikaci [17]. Případné další informace týkající se problematiky syntaktické analýzy a překladu lze též nalézt v publikacích [1, 2, 3, 15].

V předchozích dvou kapitolách jsme se po teoretické stránce věnovali modifikacím dvou formálních modelů souvisejících s překladači. Z těchto dvou modelů pro nás nyní budou zajímavé především zásobníkové převodníky, neboť zásobníkový převodník je formální model, který dokáže formálně definovat činnost syntaktického analyzátoru překladače. Syntaktický analyzátor většinou představuje jádro překladače, které řídí celý proces překladu. Při své činnosti volá lexikální analyzátor, který mu dodává jednotlivá slova vstupního řetězce rozšířená o některé další informace. Takto rozšířená slova v oblasti kompilátorů obvykle nazýváme *tokeny*. Lexikální

analyzátor obvykle bývá tvořen konečným automatem, či převodníkem, které jsme si nadefinovali v kapitole 2.

Posloupnost tokenů je přijímána syntaktickým analyzátorem. Současně je generována i odpovídající věta výstupního jazyka. Výstupní jazyk může být tvořen posloupností volání funkcí sémantického analyzátoru, případně jiným vhodným způsobem závislým na implementaci konkrétního překladače. V případě úspěšného přijetí vstupního řetězce tokenů je pak jako výstup syntaktické analýzy k dispozici posloupnost řídicích příkazů sémantického analyzátoru nebo v případě užšího prolnutí se sémantickým analyzátorem již částečně přeložená forma vstupního programu v podobě příslušného mezikódu s více či méně vyřešenými sémantickými závislostmi.

Kompilátor, jehož dvě vývojové fáze budeme sledovat v této kapitole, začal vznikat jako *paralelizující kompilátor*. Tedy kompilátor, který běžný sekvenční program upraví tak, aby byl vhodný pro běh na stroji s více výpočetními jednotkami (typicky víceprocesorový počítač). Požadavky kladené na paralelizující kompilátor jsou poněkud odlišné, než požadavky kladené na běžný sekvenční kompilátor. Těmto požadavkům tedy byl od začátku podřízen i návrh popisovaného kompilátoru. Později se však ukázalo, že vlastní paralelizace je velmi komplikovaný problém a pokusy o její řešení nepřinášely očekávané výsledky. Vzhledem k obecnému vývoji, kdy se spíše než masivní paralelizace sekvenčního kódu začaly používat jiné a méně komplikované prostředky jako třeba vícevláknové zpracování, jsem se rozhodl vývoj paralelizujícího kompilátoru ukončit. Nicméně, některé aspekty částečně vytvořeného kompilátoru mě inspirovaly k jejich využití v běžných kompilátorech.

5.1 Prvotní požadavky

Nejprve se tedy podívejme na některé požadavky na paralelizující kompilátor a jejich řešení v tomto kompilátoru. V současné době je popsána řada optimalizačních metod pro detekci a následné využití skrytého paralelizmu v sekvenčních programech (například v [31, 26]). Častým, poměrně podstatným problémem, bývá malá *granularita* tohoto paralelizmu. Pojem granularita představuje množství instrukcí, které se nachází v jednom paralelizovatelném úseku programu. Obecně je totiž nutné počítat s tím, že výpočetní jednotky zpracovávající paralelně jednotlivé úseky kódu spolu musí komunikovat. Lze předpokládat, že čím méně instrukcí se podaří zpra-

covat mezi dvěma vynucenými komunikacemi, tím více se uplatní režie paralelního zpracování. Ideálním stavem je, pokud mohou všechny výpočetní jednotky zpracovávat dlouhé, zcela nezávislé bloky instrukcí, a teprve po jejich zpracování si předat výsledky. Tím je dána jedna z možných cest — zvětšování granularity paralelizmu.

Jelikož již byla po teoretické stránce zpracována celá řada metod umožňujících detekci a využití paralelizmu, zaměřil jsem se ve své práci na optimalizaci syntaktické analýzy a překladu do intermediárního kódu produkující sice sekvenční kód, ale s větší granularitou, než bez použití této optimalizace. Následně, budou-li na takto vygenerovaný kód použity již známé optimalizační metody, budou produkovat kvalitnější paralelizovaný kód.

Většina optimalizačních metod uváděných v literatuře (například v [26, 31]) využívá ke své činnosti grafy datových závislostí. Vrcholy těchto grafů odpovídají jednotlivým příkazům programu popřípadě různě velkým blokům programu v závislosti na tom, jak detailní popis datových závislostí je v danou chvíli vyžadován. Hrany pak korespondují s datovými závislostmi příkazů odpovídajících daným vrcholům. Mají-li dva příkazy mezi sebou více datových závislostí, projeví se tato skutečnost v grafu odpovídajícím počtem hran mezi jejich dvěma uzly. Jednotlivé hrany tedy představují závislost způsobenou použitím jedné proměnné (přesněji jednoho paměťového místa) v různých příkazech. Běžnou praxí využívanou při tvorbě kompilátorů je vytváření velkého množství pomocných dočasných proměnných v průběhu překladu výrazu a jejich následné využití a maximální možná eliminace při následných optimalizacích. Tento postup je vysoce efektivní z hlediska optimalizací pro sekvenční provádění programu, pro paralelní zpracování se však příliš nehodí. Vytváření velkého počtu dočasných proměnných a vysoká úroveň dekompozice výrazů zvětšuje graf datových závislostí, takže se jeho zpracování stává časově náročnějším, ale také podstatně zvyšuje počet paralelizovatelných úseků, které lze poté v grafu datových závislostí nalézt. Mezi počtem paralelizovatelných úseků a výslednou granularitou získaného paralelizmu platí přibližně nepřímá úměrnost. Předpokládejme, že máme sekvenční program s n instrukcemi. Pokud jej rozdělíme do m paralelních bloků a zbyde nám s instrukcí, které se nepodařilo paralelizovat, je průměrná velikost jednoho paralelního bloku rovna $\frac{n-s}{m} + r$ instrukcím, kde r představuje jisté instrukce zajišťující komunikaci mezi jednotlivými výpočetními jednotkami a tedy i režii paralelního zpracování jednoho bloku kódu.

Budeme-li vycházet z uvedeného předpokladu, potom je pro nás výhodné, aby výsledný program obsahoval pokud možno co nejméně paralelizovatelných bloků kódu, zato však co největší délky.

Optimalizaci, kterou jsem pro tento účel vytvořil, jsem nazval *spojování výrazů*. Tento název přesně vystihuje i její činnost. Hlavní myšlenka metody vychází z předpokladu, že další optimalizační metody budou pracovat s grafem datových závislostí. Pro většinu existujících metod je tento předpoklad splněn. Předpokládejme, že paralelizmus přeložených výrazů ve tvaru, jaký bývá běžně produkován kompilátory, má příliš malou granularitu na to, aby se jej vyplatilo využívat. Potom je nutné veškerý potenciální paralelizmus ve výrazech upravit tak, aby jej nemohly detekovat a využít metody, které použijeme v dalších optimalizačních fázích.

Běžně používaným přístupem při překladu výrazů bývá generování instrukcí pro načtení operandů, generování instrukce pro provedení překládané operace a nakonec generování instrukce pro uložení výsledku do dočasné proměnné. Celý postup se pak opakuje pro všechny operace obsažené ve výrazu. Tento způsob práce je výhodný z hlediska dalšího možného využití mezivýsledků a následných optimalizací pro sekvenční počítač. Ovšem z hlediska paralelizujícího kompilátoru je výhodnost tohoto postupu přinejmenším sporná. Každá dočasná proměnná totiž představuje paměťové místo. Při následném vytváření grafu datových závislostí budou brány v úvahu i dočasné proměnné a tím pádem mohou být nakonec paralelizovány i velmi malé podvýrazy původních výrazů. Považoval jsem tedy za vhodné, optimalizovat již vlastní generování instrukcí pro zpracování výrazů tak, aby během jejich vyhodnocování nedocházelo k ukládání mezivýsledků a tím pádem nemohly být jednotlivé fragmenty výrazů paralelizovány. Tuto optimalizaci lze v kompilátoru poměrně jednoduše implementovat vynecháním generování přebytečných instrukcí pro načítání a ukládání mezivýsledků a doplněním mechanizmu pro šíření informace o spojovaných instrukcích v rámci jednoho výrazu. Možnost paralelizace celých výrazů zůstane samozřejmě zachována.

Syntaktická analýza využívající spojování výrazů byla původně řešena jako modifikace metody pracující rekurzivním sestupem. Jelikož se jedná především o modifikaci při zpracování výrazů, může se použití této metody zdát poněkud nevhodné v porovnání například s operátorově precedenční analýzou či LR analýzou. Pokud však vezmeme v úvahu, že volba metody byla provedena tak, aby ji bylo možné

použít pro celý proces překladu, pak už je tato volba přijatelná. Použití operátorově precedenční analýzy bylo prakticky vyloučeno a LR analýza, ačkoliv je obecně silnější, je poměrně těžkopádná a pomalá. Zbývalo tedy vybrat mezi prediktivní analýzou a rekurzivním sestupem. Nakonec jsem zvolil rekurzivní sestup pro jeho vyšší názornost a vynikající podporu pro práci se zásobníkem, neboť je k dispozici implicitní zásobník rekurze programovacího jazyka.

Jak již bylo uvedeno výše, původní princip optimalizace spojováním výrazů je jednoduchý. Při generování mezikódu se generuje kompaktní mezikód vždy pro celý výraz tak, aby v průběhu výpočtu celého výrazu nedocházelo k ukládání mezivýsledků do paměti. Tato optimalizace byla nakonec použita pouze jako modifikace běžného kompilátoru využívajícího syntaktickou analýzu rekurzivním sestupem se současným generováním mezikódu a částečnou sémantickou analýzou. Tím ale vznikly jisté problémy, které si popíšeme v následující kapitole.

5.2 Realizace kompilátoru

Nyní již můžeme detailněji popsat implementaci modifikovaného kompilátoru využívajícího spojování výrazů. Nejdříve si uvedeme obecné předpoklady a některé implementační detaily. Zásadním rozdílem mezi tradiční a modifikovanou konstrukcí kompilátoru je bezesporu nemožnost rozdělit vygenerovaný kód na základní bloky. Hranice základních bloků by totiž (především při překladu boolovských výrazů se zkráceným vyhodnocováním) mohly vznikat uprostřed výrazů, což je nepřípustné. Jak bylo uvedeno dříve, tato metoda nepoužívá ukládání mezivýsledků výpočtů do paměti. Instrukce v základním bloku by tudíž následně z pohledu běžných algoritmů pro zpracování základních bloků pracovaly s hodnotami „odnikud“ nebo ještě lépe s nějakými nedefinovanými hodnotami, neboť by se mohlo lehce stát, že zpracovávané hodnoty by nebyly v rámci základního bloku ani načteny a dokonce ani nikam uloženy. Vzniklou situaci názorně demonstruje příklad 5.1.

Pro využití spojování výrazů je tedy nutné upravit mechanizmus generování instrukcí vnitřního (intermediárního) kódu a použít zcela nový algoritmus pro přidělování registrů, který nebude, narozdíl od běžně používaných algoritmů, pracovat se základními bloky. Implementace metody spojování výrazů vychází z následujících předpokladů:

- Vygenerované instrukce jsou uloženy v datové struktuře umožňující náhodný přístup k jednotlivým prvkům.
- Každá instrukce zná umístění všech instrukcí, jejichž výsledky zpracovává, ve vygenerovaném kódu.
- Každá instrukce má informace o typech registrů, které používá.
- Každá instrukce má možnost si „zapamatovat“ číslo registru, do kterého má uložit výsledek.
- Každá instrukce používá libovolný počet vstupních operandů a maximálně jeden výstupní.
- Syntaktický analyzátor poskytuje prostředky pro přenos údajů o rozpracovaných instrukcích v závislosti na právě zpracovávaném podstromu.

Prvních šest bodů se týká pouze sémantického analyzátoru a lze je poměrně jednoduše splnit volbou vhodné datové struktury pro ukládání vygenerovaných instrukcí intermediárního kódu. Poslední bod se týká výhradně syntaktického analyzátoru a pro jeho splnění potřebujeme jistý mechanizmus přenosu údajů v průběhu překladu, který nám s výhodou poskytuje implicitní zásobník rekurze.

Podíváme-li se na strukturu syntaktického analyzátoru jako na zásobníkový převodník, pak vidíme, že tento převodník přijímá řetězec symbolů (tokenů) produkovaný lexikálním analyzátorem a generuje řetězec symbolů, který představuje posloupnost volání funkcí sémantického analyzátoru a generovaných instrukcí vnitřního kódu. Zásobníku pak odpovídá implicitní zásobník rekurze a vlastní program tvořící syntaktický analyzátor představuje množinu pravidel.

Činnost celého kompilátoru při generování instrukcí pro výpočet výrazů v příkazu přiřazení potom bude probíhat podle algoritmu 5.1.

Algoritmus 5.1 (Zpracování výrazů metodou spojování výrazů). Vstupem je překládaný výraz a výstupem jeho reprezentace ve vnitřním kódu.

- Ve výrazu se objevil operand, který ještě nebyl v rámci výrazu načten:
  - Syntaktický analyzátor požádá sémantický analyzátor o generování instrukce pro načtení požadovaného operandu.

        * Sémantický analyzátor operand buď vyhledá v tabulce symbolů, pokud se jedná o proměnnou, nebo generuje načtení konstanty, pokud se jedná o konstantu.
    - Sémantický analyzátor vrátí „ukazatel“ na tuto instrukci a syntaktický analyzátor si tento ukazatel uloží do odpovídajícího atributu.

- Ve výrazu se vyskytla operace s jedním nebo více známými operandy:
    - Syntaktický analyzátor požádá sémantický analyzátor o generování instrukce (nebo více instrukcí) pro výpočet požadované operace a jako operandy mu předá ukazatele na instrukce, s jejichž výsledky se má požadovaná operace provést.
    - Sémantický analyzátor vrátí ukazatel na instrukci, která produkuje výsledek požadované operace.

- Výraz je celý zpracován:
    - Syntaktický analyzátor požádá sémantický analyzátor o zápis hodnoty do paměti. Jako hodnotu, která se má zapsat, mu předá operand a ukazatel na instrukci, po jejímž provedení je k dispozici výsledek celého výrazu.
    - Sémantický analyzátor operand vyhledá v tabulce symbolů a vygeneruje zápis hodnoty na odpovídající paměťové místo.

Výsledkem překladu výrazu je tedy datová struktura obsahující instrukce přeloženého programu ve správném pořadí. Předávání dat mezi jednotlivými instrukcemi ale není řešeno pomocí paměťových buněk, nýbrž provázáním pomocí ukazatelů, kdy každý ukazatel představuje informaci o přenosu hodnoty produkované jednou instrukcí na vstup jiné připojené instrukce.

Pro překlad boolovských výrazů se zkráceným vyhodnocováním lze použít obvyklý princip, pouze s tím rozdílem, že jejich případné neboolovské podvýrazy jsou počítány podle výše uvedeného postupu. Jediný problém může nastat v okamžiku, kdy je potřeba použít výsledek vyhodnocení boolovského výrazu jako vstup další instrukce. Jedním z předpokladů pro možnost použití metody spojování výrazů byla existence pouze jediného výstupního operandu pro každou instrukci. Každý boolovský výraz se však při zkráceném vyhodnocování vyhodnocuje ve dvou větvích

(pravda, nepravda) a tudíž se může stát, že jeho výsledek bude poskytován současně dvěma instrukcemi a ve fázi překladu není možné rozhodnout, která z nich se má kdy použít. Instrukce zpracovávající výsledek boolovského výrazu by tedy vlastně měla k dispozici dva ukazatele na instrukce, jejichž výsledek má použít, ale pro jednu vstupní hodnotu. Tento případ je samozřejmě silně nedeterministický, tudíž nežádoucí.

Řešení lze provést více způsoby. Nejjednodušším je uložení obou výstupních hodnot do paměti, vygenerování instrukce pro načtení této hodnoty z paměti a použití načítající instrukce jako výsledku celého výrazu. Ukládání výsledku do dočasné proměnné ale jednak způsobí vytvoření nového uzlu v grafu datových závislostí a navíc představuje zbytečné odkládání a opětovné načítání mezivýsledku. Tento postup byl použit v prvním testovacím kompilátoru, ale z výše uvedených důvodů od něj bylo upuštěno a v další, přepracované, verzi už je zahrnuta podpora pro udržování vazeb s více zdrojovými instrukcemi.

Příklad 5.1 (Překlad výrazu). Mějme výraz

$$a = a * (c \text{ or } b + c > a - b) + (c \text{ and } a)$$

Jedná se o výraz obsahující několik podvýrazů, z nichž dva jsou logické a ostatní běžné aritmetické. Výsledky logických operací jsou kompatibilní s typy `int` (celočíselný) a `float` (desetinný s pohyblivou řádovou čárkou). Pravdivý výsledek logické operace je reprezentován jako hodnota 1 a nepravdivý jako 0. Celý výraz bude vyhodnocován pomocí zkráceného vyhodnocování, ale jeho výsledkem bude běžné číslo, které se uloží do proměnné a. Priorita operátorů je následující (od nejvyšší po nejnižší): Závorky, násobení, sčítání nebo odčítání, porovnání, logický součin, logický součet. Pokud by byl na vygenerovaný kód aplikován algoritmus pro rozdělení na základní bloky, dojde k rozdělení tohoto jediného výrazu na několik základních bloků. Provázání instrukcí pro uchovávání výsledků jednotlivých podvýrazů je ovšem kompilátorem provedeno v rámci celého výrazu. Konstrukce základních bloků by toto propojení porušila a výsledné přidělení registrů by bylo chybné. Situaci nejlépe vystihuje obrázek 5.1 ilustrující vazby mezi instrukcemi a případné hranice základních bloků, které by tyto vazby porušily. Čísla řádků, na nichž začíná nový základní blok, jsou sázena tučně. Jedná se o skutečný kód vytvořený první verzí překladače bez podpory vícenásobných vazeb. Instrukce, jejichž popis je sázen skloněným písmem,

jsou generovány proto, aby celý výsledek vyhodnocení logického výrazu byl tvořen jedinou instrukcí.

Poznámka: Jedná se o komentovaný skutečný kód vygenerovaný kompilátorem využívajícím metodu spojování výrazů a názvy instrukcí intermediárního kódu byly ponechány tak, jak byly použity v tomto kompilátoru. Malé „i“ u názvu instrukcí znamená, že se jedná o celočíselné operace.

5.3 Přidělování registrů

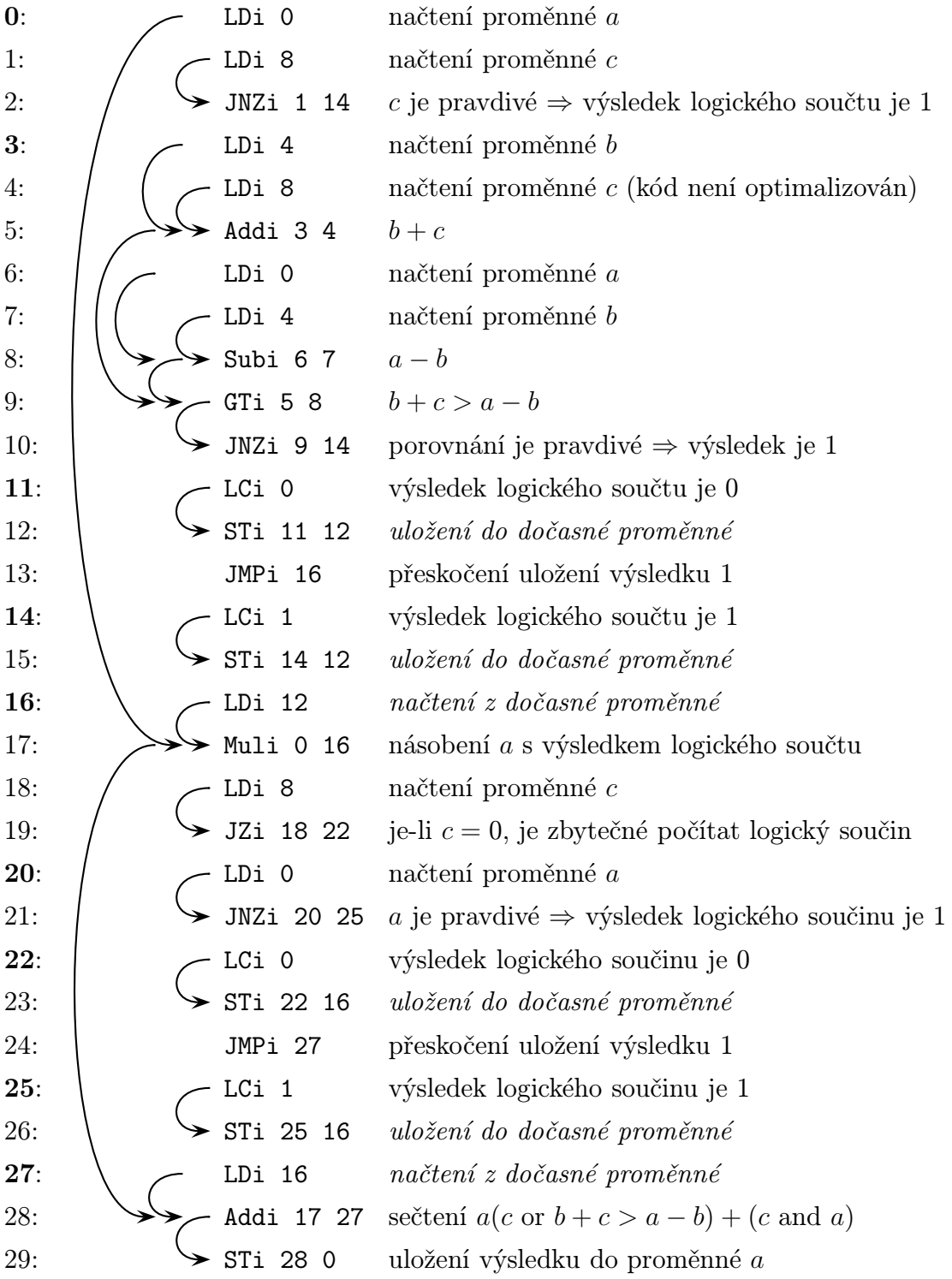
Předpokládejme, že máme program přeložený do intermediárního „pseudokódu“, kdy jednotlivé instrukce nepracují s registry, ale zatím jen se vzájemnými vazbami, jak je ukázáno v příkladu 5.1. Pro převedení do skutečného intermediárního kódu potřebujeme ještě jednotlivým vazbám přidělit skutečné registry. Přidělování registrů lze realizovat několika způsoby. První verze překladače obsahovala přidělování nekonečného počtu registrů s možností následného přemapování na skutečné registry procesoru. Při řešení přemapování však vyvstaly zbytečné problémy se správným odkládáním obsahu registrů při potřebě více registrů, než má procesor. Navíc byl kompilátor jako celek zbytečně složitý. Původní algoritmus přidělující neomezený počet registrů je tedy uveden pouze pro ilustraci vývoje tohoto překladače.

5.3.1 První verze přidělování registrů

V první verzi kompilátoru bylo přidělování registrů řešeno pomocí algoritmu, jehož zjednodušená podoba vypadá následovně.

Algoritmus 5.2. Za předpokladu, že při implementaci metody spojování výrazů byly splněny základní předpoklady uvedené v kapitole 5.2, lze vygenerovaný kód procházet v reverzním pořadí — tedy od poslední instrukce k první — a postupně přidělovat registry podle následujících pravidel pro každou instrukci:

- Zkontroluj, zda instrukce vrací výsledek a zda je přiřazen registr pro uložení výsledku:
  - Instrukce vrací výsledek:



Obrázek 5.1: Provázání instrukcí pomocí jednoduchých vazeb

        * Registr je přiřazen. Přiřaď tento registr operandu, do kterého se ukládá výsledek.
        * Registr není přiřazen. Jedná se o fatální chybu. Kam uložit výsledek?
    - Instrukce nevrací výsledek:
        * Registr je přiřazen. Opět se jedná o fatální chybu — jinou instrukcí je požadován výsledek, ačkoliv se žádný neprodukuje.
        * Registr není přiřazen. V pořádku — výsledek není generován ani požadován.

- Zpracuj všechny vstupní operandy:
    - Operand již má přiřazen vstupní registr. Podívej se na zdrojovou instrukci tohoto operandu, zda má přiřazen výstupní registr:
        * Výstupní registr není přiřazen. Přiřaď zdrojové instrukci svůj registr.
        * Výstupní registr je přiřazen. Výstupní registr zdrojové instrukce se nesmí shodovat s tvým. Generuj instrukci pro přesun (MOV), pomocí které si přesuneš výsledek z výstupního registru zdrojové instrukce do svého registru. Pokud se oba registry shodují, jedná se o fatální chybu. Dále popsaný algoritmus pro přemapování registrů by zhavaroval.
    - Operand ještě nemá přiřazen výstupní registr. Podívej se na zdrojovou instrukci tohoto operandu, zda má přiřazen výstupní registr:
        * Výstupní registr není přiřazen. Použij první volný registr daného typu a přiřaď ho jak svému operandu, tak zdrojové instrukci.
        * Výstupní registr je přiřazen. Použij její výstupní registr a přiřaď ho svému parametru.

- Uvolni registr, jehož operand je výstupní a zároveň není vstupní.

Výše popsaný způsob je použitelný pro instrukce, které mají všechny operandy stejného typu. Pro instrukce, které nemají všechny operandy stejného typu (například instrukce pro konverzi typu `int` na typ `float`), se uvedený algoritmus musí provádět vícekrát, vždy pouze pro operandy stejného typu. Přitom platí pravidlo, že

je nutné nejprve zpracovat výstupní operand a potom v libovolném pořadí vstupní operandy. Chyby, které jsou v algoritmu přidělování registrů ošetřeny, jsou chybami, které vznikly mimo tento algoritmus. Principem činnosti tohoto algoritmu je dáno, že k uvedeným chybám nikdy nedojde. Použité testy tedy slouží spíše pro odhalení chyb vzniklých v předcházejících částech překladu. Uvedené tvrzení se mnohokrát potvrdilo během vývoje a testování kompilátoru, který popsaný algoritmus přidělování registrů používal. Všechny hlášené chyby byly vždy způsobeny vinou špatné funkce mechanizmu generování instrukcí během překladu.

Tato podoba algoritmu pro přidělování registrů byla navržena ještě v době, kdy se jednalo o paralelizující kompilátor a předpokládalo se, že neomezený počet registrů bude poskytovat dostatek volnosti pro další optimalizace. Přednostně jsou ovšem přidělovány registry s číslem menším, než je zadaná hodnota. Tím je vytvořena podpora pro následné optimalizované přemapování nekonečného počtu registrů na konečný počet. Registry nad povolený počet jsou použity pouze v případě naprosté nezbytnosti a při dostatečném počtu registrů procesoru pravděpodobně během překladu nedojde k situaci, kdy by byly všechny vyčerpány.

Jak bylo uvedeno výše, tento algoritmus v kombinaci s následným přemapováním se v sekvenčním kompilátoru choval velmi těžkopádně a celý kompilátor byl zbytečně složitý. Veškeré pokusy o přemapování tedy byly ukončeny a kompilátor byl ponechán jako demonstrační s přidělováním neomezeného počtu registrů. Takto je také ponechán na přiloženém CD.

5.3.2 Druhá verze přidělování registrů

Po neúspěšném ukončení vývoje výše popsané verze kompilátoru jsem se rozhodl kompletně přepracovat celé přidělování registrů. Vývoj této nové verze byl spojen s tvorbou demonstračního kompilátoru pro výuku předmětu Základy překladačů. V souladu se zadáním projektu tohoto předmětu byl patřičně upraven vstupní jazyk a také generovaný intermediární kód. Především ale byla kompletně přepracována celá sémantická analýza včetně přidělování registrů.

Přidělování registrů nadále využívá provázání instrukcí pomocí vazeb. Jsou však podporovány vazby s libovolným počtem zdrojových instrukcí a libovolným počtem cílových instrukcí. Proces přidělování registrů probíhá vždy globálně v rozsahu jedné funkce překládaného programu. Samotné vazby jsou pro každou funkci řešeny jako

nezávislý objekt. Stejně tak generované instrukce tvoří nezávislý objekt. Zároveň platí, že tyto objekty jsou obousměrně provázané, takže každá instrukce má jednoznačně přiřazeny své vstupní a výstupní vazby a každá vazba má informace o tom, které instrukce k ní příslušejí jako zdrojové a které jako cílové.

Podobně jako v případě prvního kompilátoru, i v tomto kompilátoru probíhá nejprve generování mezikódu a současně s ním i vytváření vazeb, kdy je každá nová instrukce vkládána na konec seznamu instrukcí překládané funkce pomocí následujícího algoritmu.

Algoritmus 5.3 (Vložení instrukce). Vstupem jsou maximálně dva ukazatele na zdrojové instrukce (předpokládají se pouze binární operace) a číslo vazby, která se má použít v případě připojování další zdrojové instrukce k již existující vazbě. Výstupem jsou vytvořená instrukce a identifikátor její výstupní vazby.

- Vytvoř novou instrukci.
- Pro každý její registr (vstup či výstup této instrukce) proveď:
  - Pokud je registr vstupní nebo vstupně-výstupní:
    - Zkontroluj typovou kompatibilitu zdrojové instrukce. Pokud není stejného typu, vlož sekvenci konverzních instrukcí. V případě typové nekompatibility ukonči činnost a nahlaš chybu.
    - Propoj tento vstup instrukce s vazbou, se kterou je propojen výstup zdrojové instrukce.
  - Pokud je registr výstupní nebo vstupně-výstupní:
    - Pokud se jedná o instrukci přidávající novou vazbu, tedy o instrukci, která buď nefiguruje jako další zdrojová instrukce u vazby s více zdrojovými instrukcemi, případně je první zdrojovou instrukcí takovéto vazby, vytvoř novou vazbu a tuto instrukci s ní propoj jako zdrojovou.
  - U propojené vazby inkrementuj čítač použití.

Po vygenerování celého kódu aktuálně zpracovávané funkce je možné provést druhou fázi, a to přidělení registrů. Jak jsme si již uvedli, toto přidělování probíhá globálně nad celým vygenerovaným mezikódem této funkce. Je proveden jeden

průchod generovaným kódem ve směru toku programu (tedy od první instrukce k poslední) a do jisté míry optimalizovaným způsobem je provedeno přidělení registrů jednotlivým vazbám. K objektům instrukcí a vazeb nyní přibývají ještě další dva objekty. Je to jednak objekt, který udržuje informace o příslušnosti jednotlivých registrů k jednotlivým vazbám a dále objekt obsahující informace o využití registrů. Všechny tyto objekty využívají pro uložení svých dat obousměrně vázaný lineární seznam s jedním aktivním prvkem a „záložkou“, která umožňuje uchovat aktivitu seznamu v určeném okamžiku a kdykoliv na tento uchovaný prvek nastavit aktivitu se současným nastavením záložky na právě aktivní prvek.

Úkolem objektu využití registrů je uchovávání informace o využití registrů. Volné registry jsou udržovány na začátku seznamu. Každým použitím se registr přesouvá na konec seznamu, naopak, po každém uvolnění se přesune na začátek seznamu. Při přidělování nového registru jsou vždy přidělovány registry ze začátku seznamu, čímž je minimalizována nutnost odkládání obsazených registrů do paměti.

Údaje objektu příslušnosti registrů k vazbám jsou využívány v okamžiku rozhodování o registru, který bude přidělen v okamžiku, kdy instrukce pracuje s nějakou vazbou a této vazbě je nutné přidělit registr. Vlastní přidělování probíhá podle následujícího algoritmu.

Algoritmus 5.4 (Přidělování registrů). Vstupem jsou objekty instrukcí a vazeb mezi instrukcemi. Výstupem je objekt instrukcí s přiřazenými registry.

Seznam instrukcí je postupně procházen v pořadí vykonávání programu a pro každou z nich je provedeno následující:

- Pokud se jedná o instrukci vložení do zásobníku, je možné, že dále bude následovat volání funkce. Pokud ještě není nastavena záložka, nastav záložku na tuto instrukci. Poté přiděl registr jejímu operandu.

- Pokud se jedná o instrukci volání funkce nebo podprogramu, vygeneruj před instrukci označenou záložkou potřebné instrukce pro uložení obsahu používaných registrů do paměti. Pokud žádné místo není označeno, generuj vše před tuto aktuální instrukci. Následně za aktuální instrukci vygeneruj instrukce potřebné pro obnovu obsahu registrů po návratu z podprogramu. Je-li následující instrukce instrukcí pro vyzvednutí ze zásobníku, je na vrcholu zásobníku ná-

vratová hodnota funkce. Tuto hodnotu je tedy ještě před obnovením registrů nutné vyjmout a přidělit jí registr.

- Jedná-li se jakoukoliv jinou instrukci, přiděl registry všem jejím operandům.

Z činnosti tohoto algoritmu je zřejmý jeho nedostatek. V programu nelze libovolně používat instrukce pro vkládání do zásobníku, neboť jsou vždy považovány za ukládání parametrů volaného podprogramu či funkce. Pokud tento požadavek nelze volbou konstrukce kompilátoru splnit, či existuje-li přímý přístup k zásobníku z překládaného jazyka, je nutné doplnit instrukce o příznak ukládání parametru a ve výše uvedeném algoritmu pak počátek ukládání parametrů poznat podle tohoto příznaku.

Vlastní přidělení registru pro každý operand instrukce probíhá podle následujícího algoritmu.

Algoritmus 5.5 (Přidělení registru). Vstupem jsou objekty vazeb, přiřazení vazeb registrům a využití registrů. Výstupem je číslo registru. Stav všech vstupních objektů je však průběžně aktualizován, aby odpovídal aktuální konfiguraci.

- Inkrementuj čítač přidělení registru požadované vazby.
- Zkontroluj umístění požadované vazby. Pokud již má přidělen registr, potom:
  - Pokud je čítač přidělení roven čítači použití, je to poslední přidělení této vazby. Označ tedy přidělený registr jako volný a v objektu využití registru jej přesuň na první místo.
  - Pokud má čítač přidělení nižší hodnotu než čítač použití, bude se registr této vazby ještě používat. Je tedy pravděpodobně nejhorším kandidátem pro přidělení jiné vazby. V objektu využití registrů jej tedy přesuň na poslední místo.
  - Vrať číslo registru.
- Pokud vazba nemá přiřazený registr, potom proveď:
  - Zkontroluj první registr v seznamu objektu využití registrů
    - Pokud není volný, uvolni ho a vrať jeho číslo — je to nejlepší kandidát.

· Pokud máš k dispozici volné paměťové místo, použij ho a přesuň do něj obsah tohoto registru.
· Pokud volné paměťové místo nemáš, vytvoř ho a přesuň do něj obsah tohoto registru.
· Vazbu, která měla tento registr přiřazen, označ jako uloženou v paměti.
· Vrať číslo tohoto registru.

∗ Pokud je volný, pouze změň jeho stav na obsazeno a vrať jeho číslo.

– Pokud je vazba již uložena v paměti, vygeneruj potřebné instrukce pro načtení hodnoty z paměti do přiděleného registru. Ulož si adresu uvolněného paměťového místa pro opětovné využití v případě dalšího uvolňování registru.
– Zkontroluj čítače použití a přidělení této vazby.
∗ Pokud je čítač přidělení roven čítači použití, je to poslední přidělení této vazby. Označ tedy přidělený registr jako volný a v objektu využití registru jej přesuň na první místo.
∗ Pokud má čítač přidělení nižší hodnotu než čítač použití, bude se registr této vazby ještě používat. Je tedy pravděpodobně nejhorším kandidátem pro přidělení jiné vazby. V objektu využití registrů jej tedy přesuň na poslední místo.

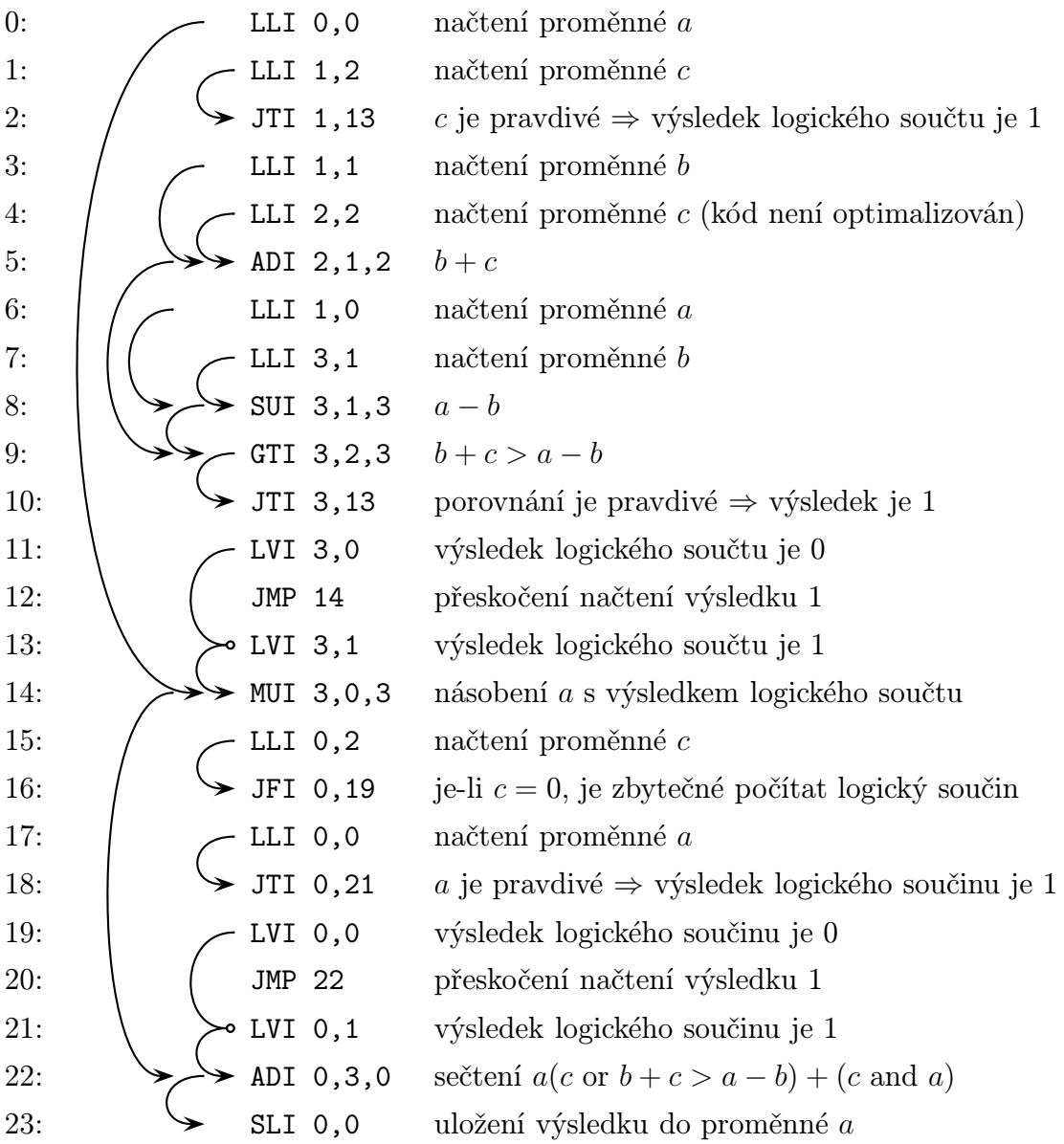
V následujícím příkladu je uveden překlad stejného výrazu jako v příkladu 5.1, tentokrát ovšem s využitím vícenásobných vazeb a druhé verze přidělování registrů tak, jak jsme si ji právě popsali.

Příklad 5.2. Mějme stejný výraz jako v příkladu 5.1, tedy

$$a = a * (c \text{ or } b + c > a - b) + (c \text{ and } a)$$

Na obrázku 5.2 je tento výraz přeložený do cílového kódu druhou verzí překladače spolu s doplněnými vícenásobnými vazbami.

Jak bylo uvedeno výše, byl tento způsob generování mezikódu a přidělování registrů implementován v demonstračním kompilátoru používaném k demonstračním účelům při výuce předmětu Základy překladačů. Kompletní zdrojové kódy včetně ukázkových programů jsou opět k dispozici na doprovodném CD.



Obrázek 5.2: Provázání instrukcí pomocí vícenásobných vazeb

5.4 Shrnutí

V této kapitole jsme se zabývali alternativním přístupem k překladu inspirovaným optimalizační metodou původně určenou pro paralelizující kompilátory. Při použití v sekvenčním kompilátoru tato metoda přináší jisté problémy, jejichž řešení jsme se věnovali. Komplexní popis i jednoduchého kompilátoru je však natolik rozsáhlá problematika, že by výrazně překročila zaměření této práce. Proto byly z kompilátoru vybrány pouze zajímavé pasáže a tyto byly popsány. Podrobnější popis obou diskutovaných verzí kompilátoru včetně specifikace překládaných programovacích jazyků je však možné najít na přiloženém CD.

Část IV

Závěr

Kapitola 6

Závěr

Hlavním cílem této práce byl alternativní pohled na problematiku překladu. Těžiště této práce spočívá především v teoretické oblasti, nicméně část práce se věnuje i praktické oblasti, tedy tvorbě kompilátoru. Celá práce byla psána tak, aby byla co nejvíce soběstačná. Čtenář se základními znalostmi z oblasti teoretické informatiky a kompilátorů tak najde v první části stručný úvod do problematiky a následně i všechny potřebné definice formálních modelů obvykle používaných v souvislosti s překladači.

Po úvodu a základních formálních definicích následuje vlastní teoretická část práce. Zde je podrobně diskutována problematika sebereprodukujících zásobníkových převodníků a omezených zásobníkových automatů.

Sebereprodukující zásobníkové převodníky představují velmi silný formální model se sílou rovnou síle Turingova stroje. I přes svoji obrovskou sílu však mají vnitřní strukturu téměř totožnou s běžnými zásobníkovými automaty. Při jejich praktické implementaci tedy není nutné řešit žádné problémy, které mohou vznikat při praktické implementaci zcela nových či významně rozšířených formálních modelů. Tato práce tudíž přináší nový, alternativní, formální model, který je schopen přijímat či generovat všechny rekurzivně spočetné jazyky. Tím se okamžitě nabízí jeho využití v kompilátorech, kde by mohl nahradit běžně používané zásobníkové převodníky a díky své síle eliminovat nutnost používání různých podpůrných částí určených pro zpracování kontextových závislostí, jako jsou například tabulky symbolů.

Jak je ukázáno na příkladu sebereprodukujícího zásobníkového převodníku přijímajícího jazyk $L(M) = \{yy^R y | y \in \Sigma^*\}$, kde y^R představuje reverzi řetězce y, lze

pro některé jazyky ležící mimo množinu bezkontextových jazyků poměrně snadno najít množinu pravidel, a tak vytvořit sebereprodukující převodník, který tento jazyk přijímá, či generuje. Nicméně, jak už bývá u podobných modelů obvyklé, je nalezení vhodné množiny pravidel pro specifikaci potřebného jazyka mimo množinu bezkontextových pravidel obecně velmi složité. Kromě některých speciálních případů, jakým je například zvolený jazyk $L(M) = \{yy^R y | y \in \Sigma^*\}$, se pro implementaci syntaktické analýzy v překladači stále jeví jako podstatně jednodušší využití obyčejného zásobníkového převodníku s kontextovými závislostmi řešenými pomocí tabulky symbolů. Dokud se nepodaří vytvořit obecný a snadno zvládnutelný postup specifikace alespoň kontextových jazyků, nelze očekávat výraznější praktické využití sebereprodukujících zásobníkových převodníků, stejně jako řady dalších mocných formálních modelů.

Druhým formálním modelem, který byl v práci diskutován jsou omezené dvouzásobníkové automaty. Dvouzásobníkové automaty představují velmi dlouho známý formální model, který má rovněž stejnou sílu jako Turingův stroj. Při praktické implementaci takovýchto modelů nás velmi často zajímají různé aspekty týkající se například časové či prostorové náročnosti výpočtu. Proto se na tyto modely často kladou různé omezující požadavky. V kontrastu se zdánlivě velmi omezujícím požadavkem na jedinou a navíc současnou obrátku v obou zásobnících v průběhu celého výpočtu, je diskutován zdánlivě benevolentní požadavek na maximální velikost rozdílu délek obou zásobníků v průběhu výpočtu. Ačkoliv je dokázáno, že první podmínka sílu automatu neovlivní, v této práci bylo dokázáno, že druhá podmínka ji degraduje na úroveň běžných zásobníkových automatů, tedy na množinu bezkontextových jazyků.

Na teoretickou část zabývající se alternativními modely souvisejícími s kompilátory navazuje praktická část. Ta se zabývá alternativním využitím jednoduché optimalizační metody. Tato metoda byla původně navržena pro paralelizující kompilátory jako přípravná metoda za účelem zvýšení granularity skrytého paralelizmu před použitím dalších optimalizačních metod. Tato kapitola se však zabývá využitím této, původně optimalizační, metody jako alternativního způsobu generování vnitřního kódu v sekvenčních kompilátorech. Generovaný vnitřní kód neukládá mezivýsledky výpočtů do paměti ani do registrů, ale využívá provázání generovaného kódu pomocí vazeb. Každá instrukce tak má k dispozici informaci, které instrukce

generují data, s nimiž má pracovat. Díky tomu generovaný vnitřní kód nemůže být dále zpracován běžnými algoritmy pro přidělování registrů. Proto byly vyvinuty dva nové algoritmy pro přidělování registrů pracující s vytvořenými vazbami. Popisu způsobu generování kódu a následně i algoritmům pro přidělování registrů v tomto kompilátoru je věnována převážná část páté kapitoly.

Rozsáhlost diskutované problematiky však natolik překračuje rámec této práce, že dokumentace k oběma verzím demonstračního kompilátoru využívajícího vytvořené algoritmy nebyla začleněna a byla ponechána pouze na přiloženém CD.

Literatura

[1] Aho, A. V., and Ullman, J. D.: *The Theory of Parsing, Translation and Compiling, Volume I: Parsing*. Prentice Hall, Englewood Cliffs, New Jersey, 1972.

[2] Aho, A. V., and Ullman, J. D.: *The Theory of Parsing, Translation and Compiling, Volume II: Compiling*. Prentice Hall, Englewood Cliffs, New Jersey, 1973.

[3] Aho, A. V., and Ullman, J. D.: *Principles of Compiler Design*. Addison-Wesley, Reading, Massachusetts, 1977.

[4] Berstel, J.: *Transductions and Context-Free Languages*. Teubner Verlag, 1979.

[5] Carroll, J., and Long, D.: *Theory of Finite Automata*. Prentice Hall, New Jersey, 1989.

[6] Dassow, J., and Paun, Gh.: *Regulated Rewriting in Formal Language Theory*. Springer, Berlin, 1989.

[7] Eilenberg, S.: *Automata, Languages, and Machines, Volume A*. Academic Press, New York, 1974.

[8] Eilenberg, S.: *Automata, Languages, and Machines, Volume B*. Academic Press, New York, 1976.

[9] Ginsburg, S.: *The Mathematical Theory of Context-Free Languages*. McGraw Hill, New York, 1966.

[10] Harrison, M. A.: *Introduction to Formal Language Theory*. Addison-Wesley, Reading, 1978.

[11] Hopcroft, J. E., and Ullman, J. D.: *Introduction to Automata Theory, Languages, and Computation*. Second Edition, Addison-Wesley, Reading, Massachusetts, 1979.

[12] Choffrut, C., and Culik II, K.: Properties of Finite and Pushdown Transducers. *SIAM Journal on Computing*, 12, pp. 300–315, 1983.

[13] Kelley, D.: *Automata and Formal Languages*. Prentice Hall, Englewood Cliffs, 1995.

[14] Kleijn, H. C. M., and Rozenberg, G.: On the Generative Power of Regular Pattern Grammars. *Acta Informatica*, Vol. 20, pp. 391–411, 1983.

[15] Lewis, P. M., II, Rosenkrantz, D. J., and Stearns, R. E.: *Compiler Design Theory*. Addison-Wesley, Reading, Massachusetts, 1976.

[16] Linz, P.: *An Introduction to Formal Languages and Automata*. D.C. Heath and Co., Lexigton, Mass, 1990.

[17] Lorenc, L.: A New Method of Optimization in Parallel Compilers. *Proceedings of 6th Spring International Conference ISIM'03 Information Systems Implementation and Modelling*. Ostrava, CZ, MARQ, pp. 87–194, 2003.

[18] Lorenc, L., and Meduna, A.: Self-Reproducing Translation Made by Pushdown Transducers. *MEMICS 2005*. Brno, pp. 59–67, 2005.

[19] Lorenc, L., and Meduna, A.: Self-Reproducing Pushdown Transducers. *Proceedings of 7th Spring International Conference ISIM'04 Information Systems Implementation and Modelling*. Ostrava, CZ, MARQ, pp. 155–160, 2004.

[20] Meduna, A.: *Automata and Languages: Theory and Applications*. Springer, London, 2000.

[21] Meduna, A.: Simultaneously One-Turn Two-Pushdown Automata. *Intern. J. Computer Math.*, Vol. 80, pp. 679–687, 2003.

[22] Meduna, A., and Kolar, D.: Regulated Pushdown Automata. *Acta Cybernetica* 14, pp. 653–664, 2000.

[23] Meduna, A., and Kolar, D.: One-Turn Regulated Pushdown Automata and Their Reduction. *Fundamenta Informaticae*, Vol. 2002, No. 16, Amsterdam, NL, pp. 399–405, 2002.

[24] Meduna, A., and Lorenc, L.: Self-Reproducing Pushdown Transducers. *Kybernetika*, Vol. 41, No. 4, pp 531–537, 2005.

[25] Meduna, A., and Lorenc, L.: A Rigorous Approach to Self-Reproducing Pushdown Translation. *Proceedings of 8th Spring International Conference ISIM'05 Information Systems Implementation and Modelling*, Ostrava, CZ, MARQ, pp. 51–58, 2005.

[26] Polychronopoulos, C. D.: *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.

[27] Revesz, G. E.: *Introduction to Formal Languages*. McGraw-Hill, New York, 1983.

[28] Rozenberg, G., and Salomaa, A. (eds.).: *Handbook of Formal Languages, Volume 1–3.*, Springer, 1997.

[29] Salomaa, A.: *Theory of Automata*. Pergamon Press, London, 1969.

[30] Salomaa, A.: *Computation and Automata*. Cambridge University Press, Cambridge, England, 1985.

[31] Wolfe, M.: *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, Inc., 1996.