VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

NOVÉ STRUKTURY A OPERACE V MATEMATICKÉ INFORMATICE

NEW STRUCTURES AND OPERATIONS IN MATHEMATICAL INFORMATICS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **RICHARD BUREŠ**
AUTHOR

VEDOUCÍ PRÁCE **Prof. RNDr. ALEXANDER MEDUNA, CSc.**
SUPERVISOR

BRNO 2018

Zadání bakalářské práce/20196/2017/xbures29

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Bureš Richard**
Obor: Informační technologie
Téma: **Nové struktury a operace v matematické informatice**
New Structures and Operations in Mathematical Informatics
Kategorie: Teoretická informatika

Pokyny:

1. Seznamte se s novými strukturami a operacemi, které zavedla moderní teoretická informatika.
2. Dle instrukcí od vedoucího zaveďte a studujte nové struktury a operace.
3. Navrhněte vhodné formální modely pro pojmy z bodu 2. Popište jejich konstrukci a implementaci.
4. Dle konzultací s vedoucím, studujte využití modelů navržených v předchozím bodě. Využijte v nich úprav řetězců založených na bázi diskutovaných modelů. Implementujte je a ověřte jejich činnost na řadě příkladů.
5. Zhodnoťte dosažené výsledky a diskutujte další možný vývoj projektu.

Literatura:

- Meduna, A. and Zemek, P.: Regulated Grammars and Automata. Springer, 2014, ISBN 978-1-4939-0368-9.
- Rozenberg, G. and Salomaa, A. (eds.): Handbook of Formal Languages, Volume 1-3. Springer, 1997, ISBN 3-540-60649-1.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Meduna Alexander, prof. RNDr., CSc.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Nové struktury a operace v matematické informatice

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Prof. Alexandra Meduny a uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Richard Bureš
14. května 2018

Poděkování

Rád bych poděkoval Prof. Alexandru Medunovi za vedení a inspirace při tvorbě této práce a také spolužákům a přátelům, kteří byli ochotni poslouchat mé stížnosti, kdykoliv jsem si s něčím nevěděl rady.

Obsah

Kapitola 1

Úvod

V této práci se zaměříme na jisté existující operace v moderní teoretické informatice a následně se pokusíme vytvořit operace nové. Nejdříve se v první části kapitoly "Operace" zaměříme na operace známé, druhá část bude věnována operacím novým. (Zde jako "nové operace" budeme brát operace, které byly v rámci této práce vymyšleny, počínaje rozdílným sjednocením.) V následné části bude proveden popis nutného postupu pro umožnění implementace těchto operací, poté se budeme věnovat implementaci samotných operací a práci uzavře kapitola o testování.

1.1 Základní definice a notace

V této práci je použito několik pojmů a zápisů, které nemusí být čtenáři známé, mohou mít více významů, nebo naopak pro jednu věc existuje několik zápisů. V této části si tedy definujeme, co budeme používat ve zbytku práce. Kupříkladu $\binom{a}{b}$ čteme jako A nad B a je to zápis ekvivalentní k $\frac{a!}{b!(a-b)!}$

1.1.1 Konečný automat a jeho zápis

Konečný automat v této práci bude definován pěticí $M = \{Q, \Sigma, R, s, F\}$, kde:

1. Q je konečná množina všech stavů automatu.

   Stavy mohou být popsány písmenem, například q, nebo n-ticí např.: (q, q'), což definuje jeden stav složený z více stavů. Kupříkladu (q_M, q_N) značí stav, který je složený ze dvou stavů, kde q_M je stav automatu M a q_N je stav automatu N.

2. Σ je abeceda (neprázdná konečná množina symbolů)

3. R je množina pravidel (nebo-li přechodů).

   Pravidla v R zapisujeme $q\alpha \longrightarrow q'$. Tento zápis znamená, že ze stavu q přecházíme pomocí symbolu α do stavu q'.

   Pravidla také můžeme zapisovat jako $qw \longrightarrow^* q'$, což znamená, že pomocí řetězce w se sérií přechodů můžeme dostat ze stavu q do stavu q'.

   Pravidla mohou být také definována pomocí rovnice:
   $R = \{q\alpha \longrightarrow q'$; podmínky a definice stavů a symbolů$\}$

Speciálním pravidlem je takzvané epsilon pravidlo/přechod, nebo-li prázdné pravidlo/prázdný přechod, což je speciální přechod, při kterém nečteme žádný vstupní znak. Zapisujeme buď jako $q \longrightarrow q'$ nebo $q\epsilon \longrightarrow q'$

4. s je počáteční stav automatu.

5. F je konečná množina koncových stavů automatu.

1.1.2 Zápisy jazyků

Jazyky můžeme zapisovat několika způsoby, buď jako výčet $L = \{a, b\}$, což je jazyk akceptující řetězce a a b, nebo rovnicí $L = \{w|$návod, jak sestrojit w, kde w je libovolný řetězec v $L\}$. Při zápisu jazyků se můžeme také setkávat se zápisy řetězců typu $u_1...u_k$, což znamená řetězec skládající se z k symbolů. Podobný typ zápisu je také kupříkladu $u_2u_4...u_{k-2}u_k; u = u_1...{}_k; u \in L$ značící řetězec skládající se z každého druhého znaku z řetězce v jazyce L.

Dalším k jazykům se vztahujícím zápisem je zápis rodin jazyků, který může být zadán buď definicí stejně jako jazyk, nebo výčtem jako: $R = \{K : \{a, b\}, L : \{c, d\}\}$, což udává rodinu jazyků skládající se z jazyků K a L.

Kapitola 2

Operace

V této kapitole si představíme jak existující a tak i definujeme nové operace nad jazyky, případně rodinami jazyků. Operace si definujeme, ukážeme si nad nimi pár příkladů a také některé z jejich vlastností.

2.1 Sjednocení (Union)

Union, neboli sjednocení, je známá operace, kdy sjednocujeme dva jazyky J a K a vzniká nám jazyk L, obsahující prvky jazyka J i K. (viz. 2.1)

$$Union(L_1, L_2) = \{w | w \in L_1 \vee w \in L_2\} \tag{2.1}$$

2.1.1 Vlastnosti

Rodiny regulárních a bezkontextových jazyků jsou vůči této operaci uzavřeny. Tedy pokud operaci použijeme nad dvěma regulárními jazyky, výsledkem bude regulární jazyk, a obdobně je tomu tak s bezkontextovými. Použijeme-li operaci nad rodinou regulárních jazyků, vznikne nám jiná, větší rodina regulárních jazyků.

Při použití nad libovolnou rodinou, jenž je podmnožinou rodiny regulárních jazyků, platí že není-li žádný z jazyků v rodině podmnožinou jazyka v téže rodině, tak velikost vzniklé rodiny je dána rovnicí 2.2, kde n je délka původní rodiny. Pokud rodina obsahuje jazyky, jenž jsou podmnožinou jazyků v téže rodině, je délka vzniklého jazyka na tomto faktu závislá, a proto obecně víme pouze to, že bude menší než hodnota daná rovnicí 2.2. Je-li však mezi jazyky v rodině taková vazba, že provedení operace $Union$ nad libovolnými dvěma jazyky rodiny nám vytvoří jazyk patřící do této rodiny, víme, že vzniklá rodina jazyků bude totožná s rodinou, nad kterou jsme tuto operaci používali. (Viz. podkapitola 2.1.2) Tento vztah budeme nazývat, jako rodina tranzitivně uzavřena nad operací.

$$\binom{n+1}{2} \tag{2.2}$$

2.1.2 Příklad

Opakované použití operace Union s prázdným jazykem:
Mějme rodinu jazyků R_1

$$R_1 = \{L_1 : \{0, 00, 000\}, L_2 : \{1, 11, 111\}, L_3 : \{0, 1\}, L_4 : \{\epsilon\}, L_5 : \{\}\}$$

Použijeme nad touto rodinou sjednocení: $R_2 = Union(R_1)$

$$\begin{aligned} R_2 = \{&L_1 : \{0, 00, 000\}, L_2 : \{1, 11, 111\}, L_3 : \{0, 1\}, L_4 : \{\epsilon\}, \\ &L_5 : \{\}, L_6 : \{0, 00, 000, 1, 11, 111\}, L_7 : \{0, 00, 000, 1\}, \\ &L_8 : \{0, 00, 000, \epsilon\}, L_9 : \{0, 1, 11, 111\}, L_{10} : \{1, 11, 111, \epsilon\}, \\ &L_{11} : \{0, 1, \epsilon\} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \} \end{aligned}$$

A použijeme-li sjednocení ještě jednou, získáme tranzitivně uzavřenou rodinu: $R_3 = Union(R_2)$

$$\begin{aligned} R_3 = \{&L_1 : \{0, 00, 000\}, L_2 : \{1, 11, 111\}, L_3 : \{0, 1\}, L_4 : \{\epsilon\}, \\ &L_5 : \{\}, L_6 : \{0, 00, 000, 1, 11, 111\}, L_7 : \{0, 00, 000, 1\}, \\ &L_8 : \{0, 00, 000, \epsilon\}, L_9 : \{0, 1, 11, 111\}, L_{10} : \{1, 11, 111, \epsilon\}, \\ &L_{11} : \{0, 1, \epsilon\}, L_{12} : \{0, 00, 000, 1, 11, 111, \epsilon\}, \\ &L_{13} : \{0, 00, 000, 1, \epsilon\}, L_{14} : \{0, 1, 11, 111, \epsilon\} \\ &\} \end{aligned}$$

Obdobný výsledek by nastal u jakékoliv konečné rodiny jazyků. Přesněji řečeno, obsahuje-li rodina jazyků k jazyků, tak nejpozději při $k-1$ iteraci se dostaneme k tranzitivně uzavřené rodině.

2.2 Průnik (Intersection)

Intersection neboli průnik je další známá operace, kde průnik jazyků L_1 a L_2 nám dává jazyk L_3, jenž obsahuje pouze řetězce přítomné v jazyce L_1, a zároveň jazyce L_2. (Viz. 2.3)

$$Intersection(L_1, L_2) = \{w | w \in L_1 \wedge w \in L_2\} \tag{2.3}$$

2.2.1 Vlastnosti

Rodina regulárních jazyků je vůči této operaci uzavřena, kdežto rodina bezkontextových jazyků nikoliv. Tedy oproti Union, pokud použijeme operaci Intersection nad dvěma jazyky, jež jsou bezkontextové, nemůžeme si být jisti, že výsledkem bude jazyk bezkontextový.

Při použití nad rodinou regulárních jazyků bude opět vzniklá rodina větší za předpokladu, že původní rodina nebyla nad touto operací tranzitivně uzavřená.

2.2.2 Příklad

Na příkladu si ukážeme opakované použití operace Intersection až do bodu, kdy se dostaneme do stavu, kdy platí, že je operace nad rodinou tranzitivně uzavřená, což bychom si mohli dokázat tak, že operaci použijeme znovu.

Opakované použití operace Intersection
Mějme rodinu jazyků R_1

$$R_1 = \{$$
$$L_1 : \{0, 1, 00\}, L_2 : \{1, 11, 00\}, L_3 : \{0, 00, 11\},$$
$$\}$$

Použitím operace Intersection získáme R_2: $R_2 = Intersection(R_1)$

$$R_2 = \{$$
$$L_1 : \{0, 1, 00\}, L_2 : \{1, 11, 00\}, L_3 : \{0, 00, 11\},$$
$$L_2 : \{1, 00\}, L_2 : \{0, 00\}, L_3 : \{00, 11\},$$
$$\}$$

A dalším použitím Intersection se dostáváme k tranzitivně uzavřené rodině R_3

$$R_3 = \{$$
$$L_1 : \{0, 1, 00\}, L_2 : \{1, 11, 00\}, L_3 : \{0, 00, 11\},$$
$$L_4 : \{1, 00\}, L_5 : \{0, 00\}, L_6 : \{00, 11\}, L_7 : \{00\}$$
$$\}$$

2.3 Doplněk (Complement)

Doplněk jazyka si můžeme definovat následujícím příkladem: Představme si, že máme jazyk L patřící do abecedy Σ. Doplněk jazyka L jsou všechny řetězce patřící do množiny řetězců Σ^* a zároveň nepatřící do jazyka L.

Pro představu si můžeme říci, že $Complement(L) = \Sigma^* - L$, kde L patří do rodiny regulárních jazyků.

2.3.1 Vlastnosti

Stejně jako u průniku je rodina regulárních jazyků vůči této operaci uzavřena, kdežto rodina bezkontextových nikoliv. Také je zcela zřejmé, že $Complement(Complement(L_1)) = L_1$, kde L patří do rodiny regulárních jazyků.

2.4 Rozdíl (Difference)

Rozdíl dvou jazyků je dosti podobný rozdílu dvou množin, tedy pokud $L_3 = Difference(L_1, L_2)$, tak L_3 obsahuje všechny řetězce patřící do jazyka L_1 a zároveň nepatřící do jazyka L_2, což si můžeme znázornit rovnicí 2.4. Mnohem výstižněji to však můžeme popsat rovnicí 2.5

$$Difference(L_1, L_2) = \{w | w \in L_1 \wedge w \notin L_2\} \tag{2.4}$$

$$Difference(L_1, L_2) = Intersection(L_1, Complement(L_2)) \tag{2.5}$$

2.4.1 Vlastnosti

Z rovnice 2.5 nám jasně plyne, že rodina regulárních jazyků je vůči této operaci uzavřena, jelikož je uzavřená vůči průniku a doplňku. Rodina bezkontextových jazyků však nikoliv.

2.5 Shuffle

Operaci shuffle můžeme definovat jako všechny kombinace, které vzniknou pomícháním znaků z řetězců u a v se zachováním pořadí znaků tak, jak byly v původním řetězci. V podstatě bychom tuto operaci mohli popsat jako prolnutí dvou řetězců (viz 2.6). Tuto operaci potom můžeme generalizovat také na jazyky, viz 2.7. Znalost a pochopení této operace nám pomůže v chápání dalších operací, jako je vkládání (kapitoly 2.6 a 2.7).

$$\begin{aligned} Shuffle(u,v) = \{ & \\ & u_1v_1...u_iv_j| \\ & \quad u = u_1...u_i \wedge u \in \Sigma* \\ & \quad v = v_1...v_j \wedge v \in \Sigma* \\ & \quad u_p \in u \wedge u_p \in \Sigma \cup \epsilon; \\ & \quad v_q \in v \wedge v_q \in \Sigma \cup \epsilon; \\ & \quad 1 \leq p \leq i \wedge 1 \leq q \leq j \\ \} & \end{aligned} \tag{2.6}$$

$$Shuffle(K,L) = \{w|w = Shuffle(u,v); u \in K \wedge v \in L\} \tag{2.7}$$

2.5.1 Vlastnosti

Uzavřenost vůči této operaci je obdobná s předchozími příklady, tedy platí pro rodinu regulárních jazyků a neplatí pro rodinu jazyků bezkontextových.

2.5.2 Příklad

Operace Shuffle nad dvěma řetězci:
Nechť existují řetězce u a v takové, že $u = "ab", v = "cd"$. $Shuffle(u,v)$ se poté bude rovnat $\{"abcd","acbd","acdb","cabd","cadb","cdab"\}$.

2.6 Sekvenční vkládání (Sequential Insertion)

Tato operace je výborně popsána v [7] (strana 23-28). Ve svém provedení jako takovém se jedná o jednoduchou operaci, kdy do libovolného řetězce z jazyka K vložíme na libovolné místo libovolný řetězec z jazyka L. Tuto operaci si tedy můžeme popsat rovnicí 2.8.

$$SequentialInserion(K,L) = \{w|w = xyz, xz \in K \wedge y \in L \wedge x,z \in \Sigma^* \cup \epsilon\} \tag{2.8}$$

2.6.1 Vlastnosti

Rodiny regulárních a bezkontextových jazyků jsou uzavřené vůči této operaci, viz [7] strana 25-27.

2.6.2 Příklad

Operace SequentialInsertion nad dvěma jazyky:
Mějme dva jazyky, $K = \{abc, def\}$ a $L = \{xy\}$
Použití operace nám poté generuje jazyk:
$SequentialInsertion(K,L) = \{xyabc, axybc, abxyc, abcxy, xydef, dxyef, dexyf, defxy\}$

2.7 Paralelní vkládání (Parallel Insertion)

Paralelní vkládání je operace velice podobná sekvenčnímu, avšak s tím rozdílem, že pokud máme jazyk K, do kterého vkládáme, tak nevkládáme pouze na jednu libovolnou pozici, nýbrž vkládáme na všechny pozice. Operace je opět výborně popsána v [7] na straně 24-28. Tuto operaci bychom si také mohli definovat rovnicí 2.9

$$\begin{aligned} ParallelInsertion(K,L) = \{ & \\ & w | w = x_0u_0u_1x_1...x_nu_nx_{n+1}, \\ & \quad x_k \in L \wedge u_1u_1...u_n \in K, \\ & \quad n \geq 0 \wedge 0 \leq k \leq n \\ \} & \end{aligned} \tag{2.9}$$

2.7.1 Vlastnosti

Rodiny regulárních a bezkontextových jazyků jsou uzavřené vůči této operaci, viz [7] strana 27-28.

2.7.2 Příklad

Operace ParallelInsetion nad jazyky:
Mějme jazyky $K = \{abc\}$ a $L = \{de\}$.
Použití operace nám poté generuje jazyk: $ParallelInsertion(K,L) = \{de\mathbf{a}de\mathbf{b}de\mathbf{c}\}$

2.8 Sekvenční Mazání (Sequential Deletion)

Sekvenční mazání je operace podobná sekvenčnímu vkládání, jedná se téměř o její přesný opak. Takto si ji však můžeme nazvat pouze neformálně, nemůžeme u této operace spoléhat na $SequentialDeletion(SequentialInsertion(L)) = L$(viz příklad). Více je tato operace popsána v [7] na straně 55-70. Tato operace je lehce popsatelná rovnicí 2.10.

$$SequentialDeletion(K,L) = \{w | w = xz; xyz \in K \wedge y \in L\} \tag{2.10}$$

2.8.1 Vlastnosti

Všechny vlastnosti si dopodrobna může čtenář přečíst ve výše uvedené knize, bylo by však vhodné podotknout, že rodiny regulárních a bezkontextových jazyků jsou vůči této operaci uzavřeny.

2.8.2 Příklad

Ukázka toho, že $SequentialDeletion(SequentialInsertion(K,L),L) \neq K$**:**
Mějme jazyky $K = \{abc\}$ a $L = \{a\}$. Použijeme-li sekvenční vkládání, dostáváme $K_2 = SequentialInsertion(K,L) = \{aabc, abac, abca\}$. Pokud následně použijeme sekvenční mazání, dostáváme $K_3 = SequentialDeletion(K_2,L) = \{abc, bac, bca\}$, což se zcela zřetelně nerovná K.

2.9 Prefixes

Prefixes je operace, jejíž aplikace na K vytváří L, kde L obsahuje všechny řetězce, které jsou prefixy všech řetězců K (Viz. 2.11).

$$Prefixes(L) = \{w | x = wy; x \in L \wedge y \in \Sigma_L\} \tag{2.11}$$

2.9.1 Vlastnosti

Theorem 1 (Uzavřenost nad regulárními jazyky)**.** *Použití operace Prefixes nad libovolným regulárním jazykem L generuje opět regulární jazyk.*

Důkaz 1. Mějme konečný automat $M = \{Q_M, \Sigma_M, R_M, s_M, F_M\}$ Pokud je operace Prefixes uzavřená nad regulárními jazyky, je tedy uzavřená nad konečnými automaty, a tedy jsme schopni vytvořit konečný automat N, který bude přijímat všechny prefixy jazyka $L(M)$.

Automat N vytvoříme tak, že vezmeme automat M a všechny stavy, které nejsou neukončující, označíme jako konečné.

Automat N je vždy konečný automat a přijímá všechny prefixy jazyka L(M), což potvrzuje, že je operace nad regulárními jazyky uzavřená. Dostáváme tedy vztah:

$$Prefixes(K(M)) = L(N)$$

Tvrzení: Platí vztah $Prefixes(K(M)) = L(N)$
Toto tvrzení platí pokud pro jakékoliv $w = uv$ přijímané automatem M, automat N přijímá řetězec u, a pro jakékoliv $w = uv$ nepřijímané automatem M, kde automat M nepřijímá ani u, automat N nepřijímá řetězec u. Bez ztráty obecnosti můžeme tvrdit, že N nemá žádné neukončující stavy, tedy vzhledem ke způsobu pro vytvoření automatu N uvedenému výše víme, že všechny stavy N jsou koncové, a tedy i počáteční stav. Z čehož, nám plyne, že je-li w nebo u přijímané automatem M, automat N bude u přijímat také, jelikož se na konci bude nacházet v jednom z jeho stavů. Naopak není-li ani w ani u přijímané automatem M, tak se během zpracování řetězce u nebo w stane, že automat M nemá stav, do kterého by přešel, tak ani automat N nebude mít stav, do kterého by přešel, a proto u nebude přijímané automatem N. □

2.9.2 Příklad

Nechť existuje jazyk $L = \{123, 456, 101\}$, aplikací operace Prefixes můžeme tedy vytvořit jazyk $K = Prefixes(L) = \{\epsilon, 1, 12, 123, 4, 45, 456, 10, 101\}$. Zde je třeba si všimnout faktu, že řetězec "123" je prefixem řetězce "123", neboť sufixem je ”, neboli ϵ.

2.10 Rozdílné sjednocení (Different Union)

Tato operace vychází z operace sjednocení, avšak s tím rozdílem, že povoluje pouze sjednocení nestejných jazyků (Viz. 2.12).

$$DifferentUnion(L_1, L_2) = \{w | (w \in L_1 \vee w \in L_2) \wedge L_1 \neq L_2\} \tag{2.12}$$

2.10.1 Vlastnosti

Operace je uzavřena nad regulárními, ne však nad bezkontextovými jazyky.

Theorem 2 (Uzavřenost nad regulárními jazyky)**.** *Použití operace DifferentUnion nad libovolnými dvěma regulárními jazyky K a L generuje opět regulární jazyk.*

Důkaz 2. Nejprve musíme zjistit, jak porovnat dva jazyky K a L. Dva jazyky jsou si rovné, pokud jsou oba dva podmnožinou toho druhého. To si můžeme převést na rozdíl a vyjádřit nerovnost následovně:

$$K \neq L \iff K - L \neq \varnothing \wedge L - K \neq \varnothing$$

Pravou stranu si dosadíme do rovnice 2.12, a jelikož víme, že rodina regulárních jazyků ja uzavřena vůči Sjednocení i Rozdílu, je tedy uzavřena vůči DifferentUnion. □

Theorem 3 (Uzavřenost nad Bezkontextovými jazyky)**.** *Předpokládejme, že použití operace DifferentUnion nad libovolnými dvěma bezkontextovými jazyky K a L generuje opět bezkontextový jazyk.*

Důkaz 3. Postupujeme stejně jako u důkazu 2, avšak zjišťujeme, že operace Rozdíl není uzavřena nad bezkontextovými jazyky. Tedy ani operace DifferentUnion nemůže být nad těmito jazyky uzavřena, jelikož nemůžeme tvrdit, že jsme schopni porovnat libovolné dva bezkontextové jazyky a rozhodnout o jejich rovnosti. □

Při použití nad rodinou jazyků nám vzniká opačný efekt (než kupříkladu u sjednocení) a to, že může vzniknout rodina menší než rodina, nad kterou jsme operaci aplikovali. Velikost rodiny zde opět záleží na tom, zda-li rodina obsahuje jazyk, jenž je podmnožinou jiného jazyka. Při opakovaném použití se však vždy dostaneme do stavu, kdy se rodina začne zmenšovat do bodu, kdy obsahuje jeden jazyk, a nakonec tedy je tato rodina prázdná. Tato vlastnost platí i pro nekonečně velké rodiny jazyků, u nichž však k nule nikdy nedojdeme, a rodina se nám zmenšuje a zároveň zůstává nekonečná. (Různě velká nekonečna obdobně jako množina celých čísel je menší nekonečno než množina čísel reálných.)

Tento efekt nastává proto, že pokud se nemůže jazyk sjednotit sám se sebou a ani se svou podmnožinou, nemá možnost být ve výsledku další rodiny, a tedy je "vyloučen".

Další vlastností, které je vhodné si všimnout je, že jazyky v rodině jazyků se vždy zvětšují, a to proto, že se do výsledku dostanou pouze jazyky, jenž jsou sjednocením dvou jiných, nebo jazyky, jež jsou nadmnožinou jazyka jiného.

2.10.2 Příklad

Na příkladu si můžeme všimnout kolísání velikosti rodin jazyků. Po prvním použití je vidět, že se velikost výsledné rodiny oproti předchozí zvětší, a to díky jazyku L_5, jenž je podmnožinou všech jazyků. Následně se velikost rodin začíná zmenšovat, až nakonec dojdeme do stavu, kdy je rodina jazyků prázdná.

Iterativně použitá operace DifferentUnion nad rodinou jazyků
Mějme rodinu jazyků R_1 a postupně aplikujme operaci DifferentUnion, dokud se nedostaneme k prázdné rodině jazyků.

$$
\begin{aligned}
R_1 &= \{L_1 : \{0, 00, 000\}, L_2 : \{1, 11, 111\}, L_3 : \{0, 1\}, L_4 : \{\epsilon\}, L_5 : \{\}\} \\
R_2 &= \{ \\
&\quad L_1 : \{0, 00, 000\}, L_2 : \{1, 11, 111\}, L_3 : \{0, 1\}, L_4 : \{\epsilon\}, \\
&\quad L_6 : \{0, 00, 000, 1, 11, 111\}, L_7 : \{0, 00, 000, 1\}, \\
&\quad L_8 : \{0, 00, 000, \epsilon\}, L_9 : \{0, 1, 11, 111\}, L_{10} : \{1, 11, 111, \epsilon\}, \\
&\quad L_{11} : \{0, 1, \epsilon\} \\
&\} \\
R_3 &= \{ \\
&\quad L_6 : \{0, 00, 000, 1, 11, 111\}, L_7 : \{0, 00, 000, 1\}, \\
&\quad L_8 : \{0, 00, 000, \epsilon\}, L_9 : \{0, 1, 11, 111\}, L_{10} : \{1, 11, 111, \epsilon\}, \\
&\quad L_{11} : \{0, 1, \epsilon\}, L_{12} : \{0, 00, 000, 1, 11, 111, \epsilon\}, \\
&\quad L_{13} : \{0, 00, 000, 1, \epsilon\}, L_{14} : \{0, 1, 11, 111, \epsilon\} \\
&\quad \} \\
R_4 &= \{ \\
&\quad L_6 : \{0, 00, 000, 1, 11, 111\}, L_{12} : \{0, 00, 000, 1, 11, 111, \epsilon\}, \\
&\quad L_{13} : \{0, 00, 000, 1, \epsilon\}, L_{14} : \{0, 1, 11, 111, \epsilon\} \\
&\quad \} \\
R_5 &= \{L_{12} : \{0, 00, 000, 1, 11, 111, \epsilon\}\} \\
R_6 &= \{\}
\end{aligned}
$$

2.11 Operace "rozdílné"(Operation Different)

Operation Diferent, neboli rozdílné (neplést s Difference, rozdíl) přijímá libovolné dva jazyky a provádí nad nimi operaci, kterou nejlépe definuje 2.13, pokud bychom ji chtěli definovat dopodrobna, byla by popsána rovnicí 2.14.

$$Different(L_1, L_2) = Union(L_1, L_2) - Intersection(L_1, L_2) \tag{2.13}$$

$$Different(L_1, L_2) = \{w | (w \in L_1 \wedge w \notin L_2) \vee (w \notin L_1 \wedge w \in L_2)\} \tag{2.14}$$

2.11.1 Vlastnosti

S ohledem na fakt, že tato operace jde rozložit na několik jednodušších operací, jsme tak schopni odvodit i chování této operace. Vzhledem k tomu, že všechny operace uvedené v 2.13 jsou operace, nad kterými je rodina regulárních jazyků uzavřená, můžeme říci, že tato rodina je uzavřená nad celou touto operací. Stejným způsobem zjistíme, že rodina bezkontextových jazyků vůči této operaci uzavřená není.

Operace při opakovaném volání nevykazuje žádné speciální vlastnosti vyjma toho, že jakékoliv použití této operace nad rodinou jazyků nám zaručí existenci prázdného jazyku. Existence prázdného jazyku nám dále zaručí, že v dalším iteraci nám nezmizí žádný jazyk, a tedy je výsledná rodina vždy stejná, nebo větší. Po určitém počtu iterací se nakonec dostáváme k rodině, která je nad touto operací tranzitivně uzavřená.

2.12 Unikátní konkatenace (Unique Concatenation)

Unique Concatenation, neboli Výlučná konkatenace, je operace, jež se chová jako konkatenace dvou jazyků, až na ten fakt, že nepřijímá takové řetězce z K a L, jichž konkatenace by patřila do K nebo L. Pro zjednodušení si určeme konkatenaci tak, že konkatenace je operace popsatelná rovnicí 2.15. Unikátní konkatenace je poté definována rovnicí 2.16. Ve zkratce: výsledek unikátní konkatenace generuje tytéž řetězce jako konkatenace normální, s výjimkou, že neobsahuje řetězce, které již byly obsaženy v jazycích, jež jsme konkatenovali, což je nejlépe popsáno rovnicí 2.17.

$$Concatenation(L_1, L_2) = \{w | w = xy; y \in L_1 \wedge y \notin L_2\} \tag{2.15}$$

$$UniqueConc(L_1, L_2) = \{w | w = xy; y \in L_1 \wedge y \notin L_2 \wedge xy \notin L_1 \wedge xy \notin L_2\} \tag{2.16}$$

$$UniqueConc(L_1, L_2) = Concatenation(L_1, L_2) - L_1 - L_2 \tag{2.17}$$

2.12.1 Vlastnosti

Díky neunikátnosti této operace jsme schopni z jejího vyjádření 2.17 vyvodit, že rodina regulárních jazyků je uzavřená vůči této operaci, avšak rodina bezkontextových jazyků nikoliv.

Rodina vzniklá touto operací je velikosti n^2 nebo $(n-1)^2 + 1$, pokud původní rodina obsahovala prázdný jazyk, kde n je velikost rodiny, nad kterou jsme operaci aplikovali.

2.13 Protkání (Interlacement)

Tato operace je inspirovaná Shuffle a vkládajícími operacemi. Tato operace je nejvíce podobná operaci Shuffle, kdy nejdříve uvažujme dva řetězce u a v. Protkáním těchto dvou řetězců dostáváme řetězec, který se skládá vždy ze symbolu řetězce u, a následně symbolu z řetězce v. Protkání dvou řetězců si tedy můžeme definovat rovnicí 2.18 s tím, že pokud dva řetězce nemají stejnou délku, nemohou být protkány, a generuje se tak řetězec prázdné délky. Tuto operaci následně můžeme generalizovat a použít nad jazyky obdobným způsobem, jako jakoukoliv jinou operaci, tedy protkáme každý řetězec z jazyka K s každým řetězcem z jazyka L (viz. rovnice 2.19). Obdobně bychom mohli operaci generalizovat i na rodinu jazyků (viz rovnice 2.20). Iterativní použití této operace nad rodinou jazyků by mohlo mít silně expandující efekt, proto si představme ještě variaci nad rodinami jazyků, která nám nebude přijímat dva stejné jazyky, tedy bude růst výrazně pomaleji, případně může za správných okolností stagnovat i klesat, definovanou rovnicí 2.21.

$$\begin{aligned} Interlacement(u, v) = \{ & \\ & w | w = u_1 v_1 u_2 v_2 ... u_k v_k; \\ & u = u_1 u_2 ... u_k \wedge v = v_1 v_2 ... v_k; \\ & k \geq 1 \wedge k = |u| = |v| \\ \} & \end{aligned} \tag{2.18}$$

$$Interlacement(K, L) = \{w | w = Interlacement(u, v); u \in K \wedge v \in L; w \neq \epsilon\} \tag{2.19}$$

$$Interlacement(R) = \{J | J = Interlacement(K, L); K, L \in R; J \neq \emptyset\} \tag{2.20}$$

$$UniqueInterlacement(R) = \{J | J = Interlacement(K, L); K, L \in R \wedge K \neq L; J \neq \emptyset\} \tag{2.21}$$

2.13.1 Vlastnosti

Theorem 4 (Uzavřenost nad regulárními jazyky)**.** *Rodina regulárních jazyků je uzavřená vůči operaci Interlacement.*

Důkaz 4. Mějme automaty $M = \{Q_M, \Sigma_M, R_M, s_M, F_M\}$ a $N = \{Q_N, \Sigma_N, R_N, s_N, F_N\}$ a jimi definované jazyky $K(M)$ a $L(N)$. Nyní jsme schopni vytvořit jazyk $K^{'}(M^{'}) = Interlacement(K, L)$:

$$\begin{aligned}
M^{'} = \{ & \\
& Q = \{0, 1\} \times Q_M \times Q_N, \\
& \Sigma = \Sigma_M \cup \Sigma N, \\
& R = \{ \\
& \quad (0, q_M, q_N)\alpha \longrightarrow (1, q_M\alpha, q_N), \\
& \quad (1, q_M, q_N)\alpha \longrightarrow (0, q_M, q_N\alpha); \\
& \quad q_M \in M \wedge q_N \in N \\
& \} \\
& s = (0, s_M, s_N), \\
& F = \{(0, q_M, q_N) | q_M \in F_M \wedge q_N \in F_N\} \\
\}
\end{aligned} \tag{2.22}$$

A jelikož jsme schopni sestrojit konečný automat přijímající tento jazyk, víme, že tento jazyk bude regulární. Tedy dostáváme vztah:

$$Interlacement(K, L) = K^{'}(M^{'})$$

Tvrzení: Platí rovnost $Interlacement(K, L) = K^{'}(M^{'})$
Důkaz takovéto rovnosti by byl mimo rozsah této bakalářské práce, avšak pokusíme si tuto rovnost "načrtnout".

Operace nám zajišťuje, že v generovaném jazyce budou všechny řetězce takové, že:

$$w = u_1v_1u_2v_2...u_nv_n; u \in K, v \in L$$

Tedy řetězec $u = u_1u_1u_2...u_n$ je přijímaný automatem M a řetězec $v = v_1v_1v_2...v_n$ je přijímaný automatem N. Aby platila rovnost, řetězec w musí být přijímaný automatem $M^{'}$, tedy musí existovat taková posloupnost přechodů, že se pomocí řetězce w dostaneme

z počátečního stavu do koncového. Bez ztráty obecnosti můžeme předpokládat, že oba automaty M a N mají právě jeden konečný stav f. Tedy vytvoříme-li automat M' dle předpisu 2.22, existuje v automatu M' právě jeden koncový stav:

$$(0, f_m, f_n); f_M \in F_M, f_n \in F_N$$

Platí-li rovnost, musí existovat posloupnost přechodů:

$$(0, s_m, s_n)w \longrightarrow^{2o} (0, f_m, f_n); o = |u| = |v|$$

Takovouto posloupnost si můžeme přepsat jako:

$$(0, s_m, s_n)wv_{o-1} \longrightarrow^{2o-1} (1, f_m, q'_n)v_{o-1} \longrightarrow (0, f_m, f_n)$$

Podrobněji jako:

$$(0, s_m, s_n)wu_{o-1}v_{o-1} \longrightarrow^{2o-2} (0, q'_m, q'_n)u_{o-1}v_{o-1} \longrightarrow (1, f_m, q'_n)v_{v-1} \longrightarrow (0, f_m, f_n)$$

Případně pro dokončení dvou celých kroků v řetězci u a v řetězci v jako:

$$\begin{aligned}(0, s_m,&s_n)wu_{o-2}v_{o-2}u_{o-1}v_{o-1}\\ &\longrightarrow^{2o-4} (0, q''_m, q''_n)u_{o-2}v_{o-2}u_{o-1}v_{o-1}\\ &\longrightarrow (1, q'_m, q''_n)v_{o-2}u_{o-1}v_{o-1}\\ &\longrightarrow (0, q'_m, q'_n)u_{o-1}v_{o-1}\\ &\longrightarrow (1, f_m, q'_n)v_{o-1}\\ &\longrightarrow (0, f_m, f_n)\end{aligned}$$

Tím bychom si následně mohli dokázat pravdivost tvrzení už jen z toho důvodu, že existuje-li v automatu M pravidlo $q_m\alpha \longrightarrow q'_m$, musí existovat $(0, q_m, q_n)\alpha \longrightarrow (1, q'_m, q_n)$, a to samé musí platit i pro pravidla automatu N. □

Na první pohled by se mohlo zdát, že konečný automat pro výsledek této operace sestrojit nedokážeme, neboť se nám sama nabízí možnost zásobníkového automatu, který by "počítal", jestli bychom měli aplikovat pravidla z prvního nebo druhého automatu. Jelikož zásobník takového automatu by vždy obsahoval jeden symbol (vyjma počátečního), tak jsme toto chování schopni replikovat nad konečnými automaty pomocí zdvojnásobení počtu stavů.

Použití této operace nad jazyky může generovat prázdný jazyk, z čehož je zřejmé, že ne všechny řetězce přispějí do výsledného jazyka, přesněji řetězec z jazyka K takové délky, že neexistuje řetězec v jazyce L stejné délky, nepřispívá do generovaného jazyka a naopak.

Použití operace nad dvěma jazyky vždy generuje jazyk nepřijímající prázdný řetězec.

Použití operace neprázdnou rodinou jazyků vždy generuje větší, nebo alespoň stejně velkou rodinu jazyků s tím, že při iterovaném použití se výsledek nikdy neustálí, a tedy není možné, aby byla rodina jazyků nad touto operací tranzitivně uzavřena.

Pokud bychom uvažovali upravenou verzi, nepřijímající dva stejné jazyky (jak jsou dva jazyky porovnávány jsme si ukazovali v kapitole 2.10), tak budeme li uvažovat rodinu jazyků R dostáváme následující vlastnosti:

1. Použití operace nad libovolnými dvěma nestejnými jazyky z R vrací prázdný jazyk:

   Generovaná rodina je prázdná

2. Obsahuje-li R pouze dva nestejné jazyky, nad nimiž operace negeneruje prázdný jazyk:

   Generovaná rodina bude obsahovat opět dva stejně velké jazyky ($Interlacement(K, L)aInterlaceme$

   (a) Při iterovaném použití nad takovou rodinou:
   Od druhého použití víme, že rodina R_i bude stejně velká jako rodina R_{i-1}, avšak jazyky v rodině R_i budou větší než v rodině R_{i-1}, a to samé platí i o délce nejdelšího řetězce.

3. Obsahuje-li rodina R více kompatibilních jazyků, víme pouze, že při iterativním použití bude od kroku 2 růst s tou výjimkou, že obsahuje-li rodina pouze dvojice kompatibilních jazyků, bude stagnovat obdobně jako v bodě 2.

2.13.2 Příklad

Použití nad dvěma řetězci:
Nechť existuje řetězec $u = abc$ a řetězec $v = def$, poté řetězce $w = Interlacement(u, v)$ a $w' = Interlacement(v, u)$ jsou:
$w = adbecf$ a $w' = daebfc$

Použití nad dvěma jazyky:
Nechť existují jazyky $K = \{01, 23, 45\}$ a $L = \{67, 890\}$, poté jazyky $K^{'} = Interlacement(K, L)$ a $L^{'} = Interlacement(L, K)$ jsou:
$K^{'} = \{0617, 2637, 4657\}$ a $L^{'} = \{6071, 6273, 6475\}$

Použití nad rodinou jazyků:
Nechť existuje rodina jazyků $R = \{L_1 : \{01, 23, 45\}, L_2 : \{67, 890\}, L_3 : \{89, ab\}\}$ poté použití operace nad touto rodinou bude vypadat následovně:

$$\begin{aligned} R^{'} = Interlacement(R) = \{&L_{1,1} : \{0011, 0213, 0415, 2031, 2233, 2435, 4051, 4253, 4455\}, \\ &L_{1,2} : 0617, 2637, 4657, L_{2,1} : \{6071, 6273, 6475\}, ..., \\ &L_{3,2} : \{5697, a6b7\}, L_{3,3} : \{8899, aabb, 8a9b, a8b9\}\} \end{aligned}$$

Použití unikátní verze operace nad rodinou jazyků:
Nechť existuje rodina jazyků $R = \{L_1 : \{01, 23, 45\}, L_2 : \{67, 890\}, L_3 : \{89, ab\}\}$, poté použití unikátní verze operace nad touto rodinou bude vypadat následovně:

$$\begin{aligned} R^{'} = UniqueInterlacement(R) = \{&L_{1,2} : \{0617, 2637, 4657\}, L_{2,1} : \{6071, 6273, 6475\}, \\ &L_{1,3} : \{0819, 2839, 4859, 0a1b, 2a3b, 4a5b\}, ..., \\ &\{L_{3,2} : \{5697, a6b7\}\} \end{aligned}$$

2.14 Plné zakázání abecedy (Full Alphabet deletion)

Tato operace je výrazně inspirovaná Paralelním mazáním [7] (strany 55-70) a je zcela aplikovatelná použitím Paralelního mazání, kde bychom místo abecedy použili jazyk, kde každý řetězec je výlučně jeden symbol. Rozdíl této operace s Paralelním mazáním je však v její aplikovatelnosti na konečné automaty (viz 2.14.1), případně v její implementaci. Tato operace je opět lehce popsatelná rovnicí 2.23.

$$FAD(K,\Sigma_2) = \{ \\ u_1u_2...u_ku_{k+1}|k \geq 1, u_i \in \Sigma^*, 1 \leq i \leq k+1 \wedge \\ \exists v_i \in \Sigma_2, 1 \leq i \leq k : u = u_1v_1...u_kv_ku_{k+1}v_{k+1} \\ kde\{u_i\} \notin \Sigma_2 \\ \} \tag{2.23}$$

2.14.1 Vlastnosti

Vzhledem k tomu, že tuto operaci aplikovat pomocí Paralelního mazání, tak víme, že stejně jako Paralelní mazání bude vůči ní rodina regulárních jazyků uzavřená. Výhodou této operace je však to, že oproti Paralelnímu mazání není ani zdaleka tak složitá. Paralelní mazaní samo o sobě vyžaduje velikou režii a aplikovatelnost nad automatem je velice složitá, naopak u Plného mazání abecedy je aplikace primitivní, neboť nám pouze stačí projít všechna pravidla automatu, ze kterého odstraňujeme abecedu a nahradit odstraňované symboly za ϵ. Tedy je operace spíše blíže homomorfismu než paralelnímu mazání.

2.15 Pop

Operace Pop je de facto operace reverzní ke konkatenaci. Tuto operaci si rozdělíme na $LPop(K,L)$ a $RPop(K,L)$. Toto rozdělení provádíme z toho důvodu, že konkatenovat můžeme z obou stran, a tedy bychom rádi z obou stran i odebírali, což musíme nějak specifikovat. Obě operace pop jsou vlastním způsobem omezená operace Sekvenční mazání. Operace si můžeme definovat následujícím způsobem:

$$LPop(K,L) = \{y|xy|inK \wedge x \in L\} \tag{2.24}$$

$$RPop(K,L) = \{x|xy|inK \wedge x \in L\} \tag{2.25}$$

2.15.1 Vlastnosti

Theorem 5 (Uzavřenost lPop nad regulárními jazyky)**.** *Rodina regulárních jazyků je uzavřená nad LPop.*

Důkaz 5. Nechť existují dva jazyky L_1 a L_2, nad abecedou Σ, a nechť existuje automat $M = \{Q, \Sigma, R, s, F\}$ přijímající jazyk L_1. Pro každé dva stavy $s, q \in Q$ nechť existuje:

$$L_{s,q} = \{w|sw \longrightarrow^* q \in M; w \in \Sigma\}$$

Poté uvažujme automat:

$$M^{'} = \{Q, \Sigma \cup \{\#\}, R^{'}, s, F\}$$

kde

$$R^{'} = R \cup \{s\# \longrightarrow q|s,q \in Q \wedge L_2 \cap L_{s,q} \neq \emptyset\}$$

kde # je nový symbol nepatřící do abecedy Σ.

Následně můžeme tvrdit, že:

$$LPop(L_1,L_2) = h(L(M^{'} \cap \Sigma^*\#\Sigma^*))$$

Následně důkaz pokračuje jako důkaz Sekvenčního mazání, viz [7] (strana 60-61), kde $A^{'} = M^{'}$. □

Theorem 6 (Uzavřenost rPop nad regulárními jazyky)**.** *Rodina regulárních jazyků je uzavřená nad RPop.*

Důkaz 5. Nechť existují dva jazyky L_1 a L_2, nad abecedou Σ, a nechť existuje automat $M = \{Q, \Sigma, R, s, F\}$ přijímající jazyk L_1. Pro každé dva stavy $q \in Q \wedge f \in F$ nechť existuje:

$$L_{q,f} = \{w | qw \longrightarrow^* f \in M; w \in \Sigma\}$$

Poté uvažujme automat:

$$M^{'} = \{Q, \Sigma \cup \{\#\}, R^{'}, s, F\}$$

kde

$$R^{'} = R \cup \{s\# \longrightarrow q | s, q \in Q \wedge L_2 \cap L_{q,f}\}$$

kde # je nový symbol nepatřící do abecedy Σ.

Následně můžeme tvrdit, že:

$$LPop(L_1, L_2) = h(L(M^{'} \cap \Sigma^* \# \Sigma^*))$$

Následně důkaz pokračuje jako důkaz Sekvenčního Mazání, viz [7] (strany 60-61), kde $A^{'} = M^{'}$. □

Při používání této operace je důležité mít na paměti, že i když o ní můžeme uvažovat jako o opaku konkatenace, tedy $RPop(Concatenation(K, L), L) = K$, tak

$Concatenation(RPop(K, L), L) \neq K$. Jedná se tak proto, při použití $xPop$ operace, nepřispívají všechny řetězce jazyka K, nýbrž pouze ty, ze kterých můžeme něco smazat.

2.15.2 Příklad

Ukázka použití LPop a RPop:
Nechť existují dva jazyky, $K = \{aba, ab, c\}$ a $L = \{a\}$.

Použití operace $LPop$ nám generuje jazyk $K^{'} = LPop(K, L) = \{ba, b\}$ a použití operace $RPop$ nám generuje jazyk $K^{''} = RPop(K, L) = \{ab\}$.

Ukázka RPop(Concatenation(K, L), L) = K:
Uvažujme stejné jazyky jako v předchozím příkladu, konkatenace $Concatenation(K, L)$ nám generuje jazyk $K^{'} = \{abaa, aba, ca\}$. Následné použití RPop nám generuje opět jazyk K, $RPop(K^{'}, L) = K$.

Ukázka Concatenation(RPop(K, L), L) ≠ K:
Opět uvažujme stejné jazyky jako v předchozích příkladech, použití operace $RPop(K, L)$ nám generuje jazyk $K^{'} = ab$, což se zcela zřetelně nerovná K.

Kapitola 3

Implementace ekosystému pro implementaci operací

3.1 Požadavky na implementaci

Vytvořit jednoduché prostředí, nad kterým bude možné zpracovávat operace nad konečnými automaty.

Prostředí musí splňovat následující požadavky:

1. Musí umět zpracovávat konečné automaty.
   (a) Přijmutí konečného automatu.
   (b) Vypsání konečného automatu.
   (c) Aplikování základních operací nad konečnými automaty neměnící přijímaný jazyk (odstranění nekoncových stavů, nedeterminismu a pod.).
2. Musí být možné tyto automaty rozšířit/přidat nový typ automatů.
3. Musí být možné vytvářet operace pro práci nad těmito automaty (Implementace operací).
4. Musí být možné přidávat nové typy automatů.
5. Musí být možné na toto prostředí navázat nebo ho používat i v jiných pracích.

3.1.1 Rozšíření

Přidat zpracování zásobníkového automatu, na němž se ukáže rozšiřitelnost ekosystému o další automaty.

3.2 Jazyk a balíčky implementace

Jako jazyk implementace byl zvolený jazyk JavaScript. Tento jazyk byl zvolen z důvodu, že je podporován internetovou komunitou, a především knihovna (prostředí) napsané v tomto jazyce půjde použít nejen na serverech a prohlížečích, ale také i na mobilních a desktopových zařízeních.

Pro zpřehlednění čitelnosti kódu je použita verze EcmaScript 2015 [2] a výše, a pro typovou kontrolu rozšíření Flow [3].

3.2.1 Balíčky použité pro běh (produkční prostředí)

- Lodash [1]

3.2.2 Balíčky použité pro vývoj

- Flow [3]
  - flow-bin [4]: spustitelný program použitý pro statickou typovou kontrolu projektu
- Babel [15]
  - babel-cli [10]: spustitelný program použitý pro překlad zdrojových kódů v novější verzi jazyka JavaScript do starší verze jazyka.
  - babel-register-cli [14]: překládá jazyk obdobně jako babel-cli, avšak až za běhu programu. Dalo by se říci, že rozšiřuje interpret programu.
  - babel-plugin-transform-object-rest-spread [11]: babel balíček pro rozšíření syntaxe jazyka
  - babel-preset-es2015 [12]: babel balíček pro překlad z EcmaScript 2015 do starších verzí.
  - babel-preset-flow [13]: babel balíček rozšiřující syntaxi o datové typy pro Flow.

Zde by mohla vzniknout otázka, proč používat Flow a Babel?

Flow je v implementaci použito, jak je výše zmíněno, pro typovou kontrolu. Není nijak nutné používat flow v dalších rozšířeních projektu, avšak věřím, že typovost dodá programu více spolehlivosti, a také výrazně urychluje vývoj už tím, že zamezuje "runtime" chybám vzniklým právě při nekompatibilních typech u netypovaných jazyků. Výrazným soupeřem Flow je TypeScript, který je stejně vhodný. Výběr Flow jako typové nadstavby byl zvolen především pro svou kompatibilitu s Reactem (reactjs.org), což může zjednodušit případný vývoj grafického prostředí v budoucí projektech.

Babel je v implementaci použit tak, jak definují balíčky ve výše uvedeném seznamu. Nepoužití Babel by zamezovalo použití Flow a kód napsaný v EcmaScript 2015 by byl omezený pouze na nejnovější verze interpretů. Především kdokoliv, kdo by chtěl na tomto kódu stavět, by musel být obeznámen s EcmaScript 2015, kdežto takto stačí pouze znalost tradičního (slangově oldschool) JavaScriptu, nebo naopak je možné použít zcela jiný přístup, například beztřídní.

Upozornění: Ke konci dubna 2018 přešel Babel na novou verzi 7.0.0-beta.46. Vzhledem k tomu, že se nejedná o plné uvolnění Babel 7 a že tato práce je psána před vypuštěním Babel 7, jsou výše uvedené Babel knihovny závislé na verzi 6.*. Tento fakt nemá sebemenší vliv na implementaci, ani případné používání praktické části této práce v jiných projektech. Pouze je potřeba, aby čtenář věděl, že výše uvedené knihovny jsou nyní "depracated", a tedy při navazujících pracích je doporučováno používat Babel 7, který má však jiný ekosystém. (Nebude-li však navazující práce používat Babel vůbec, tento fakt ji nijak neovlivní, jelikož práce obsahuje i překompilovanou verzi JavaScriptových souborů.)

3.2.3 Testovací prostředí

Pro testovací prostředí byl vybrán framework AVA [16] a pro zjištění pokrytí kódu je používán Istanbul Code Coverage[5], respektive jeho verze pro příkazovou řádku "nyc"[6].

Testy jako takové je možno spouštět nad přeloženou i nepřeloženou verzí JavaScriptu tak, aby bylo možné pouštět testy při vývoji, ale zároveň i otestovat výstupní kód.

Dále při každém pushi na git (Github), jsou projekty spuštěny pomocí služby Travis CI [9].

3.3 Vstupně výstupní formát automatu

Formát jazyka bude nejlépe vysvětlen jeho popisným příkladem viz 3.2, kde <cokoliv>, není součástí formátu, nýbrž popis hodnoty. Důležité je zmínit, že formát je v zápisu JSON, jeho struktura je odvozena od množinového zápisu automatu, a uvedený příklad popisuje povinné hodnoty pro konečný automat a je možné ho jakkoliv rozšířit, bez porušení kompatibility a při zachování stávající struktury.

```
{

    "states": [
        {name: <unikátní název stavu (řetězec)>},
        {name:  <unikátní název stavu (řetězec)>
        ],
    "alphabet": [<symbol abecedy (řetězec)>,...],
    "rules": [
        {
            "from": {"state": {"name": <název existujícího stavu (řetězec)>},},
            "to": {"state": {"name": <název existujícího stavu (řetězec)>}},
            "symbol": <existující symbol abecedy (řetězec)>,
        },
        {
            "from": {"state": {"name": <název existujícího stavu (řetězec)>},},
            "to": {"state": {"name": <název existujícího stavu (řetězec)>}},
            "symbol": <existující symbol abecedy (řetězec)>,
        }
    ],
    "finalStates": [{name: <název existujícího stavu (řetězec)>}],
    "initialState": {name: <název existujícího stavu (řetězec)>}
}
```

Výpis 3.1: Přijímaný formát konečného automatu

V rámci rozšíření pro zásobníkové automaty je tento formát obohacen o následující klíče:

```
{
  "rules": [
    {
      "from": {
        "state": {"name": <název existujícího stavu (řetězec)>},
        "stackTop": <symbol, co musí být na vrcholu zásobníku (řetězec délky 1)>
      },
      "to": {
        "state": {"name": <symoly, které budou na vrcholu zásobníku (řetězec délky 0-2)>}
      },
      "symbol": <existující symbol abecedy (řetězec)>,
    },
  ],
  "initialStackSymbol": <(řetězec o délce 1)>,
  "stackAlphabet": [<symbol zásobníkové abecedy(řetězec o délce 1)>,...]
}
```

Výpis 3.2: Rozšíření pro zásobníkový automat

Je důležité si dát velký pozor na to, aby symbol zásobníkové abecedy byl řetězec délky jedna.

3.4 Implementace

Samotná implementace je provedena třídně objektově, kde každá součást automatu a automat sám je objekt, obdobně jako je tomu u formátu výše. Nejjednodušší popis bude pomocí třídní reprezentace viz 3.3, obdobně jako výše popsaný formát.

```
class Automata = {
    states: { <název stavu (řetězec)>: State };
    alphabet: Alphabet;
    rules: Rule[]; //pole Rule
    initialState: State;
    finalStates: { <název stavu (řetězec)>: State };
}

class State = {
    name: <(řetězec)>;
    isInitial: boolean;
    isFinal: boolean;
    isNonterminating: boolean;
}

class Alphabet = <Pole s pouze unikátními prvky (pole)>

class Rule = {
    from: {state:State};
    to: {state:State};
    symbol: string;
}
```

Výpis 3.3: Třídní hierarchie implementace

3.5 Implementace libovolné operace nad tímto ekosystémem

Každá operace, jenž je možné provést nad daným automatem, je pro tento typ automatu pomocí implementovatelná. Tato kapitola se bude věnovat implementaci hypotetické operace.

Představme si tedy unární operaci *allowEmpty* jejímž cílem je pouze donutit automat, aby přijímal prázdný řetězec. (viz 3.4 a 3.5)

```
//@flow
import FA from "../Automata/FA/FA.js";

/**
* Generuje automat přijímající prázdný řetězec
* @param {FA} automata
*/
export default function allowEmpty(automata:FA):FA {
  let newAutomata = automata.copy();
  newAutomata.initialState.isFinal = true;
  return newAutomata;
}
```

Výpis 3.4: Ukázka implementace operace za použití EcmaScript 2015 a Flow

```
var FA = require("../Automata/FA/FA.js").default;

/**
* Generuje automat přijímající prázdný řetězec
* @param {FA} automata
*/
function allowEmpty(automata) {
  var newAutomata = automata.copy();
  newAutomata.initialState.isFinal = true;
  return newAutomata;
}
window.exports = {default:allowEmpty};
```

Výpis 3.5: Ukázka implementace operace pomocí standardního JavaScriptu

K použití operace pak už pouze stačí vytvořit automat a zavolat tuto operaci tak, že automat je jejím argumentem. (příklad viz 3.6)

```
import FA from "../Automata/FA/FA.js";
import allowEmpty from 'allowEmpty.js';
let automata = new FA(/*...*/);
let newAutomata = allowEmpty(automata);
----------------------------------------------------------------------
var FA = require("../Automata/FA/FA.js").default;
var allowEmpty = require('allowEmpty.js').default;
var automata = new FA(/*...*/);
var newAutomata = allowEmpty(automata);
```

Výpis 3.6: Ukázka použití operace

3.6 Zprovoznění

Pro zprovoznění práce je nutné mít nainstalováno následující:

1. *node* verze 9.11.1 a vyšší https://nodejs.org/en/download/
2. *npm* verze 5.8.0 a vyšší https://www.npmjs.com/get-npm,

Práce jako taková může běžet na serveru EVA bez potřeby jakýchkoliv doplňků.

Zprovoznění pro použití:

1. Získání kódů bakalářké práce
2. cd {hlavní složka práce}
3. npm install –production

Zprovoznění pro úpravu:

1. Získání kódů bakalářké práce
2. cd {hlavní složka práce}

3. npm install

Spuštění testů: Zprovozníme pro úpravu, můžeme použít i zprovoznění pro použití, ale pak je potřeba zavolat: *npm install ava*
Máme-li zprovozněno pouze pro použití s AVA, můžeme spustit testy pomocí *npm test*. Máme-li zprovozněno pro úpravu, můžeme použít nasledující příkazy:

1. *npm test* #spustí testy
2. *npm run build* #vygeneruje novou složku **dist** dle změn v **src**
3. *npm run testbuild* #spustí build a následně testy
4. *npm run testES6* #spustí testy nad src
5. *npm run coverage* #spustí testy s "code coverage"
6. *npm run lcovCoverage* #spustí testy s "code coverage", které generují *coverage/lcov-report/index.html*, což je proklikatelná verze "code coverage"

Kapitola 4

Použití operací nad automaty

Spojíme znalosti z 2 a 3 a podíváme se na použití dříve uvedených operací nad automaty.

Ukázky kódů v této kapitole budou pro lepší čitelnost přebírat syntaxi z reálného kódu. Kapitola se bude soustředit především na sémantickou část implementace. Ne všechny uvedené ukázky tedy budou celistvé kusy kódu. Pro podrobné dokreslení bude u každého příkladu uvedena poloha skutečné implementace, již poloha bude uvedena u každého příkladu. Dále v této kapitole už nebudeme používat standardní JavaScript, nýbrž pouze EcmaScript 2015 s Flow, a to z toho důvodu, že by jinak tato kapitola zbytečně nabyla na velikosti.

4.1 Sjednocení (Union)

Návod, jak provést tuto operaci, je předveden v [8] (strany 49-50), a proto se podíváme pouze na její implementaci.

4.1.1 Implementace

V této operaci si ukážeme nejen jak provést sjednocení, ale jak ho provést nad několika typy automatů. I když se jedná o relativně jednoduchou operaci, ukážeme si zde její kód.

```
1  export default function union(left: (Automata | PA | FA), right: (Automata | PA | FA)) {
2    //Převedeme si automat na jeho serializovatelnou reprezentaci
3    let plainLeft: T_AnyPlainAutomata = toPlainLeft(left),
4        plainRight: T_AnyPlainAutomata = toPlainRight(right);
5    ...
6    let plainUnion:T_AnyPlainAutomata = {
7        /*nový automat*/
8    };
9    ...
10   let paFun = /*funkce pro zásobníkový automat*/;
11   let faFun = /*funkce pro konečný automat*/;
12   ...
13   // Nastavíme jaká funkce se má volat pro kterou kombinaci automatu
14   return overload([
15       {parameters: [{value: left, type: FA}, {value: right, type: PA}], func: paFun},
16       /*další kombinace automatu*/
17   ]);
18 }
```

Výpis 4.1: Ukázka implementace operace Union (*src/operations/union.js*)

4.2 Průnik (Intersection)

4.2.1 Provedení nad automatem

Průnik dvou automatů je jednoduchá, avšak dosti zdlouhavá operace. Její aplikaci si předvedeme na příkladu. Mějme dva automaty :
$M = \{Q_M, \Sigma_M, R_M, s_M, F_M\}$ a $N = \{Q_N, \Sigma_N, R_N, s_N, F_N\}$
Průnik poté provedeme následovně:

$$
\begin{aligned}
Intersection(M, N) = \{ & \\
& Q = Q_M \times Q_N, \\
& \quad \Sigma = \Sigma_M \cap \Sigma_N \\
& \quad R = \{(q_M, q_N)\alpha \longrightarrow (q_M\alpha, q_N\alpha); \\
& \qquad q_M \in Q_M \wedge q_N \in Q_N \wedge \alpha \in \Sigma \\
& \quad \}, \\
& \quad s = (s_M, s_N), \\
& \quad F = F_M \times F_N \\
\}
\end{aligned}
\tag{4.1}
$$

4.2.2 Implementace

Implementace zde pouze opisuje postup vytvoření automatu. (Soubor *src/operations/intersectionFA.js*)

4.3 Doplněk (Complement)

4.3.1 Provedení nad automatem

Provedení této operace nad automatem vyžaduje, aby měl automat takzvaný uklízecí stav (trap state). Uklízecí stav je takový stav, který je nekoncový, a jsou do něj odvedeny přechody, které nemohou vést do jiných stavů.

4.3.2 Implementace

Implementace této operace je ve své podstatě velice jednoduchá, jediné, co je potřeba, je uvědomit si teorii uvedenou výše, tedy že musíme mít uklízecí stav. Poté pouze uděláme z ukončujících stavů neukončující, a naopak.
(*src/operations/complementFA.js*)

4.4 Rozdíl (Difference)

4.4.1 Provedení nad automatem

Zde pouze sledujeme myšlenku ze sekce 2.4 a používáme již existující operace.

4.4.2 Implementace

(*src/operations/differenceFA.js*)

4.5 Rozdílné sjednocení (Different Union)

4.5.1 Provedení nad automatem

Hlavní podmínkou provedení této operace je porovnat dva jazyky. Jsme-li schopni rozhodnout o jejich rovnosti, poté je uzavřenost a celková proveditelnost operace ponechána jen na proveditelnosti sjednocení. Dva jazyky můžeme považovat za shodné, pokud rozdíl těchto jazyků generuje vždy prázdnou množinu v libovolném pořadí použití. Zde je však vidět, že ne všechny typy jazyků lze porovnávat, jelikož ne všechny typy jazyků jsou uzavřeny nad operací rozdíl, Dva automaty považujeme za shodné, pokud přijímají tentýž jazyk.

4.5.2 Implementace

V implementaci si musíme dávat pozor na porovnávání přijímaných jazyků, avšak pokud se budeme držet teorie a použijeme dříve implementovanou operaci rozdílu. je implementace pouze otázkou dvou podmínek. (*src/operations/differentUnionFA.js*)

4.6 Operace "Rozdílné"(Operation Different)

4.6.1 Provedení nad automatem

Provedení je stejné jako nad jazyky, viz 2.11, tedy použijeme již existující operace.

4.6.2 Implementace

(*src/operations/operationDifferentFA.js*)

4.7 Konkatenace (Concatenation)

Návod, jak provést tuto operaci, je předveden v předmětu [8] (strana 51-52), a její implementace je s ní totožná.

4.7.1 Implementace

(*src/operations/concatenationFA.js*)

4.8 Unikátní Konkatenace (Konkatenace)

Tuto operaci lze provést pokud víme, jak funguje konkatenace (4.7) a rozdíl (4.4) . Následně můžeme operaci implementovat použitím vzorce 2.17.

4.9 Implementace

(*src/operations/uniqueConcatenationFA.js*)

4.10 Předpony (Prefixes)

4.10.1 Provedení nad automatem

Zde je potřeba si uvědomit co je to prefix řetězce. Je to prázdný řetězec, první znak, první dva znaky a tak dále, až po poslední znak včetně. A tedy potřebujeme jen donutit automat, aby přijímal všechny řetězce, které mohou vést k řetězci, jenž by byl přijat původním automatem. Najdeme si tedy všechny ukončující stavy, uděláme z nich stavy koncové.

4.10.2 Implementace

```
1  //@flow
2  /*Import potřebných závislostí*/
3
4  /**
5  * Generuhe automat přijímající prefixy zadaného automatu
6  * @param {FA} automata
7  * @return {FA}
8  */
9  export default function prefixes(automata: FA): ?FA {
10  //vytvoříme nový automat ze starého
11  let resAutomata = automata.clone();
12  resAutomata.removeTrapStates(); //odstraníme uklízecí stavy
13
14   //všechny stavy se stávají koncovými
15  for(let state of objectValues(resAutomata.states)) {
16    state.setAsFinal();
17  }
18  resAutomata.finalStates = _.clone(resAutomata.states);
19
20   //pro úhlednost přidáme uklízecí stav
21   resAutomata.ensureOneTrapState();
22
23   return resAutomata;
24 }
25
```

Výpis 4.2: Ukázka implementace Předpon (*src/operations/prefixesFA.js*)

4.11 (Shuffle)

4.11.1 Provedení nad automatem

Aplikace se zpočátku může zdát složitá. Uvědomíme-li si však, že můžeme používat složení automatů v kartézském součinu pro sledování cesty, ve kterém automatu se pohybujeme, zjistíme, že stačí provést pouze následující:
Mějme dva automaty :

$M = \{Q_M, \Sigma_M, \delta_M, s_M, F_M\}$ a $N = \{Q_N, \Sigma_N, \delta_N, s_N, F_N\}$

Promíchání poté provedeme následovně:

$$
\begin{aligned}
Shuffle(M,N) = \{ & \\
& Q = Q_M \times Q_N, \\
& \Sigma = \Sigma_M \cup \Sigma_N \\
& R = \{ \\
& \qquad (q_M, q_N)\alpha \longrightarrow (q_M\alpha, q_N) \vee (q_M, q_N\alpha); \\
& \qquad q_M \in Q_M \wedge q_N \in Q_N \wedge \alpha \in \Sigma \\
& \qquad \}, \\
& s = (s_M, s_N), \\
& F = F_M \times F_N \\
& \}
\end{aligned}
\tag{4.2}
$$

4.11.2 Implementace

Jak je uvedeno výše v zápisu automatu, promíchání probíhá tak, že se posouváme dle pravidla automatu M, nebo automatu N. Viz následující ukázka kódu, kde si ukážeme generování pravidel pro Shuffle.

```
1  /**Anotace*/
2  function createRules(
3    left: FA,
4    right: FA,
5    newStates: { [key: string]: MergedState }
6  ): Rule[] {
7    let newRules = [];
8    //pro každý nový stav generujeme pravidla
9    for (let newState: MergedState of objectValues(newStates)) {
10     //generujeme pravidla z levého automatu
11     let leftRules = /*Filtrovaná pravidla levého automatu, pro tento stav*/;
12     for (let rule: Rule of leftRules) {
13       newRules.push(new Rule({
14         from: {state: newState},
15         symbol: rule.symbol,
16         to: {state: newStates[MergedState.createName(rule.to.state,   newState.oldRight)]}
17       }));
18     }
19
20     //generujeme pravidla z pravého automatu
21     /*Obdobně jako pro lévé*/
22       ...
23         to: {state: newStates[MergedState.createName(newState.oldLeft, rule.to.state)]}
24       ...
25   }
26   return newRules;
27 }
```

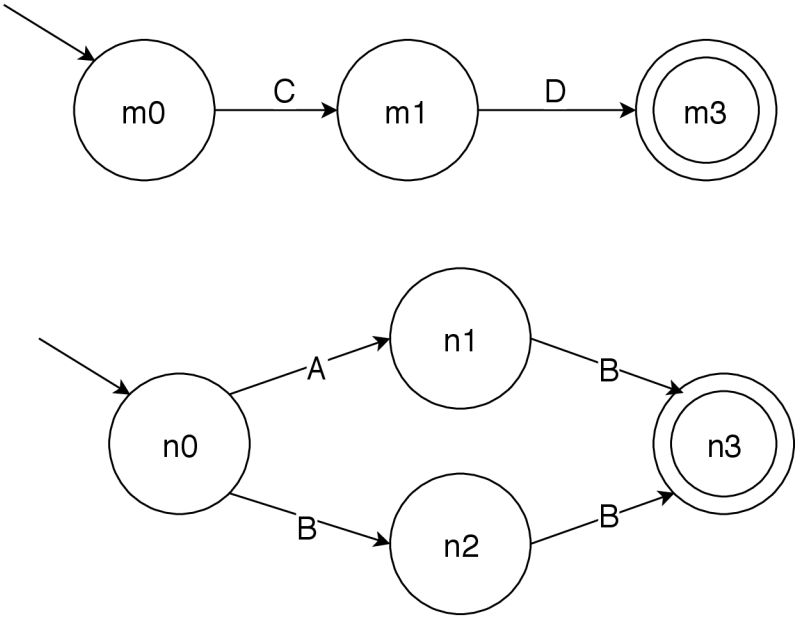
Výpis 4.3: Ukázka generování přechodů pro shuffle (*src/operations/shuffleFA.js*)

4.12 Sekvenční Vložení (Sequential Insertion)

4.12.1 Provedení nad automatem

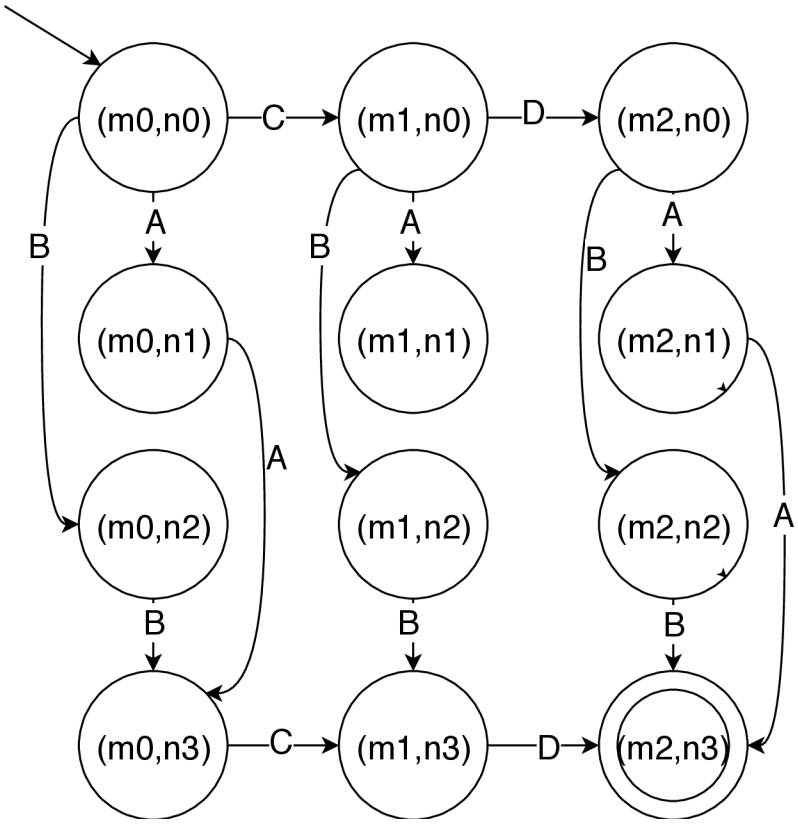
Vzhledem k tomu, že tato operace je podobná operaci Shuffle, můžeme nad ní uvažovat velice podobně. Zde však s tím rozdílem, že musíme vložit řetězec z druhého jazyka nerozdělený. Nejlépe ukážeme na příkladu:

Mějme dva jazyky $K(M) = \{CD\}$ a $L(N) = \{AA, BB\}$ a automaty, kterými jsou definovány (viz 4.1).



Obrázek 4.1: Příklad automatů pro Sequential Insertion

Nyní nad těmito jazyky budeme chtít provést sekvenční vložení N do M, tedy $SequentialInsertion(M, N)$. Zde si je potřeba uvědomit, že můžeme provést kartézský součin k tomu, abychom sledovali, ve kterém jsme právě automatu. Pro jednodušší pochopení si představme, že automat M je osa X a automat N je osa Y. Sekvenční vložení nám pak říká, že musíme řetězec přijímaný automatem N vložit kamkoliv do řetězce přijímaného automatem M. Tedy se můžeme pohybovat přechody po ose X libovolně, ale ve chvíli, kdy se přesuneme po ose Y, musíme po této ose dojít až na konec řetězce přijímaného automatem N (viz. 4.2).



Obrázek 4.2: Výsledek po použití operace Sequential Insertion

Tento příklad si můžeme zobecnit následujícím způsobem. Mějme dva automaty : $M = \{Q_M, \Sigma_M, R_M, s_M, F_M\}$ a $N = \{Q_N, \Sigma_N, R_N, s_N, F_N\}$
Sekvenční vložení N do M provedeme následovně:

$$
\begin{aligned}
Insertion(M,N) = \{ & \\
& Q = Q_M \times Q_N, \\
& \Sigma = \Sigma_M \cup \Sigma_N \\
& R = \{(q_M, q_N)\alpha \longrightarrow \begin{cases} (q_M\alpha, q_N) & if \quad q_N = s \vee q_N \in F_N \\ (q_M, q_N\alpha) & else \end{cases}; \\
& \qquad q_M \in Q_M \wedge q_N \in Q_N \wedge \alpha \in \Sigma \\
& \qquad \}, \\
& s = (s_M, s_N), \\
& F = F_M \times F_N \\
\}
\end{aligned}
\tag{4.3}
$$

4.12.2 Implementace

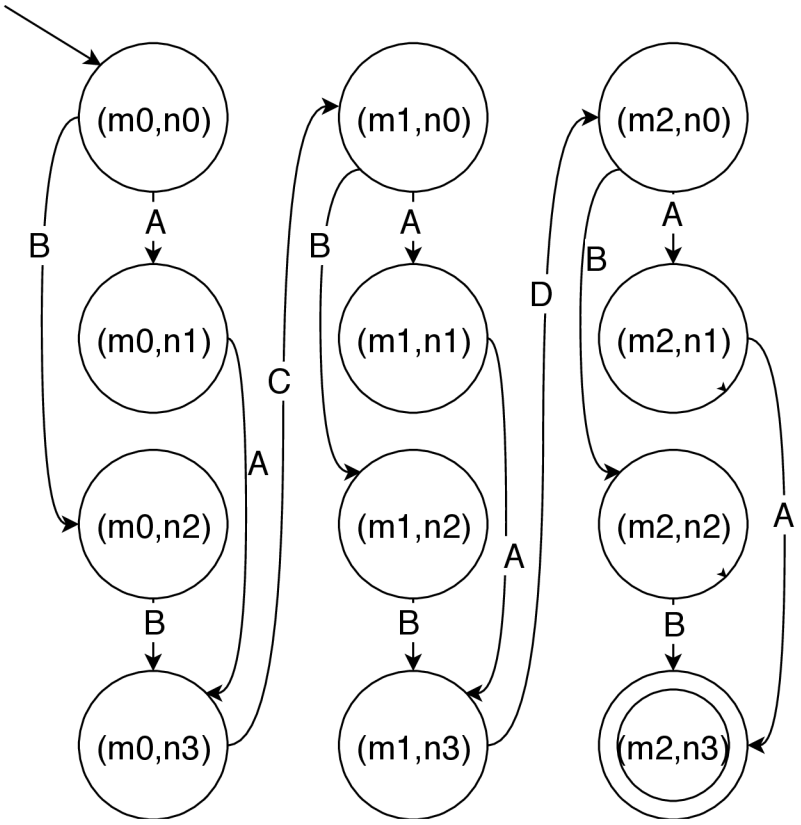
Pravidla se vytváří dle výše uvedeného automatu, obdobně jako tomu bylo v ukázce 4.3 (*src/operations/sequentialInsertionFA.js*).

4.13 Paralelní vkládání (Paralel Insertion)

4.13.1 Provedení nad automatem

Provedení paralelního vkládání je velice podobné vkládání sekvenčnímu. Nejjednodušší bude si opět uvést příklad. Mějme tedy opět dva jazyky $K(M) = \{CD\}$ a $L(N) = \{AA, BB\}$ a automaty, kterými jsou definovány, viz výše uvedený příklad pro sekvenční vkládání (viz 4.1).

Rozdíl je zde v tom, že při paralelním vkládání musíme vždy po zpracování znaku z řetězce jazyka $K(M)$, zpracovat celý řetězec z jazyka $L(N)$. Abychom použili analogii na na osy X a Y, tak vždy, když se chceme posunout po ose X, musíme se posunout po ose Y, až do konečného stavu (viz 4.3).



Obrázek 4.3: Automat generovaný operací ParalelInsertion(N,L)

Tento příklad si můžeme zobecnit následujícím způsobem. Mějme dva automaty : $M = \{Q_M, \Sigma_M, R_M, s_M, F_M\}$ a $N = \{Q_N, \Sigma_N, R_N, s_N, F_N\}$

Paralelní vkládání poté můžeme zobecnit následovně:

$$
\begin{aligned}
ParalelInsertion(M,N) = \{ & \\
& Q = Q_M \times Q_N, \\
& \Sigma = \Sigma_M \cup \Sigma_N \\
& R = \{(q_M,q_N)\alpha \longrightarrow \begin{cases} (q_M\alpha, s_N) & if \quad q_N \in F_N \\ (q_M, q_N\alpha) & else \end{cases}; \\
& \qquad q_M \in Q_M \wedge q_N \in Q_N \wedge \alpha \in \Sigma \\
& \qquad \}, \\
& s = (s_M, s_N), \\
& F = F_M \times F_N \\
\}
\end{aligned}
\tag{4.4}
$$

4.13.2 Implementace

Pravidla se vytváří dle výše uvedeného automatu, obdobně jako tomu bylo v ukázce 4.3 (*src/operations/parallelInsertionFA.js*)

4.14 Protkání (Interlacement)

V této operaci postupujeme tak, že vytváříme automat přesně tak, jak je automat vytvořen v důkazu.

4.14.1 Implementace

Pravidla se vytváří dle výše uvedeného automatu, obdobně jako tomu bylo v ukázce 4.3 (*src/operations/interlacementFA.js*)

4.15 Sekvenční mazání (Sequential Deletion)

4.15.1 Implementace nad automaty

Implementace sekvenčního mazání nad automaty se řídí postupem uvedeným v důkazu [7] strana 59-61.

Vytvoření výsledného automatu si však můžeme zkrátit do následujících kroků:
(Pro kroky uvažujme zadané automaty $M = \{Q_M, \Sigma, R_M, s_M, F_M\}$ a $N = \{Q_N, \Sigma, R_N, s_N, F_N\}$, s tím, že chceme vytvořit $M'' = SequentialDeletion(M,N)$)

1. Nalezneme všechny pod-jazyky/pod-automaty automatu M (nazveme je jako jazyky $L_{q,q'}$).
2. Vytvoříme nový automat $M' = \{Q_M, \Sigma \cup \#, R_{M'}, s_M, F_M\}$

   Kde # je nový symbol nenacházející se v Σ

   A kde $R_{M'} = R_M \cup \{q\# \longrightarrow q' | q, q' \in Q_M \wedge L_2 \cap L_{q,q'}\}$
3. Provedeme homomorfismus a získáme $M'' : M'' = h(L(M') \cap \Sigma^*\#\Sigma^*)$

   Kde $h : h(\#) = \epsilon, h(\alpha) = \alpha; \forall a \in \Sigma$

Což ve výsledku znamená, že v $L(M')$ necháme jen ty řetězce, které obsahují #, a následně nahradíme všechny # za ϵ,

4.15.2 Implementace

Implementace této operace je relativně složitá záležitost, a proto jí uvedeme celou v příloze A.

4.16 Zakázání abecedy (Full Alphabet deletion)

Provedení této operace je dosti primitivní, a tedy i její implementace je snadná. De facto pouze nahradíme všechny znaky v zadané abecedě za ϵ
(*src/operations/fullAlphabetDeletionFA.js*)

4.17 (Pop)

4.17.1 Implementace nad automaty

Implementace této operace je dosti podobná Sekenčnímu mazání, s následujícími rozdíly:

1. Při implementaci LPop:

   Použijeme stejný postup až na to, že libovolné q nahradíme za s_M.

2. Při implementaci RPop:

   Použijeme stejný postup s tím rozdílem, že libovolné q nahradíme ta libovolné f kde $f \in F_M$

4.17.2 Implementace

Jak bylo zmíněno výše, implementace je téměř totožná s implementací SequentialDeletion, a to i co se týče kódů. Při Implementaci tedy pouze uvažujeme změny popsané výše, ale jinak implementujeme stejně jako SequentialDeletion (viz příloha A).

Kapitola 5

Testování

Testování v rámci implementace této práce je provedeno stejně jak pro prostředí tak pro operace samotné. Testování je zde na úrovni integrační až systémové, tedy žádná metoda-/funkce není testována sama o sobě, ale tak, jak patří k celku.

5.1 Testování prostředí

Testování prostředí, zde míněno jako testování automatu, probíhalo jak nad konečnými tak nad zásobníkovými automaty. Testy pro základní operace, včetně instanciace, celého automatu, jsou dělány tak, že se kontroluje stav instance proti zadanému automatu.

```
1  runTest('automataProcessed', test => {
2    let automata = new FA(plain);
3    test.is(Object.keys(automata.states).length, 2);
4    test.true(automata.states['s'].isInitial);
5
6    ...
7
8    test.is(automata.alphabet.length, 1);
9    test.is(automata.alphabet[0], 'a');
10
11   ...
12
13   test.is(automata.rules[0].to.state.name, 'f');
14   test.is(automata.rules[1].from.state.name, 'f');
15   test.is(automata.rules[1].to.state.name, 'f');
16 });
```

Výpis 5.1: Ukázka testů automatu (*tests/automata.test.js*)

Výše uvedený test by se dal nazvat i jednotkovým testem, ale vzhledem k tom, že na to nebyl kladen důraz, a implementace by se teoreticky mohla změnit tak, že by toto tvrzení již neplatilo, nebudeme ho tak uvádět.

Další testy prostředí by tetovaly vstupy a výstupy, aby se zjistilo, jestli se kupříkladu metoda nad automatem chová dle požadovaných vlastností.

5.2 Testování operací

Testování operací je provedeno tak, že je operace vždy testována nad automatem a kontroluje se, jestli to, co s automatem provedla, odpovídá tomu, co by měla v teorii dělat.

U jednodušších operací, jako je kupříkladu sjednocení, bylo možné provést testování každého stavu automatu a zjistit, jestli je nastaven tak, jak má být, si ukážeme na příkladu:

```
runTest('unionAutomata', test => {
  let l_automata = new FA(plain), r_automata = new FA(plain);

  let unionAutomata = union(l_automata, r_automata);

  test.true(unionAutomata.states['l_s'] instanceof State);
  ...
  test.is(unionAutomata.alphabet.length, 1);
  test.is(unionAutomata.alphabet[0], 'a');

  test.is(Object.keys(unionAutomata.finalStates).length, 2);
  ...
  test.is(unionAutomata.rules[4].to.state.name, 'r_f');
  test.is(unionAutomata.rules[5].from.state.name, 'r_f');
  test.is(unionAutomata.rules[5].to.state.name, 'r_f');
});
```

Výpis 5.2: Ukázka testů jednoduché operace (*tests/union.test.js*)

U složitějších operací by takovýto postup nebyl tak jednoduše možný, jelikož predikce toho, co by bylo výstupním automatem, by musela být naprosto přesná, což by dělalo pokus o takovéto testování příliš náročný. Proto byl u složitějších operací zvolen přístup testování přes přijímané řetězce, tedy pomocí metody Automata.accepts. Tím se kontroluje, jestli automat přímá řetězce, které má, a nepřijímá ty, které nemá. Viz ukázka níže.

```
runTest('simple', test => {
  let automata = parallelInsertion(new FA(simple2), new FA(simple1));
  test.true(automata.accepts('aacaadaa'));
  test.true(automata.accepts('aacaadbb'));
  ...
  test.true(automata.accepts('bbcbbdbb'));

  test.false(automata.accepts('aaacadaa'));
  ...
  test.false(automata.accepts('bcbaadaa'));
});
```

Výpis 5.3: Ukázka testů složitější operace (*tests/parallelInsertion.test.js*)

5.3 Code Coverage

Jak bylo uvedeno výše, pokrytí testů bylo prováděno pomocí Istanbul [5], a proto si zde uvedeme pouze celkový výstup převedený do tabulky:

Výrazy%	Větve%	Funkce%	Řádky%
97.66	86.97	96.94	98.07

Tabulka 5.1: "Code Coverage"Implementace

Kapitola 6

Závěr

V této práci jsme se seznámili s několika existujícími, i když ne všem potencionálním čtenářům známými operacemi(Existující, ale obecně ne tolik známou operací, může být kupříkladu Paralelní mazání.). Následně jsme si ukázali několik vlastních operací a zabývali se jejich vlastnostmi. Tato práce důrazně poukazuje na fakt, že operace v teoretické informatice nejsou uzavřená část, a že je pořád stále čím přispívat, a to i co se týče již existujících operací (kupříkladu co se týče iterativního použití nad určitými rodinami jazyků).

Dalším z mých cílů v této práci bylo vytvořit univerzální nástroj, na který bude moci kdokoliv navázat a mimo jiné mu udělat grafické rozhraní a použít pro výuku. Tento cíl je zcela splněn a implementace je umístěna na stránce www.npmjs.com/package/automata-operations, odkud je možné ji získat pouhým "npm install automata-operations".

Doufám, že má práce otevírá různé možnosti kontinuity - jako například vytvoření interaktivního nástroje pro výuku základů končených automatů a operací nad nimi

Literatura

[1] Dalton, J.-D.; aj.: *Lodash*. Duben 2018, [Online; verze 4.17.10 a vyšší; navštíveno 03.05.2018].
URL https://lodash.com/

[2] ECMA International: *Standard ECMA-262 - 6th Edition*. Červen 2015.
URL https://www.ecma-international.org/ecma-262/6.0/

[3] Facebook: *Flow*. Duben 2018, [Online; navštíveno 03.05.2018].
URL https://flow.org/

[4] Facebook: *flow-bin*. Duben 2018, [Online; verze 0.71.0 a vyšší; navštíveno 03.05.2018].
URL https://github.com/flowtype/flow-bin

[5] Istanbuljs tým: *Istanbul*. Apr 2018, [Online; navštíveno 03.05.2018].
URL https://istanbul.js.org/

[6] Istanbuljs tým: *nyc*. Apr 2018, [Online; verze 11.7.1 a vyšší ; navštíveno 03.05.2018].
URL https://github.com/istanbuljs/nyc/

[7] Kari, L.: *On Insertion and Deletion in Formal Languages*. University of Turku, 1991.

[8] Meduna, A.: *Formal Languages and Computation, Models and their applications*. CRC Press, 2014.

[9] Travis CI tým: *Travis CI*. [Online; navštíveno 08.05.2018].
URL https://about.travis-ci.com

[10] Tým Babel: *babel-cli*. Říjen 2017, [Online; verze 6.26.0; navštíveno 03.05.2018].
URL https://github.com/babel/babel/tree/master/packages/babel-cli

[11] Tým Babel: *babel-plugin-transform-object-rest-spread*. Říjen 2017, [Online; verze 6.26.0; navštíveno 03.05.2018].
URL
https://www.npmjs.com/package/babel-plugin-transform-object-rest-spread

[12] Tým Babel: *babel-preset-es2015*. Říjen 2017, [Online; verze 6.26.0; navštíveno 03.05.2018].
URL
https://github.com/babel/babel/tree/master/packages/babel-preset-es2015

[13] Tým Babel: *babel-preset-flow*. Říjen 2017, [Online; verze 6.23.0; navštíveno 03.05.2018].

URL
https://github.com/babel/babel/tree/master/packages/babel-preset-flow

[14] Tým Babel: *babel-register-cli*. Srpen 2017, [Online; verze 5.0.0; navštíveno 03.05.2018].
URL https://github.com/babel/babel/tree/master/packages/babel-cli

[15] Tým Babel: *Babel*. Duben 2018, [Online; verze 6.26.3; navštíveno 03.05.2018].
URL https://flow.org/

[16] Wubben, M.; Sorhus, S.; Demedes, V.: *AVA*. Duben 2017, [Online; verze 1.0.0-beta.3; navštíveno 03.05.2018].
URL https://github.com/avajs/ava

Příloha A

Implementace operace Sequential deletion

```
1  //@flow
2  import FA from '../Automata/FA/FA';
3  import Alphabet from "../Automata/Alphabet";
4  import {objectValues} from "../Automata/services/object";
5  import State from "../Automata/State/State";
6  import Rule from "../Automata/Rule";
7  import intersectionFA from "./intersectionFA";
8
9  /**
10 * Operace sekvenčního mazání
11 * @param {FA} left
12 * @param {FA} right
13 * @param {string} specialSymbol
14 * @return FA
15 */
16 export default function sequentialDeletion(left: FA, right: FA, specialSymbol: string = '#'
       ): FA {
17 //naklonujeme automaty, abychom nezasahovali do puvodních
18   left = left.clone();
19   right = right.clone();
20   //odstraníme epsilon přechody
21   left.removeEmptyRules();
22   right.removeEmptyRules();
23   //sjednotíme abecedy
24   left.alphabet = right.alphabet = new Alphabet(...right.alphabet, ...left.alphabet,
       specialSymbol);
25
26   // vytvoříme kopii levého automatu (jako wprapper pro L_q,q')
27   let copyLeft = left.clone();
28   // tato kopie
29   copyLeft.initialState.isInitial = false;
30   for (let state of objectValues(copyLeft.finalStates)) {
31     state.isFinal = false;
32   }
33
34   //převedeme stavy do pole
35   let copyLeftStates = objectValues(copyLeft.states);
36
37   //pro všechny kombinace q a q' jako počáteční a koncový stav si přidáme #   pravidla do
       levého automatu
```

```
38    // (levý automat se tak rovnou stává i automatem v teorii jako M')
39    for (let newInitialState: State of copyLeftStates) {
40      for (let newFinalState: State of copyLeftStates) {
41        //nastavíme procházené stavy jako počáteční a koncový
42        copyLeft.initialState = newInitialState;
43        newInitialState.isInitial = true;
44        copyLeft.finalStates = {[newFinalState.name]: newFinalState};
45        newFinalState.isFinal = true;
46
47        //naklonujeme takto upravený levý automat a zbavíme se nepotřebných stavu
48        let partLeft = copyLeft.clone();
49        partLeft.removeUnreachableStates();
50
51        //pokud nový počáteční a koncový stav propojeny
52        if (Object.keys(partLeft.finalStates).length > 0) {
53          //zbavíme se dalších nepotřebných stavu
54          partLeft.removeTrapStates();
55
56          //provedeme prunik s pravým automatem a zbavíme se nepotřebných stavu
57          let intersection = intersectionFA(partLeft, right);
58          intersection.removeUnreachableStates();
59          // pokud stále existuje cesta mezi q a q', vytvoříme mezi nima # přechod
60          if (Object.keys(intersection.finalStates).length > 0) {
61            left.rules.push(new Rule({
62              from: {state: left.states[newInitialState.name]},
63              to: {state: left.states[newFinalState.name]},
64              symbol: specialSymbol
65            }));
66          }
67        }
68
69        //uklidíme po práci cyklu
70        newFinalState.isFinal = false;
71        newInitialState.isInitial = false;
72      }
73    }
74
75    return specialRulesToResultAutomata(left, right, specialSymbol);
76  };
77
78  /**
79  * Převádí automat s # přechody na h(L(M') (prunik) Sigma*#Sigma*)
80  * @param {FA} left (jako M')
81  * @param {FA} right
82  * @param {string} specialSymbol
83  * @return FA
84  */
85  export function specialRulesToResultAutomata(left: FA, right: FA, specialSymbol: string):
        FA {
86    //Vytovříme automat přijímající Sigma*#Sigma*
87    let sigmaIterSpecialSigmaIter = new FA();
88
89    //vytvoříme abecedu
90    sigmaIterSpecialSigmaIter.alphabet = left.alphabet;
91
92    //vytvoříme stavy
93    let initState = new State({name: 'start', isInitial: true});
94    let finalState = new State({name: 'end', isFinal: true});
```

```
95    sigmaIterSpecialSigmaIter.states = {[initState.name]: initState,  [finalState.name]:
        finalState};
96    initState.isInitial = true;
97    sigmaIterSpecialSigmaIter.initialState = initState;
98    finalState.isFinal = true;
99    sigmaIterSpecialSigmaIter.finalStates = {[finalState.name]: finalState};
100
101   // vytvoříme pravidla
102   sigmaIterSpecialSigmaIter.rules = [new Rule({
103     from: {state: initState},
104     to: {state: finalState},
105     symbol: specialSymbol
106   })];
107   for (let symbol of [...left.alphabet, ...right.alphabet]) {
108     sigmaIterSpecialSigmaIter.rules.push(new Rule({
109       from: {state: initState},
110       to: {state: initState},
111       symbol: symbol
112     }));
113     sigmaIterSpecialSigmaIter.rules.push(new Rule({
114       from: {state: finalState},
115       to: {state: finalState},
116       symbol: symbol
117     }));
118   }
119
120   //provedeme prunik M' s Sigma*#Sigma* a nahradíme # za prázné přechody
121   let intersection = intersectionFA(left, sigmaIterSpecialSigmaIter);
122   let index = intersection.alphabet.indexOf(specialSymbol);
123   if (index !== -1) intersection.alphabet.splice(index, 1);
124   for (let rule of intersection.rules) {
125       if (rule.symbol === specialSymbol) {
126         rule.symbol = '';
127       }
128   }
129     //odstraníme nepotřebné stavy
130   intersection.removeTrapStates();
131   intersection.removeUnreachableStates();
132
133   return intersection;
134 }
```

Výpis A.1: Implementace Sequential deletion (*src/operations/sequentialDeletionFA.js*)