XML, czyli drzewa z danymi
semestr letni 2010/2011

11 lipca 2011

1 Model dokumentu XML

Założenia Korzystając z używanego na wykładzie modelu dokumentów XML trzeba zdawać sobie sprawę z poniższych rzeczy:

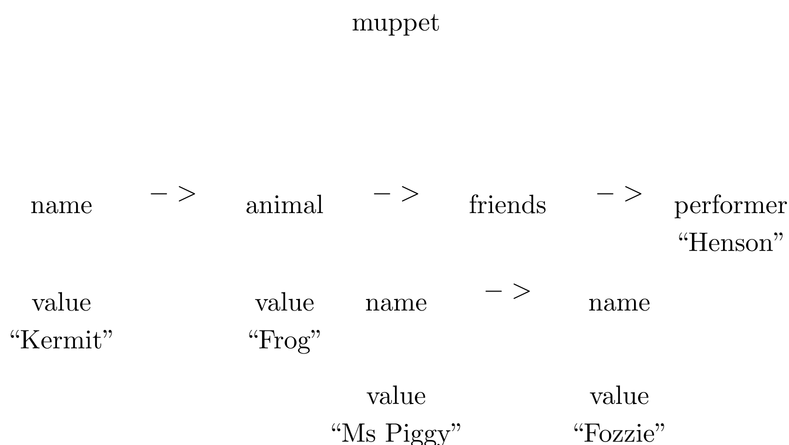
1. Czy kolejność elementów jest istotna?
   Może być, ale nie musi. Można sobie z tym zawsze poradzić.
2. Czy atrybuty i wartości elementów są rozróżnialne?
   Przyjmiemy, że nie są.
3. Czy zawsze ważne są wartości elementów?
   Nie w każdym wypadku będą nas interesowały.
4. Czy dokument XMLowy jest drzewem?
   W ogólności nie, ale dla nas tak.
5. Czy wspomniane drzewo jest **urangowane**?
   W praktycznie każdym interesującym przypadku tak nie jest. Przyjmiemy zatem, że nie.

Drzewo **urangowane** to takie, w którym każdy węzeł albo jest liściem, albo posiada tę samą liczbę dzieci co inne węzły nie będące liśćmi.

Przykład Weźmy poniższy dokument XML:

```
<muppet performer='Henson'>
   <name>Kermit</name>
   <animal>Frog</animal>
   <friends>
      <name>Ms Piggy</name>
      <name>Fozzie</name>
   </friends>
</muppet>
```

A oto i nasz model, który reprezentuje powyższy dokument XML:



2 Drzewa

Drzewa binarne Drzewo **binarne** *domt* to skończony podzbiór $\{0,1\}^*$, który dodatkowo jest zamknięty na prefiksy: $\forall w \in domt \quad (v \sqsubseteq w \Rightarrow v \in domt)$.

Drzewa etykietowane Drzewo etykietowane to funkcja $t : domt \to \Sigma$, gdzie Σ jest skończonym zbiorem etykiet.

Przykład Poniżej jest etykietowane drzewo binarne oraz jego reprezentacja:

ε	a
0	b
1	a
10	c
11	c

Drzewa z dowolną liczbą dzieci w węźle Na wykładzie będą nas interesować drzewa w których węzły mogą mieć dowolną liczbę dzieci. Takie drzewo $domt \subset \mathbb{N}^*$ tak samo jak drzewo binarne jest zbiorem skończonym i zamkniętym na prefiksy. Dodatkowo musi spełniać warunek oszczędności:

$$\forall v \in domt \quad \forall i \in \mathbb{N} \quad (vi \in domt \Rightarrow v0 \in domt, v1 \in domt, ..., v(i-1) \in domt)$$

3 Automaty

Automaty na słowach Dla przypomnienia: automat na słowach to krotka $\langle \Sigma, Q, q_I, \delta, F \rangle$, gdzie zbiór Σ jest alfabetem, nad którym działa automat, Q zbiorem stanów automatu, $q_I \in Q$ stanem początkowym, $\delta \subseteq Q \times \Sigma \times Q$ relacją przejścia między stanami, a $F \subseteq Q$ zbiorem stanów akceptujących. Aby automat był deterministyczny relację przejścia należy zastąpić funkcją przejścia.

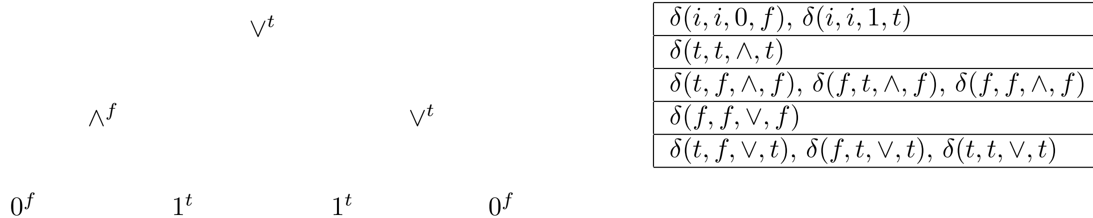
Automaty na drzewach binarnych Aby automat $\langle \Sigma, Q, q_I, \delta, F \rangle$ działał na drzewach, zamiast na słowach, wystarczy zmodyfikować odpowiednio relację przejścia. Jej nowa postać to $\delta \subseteq Q \times Q \times \Sigma \times Q$. Istnieją dwa rodzaje automatów: takie, które przechodzą drzewa z dołu do góry i takie, które robią to z góry do dołu.

Automat idący z dołu do góry akceptuje drzewo, jeśli stan $q_r \in Q$ wyliczony w korzeniu należy do zbioru F. Stan $q \in Q$ dla danego węzła v i jego etykiety $a \in \Sigma$ wyliczamy tak: patrzymy, czy spełniona jest relacja $\delta(q_l, q_r, a, q)$, gdzie q_l oraz q_r są stanami wyliczonymi odpowiednio w lewym i prawym dziecku; jeśli tak, to węzeł v dostaje znajdzie się w stanie q (zauważmy że, jeśli spełnione są relacje $\delta(q_l, q_r, a, q_1)$ oraz $\delta(q_l, q_r, a, q_2)$, to automat wybierze niedeterministycznie jeden ze stanów q_1, q_2). Jeśli węzeł jest liściem, albo posiada tylko jedno dziecko, to w miejsca brakujących dzieci dokładamy węzły ze stanem wyliczonym do stanu początkowego q_I. Teraz można już wyliczyć stany dla każdego węzła drzewa, w szczególności stan q_r dla korzenia.

Automat z góry do dołu działa na odwrót, z tą różnicą, że wyliczony stan węzła $q \in Q$ i jego etykieta $a \in \Sigma$ definiują za pomocą relacji δ stany $\{q_l, q_r\}$ w jego dzieciach, przy czym korzeniowi przypisujemy stan początkowy q_I. Jeśli węzeł nie ma dziecka, któremu chciałby przypisać stan, to umieszcza ten stan w zbiorze stanów końcowych $Q_F \subseteq Q$, na początku $Q_F = \emptyset$. Automat akceptuje drzewo jeśli $Q_F \subseteq F$. Innymi słowy, gdybyśmy utworzyli brakujące dzieci dla wyliczonych stanów, to dla każdego takiego nowego węzła jego stan musi należeć do zbioru stanów akceptujących F.

Przykład Automat $\langle \Sigma, Q, i, \delta, F \rangle$ z dołu do góry definiujący język drzew reprezentujących formuły logiczne ewaluujące się do wartości logicznej "prawda". $\Sigma = \{0, 1, \vee, \wedge\}$, $Q = \{i, f, t\}$, $F = \{t\}$.

Przykładowe drzewo należące do języka i relacja przejścia:



$\delta(i, i, 0, f)$, $\delta(i, i, 1, t)$
$\delta(t, t, \wedge, t)$
$\delta(t, f, \wedge, f)$, $\delta(f, t, \wedge, f)$, $\delta(f, f, \wedge, f)$
$\delta(f, f, \vee, f)$
$\delta(t, f, \vee, t)$, $\delta(f, t, \vee, t)$, $\delta(t, t, \vee, t)$

Fakt Automaty z dołu do góry (zarówno niedeterministyczny jak i deterministyczny) oraz z góry do dołu (tylko niedeterministyczny) posiadają taką samą siłę wyrazu. Automat deterministyczny z góry do dołu jest słabszy.

4 Drzewa nieurangowane

Automaty na drzewach nieurangowanych W drzewie nieurangowanym każdy wierzchołek może mieć dowolną ilość dzieci (będziemy przyjmować, że każdy wierzchołek ma skończenie wiele dzieci). Z tego powodu automat akceptujący drzewa nieurangowane musi mieć nieco inną strukturę niż automat dla drzew binarnych. Intuicyjnie można przyjąć, że jego postać to $\langle \Sigma, Q, \delta, F \rangle$, gdzie relacja pomiędzy stanem i etykietą rodzica a stanami dzieci to $\delta \subseteq Q \times \Sigma \times P(Q)$. Zauważmy, że przy takiej definicji relacji δ stan początkowy jest niepotrzebny — dla liścia zbiór stanów jego dzieci jest zbiorem pustym. Powyższa definicja ma jednak pewną wadę w stosunku do definicji automatu na drzewach binarnych — w zbiorze stanów dzieci nie znamy ani kolejności ich występowania, ani ich liczebności. Dlatego relację δ zdefiniujemy jako $\delta \subseteq Q \times \Sigma \times RegExp(Q)$, gdzie $RegExp(Q)$ to języki (wyrażenia) regularne nad alfabetem będącym zbiorem stanów. Wtedy liściowi ze znakiem a można będzie przypisać stan q jeśli $\delta(q, a, P)$ dla takiego P, które opisuje język, do którego na należy słowo puste.

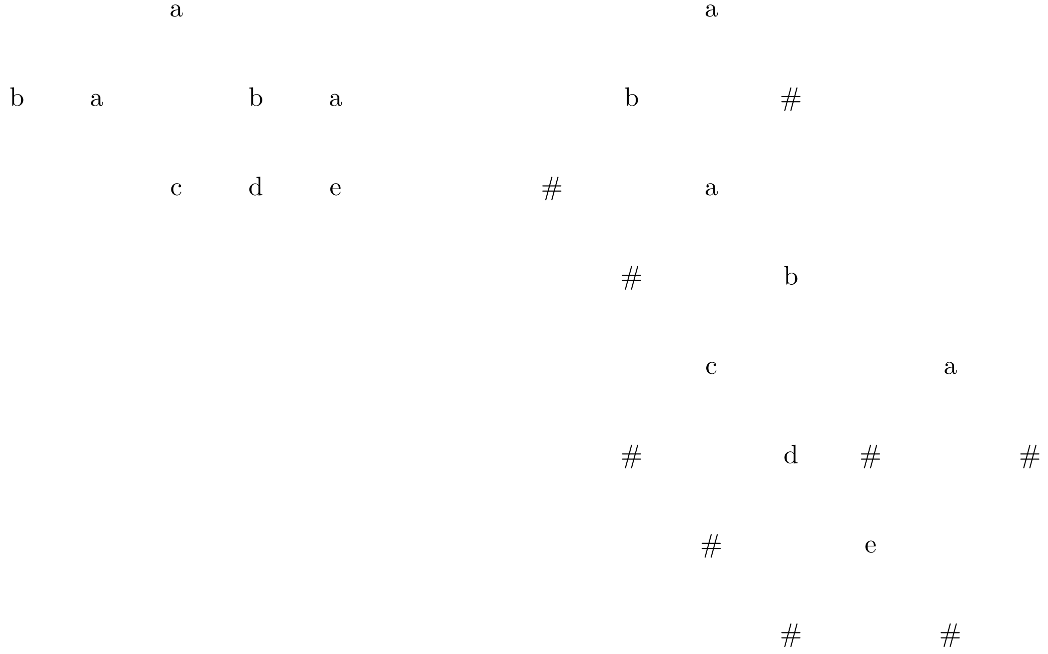
Przykład Automat akceptujący wyrażenia logiczne dające w wyniku wartość *true* wraz z przykładowym akceptowanym drzewem wyrażenia oraz relacją (w tym przypadku funkcją) przejścia między stanami.

$Q = \{t, f\}$
$\Sigma = \{1, 0, \wedge, \vee\}$
$F = \{t\}$
$\delta(t, 1) = \{\epsilon\}$
$\delta(f, 1) = \emptyset$
$\delta(t, 0) = \emptyset$
$\delta(f, 0) = \{\epsilon\}$
$\delta(t, \wedge) = tt^*$
$\delta(f, \wedge) = (t + f)^* f (t + f)^*$
$\delta(f, \vee) = f f^*$
$\delta(t, \vee) = (t + f)^* t (t + f)^*$

$\wedge$
1 $\vee$
$\wedge$ 0
1 1

Przejścia między drzewami binarnymi a drzewami nieurangowanymi Niektóre zagadnienia wygodnie jest rozważać w przestrzeni drzew binarnych, inne w przestrzeni drzew nieurangowanych. Wygodnie jest zatem dysponować możliwością przekształcenia drzewa nieurangowanego do drzewa binarnego. Takie przekształcenie uzyskujemy, wykorzystując np. kodowanie *„first child, next sibling"*.

Przykład Kodowanie drzewa nieurangowanego jako drzewo binarne. "Przekręcamy" drzewo nieurangowane w prawo o 45 stopni i ścieżka w prawo w dół reprezentuje poprzednią ścieżkę do bardziej prawych braci, a ścieżka w prawo w dół od lewego syna reprezentuje dawnych wszystkich synów. Analogicznie odkodowujemy drzewa binarne robiąc z nich drzewo nieurangowane.



Fakt Dla każdego automatu na drzewach nieurangowanych A istnieje automat na drzewach binarnych B taki, że $L(B) = \{\ t \in T^2_{\Sigma\cup\{\#\}} \ \mid\ \exists_{s\in L(A)} encode(s) = t\}$. Mówiąc po ludzku, jeśli jest automat akceptujący język na drzewach nieurangowanych, to też jest automat akceptujący jego tłumaczenie na język drzew binarnych.

Dowód Najpierw podzielmy wyrażenie regularne E na dwie części E_L i E_P tak, by konkatenacja tych części opisywała słowa z języka. Do tego różnych takich podziałów musi być skończenie wiele, a wszystkie języki generowane przez konkatenacje takich podziałów mają się sumować do $L(E)$. Rozpatrzmy taką (przykładową) sytuację: $E = a_2a_3(a_1a_2a_3)^*a_1$. Muszę to wyrażenie umieć podzielić na następujące części:
$[a_2a_3(a_1a_2a_3)^*a_1] \to [][a_2a_3(a_1a_2a_3)^*a_1]$,
$[a_2a_3(a_1a_2a_3)^*a_1] \to [a_2][a_3(a_1a_2a_3)^*a_1]$,
$[a_2a_3(a_1a_2a_3)^*a_1] \to [a_2a_3(a_1a_2a_3)^*a_1][a_2a_3(a_1a_2a_3)^*a_1]$,
$[a_2a_3(a_1a_2a_3)^*a_1] \to [a_2a_3(a_1a_2a_3)^*a_1a_2][a_3(a_1a_2a_3)^*a_1]$,
$[a_2a_3(a_1a_2a_3)^*a_1] \to [a_2a_3(a_1a_2a_3)^*][(a_1a_2a_3)^*a_1]$,
$[a_2a_3(a_1a_2a_3)^*a_1] \to [a_2a_3(a_1a_2a_3)^*a_1][]$,
Ogólny schemat podziału dla wyrażenia z niezagnieżdżoną gwiazdką bez alternatyw jest następujący: szukamy wszystkich podziałów wyrażenia $a_1...a_k(b_1...b_l)^*c_1...c_m$. Podziały, w których jeden ze składników ma wszystkie litery b, są proste. Podział, w którym część litrerek b jest po jednej, a część po drugiej stronie, robimy oddzielając od siebie literki b ze środkowego nawiasu $a_1...a_k(b_1...b_l)^*(b_1...b_l)(b_1...b_l)^*c_1...c_m$. Pozostaje jeszcze podział: pomiędzy nawiasami zgwiazdkowanymi $a_1...a_k(b_1...b_l)^*(b_1...b_l)^*c_1...c_m$ i przypadek, gdy wyrażenie zgwiazdkowane w ogóle nie występuje, czyli wszystkie podziały słowa $a_1...a_kc_1...c_m$. Dla alternatywy: $a_1...a_k(b_1...b_l \cup c_1...c_m)d_1...d_n$ należy dokonać wszystkich możliwych podziałów dla obu opcji.
Zatem należy umieć wygenerować wszystkie podziały wyrażenia $(P_1P_2...P_k)^*$, takiego, że P_i jest pojedynczą literką, alternatywą wyrażeń regularnych lub zgwiazdkowanym wyrażeniem regularnym. Zróbmy podział wewnątrz środkowego nawiasu wyrażenia $(P_1P_2...P_k)^*(P_1P_2...P_k)(P_1P_2...P_k)^*$. Najpierw zlikwidujmy z niego alternatywy i rozpatrzmy wszystkie kombinacje. Niech $(P_1P_2...P_k)^*(Q_1Q_2...Q_l)(P_1P_2...P_k)^*$

będzie jedną z tych kombinacji. Wówczas każde z Q_i jest albo pojedynczą literą, albo zgwiazdkowanym wyrażeniem regularnym. Jeśli podział przebiega między Q_i i Q_{i+1}, to łatwo go zrobić. Jeśli ma przebiegać wewnątrz Q_i takiego, że jest to $(a_1a_2...a_k)^*$, to należy to zrobić tak, jak w przypadku dla niezagnieżdżonych gwiazdek. Jeśli zaś wewnątrz Q_i są jakieś alternatywy, robimy podział rekurencyjnie, bo $Q_i = (P_1P_2...P_m)^*$. Konstrukcja jest skończona, bo za każdym wywołaniem rekurencyjnym odwołuję sie do wyrażenia z mniejszą ilością gwiazdek i alternatyw.
Zatem umiem podzielić wyrażenie regularne na dwa zadane na początku podwyrażenia. Zauważmy też, że jeśli mam słowo wav takie, że $w, v \in \Sigma^*$, $a \in \Sigma$, $wa \in E_L$ oraz $v \in E_R$, to jest jakiś inny podział E'_R i E'_L taki, że $w \in E'_L$ i $av \in E'_R$.
Teraz dla wszystkich reguł generujemy wszystkie możliwe podziały. Niech stanami w nowym automacie będą pary (E_L, E_R) dla pewnego E, które występowało po prawej stronie jakiejś reguły ze starego automatu. Dla każdej pary podziałów E_R, E_L i E'_R i E'_L takiej, że jeśli słowo $wav \in L(E)$, to jeśli $wa \in E'_L$, $v \in E'_R$, to $w \in E_L$, $av \in E_R$ definiujemy przejście: $(E_L, E_R) \to ([], [F]), (E'_L, E'_R)$. F jest tu pewnym wyrażeniem regularnym, takim, że w pierwotnym automacie było przejście $q \to F$, gdzie q jest stanem, o który różnią się E_L i E'_L (a także E_R i E'_R).
Oczywiście, jeśli wierzchołek był wcześniej liściem, to zgodnie z powyższą procedurą jako lewego syna dostanie $([], [\epsilon]) = nil$, który to stan jest nowym liściem (a więc dokładamy jeszcze regułę $nil \to \epsilon$).
Zauważmy, że nowy automat robi dokładnie to, co chcemy. Będąc w jakimś węźle patrzy, czy jego lewy syn (dawniej: wszystkie dzieci) pasuje do wyrażenia regularnego zadanego swoim starym wyrażeniem regularnym, dodatkowo zbierając informacje o tym, czy wciąż pozostaje w słowie opisywanym przez wyrażenie regularne ojca swojego najstarszego lewego przodka (czyli dawniej: swojego ojca).

Fakt Dla każdego automatu na drzewach binarnych A istnieje automat na drzewach nieurangowanych taki, że $L(A) = \{\ t \quad | \quad encode(t) \in L(B)\ \}$.

Dowód Załóżmy, że język drzew binarnych składa się z takich drzew, które mają tylko wierzchołki o dwóch bądź zero synach. Wtedy wszystkie przejścia w funckji przejścia są postaci $q \to p_1p_2$ lub $q \to \epsilon$. Utwórzmy graf, którego wierzchołkami są stany, a krawędź między stanem p i q jest wtedy, gdy istnieje produkcja $p \to rq$ dla pewnego r. W takim grafie łatwo znaleźć wszystkie cykle proste. Następnie dla wszystkich par (q, l), gdzie q jest dowolnym stanem, a l to stan występujący w liściu (czyli taki, że jest produkcja $l \to \epsilon$), sprawdzamy wszystkie ścieżki niezawierające cyklu, które prowadzą w grafie od q do l. Wyrażenie regularne konstruujemy w następujący sposób: bierzemy alternatywę ścieżek, a dla każdej ścieżki w miejsce, w którym można dołożyć cykl, dokładamy go pod gwiazdką, rekurencyjnie sprawdzając, czy dokładając cykl nie poszerzyliśmy tym samym zbioru cykli, które można dołożyć. Jeśli w któreś miejsce da się dołożyć kilka cykli, dokładamy zgwiazdkowaną alternatywę. Wyrażenie regularne zostaje podpięte pod taki stan p, który ma produkcję $p \to qr$ dla pewnego r. Tak utworzone reguły tworzą odpowiadający język na drzewach nieurangowanych.

5 DTD

Definicja DTD (czyli *Data Type Definition*) to jeden ze sposobów definiowania typów dokumentów XMLowych. DTD jest również automatem niedeterministycznym na drzewach, jednak o konstrukcji innej niż dotychczas — zakładamy, że każda etykieta jest jednocześnie stanem. Formalnie jest to krotka $\langle \Sigma, \delta, \gamma_0 \rangle$, gdzie γ_0 to korzeń drzewa/dokumentu, Σ to zbiór etykiet, a δ to *funkcja* postaci $\delta : \Sigma \to RegExp(\Sigma)$.

Definicję tę interpretujemy następująco — drzewo t spełnia D $= \langle \Sigma, \delta, \gamma_0 \rangle$ o ile:

- $t(\epsilon) = \gamma_0$ (puste drzewo jest akceptowane),
- jeśli $t(v) = \gamma$, to $t(v_0)t(v_1)\ ...\ t(v_n) \in L(\delta(\gamma))$ (etykiety wierzchołków dzieci $v_0\ ...\ v_n$ spełniają wyrażenie regularne przypisane do ich rodzica v).

Automat rozpoczyna pracę w liściach. Idąc do góry patrzy, czy słowo powstałe ze stanów dzieci należy do języka opisanego wyrażeniem regularnym przypisanym do rodzica; w przypadku braku dopasowania zgłasza błąd, wpp. kontynuuje działanie. Warunkiem akceptacji drzewa jest przypisanie korzeniowi stanu γ_0.

Definicja Wyrażenie regularne nazwiemy deterministycznym, jeśli w wyprodukowanym przez nie słowie, dla każdej litery możemy dokładnie wskazać element wyrażenia regularnego, z którego ona pochodzi. Deterministyczne wyrażenia regularne są stosowane w standardach DTD/EDTD definiowania dokumentów XML.

Przykład Podać DTD takie, że najmniejsze drzewo je spełniające ma rozmiar wykładniczy względem rozmiaru δ (ilości przejść stanów).

Rozwiązaniem jest odpowiednia funkcja δ postaci:

Postać funkcji δ:

$$\begin{array}{rcl} a_1 & \rightarrow & a_2 a_2 \\ a_2 & \rightarrow & a_3 a_3 \\ a_3 & \rightarrow & a_4 a_4 \\ & \dots & \\ a_n & \rightarrow & a_{n+1} a_{n+1} \\ a_{n+1} & \rightarrow & \epsilon \end{array}$$

Twierdzenie Regularny język drzew jest definiowalny przez DTD $\Leftrightarrow$ język drzew jest zamknięty ze względu na zamianę poddrzew ukorzenionych w wierzchołkach o tej samej etykiecie.

Dowód

$\Rightarrow$ Oczywiste — jeśli dwa drzewa z języka mają wierzchołek o tej samej etykiecie, biegi automatu akceptującego DTD w obydwu z nich przypisują tym wierzchołkom ten sam stan różny od błędu, zatem zamiana poddrzew nie wpłynie na kontynuacje tych biegów i przypisanie stanu w korzeniach.
$\Leftarrow$ W regularnym języku drzew o podanej własności, dla każdej etykiety a istnieje zbiór stanów Q_a, które można jej przypisać. Skoro można zamieniać ze sobą w dowolny sposób wierzchołki o tych samych etykietach, wszystkie stany z takiego zbioru można zastąpić jednym, charakterystycznym dla litery stanem q_a. Ponieważ język drzew jest regularny, dla każdej etykiety a istnieje wyrażenie regularne na stanach dzieci, w którym wystąpienia stanów ze zbioru Q_a można zastąpić stanem q_a, lub, równoważnie — *etykietą* a, tworząc tym samym regułę dla DTD, będącą wyrażeniem regularnym nad alfabetem etykiet wierzchołków.

6 EDTD

Definicja EDTD (czyli *Extended Data Type Definition*) to kolejny ze sposobów na opisanie struktury dokumentu XML. Jest to automat niedeterministyczny postaci $\langle \Sigma, \Delta, \delta, \tau_0 \rangle$, gdzie τ_0 to typ początkowy, Σ to zbiór etykiet wierzchołków, Δ jest tzw. zbiorem typów, konstruowanych na podstawie etykiet, przez „dodanie” indeksowania[1] — $\Delta \subseteq \Sigma^{\mathbb{N}} = \{ a^i \mid a \in \Sigma \wedge i \in \mathbb{N} \}$. Funkcja δ jest postaci $\delta : \Delta \rightarrow RegExp(\Delta)$. Działanie automatu interpretuje się w następujący sposób — zaczyna on swoje działanie od liści w drzewie. Dla każdego wierzchołka v sprawdzamy, jakie wyrażenie regularne na typach jest spełniane przez wyrażenie złożone z typów jego dzieci — argument funkcji δ, dla którego zwraca ona szukane wyrażenie, staje się stanem wierzchołka v. Drzewo jest akceptowane przez EDTD wtedy, gdy da się je otypować tak, by w każdym wierzchołku z literą $\sigma \in \Sigma$ typ był równy σ^i dla jakiegoś i.

Definicja EDTD jest „single-type”, gdy w każdym wyrażeniu $\delta(a^i)$ występuje tylko jeden typ odpowiadający każdemu $\tau \in \Sigma$ — przykładowo, przejście $a^1 \rightarrow a^2 a^2$ jest dozwolone, ale nie $a^1 \rightarrow a^2 a^3$.

Definicja Język drzew jest zamknięty na ancestor-guarded subtree exchange wtedy, gdy dla dowolnych t_1 i t_2 należących do języka i dla $v_1 \in t_1$ i $v_2 \in t_2$ takich, że od korzenia t_1 do v_1 (włącznie) prowadzi słowo w i od korzenia t_2 do v_2 prowadzi to samo słowo w, gdy zamienimy miejscami (wraz z ich poddrzewami) węzły v_1 i v_2, to powstałe przez tę zamianę drzewa także będą należeć do języka.

[1] Intuicja: dodajemy informację o występowaniu etykiety w pewnym kontekście, np. tag *name* może oznaczać imię człowieka, ale równie dobrze zwierzaka domowego.

Definicja Drzewo t_w o wierzchołkach etykietowanych elementami zbioru Δ nazwiemy świadkiem dla drzewa t etykietowanego znakami z Σ, jeśli t należy do języka akceptowanego przez $\langle \Sigma, \Delta, \delta, \tau_o \rangle$, a t_w powstaje przez otypowanie wierzchołków drzewa t.

Twierdzenie Weźmy L regularny. Równoważne są stwierdzenia:

- (1) L jest zamknięty na *ancestor-guarded subtree exchange*
- (2) L jest opisywalny przez EDTD, który jest *single-type*

Dowód

(2) $\Rightarrow$ (1) Zauważmy najpierw, że każde $t \in L$ ma dokładnie jednego świadka, a typ każdego wierzchołka w świadku zależy tylko od ścieżki prowadzącej do niego z korzenia. Spróbujmy otypować t idąc od góry. Wiemy, że w korzeniu, w którym jest litera σ_0, musi być typ τ_0. Weźmy dowolny otypowany wierzchołek v, do którego z korzenia idzie się po słowie w, a w v jest litera a. W v jest typ a^i. Dla dowolnego σ, który jest etykietą dziecka v definiujemy typ σ^j, gdzie j jest takie, że σ^j występuje w wyrażeniu regularnym $\delta(a^i)$. Typ ten jest wyznaczony jednoznacznie, bo D jest single-type, zatem gdy otypujemy całe drzewo, będzie ono wyznaczone jednoznacznie, a typowanie zależy tylko od tego, co jest powyżej.
Weźmy dwa drzewa $t_1, t_2 \in L$ i $v_1 \in t_1$, $v_2 \in t_2$, takie, że do v_1 i do v_2 schodzi się od korzenia po tym samym słowie w. Zauważmy, że jeśli typowanie zależy tylko od tego, co jest wyżej, to v_1 i v_2 będą miały te same typy, bo typowanie ścieżek powyżej jest jednoznacznie wyznaczone. Teraz gdy zamienimy ze sobą poddrzewa o korzeniach w v_1 i w v_2, dostaniemy również drzewa posiadające świadków (bo w obu zamienionych poddrzewach ścieżki do korzenia się nie zmienią).
(1) $\Rightarrow$ (2) Znajdziemy EDTD z treści (2).
Zdefiniujmy $typy(wa) = \{\tau \in \Delta \mid D$ może wierzchołkowi o etykiecie a i ścieżce od korzenia w przypisać typ $\tau\}$. Zdefiniujmy $\widetilde{D} = \langle \Sigma, \widetilde{\Delta}, \widetilde{\delta}, \widetilde{\tau_0} \rangle$ w następujący sposób. Niech $\widetilde{\Delta} = \{\Gamma \in P(\Delta) \mid \exists w \in \Sigma^*$ takie, że $typy(w) = \Gamma\}$. Niech $\widetilde{\tau_0} = \{\tau_0\}$. Niech wreszcie $\widetilde{\delta}$ będzie taką funkcją, że $\delta(\{a^{i_1}, a^{i_2}, ..., a^{i_k}\}) = \delta'(a^{i_1})|\delta'(a^{i_2})|...|\delta'(a^{i_k})$, a $\delta'(a^{i_j})$ to $\delta(a^{i_j})$, w której wszystkie wystąpienia typu b^l zastąpiono przez $typy(wab)$.
Należy zauważyć, że jeśli $typy(wa) = typy(w'a)$, to także $typy(wab) = typy(w'ab)$, a więc $\widetilde{\delta}$ jest dobrze określone. Istotnie, jeśli dało się dojść po jakimś słowie w do wierzchołka v z etykietą a i możliwym stanem w v jest a^i, to $\delta(a^i)$ jest wyrażeniem regularnym, w którym występują $b^{i_1}, b^{i_2}, ..., b^{i_k}$. Wtedy zarówno $typy(wab)$ jak i $typy(w'ab)$ będą zawierały $b^{i_1}, b^{i_2}, ..., b^{i_k}$. Rozumowanie powtórzone dla wszystkich a^i da w wyniku, że $typy(wab) = typy(w'ab)$.
Zauważmy teraz, że $\widetilde{D}$ jest single-type. Tak jest, bo funkcja przejścia zwraca zawsze wyrażenia regularne, w których są tylko $typy(w)$, a więc wszystkie możliwe biegi automatu dla danego ciągu znaków. Jeśli więc pierwotne δ miało wyrażenie regularne, którego wynikiem było kilka typów tej samej litery, wszystkie te typy znajdą się w zbiorze *typy* i pojawi się on wszędzie tam, gdzie poprzednio były typy tej litery.
Oczywiście wszystkie drzewa zaakceptowane przez D będą także zaakceptowane przez $\widetilde{D}$, bo $\widetilde{D}$ symuluje działanie D. Pokażmy teraz, że $\widetilde{D}$ nie zaakceptuje już nic więcej.

Weźmy dowolne drzewo $g \in L(\widetilde{D})$. Chcemy pokazać, że $g \in L(D)$. Zrobimy to indukcyjnie, krok będzie polegał na tym, że z pewnego podzbioru wierzchołków z g, nazwijmy go S (S domknięty ze względu na relację "przodek"), takiego, że potrafimy skonstruować drzewo $d_1 \in L(D)$, takie, że S jest także podzbiorem d_1 (zamkniętym na relację "przodek"), skonstruujemy drzewo z $L(D)$, które będzie się zgadzało na wszystkich wierzchołkach z S i jeszcze na wszystkich synach jednego z tych wierzchołków.

Zacznijmy od tego, że wszystkie drzewa akceptowane przez $L(D)$ i przez $L(\widetilde{D})$ mają taką samą literę w korzeniu. Zatem potrafimy znaleźć takie drzewo z $L(D)$, które ma taki sam korzeń jak g.

Teraz załóżmy, że mamy zbiór S takich wierzchołków, na których g zgadza się z pewnym drzewem $t_1 \in L(D)$, które umiemy skonstruować. Weźmy wierzchołek v z g, który jest w S, ale którego dzieci nie są w S. Do v da się dojść z korzenia po słowie w, v ma etykietę a. Weźmy $v_1, ..., v_n$ - dzieci v. Zostały one otypowane w g zgodnie z $\widetilde{\delta}$, która dla stanu $\{a^{i_1}, a^{i_2}, ..., a^{i_k}\}$ zwraca alternatywę k wyrażeń regularnych $\delta'(a^{i_1})|\delta'(a^{i_2})|...|\delta'(a^{i_k})$. Jedno z nich musiało zatem zostać użyte do otypowania $v_1, ..., v_n$. Wyrażenie to powstało z funkcji δ, która przypisała je jakiemuś $a^j \in \{a^{i_1}, a^{i_2}, ..., a^{i_k}\}$. Istnieje $t' \in L(D)$ z takim wierzchołkiem u, który od korzenia t' do u ma ścieżkę wa, a typem u jest a^j. Ponieważ a^j jest osiągalny w u po ścieżce wa, można też zażądać, aby dziećmi u były $v_1, ..., v_n$, bo u ma typ a^j, który jest

tak dobrany, by δ przypisała mu wyrażenie regularne akceptujące słowo utworzone z etykiet $v_1, ..., v_n$. Teraz wystarczy zamienić ze sobą poddrzewo o korzeniu v z drzewa t_1 z poddrzewem o korzeniu u z drzewa t'. Możemy to zrobić, bo w $L(D)$ mamy *ancestor-guarded subtree exchange*. Tak zmienione t_1 ma $S \cup \{v_1, ..., v_n\}$ wierzchołków zgadzających się z g. Taka sama zamiana między t_1 a t' może służyć do przycinania zbędnych poddrzew, czyli uzyskaniu liścia w t_1 w miejscu, w którym g także jest liść. To kończy dowód.

7 XPath

7.1 Nawigacja w dokumencie

Rozważmy relacje binarne R_α pomiędzy wierzchołkami dokumentu. Będziemy te relacje nazywać *relacjami osiowymi.* Za relacje pierwotne uznajemy *child* (dziecko) oraz *next-sibling* (następny brat). Definiujemy też oś *self* w następujący sposób: $R_{self} = \{(n, n) : n \in Node\}$ ($Node$ oznacza zbiór wierzchołków dokumentu).

Następnie definiujemy $R_{descendant}$ (potomek) jako domknięcie przechodnie R_{child}, oraz $R_{descendant-or-self}$ jako domknięcie zwrotno-przechodnie R_{child}. $R_{following-sibling}$ jest domknięciem przechodnim $R_{next-sibling}$.

Poprzez relację odwrotną do R rozumiemy relację $\{(n, n') : R(n', n)\}$. Relacje osiowe R_{parent}, $R_{ancestor}$, $R_{ancestor-or-self}$, $R_{preceding-sibling}$, $R_{previous-sibling}$ definiujemy jako relacje odwrotne do R_{child}, $R_{descendant}$, $R_{descendant-or-self}$, $R_{next-sibling}$, $R_{following-sibling}$, odpowiednio.

$R_{following}$ jest złożeniem relacji: $R_{ancestor-or-self} \circ R_{following-sibling} \circ R_{descendant-or-self}$. $R_{preceding}$ to relacja odwrotna do $R_{following}$.

Osie *next-sibling*, *following-sibling*, *previous-sibling*, *preceding-sibling*, *following* oraz *preceding* będziemy nazywać *osiami poziomymi.*

7.2 NavXPath

Zacznijmy od podjęzyka XPath zajmującego się wyłącznie strukturą nawigacyjną dokumentu. Podjęzyk ten nazwiemy NavXPath. Składa się on z wyrażeń, które na wejściu przyjmują wierzchołek, a na wyjściu dają zbiór wierzchołków lub wartość logiczną (prawda/fałsz). Wyrażenia ostatniego typu nazywamy *filtrami.* Według popularnej konwencji $p, p', \ldots$ oznaczają dowolne wyrażenie XPath, a $q, q', \ldots$ oznaczają filtry.

Gramatyka poprawnych wyrażeń NavXPath jest następująca:

$$\begin{aligned} p &\rightarrow step \mid p/p \mid p \cup p \\ step &\rightarrow axis \mid step[q] \\ q &\rightarrow p \mid lab() = L \mid q \wedge q \mid q \vee q \mid \neg q \end{aligned}$$

axis oznacza jedną z osi opisanych powyżej. L to dowolna etykieta należąca do alfabetu Σ.

7.3 Semantyka NavXPath

Wyrażenie p NavXPath interpretujemy jako funkcję $[\![p]\!]_{NodeSet} : Node \rightarrow NodeSet$, gdzie $NodeSet$ oznacza podzbiór wierzchołków dokumentu. Filtr q interpretujemy jako predykat unarny $[\![q]\!]_{Bool} : Node \rightarrow Bool$, gdzie $Bool$ oznacza wartość logiczną (prawda/fałsz).

Semantyka wyrażeń p zdefiniowana jest następująco:

$$[\![axis]\!]_{NodeSet}(n) = \{n' : R_{axis}(n, n')\}$$

$$[\![step[q]]\!]_{NodeSet}(n) = \{n' \in [\![step]\!]_{NodeSet}(n) \wedge [\![q]\!]_{Bool}(n')\}$$

$$[\![p_1/p_2]\!]_{NodeSet}(n) = \{v : \exists w \in [\![p_1]\!]_{NodeSet}(n) \wedge v \in [\![p_2]\!]_{NodeSet}(w)\}$$

$$[\![p_1 \cup p_2]\!]_{NodeSet}(n) = [\![p_1]\!]_{NodeSet}(n) \cup [\![p_2]\!]_{NodeSet}(n)$$

Semantyka filtrów q zdefiniowana jest następująco:

$$[\![lab() = L]\!]_{Bool}(n) \iff Lab_L(n)$$

$$[\![p]\!]_{Bool}(n) \iff [\![p]\!]_{NodeSet}(n) \neq \emptyset$$

$$[\![q_1 \wedge q_2]\!]_{Bool}(n) \iff [\![q_1]\!]_{Bool}(n) \wedge [\![q_2]\!]_{Bool}(n)$$

$$[\![q_1 \vee q_2]\!]_{Bool}(n) \iff [\![q_1]\!]_{Bool}(n) \vee [\![q_2]\!]_{Bool}(n)$$

$$[\![\neg q]\!]_{Bool}(n) \iff \neg [\![q]\!]_{Bool}(n)$$

7.4 Ewaluacja zapytań NavXPath

7.4.1 Znajdywanie wierzchołków spełniających q

Wejście t - dokument, q - zapytanie unarne NavXPath

Wyjście zbiór wierzchołków spełniających q: $\{x \in dom(t) : [\![q]\!]_t(x)\}$, gdzie $dom(t)$ oznacza zbiór wierzchołków t.

Czas działania $O(|q| * |t|)$

Algorytm Definiujemy funkcję $oblicz(q)$, która dla zapytania NavXPath zwraca zbiór wierzchołków spełniających to zapytanie.

$$oblicz(q_1 \cap q_2) = oblicz(q_1) \cap oblicz(q_2)$$
$$oblicz(q_1 \cup q_2) = oblicz(q_1) \cup oblicz(q_2)$$
$$oblicz(\neg q) = dom(t) \setminus oblicz(q)$$
$$oblicz(lab() = L) = \{x \in dom(t) : lab(x) = L\}$$
$$oblicz(p) = obliczp(p, dom(t))$$

Użyta powyżej funkcja *obliczp* zdefiniowana jest następująco:

$$obliczp(p, S \subseteq dom(t)) = \{x \in dom(t) : \exists y \in S \; y \in [\![p]\!]_t(x)\}$$
$$obliczp(p_1/p_2, S) = obliczp(p_1, obliczp(p_2, S))$$
$$obliczp(p_1 \cup p_2, S) = obliczp(p_1, S) \cup obliczp(p_2, S)$$
$$obliczp(step[q], S) = obliczp(step, S \cap oblicz(q))$$
$$obliczp(axis, S) = \{x \in dom(t) : \exists y \in S \; (x, y) \in R_{axis}\}$$

Zauważmy, że w przypadku większości powyższych równań bardzo łatwo znaleźć algorytm realizujący je w czasie proporcjonalnym do wielkości drzewa. Pewne wątpliwości może budzić $obliczp(axis, S)$ (w ogólności relacja binarna może mieć kwadratowy rozmiar). Jednakże dostępne osie są ściśle określone, i dla nich jest możliwe efektywne znajdywanie *obliczp*. Weźmy dla przykładu oś *descendant*. Dla każdego wierzchołka z S zaznaczamy wszystkie wierzchołki na drodze do korzenia (zauważmy, że poruszamy się tutaj "od tyłu": szukamy wierzchołków, dla których *descendant* należy do S). Przez zaznaczanie rozumiem dodawanie do zbioru zwracanych wierzchołków. Zauważmy, że kiedy idąc od wierzchołka do korzenia napotkamy już zaznaczony wierzchołek, nie musimy zaznaczać dalej, bo wiemy że te wierzchołki już zostały oznaczone wcześniej. Dzięki temu każdy wierzchołek zostanie odwiedzony co najwyżej jeden raz.

7.4.2 Spełnialność zapytania

Wejście zapytanie unarne q

Wyjście wartość logiczna: czy istnieje drzewo t takie, że q jest prawdziwe w jakimkolwiek jego wierzchołku

Spełnialność zapytania można sprawdzić poprzez konwersję do automatu.

Czas działania Problem spełnialności jest EXPTIME-zupełny, aczkolwiek po konwersji zapytania na automat jego (nie)pustość możemy sprawdzić w czasie wielomianowym. Zatem istnieją takie zapytania, dla których rozmiar automatu musi być wykładniczy w stosunku do rozmiaru zapytania.

Konstrukcja automatu Stan automatu przechowuje informację, które podformuły q są prawdziwe w danym wierzchołku ($Q = 2^{podf(q)}$). Na przykład dla zapytania $(child[q_1] \cup parent[q_2])/descendant[q_3]$ mamy następujące podformuły:

- q_1, q_2, q_3 i ich podformuły
- $descendant[q_3]$
- $child[q_1]/descendant[q_3]$
- $parent[q_2]/descendant[q_3]$
- $(child[q_1] \cup parent[q_2])/descendant[q_3]$

Dla uproszczenia (ale bez straty ogólności) zakładamy, że w formule nie występują podformuły postaci $axis[z]$ inne niż $self[z]$. Można tak założyć, ponieważ dowolną formułę postaci $axis[z]$ można zastąpić przez $axis/self[z]$. Zakładamy też, że nie występują osie *following* ani *preceding*. Można tak założyć, ponieważ są one złożeniem innych osi.

Automat niedeterministyczny zgaduje zbiór spełnionych podformuł dla każdego wierzchołka, a następnie sprawdza, czy one rzeczywiście są spełnione i czy przyporządkowanie podformuł do wierzchołków jest spójne (z, p oznacza zapytanie, natomiast q oznacza stan automatu):

$$z_1 \cap z_2 \in q \iff z_1 \in q \wedge z_2 \in q$$

$$z_1 \cup z_2 \in q \iff z_1 \in q \vee z_2 \in q$$

$$\neg z \in q \iff z \notin q$$

$$(p_1 \cup p_2)/p_3 \in q \iff p_1/p_3 \in q \vee p_2/p_3 \in q$$

$$child/p \in q \iff \text{w pewnym dziecku jest prawdziwe } p$$

$$p \in q \iff \text{w każdym dziecku } parent/p \in q$$

$$descendant/p \in q \iff \text{w którymś dziecku jest } p \text{ lub } descendant/p$$

$$p \in q \vee ancestor/p \in q \iff \text{w każdym dziecku } ancestor/p$$

$$self[z]/p \in q \iff z \in q \wedge p \in q$$

W korzeniu osie *ancestor*, *parent*, ani osie poziome nigdy nie są prawdziwe - wymuszamy to przez zbiór stanów początkowych.

Osie poziome sprawdza automat "poziomy".

Jeżeli dany jest schemat, możemy zrobić przecięcie automatu dla zapytania i schematu w czasie wielomianowym. Dzięki temu możemy rozstrzygać problem czy dla danego zapytania istnieje drzewo zgodne ze schematem spełniające to zapytanie w czasie wykładniczym od rozmiaru zapytania (konstrukcja automatu) oraz wielomianowym od schematu (przecięcie automatów).

7.5 NavXPath a logika pierwszego rzędu

Twierdzenie Dla każdego $q \in NavXPath$ istnieje $\varphi \in FO_2$ (logika pierwszego rzędu z dwiema zmiennymi; zakładamy, że w sygnaturze sa predykaty unarne $L_{lab}(x_i)$ do sprawdzania etykiety w danym wierzchołku oraz predykaty binarne $\chi(x_i, x_j)$ odpowiadające osiom *child*, *descendant*, *next-sibling* oraz *following-sibling*; pozostałe osie można uzyskać przez ich złożenie lub odwrócenie), oraz dla każdego $\varphi \in FO_2$ istnieje $q \in NavXPath$ takie, że

$$q \text{ jest prawdziwe w } t, x \iff \varphi(x) \text{ jest prawdziwe w } t$$

$NavXPath \Rightarrow FO_2$ Zapytania unarne odpowiadają formułom FO_2 z jedną zmienną wolną. Większość typów zapytań unarnych odpowiada trywialnie zdaniom logiki pierwszego rzędu, np. $q \wedge q$. Nietrywialne są zapytania ścieżkowe postaci p (według przyjętej konwencji q oznacza zapytanie unarne, filtr; natomiast p oznacza dowolne zapytanie NavXPath). Zapytania ścieżkowe odpowiadają alternatywie formuł postaci

$$\psi(x_1, x_n) = \exists x_2 \exists x_3 \dots \exists x_{n-1} \varphi_1(x_1) \wedge \chi_1(x_1, x_2) \wedge \varphi_2(x_2) \wedge \chi_2(x_2, x_3) \dots \chi_{n-1}(x_{n-1}, x_n) \wedge \varphi_n(x_n)$$

gdzie $\varphi_i \in FO_2$, a χ_i to osie. Chcemy dostać formułę FO_2 dla $\exists x_n \, \psi(x_1, x_n)$. Jest ona postaci (z dokładnością do parzystości n):

$$\psi'(x_1) = \varphi_1(x_1) \wedge (\exists x_2 \chi_1(x_1, x_2) \wedge \varphi_2(x_2) \wedge (\exists x_1 \chi_2(x_2, x_1) \wedge \varphi_3(x_1) \dots \exists x_1 \chi_{n-1}(x_2, x_1) \wedge \varphi_n(x_1) \dots))$$

$NavXPath \Leftarrow FO_2$ Dostajemy formułę FO_2 postaci:

$$\psi(x_1) = \exists x_2 \psi'(x_1, x_2)$$

Jeżeli oryginalnie kwantyfikator nie znajduje się na początku, możemy go wyciągnąć przed całą formułę. Zauważmy, że $\psi'(x_1, x_2)$ jest kombinacją boolowską $\chi_i(x_j, x_k)$, $L_{lab}(x_i)$, $x_1 = x_2$, $x_1 \neq x_2$ oraz $\exists x_i(\dots)$. Sprowadzamy formułę do postaci DNF (disjunctive normal form), czyli alternatywy koniunkcji. Każdą koniunkcję chcemy sprowadzić do postaci $\varphi_1(x_1) \wedge \varphi_2(x_2) \wedge \chi(x_1, x_2)$ (gdzie φ lub χ mogą być trywialne). Przez χ rozumiem tutaj oś, $=$ lub $\neq$.

Dla każdej pary $\chi_1(x_1, x_2)$, $\chi_2(x_1, x_2)$ w oryginalnej koniunkcji, zachodzi jeden z dwóch przypadków. Jeżeli $\chi_1(x_1, x_2)$ oraz $\chi_2(x_1, x_2)$ są sprzeczne, możemy je zastąpić przez fałsz. W przeciwnym wypadku możemy $\chi_1(x_1, x_2) \wedge \chi_2(x_1, x_2)$ zastąpić przez $\chi_3(x_1, x_2)$. Na przykład koniunkcję osi *child* i *descendant* możemy zastąpić przez *child*.

Po przekształceniu formuły do postaci koniunkcji wyrażeń $\varphi_1(x_1) \wedge \varphi_2(x_2) \wedge \chi(x_1, x_2)$ odpowiadające jej zapytanie NavXPath jest koniunkcją zapytań postaci:

$$self[\text{formuła dla } \varphi_1]/\chi[\text{formuła dla } \varphi_2]$$

przy czym dla $x_1 = x_2$ używamy w miejsce χ osi *self*, a dla $x_1 \neq x_2$ używamy alternatywy osi *descendant*, *ancestor*, *following*, *preceding*. Jeżeli w formule nie ma osi, w miejsce χ wstawiamy *ancestor-or-self/descendant-or-self* (wierzchołki x_1 i x_2 nie są powiązane w żaden szczególny sposób).

Konwersja do postaci DNF powoduje wykładniczy wzrost wielkości formuły. Przykładowa formuła dla której tak się dzieje wygląda następująco:

$$\exists x \exists y ((\varphi_1(x) \iff \varphi_1(y)) \wedge \dots \wedge (\varphi_n(x) \iff \varphi_n(y)))$$

8 Ewaluacja zapytań z danymi.

8.1 Wstęp - FOXPath

Rozważmy teraz zapytania takie jak w NavXPATH, operujące również na danych. Zakładamy, że dane pojawiają się tylko w tekście, to znaczy zakładamy, że dokument XML jest bez etykiet:

```
<abc atr='wartosc'> </abc> → źle
<abc> wartosc </abc>
```

Chcemy móc sprawdzić również równość danych (lub stałych) w powiązaniu ze ścieżkami. Do tej pory mieliśmy gramatykę:

$$p \to step \mid p/p \mid p \cup p$$
$$step \to axis \mid step[q]$$
$$q \to p \mid lab() = L \mid q \vee q \mid \neg q$$

Dodajemy wyrażenia dla danych:

$$q \to p\, Rel\, p' \mid p \;\; Rel \;\; 'const'$$

Gdzie $Rel \in \{=, \neq, <, >, \leqslant, \geqslant, <_L, >_L, \leqslant_L, \geqslant_L\}$, a porządki z indeksem L oznaczają porządki leksykograficzne na napisach.

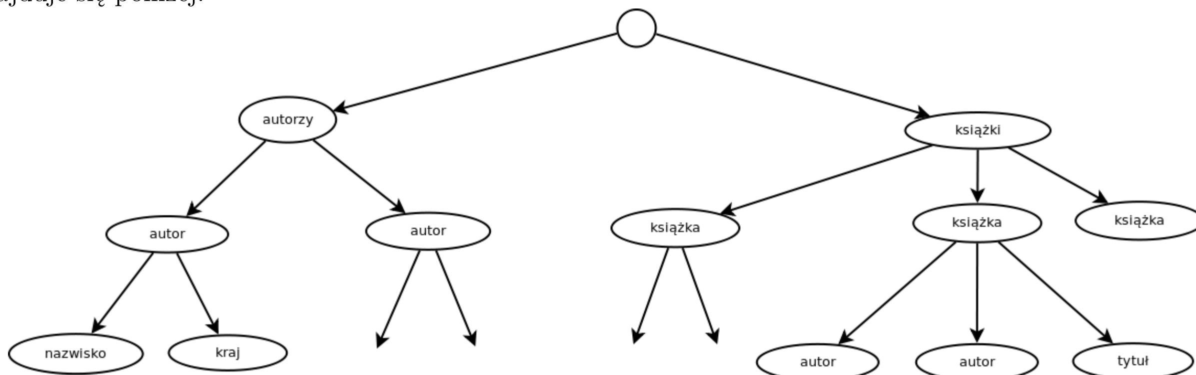
8.2 Semantyka operacji FOXPath

Konstrukcja $p\, Rel\, p'$ dla konkretnego wierzchołka x oznacza, że istnieją wierzchołki do których da się dojść odpowiednio ścieżkami p i p' takie, że ich zawartość jest ze sobą w relacji Rel. Analogicznie dla $p\, Rel\, 'const'$ porównujemy zawartości na ścieżce p ze stałą. Formalnie:

$$[\![p\, Rel\, p']\!](x) = \exists x_1, x_2 \;\; x_1 \in [\![p]\!](x) \wedge x_2 \in [\![p']\!](x) \wedge dana(x_1)\; Rel\; dana(x_2)$$
$$[\![p\, Rel\, const]\!](x) = \exists y \;\; y \in [\![p]\!](x) \wedge dana(y)\; Rel\; 'const'$$

8.3 Przykład

Wyobraźmy sobie katalog książek, wraz z informacjami o autorach. Schamat przykładowego katalogu znajduje się poniżej:



Chcemy z takiej struktury wybrać książkie takie, że chociaż jeden z autorów danej książki pochodzi z Polski. Można to zrealizować następującym zapytaniem FOXPath:

$$lab() = ksiazka \wedge (child[lab() = autor] = ancestor/descendant[lab() = nazwisko \wedge parent/child[lab() = kraj] =' Polska']$$

Takie zapytanie wybierze zawartość etykiet *książka*, w których dziecko (autor) będzie miało zawartość równą pewnemu węzłowi (autorowi), którego dziecko (kraj) ma wartość *Polska*. Fragment zapytania *ancestor/descendant* mówi o lokalizacji węzłów, w których znajdują się informacje o autorach.

8.4 Ewaluacja zapytań FOXPath

8.4.1 Wstęp

W tym miejscu przedstawimy algorytm ewaluacji zapytań FOXPath, najpierw analizując zapytanie z jednym wystąpieniem porównywania danych. Konkretnie, mając dane zapytanie Q postaci $p_1 Rel p_2$ takie, że $|Q| = q$ oraz drzewo T, $|T| = t$ szukamy zbioru $X = \{x : [\![Q]\!](x) = True\}$. Na początku przedstawimy nieco prostrzy algorytm o większej złożoności czasowej, następnie pokażemy jak tę złożoność w prosty sposób poprawić.

8.4.2 Algorytm podstawowy

khkj

Przygotowanie Przed rozpoczęciem algorytmu wykonamy prosty preprocessing w czasie $O(t)$, polegający na ponumerowaniu danych występujący w dokumencie liczbami od 1 do n (jeżeli dwie dane są równe to otrzymają ten sam numer). Odwołamy się do tego w opisie algorytmu.

Opis W alogrytmie będziemy przechodzić niezależnie ścieżki p_1 oraz p_2 od tyłu i dla każdego węzła x trzymać wszystkie dane znajdujące się w węzłach, do których da się dotrzeć z x dotychczas przetworzonym fragmentem ścieżki. Dane te można interpretować jako zbiór - worek, który na początku zawiera wartość z węzła self, a w trakcie algorytmu będzie modyfikowany w zależności od tego jak będzie się rozwijała ścieżka w zapytaniu Q. Weźmy np. zapytanie $parent/ancestor$. Rozpoczynamy od przetworzenia $ancestor$. Weźmy pierwszy węzeł x. x musi przekazać swój worek wszystkim swoim potomkom (bo dla nich ścieżka *ancestor* prowadzi między innymi do niego). Aby wykonać tę operację dla każdego x można, począwszy od korzenia, przekazywać worki w dół "kaskadowo", co zapewnia przetworzenie fragmentu *ancestor* dla wszystkich wierzchołków w czasie $O(t^2)$. Jedno przekazanie w dół może zająć potencjalnie $O(t)$, bo tak dużego rozmiaru mogą być worki a przekazywać będziemy $O(t)$ razy. Łatwo sprawdzić, że dla osi *parent* oraz $following$ można powyższą operację wykonać analogicznie, zmieniając jedynie kierunek. Dla pojedynczych kroków takich jak *child* aktualizację da się zrobić nawet w czasie $O(t)$. Przykładowo dla *child* wystarczy, że każdy węzeł przerzuci swój worek swojemy ojcu (korzeń go po prostu usunie). Analogicznie dla *parent* oraz $next-sibling$.

Po wykonaniu powyższych czynności dla każdego wierzchołka x otrzymujemy dwa zbiory danych odpowiadające workom z powyższego algorytmu. Nazwijmy te zbiory X_1 oraz X_2. X_1 zawiera wszystkie dane z wierzchołków, do których da się dojść scieżką p_1 i analogicznie X_2 zawiera wszystkie dane z wierzołków, do których da się dojść z x ścieżką p_2. Teraz możemy w liniowym czasie ze względu na wielkość tych zbiorów sprawdzić czy:

$$\exists(x,y) : x \in X_1 \wedge y \in X_2 \wedge x \; Rel \; y$$

Dla relacji $\neq, <, >$ wystarczy liniowo wyszukać maksimum/minimum X_1 oraz X_2 a następnie sprawdzić wzajemne położenie zbiorów. Dla równości możemy wrzucać dane do kubełków i jeżeli dwie dane wpadną do tego samego kubełka to znaleźliśmy dane równe sobie. Przypomina to sortowanie kubełkowe i jest możliwe dzięki temu, że na początku wykonaliśmy preprocessing i operujemy na liczbach $1..|D|$ zamiast danych samych w sobie.

Otrzymujemy więc, że x należy do zbioru wierzchołków spełniających zapytanie $p_1 \; Rel \; p_2$ wtedy i tylko wtedy, gdy powyższa formuła jest spełniona.

Rozważmy teraz zapytania z zagnieżdżonymi zapytaniami typu $p_1 Rel p_2$. Ewaluujemy najpierw najbardziej zagnieżdżone zapytanie i otrzymujemy pewien zbiór. Ten zbiór traktujemy jak filtr logiczny, który wskazuje węzły spełniające dany warunek. Np. $ancestor[p_1 \; Rel \; p_2]$: obliczamy $p_1 \; Rel \; p_2$ (jako podproblem) i przekazujemy potomkom worki tylko z tych węzłów, które spełniają to zapytanie.

Złożoność czasowa Ewaluacja każdego zapytania typu $Q = p_1 \; Rel \; p_2$ zajmuje czas $O(t^2 * q)$. Dla każdego framentu ścieżki musimy zaktualizować zbiory. Ponieważ pesymistycznie każdy fragment ścieżki może być osią, to robimy kaskadowo (wg powyższego algorytmu) w czasie $O(t^2)$. Fragmentów ścieżek jest zaś $O(q)$.

Ewaluacja pełnego zapytania daje również czas $O(t^2 * q)$. Jest tak dlatego, że zapytania typu $Q = p_1\, Rel\, p_2$ liczmy od najbardziej zagnieżdżonych a więc kazdy fragment dużego zapytania przetworzmymy tylko raz.

8.4.3 Usprawnienie algorytmu

Rozważmy teraz zapytania FOXPath, w których nie występuje relacja równości na danych. Wtedy możemy zastąpić worki z powyższego algorytmu parami danych d_{min} i d_{max} które odpowiednio oznaczają najmniejszą i największą daną, która występowałaby w worku. Zakładamy tu istnienie porządku na danych, w wypadku liczb naturalnego, dla napisów np. leksykograficznego. Teraz sprawdzenie dla danego wierzchołka czy jego worki się przecinają polega na sprawdzeniu zakresów występujących w nich danych. Zatem wykonujemy to w czasie $O(1)$. Podobnie propagacja worków usprawnia się z liniowej do czasu stałego (aktualizacja zakresów). W ten sposób otrzymujemy algorytm działający w czasie $O(t * q)$.

8.4.4 Dalsze usprawnienie algorytmu dla równości

Algorytm da się usprawnić również wtedy, gdy dołączymy do zbioru relacji $=, \leqslant$ *oraz* $\geqslant$. Usprawnienie daje złożoność liniowo-logarytmiczną względem rozmiaru drzewa kosztem złożoności sześciennej względem rozmiaru zapytania. Jest to istotnie usprawnienie, gdyż zapytania są zazwyczaj niewielkiego rozmiaru w stosunku do rozmiaru drzew. Algorytm, który działa we wspomnianym czasie jest skomplikowany, jego pełny opis można znaleźć w następnym rozdziale.

8.5 AggXPath

Gramatykę FOXPath można rozszerzyć jeszcze dalej i umożliwić wykonywanie zapytań ze zliczaniem:

$$\begin{gathered} p \to step \mid p/p \mid p \cup p \\ step \to axis \mid step[q] \\ q \to p \mid lab() = L \mid q \vee q \mid \neg q \mid p\, Rel\, p' \mid p\;\; Rel\; 'const' \mid i \\ i \to count(p) \mid sum(p) \mid i + i \mid i * i \mid ... \mid' const' \end{gathered}$$

8.6 Semantyka operacji AggXPath

count(p) oznacza liczbę węzłów, do których da się dojść ścieżką p. *sum*(p) oznacza sumę danych w węzłach, do których da się dojść ścieżką p. Formalnie:

$$[\![count(p)]\!](x) = |[\![p]\!](x)|$$

$$[\![sum(p)]\!](x) = \sum_{y \in [\![p]\!]} d(y).$$

8.7 Ewaluacja zapytań AggXPath

Na wykładzie podano bez dowodu, że na pewno da się zrobić ewaluację zapytań AggXPath w czasie nie gorszym niż $O(t^2 * q^2)$.

8.8 AggXPath a FO

Można pokazać, że w AggXPath da się wyrazić FO(oś). Korzystamy z faktu, że FO(oś) = logika temporalna $Until_{\alpha}, gdzie \alpha =$ oś. Pokażmy to dla *descendant*. Weźmy formułę $\phi_1 Until_{descendant} \phi_2$. Aby zapisać ją w AggXPath wystarczy zapytanie: $count(desc[\phi_2] \neq count(desc[\neg\phi_1]/desc[\phi_2])$. Innymi słowy sprawdzamy czy ilość potomków gdzie występuje ϕ_2 nie jest przypadkiem taka sama jak ilość potomków z występującym ϕ_2 takich, że po drodze do nich występuje gdzieś $\neg\phi_1$.

8.9 XPath 1.0

Jako ciekawostkę warto dodać, że prawdziwy XPath 1.0 to AggXPath poszerzony o możliwość sprawdzania pozycji. Rozszerzenie gramatyki wyglądałoby następująco:

$\ldots$
$i \rightarrow ...|position()|last()|$
$\ldots$

9 Analiza statyczna zapytań XPath

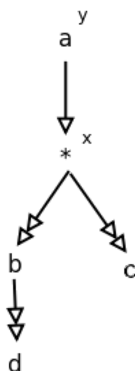
Typowe pytania Przy analizie zapytań XPath często mogą interesować nas następujące kwestie:

- spełnialność
- minimalizacja (znaleźć równoważne zapytanie, które byłoby "małe")
- równoważność
- **inkluzja:** $q \subseteq q' \equiv \forall t\ q(t) \subseteq q'(t)$,
  gdzie t - drzewo, $q(t)$ - zbiór par zwracanych przez zapytanie q

9.1 Inkluzja zapytań

Założenia Będziemy rozważać zapytania używające jedynie: osi *child*, $descendant - or - self$ oraz $*$ dla oznaczenia dowolnej etykiety (ozn. XPath(/, //, $*$)). W szczególności w rozważanych zapytaniach nie występuje alternatywa ani negacja. Zakładamy brak ograniczeń na kształt dokumentu, będziemy więc rozważać nieskończony zbiór etykiet (co nawet okaże się pomocne).

Wzorce Wzorcem będziemy nazywać "prawie" drzewa, powstałe z zapytań XPath, stanowiące schemat drzew spełniających zapytanie. "Prawie", bo dopuszczamy krawędzie typu $descendant - or - self$ symbolizujące dowolną ilość krawędzi (do końca tego rozdziału oznaczane przez podwójną strzałkę).



Rysunek 1: wzorzec dla zapytania $self[a]/child[desc-or-self[b]/desc-or-self[d]][desc-or-self[c]]$, x i y oznaczają wierzchołki zwracane przez zapytanie.

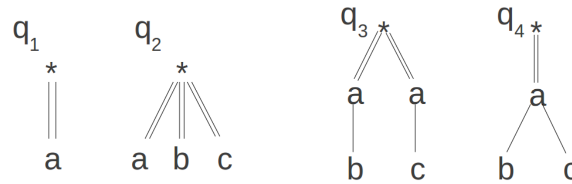
Niech $h : dom(q) \rightarrow dom(t)$ będzie homomorfizmem z wierzchołków wzorca w wierzchołki drzewa, takim że:

- etykieta u = etykieta h(u) (chyba, że u = $*$)
- q : krawędź child u, v $\rightarrow$ t: krawędź child h(u), h(v) (tu powinien być rysunek)
- q: krawedź descendant u, v $\rightarrow$ t: ścieżka (być może długości zero) między h(u) i h(v) (tu też powinien być rysunek)

9.1.1 Prostszy (?) problem

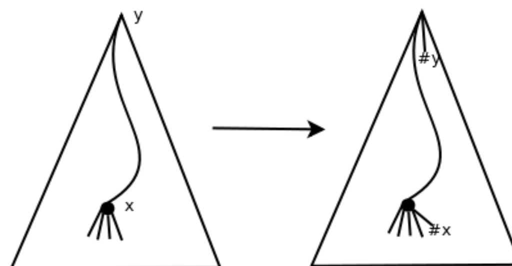
Zanim rozwiążemy problem inkluzji zapytań XPath(/, //, $*$), rozważymy prostszy problem:

Problem Dane q, q' - zapytania boole'owskie (interesuje nas tylko czy wzorzec można znaleźć, nie rozważamy zwracanych przez zapytania par). Pytamy, czy $q \subseteq_0 q'$, tj. czy $\forall t\ (t \models q) \Rightarrow (t \models q')$.



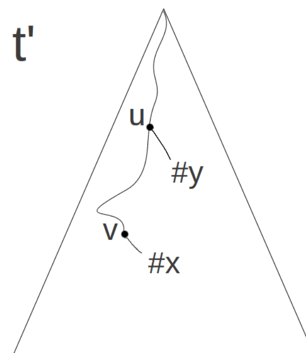
Przykład: $q_2 \subseteq_0 q_1,\ q_4 \subseteq_0 q_3$

Pokażemy, że problem inkluzji zapytań XPath(/, //, ∗) redukuje się (w $LOGSPACE$) do problemu inkluzji zapytań boole'owskich. Niech (y, x) będzie parą zwracaną przez zapytanie XPath q. Oznaczmy przez $q_{\#}$ drzewo, w którym do y i x dodajemy po jednym dziecku z nieistniejącą dotąd etykietą (odpowiednio #x i #y).



Teza: $q \subseteq_2 q' \Leftrightarrow q_{\#} \subseteq_0 q'_{\#}$

Dowód $\Leftarrow$ $\quad q_{\#} \subseteq_0 q'_{\#},\ \ t,\ \ u, v \in dom(t),\ < u, v > \in q(t)$
Niech t' będzie drzewem takim jak t, z dodanymi dziećmi #y i #x odpowiednio do wierzchołków y i x.



Mamy $t' \models q_{\#}$, zatem również $t' \models q'_{\#}$ (bo $q_{\#} \subseteq_0 q'_{\#}$). Możemy więc wkleić $q'_{\#}$ w t'. Wiemy również, że dodane etykiety #y i #x przechodzą na #y i #x w $q'_{\#}$, bo te etykiety nie występują w innych miejscach. Zatem jeśli wytniemy wierzchołki #y i #x z drzewa i wzorca, otrzymamy $< u, v > \in q'(t)$.

Dowód $\Rightarrow$ $\quad q \subseteq_2 q',\ t \models q_{\#}$, musimy pokazać, że $t \models q'_{\#}$, dla każdego t.
Nie możemy tak po prostu wyciąć wierzchołków #y i #x, bo takich wierzchołków może być więcej niż dwa, mogą być używane przez inny fragment wzorca (z gwiazdką) itp.
Bierzemy wierzchołki z #y i #x, zamieniamy im etykiety np. na 'a', a ich ojcom dodajemy po nowym synu z nieużywaną dotąd etykietą #Y / #X. Nazwijmy takie nowe drzewo t'. Drzewo t' także spełnia $q_{\#}$, gdyż w $q_{\#}$ były tylko dwa wierzchołki z krzyżykami, zatem pozostałe krzyżyki mogły być dopasowane we wzorcu jedynie przez gwiazdkę (lub nic), a litera 'a' również spełnia gwiazdkę.

Mamy więc $< u, v > \in q(t') \underset{q \subseteq q'}{\Longrightarrow} < u, v > \in q'(t')$, zatem także $t' \models q'_{\#}$. Musimy jeszcze pokazać, że $t \models q'_{\#}$. Tutaj wystarczy wziąć ten sam homomorfizm co przu t', przywracając jedynie stare etykiety.

9.1.2 Zawieranie zapytań używających /, // i *

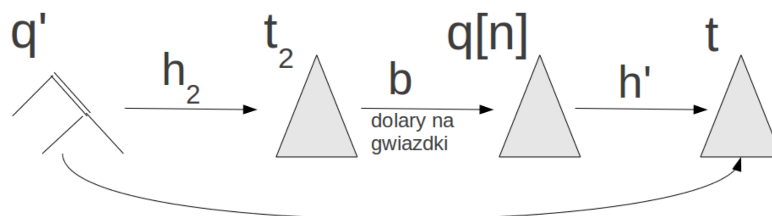
Definicje

- $Mod(q) := \{t \mid t \models q\}$
- Niech q zawiera d krawędzi typu *descendant*, ozn. $r_1, r_2, ..., r_d$. Ustalmy krotkę $\bar{n} \in \mathbb{N}^d$. Definiujemy *uszczegółowienie* $q[\bar{n}]$ w następujący sposób: bierzemy wzorzec q i każdą z krawędzi r_i typu *descendant* zamieniamy na ścieżkę krawędzi typu *child* długości n_i.
- $(q[\bar{n}])^\$:= q[\bar{n}]$, w którym wszystkie gwiazdki zamieniamy na symbol \$ nienależący do alfabetu Σ. Otrzymujemy w ten sposób drzewo nad alfabetem $\Sigma \cup \{\$\}$.
- **Model kanoniczny** $mod^\$(q) := \{(q[\bar{n}])^\$ \mid \bar{n} \in \mathbb{N}^d\}$. Mamy oczywiście $mod^\$(q) \subseteq Mod(q)$.
  $mod^\$_k := \{(q[\bar{n}])^\$ \mid \bar{n} \in \{0, 1, ..., k\}^d\}$ - model rozmiaru k.
- Gwiazdkowa długość wzorca : długość najdłuższego łańcucha gwiazdek połączonych krawędzią *dziecko*.

Fakt 1. $t \models q[\bar{n}] \;\Rightarrow\; t \models q$

Fakt 2. Jeśli $h : q \to t$ homomorfizm, to istnieje jedyne uszczegółowienie $q[\bar{n}]$ i jedyny homomorfizm $h' : q[\bar{n}] \to t$ taki, że na wierzchołkach q zachodzi $h = h'$.

Lemat. Jeśli istnieje $t \models q$, ale $t \not\models q'$, to istnieje także $t' \in mod^\$_k(q)$ takie, że $t' \models q$ oraz $t' \not\models q'$, gdzie $k = 1 +$ gwiazdkowa długość wzorca q'.
Dowód. Niech $t \models q$, $t \not\models q'$, $h : q \to t$.
Zatem istnieje $h' : q[\bar{n}] \to t$. Niech $t_2 = (q[\bar{n}])^\$$. Mamy $t_2 \in mod^\$(q)$. Pokażemy, że w t_2 q' nie jest spełnione. Załóżmy przeciwnie, że $t_2 \models q'$. Istnieje więc homomorfizm h_2 z q' w t_2. Ale istnieje także przekształcenie b, zamieniające dolary na gwiazdki, które przekształci t_2 w $q[\bar{n}]$ oraz przekształcenie $h' : q[\bar{n}] \to t$.



Składając h_2, b i h' otrzymamy $t \models q'$, czyli sprzeczność. Zatem jeśli istnieje $t \models q$, ale $t \not\models q'$, to istnieje także $t' \in mod^\$(q)$ takie, że $t' \models q$ oraz $t' \not\models q'$. Pokażemy jeszcze, że t' ma rozmiar ograniczony przez k, a dokładnie że $t' \in mod^\$_k(q)$.
Jeśli w t_2 istnieje ścieżka dolarowa dłuższa niż k, to można skrócić ją do długości k (oznaczmy takie skrócone drzewo przez t') i dalej będzie to kontrprzykład ($t' \not\models q'$).
Załóżmy przeciwnie, że $t' \models q'$. Z definicji k wiemy, że w q' nie ma ścieżek złożonych z samych gwiazdek i dzieci o długości większej niż $k - 1$. Zatem ścieżki dolarowe długości $\geqslant k$ w t' przynajmniej w części powstały z krawędzi *descendant* we wzorcu q', a więc możemy je wydłużać otrzymując drzewo nadal spełniające q'. W ten sposób otrzymujemy, że $t_2 \models q'$ - sprzeczność.

Algorytm co-NP. Pytamy czy $q \subseteq q'$, $q, q' \in XPath(/, //, *)$. Dzięki lematowi wiemy, że jeśli istnieje kontrprzykład na zawieranie tych zapytań, to jest on "mały" ($\in mod^\$_k(q)$, gdzie $k = 1 +$ gwiazdkowa długość wzorca q'). Możemy więc zgadywać potencjalny kontrprzykład i sprawdzać, czy rzeczywiście jest kontrprzykładem.

9.2 Inkluzja zapytań dla XPath(/, //, [], $*$, $\cup$) i schematu DTD

Dane są boolowskie zapytania p i q oraz schemat s, chcemy sprawdzić zawieranie $p \subseteq_s q$, czyli $\forall t \in L(s)\ t \models p \implies t \models q$. Pokażemy konstrukcję automatów równoważnych p i q, które będą deterministyczne i pojedynczo wykładnicze (tzn. rozmiaru $2^{w(\|p\|,\|q\|,\|s\|)}$, gdzie w jest pewnym wielomianem). Dzięki ich determinizmowi będziemy mogli uzyskać dopełnienie dla $L(q)$ i sprawdzić niepustość $L(p) \cap \overline{L(q)}$.

Automat dla XPath(/, //, [], $*$, $\cup$) Niech p' będzie boolowskim zapytaniem z XPath(/, //, [], $*$, $\cup$). Budujemy automat na drzewach nieurangowanych $A_{p'} = (\Sigma, Q, \delta, F)$ gdzie $\delta : Q \times \Sigma \to \mathcal{P}(Q^*)$. Definiujemy Q i F:

$$\begin{aligned} Q = \{(S_1, S_2) \mid\ & S_1 \text{ – zbiór podformuł } p \text{ prawdziwych ``tu''}, \\ & S_2 \text{ – zbiór podformuł } p \text{ prawdziwych w poddrzewie}\}, \\ F = \{(S_1, S_2) \mid\ & p \in S_1\}. \end{aligned}$$

Wreszcie definiujemy δ:

$$\delta\left((S_1, S_2), a\right) = \bigcap_{r \in S_1} L_{a,r} \cap \bigcap_{r \notin S_1} \overline{L_{a,r}} \cap \bigcap_{s \in S_2} \left(L_{a,s} \cup \bigcup_{(R_1,R_2) \in Q, s \in R_2} Q^*(R_1, R_2)Q^* \right).$$

Intuicyjnie, język $L_{a,f}$ mówi jakie stany (czyli zbiory podformuł) w dzieciach wierzchołka v o etykiecie a gwarantują, że w v prawdziwa będzie podformuła f. Bardziej formalnie, f możemy traktować jako formułę postaci $f = f_1 \cup \cdots \cup f_k$, gdzie $1 \leqslant k \leqslant \|f\|$ oraz pojedynczy wzorzec f_i nie korzysta z alternatywy przed użyciem pierwszej osi, czyli jest drzewem postaci:

x

$u_1 \ \dots \ u_m \qquad u'_1 \ \dots \ u'_n$

gdzie x jest etykietą korzenia, u_j są podformułami f_i do których z korzenia prowadzi oś *child*, a u'_j podformułami do których z korzenia prowadzi oś *descendant*. *Uwaga:* podformuły u_j oraz u'_j mogą oczywiście zawierać alternatywę.
Zdefiniujemy najpierw język L'_{a,f_i}, analogiczny do $L_{a,f}$, ale dla formuł postaci f_i. Ustalmy więc dowolne i i przyjmijmy, że oznaczenia u_j, u'_j dotyczą formuły f_i. Jeśli x nie pasuje do a (nie używamy tu pojęcia równości, ponieważ x może być równe "$*$"), to bierzemy $L'_{a,f_i} = \emptyset$. W przeciwnym razie:

$$(S_1, S'_1) \cdots (S_l, S'_l) \in L'_{a,f_i} \iff \forall_{1 \leqslant i \leqslant m} \exists_j\ u_i \in S_j \wedge \forall_{1 \leqslant i \leqslant n} \exists_j\ u'_i \in S'_j$$

Pokażemy konstrukcję automatu rozpoznającego ten język. Automat ten jest deterministyczny i ma rozmiar wielomianowy względem $|Q|$ i 2^{n+m}, a więc wykładniczy względem $\|p'\|$.

- Zbiór stanów to $\mathcal{P}(\{u_1, \dots, u_m\}) \times \mathcal{P}(\{u'_1, \dots, u'_n\})$.
- Zbiór symboli to Q z automatu $A_{p'}$.
- Stan początkowy to $\{u_1, \dots, u_m\} \times \{u'_1, \dots, u'_n\}$.
- Przejście ze stanu (U, U') po symbolu (S_1, S_2) prowadzi do stanu $(U - S_1, U' - S_2)$.
- Akceptujemy w stanie $(\emptyset, \emptyset)$.

Wracamy do $L_{a,f}$. Formuła f jest prawdziwa, gdy którakolwiek z formuł f_i jest prawdziwa, a więc $L_{a,f} = \bigcup L'_{a,f_i}$. Automat produktowy dla $L_{a,f}$ oparty o automaty dla języków L'_{a,f_i} jest deterministyczny i ma rozmiar wykładniczy względem $\|p'\|$.

Problemu inkluzji XPath(/, //, [], ∗, ∪) w obecności schematu DTD Dane są zapytania p, q i schemat s. Pytamy, czy $p \subseteq_s q$? Chcemy znaleźć $t \models s$ takie, że $t \models p \wedge t \not\models q$. Niech A_s będzie automatem odpowiadającym s, A_p – automatem równoważnym p (wg konstrukcji z poprzedniego paragrafu), analogicznie A_q dla q. Drzewo t jest dobrym kontrprzykładem, gdy $t \in L(s \cap p \cap \overline{q}) = L(A_s) \cap L(A_p) \cap \overline{L(A_q)}$. Z racji tego, że automat A_q jest deterministyczny, mamy $\overline{L(A_q)} = L\left(\overline{A_q}\right)$. Konstruujemy automat produktowy A odpowiadający $L(s \cap p \cap \overline{q})$, przyjmując:

$$\delta((x,y,z),a) = \delta^{A_s}(x,a) \times \delta^{A_p}(y,a) \times \delta^{A_q}(z,a),$$
$$F = \left\{(x,y,z) \mid x \in F^{A_s}, y \in F^{A_p}, z \in \overline{F^{A_q}}\right\}.$$

Rozmiar tego automatu jest ograniczony przez $2^{w(\|p\|,\|q\|,\|s\|)}$, dla pewnego wielomianu w. W czasie wielomianowym względem rozmiaru automatu A jesteśmy w stanie sprawdzić, czy akceptuje on jakiekolwiek drzewo – algorytm jest $EXPTIME$-zupełny.

9.3 Inkluzja zapytań dla XPath(/, //, [], ∗, ∪) i Σ skończonego

Dane są boolowskie zapytania p i q. Chcemy sprawdzić zawieranie $p \subseteq q$, czyli $\forall t\ t \models p \implies t \models q$, zakładając, że alfabet Σ jest skończony. Przedstawimy model obliczeniowy automatów alternujących oraz kilka ich własności, które pozwolą na stworzenie automatu alternującego A takiego, że $L(A) = L(p \cap \overline{q})$. Wreszcie pokażemy, że w ogólności problem niepustości dla automatów alternujących jest $EXPTIME$-zupełny, jednak dla problemu inkluzji wystarczy szukać kontrprzykładu, który ma co najwyżej $\|p\|$ liści. To spostrzeżenie doprowadzi do algorytmu $PSPACE$-zupełnego.

Lemat (†) Jeśli istnieje t takie, że $t \models p, t \not\models q$, to istnieje t' takie, że $t' \models p, t' \not\models q$ oraz:

- t' ma co najwyżej $\|p\|$ liści,
- rozgałęzeienie t' wynosi co najwyżej $\|p\|$ (wynika z pierwszego punktu).

Dowód Niech t będzie drzewem takim, że $t \models p, t \not\models q$. Dla każdej sumy ("∪") w p ewaluowanym na t, pewien jej człon jest spełniony – istnieje więc p' w którym każda suma zastąpiona jest odpowiadającym jej, spełnionym członem. Stąd $t \models p'$ i istnieje homomorfizm $h : p' \to t$. Niech t' będzie najmniejszym (w sensie liczby wierzchołków) drzewem zawierającym wszystkie wierzchołki z h. Z jednej strony mamy $t' \models p'$ – wystarczy wziąć homomorfizm h. Z drugiej strony, ze względu na monotoniczność zapytań z XPath(/, //, [], ∗, ∪) dostajemy $t' \not\models q$. Pozostaje pokazać, że t' ma co najwyżej $\|p\|$ liści. W tym celu zauważmy, że każdy liść $u \in t'$ jest w h – jeśli tak by nie było, to t' nie spełniałoby warunku minimalności. □

Automaty alternujące, na drzewach urangowanych, $\leqslant k$-arnych Opisywany automat alternujący to obiekt $(\Sigma, Q, q_0, \delta, F)$, gdzie $\delta : Q \times \Sigma \to \mathcal{B}^+(Q \times \{\varepsilon, 1, \ldots, k\})$, a $\mathcal{B}^+$ to kombinacja boole'owska używająca spójników $\wedge$ i $\vee$, bez negacji. Obliczenie automatu w wierzchołku v ze stanem q_v i etykietą a_v polega na sprawdzeniu każdego zbioru $C_v \subseteq Q \times \{\varepsilon, 1, \ldots, k\}$, dla którego spełniona jest kombinacja $\delta(q_v, a_v)$. Jeśli istnieje $C_v = \{(f, \varepsilon)\}$, gdzie $f \in F$, to obliczenie jest *akceptujące*. W przeciwnym razie obliczenie jest akceptujące, jeśli istnieje zbiór C_v taki, że:

- Jeśli $(q', \varepsilon) \in C_v$, to obliczenie v ze stanem q' (rodzaj ε-przejścia) jest akceptujące.
- Jeśli $(q, i) \in C_v$, to obliczenie dla i-tego dziecka v w stanie q jest akceptujące.

Drzewo jest akceptowane, jeśli obliczenie automatu dla korzenia w stanie q_0 jest akceptujące.
O obliczeniu A na drzewie t można też myśleć w kategorii agentów odpowiadających *prawda* lub *fałsz*. Agent z przydzielonym stanem q i wierzchołkiem v o etykiecie a wybiera kombinację $b = \delta(q, a)$. Dla każdego członu $c_i = (q_i, k_i)$ z b sprawdza, czy $q_i \in F$ oraz $k_i = \varepsilon$. Jeśli tak, wstawia w b, w miejsce członu c_i wartość *prawda*. W przeciwnym razie wysyła nowego agenta w stanie q_i do k_i-tego dziecka v (dla $1 \leqslant k_i \leqslant k$) lub do wierzchołka v (dla $k_i = \varepsilon$), a wartość zwróconą przez nowego agenta wstawia w b, w miejsce członu c_i. Na koniec agent zwraca *prawda* wtedy i tylko wtedy, gdy ewaluacja b z podstawionymi wartościami jest prawdziwa.
Obliczenie rozpoczyna się jednym agentem z przydzielonym stanem q_0 i korzeniem drzewa. Drzewo jest akceptowane, gdy agent ten zwróci *prawda*.

Lemat Dla każdego zapytania p z XPath(/, //, [], $*$, $\cup$) istnieje równoważny, wielomianowy, konstruowalny w czasie wielomianowym, automat alternujący A na drzewach o rozgałęzieniu $\leqslant k$.

Dowód Niech p będzie zapytaniem z XPath(/, //, [], $*$, $\cup$). Konstruujemy automat alternujący $A = (\Sigma, Q, q_0, \delta, F)$, gdzie $Q = \{p' \mid p' \text{ jest podformułą } p\} \cup \{\top\}$, $q_0 = p$, $F = \{\top\}$. Aby określić δ, zdefiniujemy pomocniczy operator Δ. $\Delta(p')$ będzie zwracać wszystkie przejścia automatu dla formuły p'. Przejścia te będziemy konstruować rekurencyjnie ze względu na budowę p'.
Pojawia się jedna subtelność: jak wygenerować przejścia dla $p_1//p_2$ mając dane przejścia dla p_1 oraz p_2. Jak zrobić to samo dla $p_1[p_2]$? Intuicyjnie, chcielibyśmy mieć możliwość decydowania jakie przejście wykonać po dopasowaniu wzorca p_1. W tym celu posłużymy się *znacznikiem* "$\star$", którego każde wystąpienie będzie symbolizować, że *"to przejście nie jest jeszcze określone"*. W stosownym momencie wszystkie wystąpienia znacznika $\star$ będziemy zamieniać na ukonkretnione przejście b, przy pomocy operatora podstawienia: $R[\star \mapsto b] = \{r[\star \mapsto b] \mid r \in R\}$. Dalej dla wygody przyjmiemy, że $(x, \downarrow) \equiv ((x, 1) \vee \cdots \vee (x, k))$ oraz "_" oznacza *dowolny element ze zbioru* $\{1, \ldots, k\}$.

postać p'	$\Delta(p')$	
$/p_1$	$\{(p', _) \to (p_1, \varepsilon)\} \cup \Delta(p_1)$	dla $/p_1 = p$
$/p_1$	$\{(p', _) \to (p_1, \downarrow)\} \cup \Delta(p_1)$	dla $/p_1 \neq p$
$//p_1$	$\{(p', _) \to (p_1, \varepsilon) \vee (p', \downarrow)\} \cup \Delta(p_1)$	
σ	$\{(p', \sigma) \to \star\}$	dla $\sigma \in \Sigma$
$*$	$\{(p', _) \to \star\}$	
$p_1 \cup p_2$	$\{(p', _) \to (p_1, \varepsilon) \vee (p_2, \varepsilon)\} \cup \Delta(p_1) \cup \Delta(p_2)$	
$p_1[p_2]$	$\{(p', _) \to (p_1, \varepsilon)\} \cup \Delta(p_1)[\star \mapsto (p_2, \varepsilon) \wedge \star] \cup \Delta(p_2)[\star \mapsto (\top, \varepsilon)]$	
p_1/p_2	$\{(p', _) \to (p_1, \varepsilon)\} \cup \Delta(p_1)[\star \mapsto (/p_2, \varepsilon)] \cup \Delta(/p_2)$	
$p_1//p_2$	$\{(p', _) \to (p_1, \varepsilon)\} \cup \Delta(p_1)[\star \mapsto (//p_2, \varepsilon)] \cup \Delta(//p_2)$	

Pozostaje wziąć $\delta = \Delta(p)[\star \mapsto (\top, \varepsilon)]$.
Pomijamy formalny dowód poprawności konstrukcji δ. $\square$

Dopełnienie automatu alternującego Niech dany będzie automat alternujący A. Dopełnienie $\overline{A}$ powstaje z A przez zamianę:

- koniunkcji z alternatywą w kombinacjach $\mathcal{B}^+$,
- stanów akceptujących z nieakceptującymi.

Przecięcie automatów alternujących Niech dane będą automaty alternujące $A = (\Sigma, Q^A, q_0^A, \delta^A, F^A)$ i $B = (\Sigma, Q^B, q_0^B, \delta^B, F^B)$. Wówczas automat $P = (\Sigma, Q, q_0, \delta, F)$, gdzie $Q = Q^A \cup Q^B \cup \{q_0\}$, $F = F^A \cup F^B$ oraz $\delta = \delta^A \cup \delta^B \cup \{(q_0, _) \to (q_0^A, \varepsilon) \wedge (q_0^B, \varepsilon)\}$ realizuje przecięcie automatów A i B.

Sprawdzanie niepustości (w ogólności) Wiemy, że problem sprawdzenia, czy $L(A_1) \cap \cdots \cap L(A_n) \neq \emptyset$ dla danych automatów $A_1, \ldots, A_n$ jest $EXPTIME$-zupełny. Zauważmy, że stosunkowo łatwo stworzyć automat alternujący A reprezentujący $A_1 \cap \cdots \cap A_n$, a więc problem niepustości tego przecięcia redukuje się do problemu $L(A) \neq \emptyset$. Stąd w ogólności problem niepustości dla automatów alternujących jest $EXPTIME$-zupełny.

Sprawdzanie niepustości (dla inkluzji) Mamy automat alternujący A taki, że $L(A) = L(p \cap \overline{q})$ i chcemy sprawdzić, czy akceptuje jakiekolwiek drzewo. Przypomnijmy przedstawiony wcześniej lemat (†) który mówi, że dla rozważanego problemu inkluzji wystarczy poszukiwać kontrprzykładów o co najwyżej $\|p\|$ liściach. Rozpatrzmy następujący niedeterministyczny algorytm $\mathcal{A}$, który zgaduje drzewo t i jednocześnie symuluje bieg A na tym drzewie. Algorytm w k-tym kroku oblicza zbiory prawdziwych stanów w wierzchołkach warstwy L_k drzewa t, gdzie jako warstwę L_k rozumiemy wszystkie wierzchołki drzewa, których ścieżka do korzenia ma długość k. Obliczenie L_{k+1} wygląda następująco (N_F oznacza liczbę wierzchołków zaakceptowanych w poprzednich krokach):

1. $\mathcal{A}$ wybiera dzieci wierzchołków z warstwy L_k:

- Dla każdego wierzchołka v z L_k algorytm niedeterministycznie wybiera liczbę jego dzieci z przedziału $\{0, \ldots, \|p\|\}$.
- Dla każdego z tych dzieci algorytm wybiera etykietę oraz zbiór stanów w nim prawdziwych, po czym informacje te zapisuje w L_{k+1}.
- Jeśli liczba wierzchołków w L_{k+1} przekracza $\|p\| - N_F$, to utworzona warstwa L_{k+1} jest niepoprawna i obliczenie nie jest kontynuowane.
- Jeśli wierzchołkowi v nie zostały przydzielone dzieci, to jest on liściem i w takim przypadku zbiór jego stanów musi być akceptujący, czyli $Q_v \subseteq F$. Jeśli tak nie jest, to utworzona warstwa L_{k+1} jest niepoprawna i obliczenie nie jest kontynuowane. Jeśli tak jest, algorytm zwiększa o jeden liczbę wierzchołków zaakceptowanych N_F.

2. Algorytm weryfikuje zgodność wybranych dzieci z regułami z δ:

   - Niech a_v i Q_v będą odpowiednio etykietą i zbiorem stanów dla wierzchołka v, zapisanymi w L_k.
   - Dla każdego $q \in Q_v$ algorytm wybiera kombinację $b = \delta(q, a_v)$.
   - Algorytm sprawdza, czy kombinacja b (ze stanami Q_v w v oraz wybranymi wcześniej stanami w dzieciach) jest prawdziwa.
   - Jeśli nie – utworzona warstwa L_{k+1} nie jest poprawna i obliczenie nie jest kontynuowane.

3. Algorytm sprawdza czy symulowany bieg na drzewie t jest akceptujący:

   - Jeśli warstwa L_{k+1} jest pusta i algorytm doszedł do tego kroku, to znaczy, że wszystkie wierzchołki stały się liściami i zostały zaakceptowane. Tak więc A akceptuje pewne drzewo o co najwyżej $\|p\|$ liściach.

Algorytm startuje z $N_F = 0$ i warstwą L_0 zawierającą wyłącznie korzeń drzewa t z niedterministycznie wybranymi: etykietą i zbiorem prawdziwych stanów. Na podstawie lematu (†) każda warstwa zawiera co najwyżej $\|p\|$ wierzchołków. Z każdym z wierzchołków skojarzona jest jego etykieta oraz zbiór stanów, a więc rozmiar warstwy jest ograniczony przez $\|p\| \cdot |Q| \cdot \lg |\Sigma|$. Zauważmy, że obliczenie L_{k+1} zależy wyłącznie od L_k. Stąd, algorytm w każdym kroku musi przechowywać tylko dwie warstwy. Pozostaje kwestia zapętlania się algorytmu. Zauważmy jednak, że dwie warstwy L_i oraz L_j nie mogą być identyczne dla $i < j$ – jeśli istniałaby akceptująca kontynuacja biegu od warstwy L_j to ta sama kontynuacja byłaby akceptująca od warstwy L_i. Wprowadzamy więc licznik K kroków algorytmu – przerywamy obliczenie algorytmu, jeśli przekroczy on liczbę wszystkich konfiguracji N:

$$N = \sum_{i=0}^{\|p\|} \left(2^{|Q|} \cdot |\Sigma|\right)^i \leqslant \|p\| \cdot 2^{|Q| \cdot \|p\|} \cdot |\Sigma|^{\|p\|}$$

Na licznik K potrzebujemy więc $\lg N = \lg\|p\| + |Q| \cdot \|p\| + \|p\| \lg |\Sigma| \leqslant \|p\| \cdot (|Q| + |\Sigma| + 1)$ bitów. Powyższy algorytm jest nideterministyczny $PSPACE$-zupełny. Należy go zdeterminizować (co na podstawie twierdzenia Savitch'a kwadratowo zwiększa potrzebną pamięć, więc algorytm pozostaje w $PSPACE$), po czym zanegować zwracaną odpowiedź.

10 Przekształcenia XML

XSLT

Szukamy metody na przekształcenie dokumentu XML w inny w sensowny sposób. Przykładem technologii temu służącej jest XSLT. Ze struktury o schemacie

- organization → group+ topmgr
- topmgr → employee+
- group → (mgr group+) | employee+
- mgr → employee
- employee : @id
- group : @id

opisującej organizację możemy wyłuskać strukturę opisującą ludzi spełniających konkretne wymagania (w tym przypadku kierowników, którzy nie nazywają się 'Bill', wraz z ich podwładnymi):

```
template start('organization')
  return <result>
           select-topmgr(P1)
         </result>
end
P1 = /organization/topmgr/employee

template select-topmgr('employee')
  vardef
    varID := @ID
  return
    if varID =/= 'Bill' then
      display(P2, varID)
    else
      SKIP

end
P2 = //group[mgr/employee[@id=$varID]]/group//employee

template display('employee', vID)
  vardef
    myID := @ID
  return
    <pair mgrID = $UID
          empID = $myID />
end
```

Powyższy przykład nie jest koniecznie zgodny ze potocznie używaną składnią, ale przedstawia ideę działania XSLT. Częstym problemem z XSLT jest stwierdzenie czy wygenerowany dokument pasuje do zadanego schematu, ponieważ jak widać nie musi on nawet generować poprawnego XML. Wrócimy do tego problemu w dalszej części.

XQuery

Inną technologią o mniejszej sile wyrazu, ale łatwiej sprawdzalnej poprawności jest XQuery. Przykładowe zapytanie XQuery generujące taki sam rezultat jak wcześniejszy XSLT zamieszczone jest poniżej:

```
<result>
  for $top in /organization/topmgr/employee[@id =/= 'Bill']
  for $grp in //group
  for $emp in $grp/group//employee
  where $top/@id = $grp/mgr/employee/@id
  return
  <pair>
    $top
    $emp
  </pair>
</result>
</result>
```

Ogólny model transducera na drzewach binarnych

Oba powyższe praktyczne sposoby na przekształcanie XML są częściowo lub całkowicie przekształcalne do Transducerów, których definicja i analiza znajdują się w kolejnych sekcjach. Rozwiązany też zostanie problem *type-checking*u czyli sprawdzenia czy wynik przekształcenia pasuje do zadanego schematu.

Wymagania

Wprowadzimy najbardziej ogólny model. Chcemy, aby spełniał następujące postulaty:

1. Maszyna dostaje na wejściu drzewo i zwraca drzewo
2. Drzewo wynikowe budowane jest od góry do dołu, równolegle
3. Transducer swobodnie chodzi po drzewie wejściowym, tzn. każdy wątek swobodnie porusza się po drzewie wejściowym wiedząc czy idzie się w $\nearrow$, $\searrow$, $\nwarrow$ czy $\swarrow$. Potrzebne jest to np. do transformacji

4. Transducer nie może łatwo się gubić, tzn. musi zaznaczać elementy. Zaznaczanie odbywa się za pomocą znaczników-kamieni(*pebbles*). Automat w danym momencie ma k numerowanych kamieni.

Formalna definicja

k-pebble tree transducer T to $\langle \Sigma, \Pi, Q, q_0, \delta \rangle$. Σ - alfabet wejściowy, Π - alfabet wyjściowy, Q - stany, q_0 - stan początkowy.

$$Q = Q_1 \cup Q_2 \cup \ldots \cup Q_k \quad Q_i \text{ są parami rozłączne}$$
$$q_0 \in Q_1$$

Niech $\bar{b}$ to wektor binarny opisujący, które kamienie znajdują się w bieżącym wierzchołku, a $q^{(i)}$ oznacza $i-$ ty kamień. Wtedy relacja przejścia jest postaci:

$$
\begin{aligned}
(a, \bar{b}, q^{(i)}) &\xrightarrow{\text{chodzenie}} (q_2^{(i)}, \textit{kierunek}) \qquad \textit{kierunek} \in \{\textit{stay}, \nearrow, \searrow, \nwarrow, \swarrow\} \\
(a, \bar{b}, q^{(i)}) &\xrightarrow{\text{nowy kamień}} (q_2^{(i+1)}, \textit{place-new-pebble}) \\
(a, \bar{b}, q^{(i)}) &\xrightarrow{\text{podnoszenie}} (q_2^{(i-1)}, \textit{pick-current-pebble}) \\
(a, \bar{b}, q^{(i)}) &\xrightarrow{\text{wypisywanie}} (a', (q_1^i, q_2^i), \textit{OUTPUT}) \\
& \qquad (\textit{nil}) \quad \text{- koniec tej części drzewa}
\end{aligned}
$$

T - niedeterministyczny transducer, wtedy $T(t) = \{t' \mid t' \text{ jest wynikiem działania } T \text{ na } t\}$

Przykłady

1. *k-pebble tree transducer* potrafi zasymulować top-down transducer.

   Tylko przejścia które powodują "rozmnożenie" są problematyczne. Rozwiążemy to na przykładzie przekształcenia:
   $$q(A(x_1, x_2)) \to S(q_1(x_1), B_2(q_2(x_2), B_3(q_3(x_1), q_4(x_2))))$$

   Transducer w stanie q zamienia wierzchołek A w wierzchołek S o dwóch synach i trzech miejscach do wypełnienia przez cztery kopie transducera w stanach q_1, q_2, q_3, q_4. Wystarczy przepisać taką operację przez dodanie dodatkowych stanów i wędrowania po drzewie wejściowym. Kamienie nie są potrzebne do realizacji top-down transducera.

   $$
   \begin{aligned}
   (A, \ldots, q) &\to (S, (q_1, q'), \textit{OUTPUT}) \\
   (\ldots, \ldots, q') &\to (q_1', \nwarrow) \\
   (\ldots, \ldots, q_1') &\to (B_2, (q_2, q_2'), \textit{OUTPUT}) \\
   (\ldots, \ldots, q_2') &\to (q_3', \nwarrow) \\
   (\ldots, \ldots, q_3') &\to (B_3, (q_3, q_4), \textit{OUTPUT})
   \end{aligned}
   $$

   Ten przykład pokazuje, że dzięki możliwości poruszania się po drzewie można dowolny term z relacji przejścia oryginalnego transducera przepisać na ścieżki opisywalne naszym chodzącym transducerem. Alternatywnie możnaby też skorzystać z jednego kamienia i kłaść go w startowym stanie przed rozwidleniem i korzystając z niego wskakiwać spowrotem w tranducerach kopiach, jednak struktura termu jest zawsze znana i możemy dokładnie obliczyć ścieżkę powrotu.

2. Jak przesunąć kamień do kolejnego wierzchołka w pre-order?

   Po wejściu w stan q_s poniższy fragment relacji przejścia przesunie się do nastepnego wierzchołka i pozostanie w stanie q_n lub q_f jeśli nie ma więcej wierzchołków. Operacje podnoszenia i kładzenia kamienia należy dodać odpowiednio przed q_s i po q_n. Wektor kamieni został zignorowany w poniższych elementach relacji. a_2 to wierzchołek wewnętrzny, a_0 - liść, r - korzeń.

$(a_2, \ldots, q_s) \to (q_n, \swarrow)$	przesunięto
$(a_0, \ldots, q_s) \to (q_3, \textit{stay})$	liść, będziemy szukać poprzednika
$(a, \ldots, q_3) \to (q_3, \nwarrow)$	do góry w lewo do poprzednika
$(a, \ldots, q_3) \to (q_4, \nearrow)$	raz w prawo do poprzednika
$(a, \ldots, q_4) \to (q_n, \searrow)$	przenosimy z poprzednika do następnika
$(r, \ldots, q_3) \to (q_f, \textit{stay})$	nie ma więcej wierzchołków, byliśmy w ostatnim

Problem *type-checking*u

Mamy dane transducer T oraz języki L i M. Czy $T(L) \subseteq M$? ($\forall_{t \in L} T(t) \subseteq M$).

k-pebble tree automata

Najpierw definiujemy *k-pebble tree automata* $A = \langle \Sigma, Q, q_0, \delta \rangle$. Oznaczenia te same, ale δ bez wypisywania. Akceptujemy wtw, gdy bieg się zakończy, czyli we wszystkich liściach będą *nil*e.

Pokażemy, że $T^{-1}(M)$ jest rozpoznawany przez *k-pebble tree automata.*

Fakt Dla dowolnego *k-transducera* i zwykłego automatu A istnieje *k-pebble tree automata* B taki, że $L(B) = T^{-1}(L(A))$.

$$\begin{aligned} A &= \langle \Gamma, P, p_0, \delta^A, F^A \rangle \\ T &= \langle \Sigma, \Gamma, Q, q_0, \delta^F \rangle \\ B &= \langle \Sigma, Q \times P, (p_0, q_0), \delta^B \rangle \end{aligned}$$

$$\begin{aligned} &\delta^T \ni (a, \bar{b}, q^{(i)}) \to (q_2, \mathit{akcja}) \\ &\qquad \Downarrow \\ &\delta^B \ni (a, \bar{b}, (q^{(i)}, p)) \to (q_2, p, \mathit{akcja}) \\ &\qquad \mathit{akcja} \in \{\nearrow, \searrow, \nwarrow, \swarrow, \epsilon, \mathit{pick\text{-}current\text{-}pebble}, \mathit{place\text{-}new\text{-}pebble}\} \end{aligned}$$

$$\begin{aligned} &\delta^T \ni (a, \bar{b}, q) \to (a^0, \mathit{nil}) \\ &\qquad \Downarrow \\ &\delta^B \ni (a, \bar{b}, q) \to \begin{cases} \text{OK} & \text{gdy } p \xrightarrow{a^0} \genfrac{}{}{0pt}{}{\nearrow f}{\searrow f} \in \delta^A \\ \text{ŹLE} & \text{wpp.} \end{cases} \end{aligned}$$

$$\begin{aligned} &\delta^T \ni (a, \bar{b}, q) \to (a', (q_1, q_2), \mathit{OUTPUT}) \\ &\qquad \Downarrow \forall_{p, a' \to (p_1, p_2) \in \delta^A} \\ &\delta^B \ni (a, \bar{b}, (q, p)) \to ((q_1, p_1), (q_2, p_2), \mathit{branch}) \end{aligned}$$

Widać, że powyższa transformacja wiedzie nas do celu.

Chcielibyśmy pokazać, że *k-pebble tree automata* jest równoważny zwykłym automatom.

Szkic dowodu:

$$\textit{k-pebble tree automata} \rightsquigarrow\!\rightsquigarrow \text{MSO} \underset{\text{oczywiste}}{\rightsquigarrow\!\rightsquigarrow} \text{automat.}$$

Type-checking $T(L) \subseteq M$

Pokazaliśmy, że $T^{-1}(M)$ jest rozpoznawalny za pomocą *k-pebble tree automata* $\subseteq$ *MSO* $\subseteq$ *TA*

Dowiedziemy, że $L(A)$, gdzie A jest *k-pebble tree automata*, jest definiowalny za pomocą formuły *MSO*, tzn.

$$\textit{k-pebble TA} \subseteq \textit{MSO}$$

Mamy automat $A = \langle \Sigma, Q, q_0, \delta \rangle$. $\Sigma = \Sigma_2 \cup \Sigma_0$, a wierzchołki wewnętrzne drzewa mają etykiety z Σ_2, a liście z Σ_0.

$$Q = Q_1 \cup \ldots \cup Q_k \quad Q_i \text{ są rozłączne}$$

Przejścia są postaci:

$$
\begin{aligned}
&(q^{(i)}, a, \bar{b}) \to (q_2^{(i)}, dir) \qquad dir \in \{stay, \nearrow, \searrow, \nwarrow, \swarrow, \epsilon\}\\
&(q^{(i)}, a, \bar{b}) \to (q_2^{(i+1)}, \textit{place-new-pebble}, \bar{b}')\\
&(q^{(i)}, a, \bar{b}') \to (q_2^{(i-1)}, \textit{pick-current-pebble}, \bar{b})\\
&(q^{(i)}, a_2, \bar{b}) \to ((q_1^{(i)}, q_2^{(i)}), branch)\\
&\qquad \text{dla } a_2 \in \Sigma_2\\
&(q^{(i)}, a_0, \bar{b}) \to \begin{cases} OK & \text{akceptacja w liściu}\\ REJECT & \text{odrzucenie drzewa}\end{cases}\\
&\qquad \text{dla } a_0 \in \Sigma_0
\end{aligned}
$$

Po operacji *pick-current-pebble* teleportujemy się do wierzchołka, w którym jest $(i-1)$-szy kamień. $q^{(i)} \in Q$: Operacja *pick-new-pebble* jest legalna dla $i < k$, natomiast *pick-current-pebble* jest legalna dla $i > 1$.

Automat jest niederministyczny.

Przechodzimy do konstrukcji formuły dla automatu A.

Docelowo chcemy kodować konfiguracje automatu i powiedzieć, że konfiguracja akceptująca jest osiągalna z konfiguracji początkowej.

Pierwsze przybliżenie

$$
\begin{aligned}
&\phi(x,y) = \text{“y jest potomkiem x”}\\
&\forall_S\ S(x) \land closed(S) \Rightarrow S(y)\\
&\text{gdzie } closed = \forall_{z_1,z_2} S(z_1) \land \begin{bmatrix} succ_1(z_1,z_2) \lor succ_2(z_1,z_2)\\ \Downarrow \\ S(z_2)\end{bmatrix}
\end{aligned}
$$

Drugie przybliżenie

$$
\begin{aligned}
&\Sigma_2 = \{\land, \lor\}, \Sigma_0 = \{0,1\} \quad \text{(czyli wyrażenia logiczne)}\\
&\phi = \text{“wyrażenie ma wartość 1”}\\
&\phi = (\forall_x\ 1(x) \to S(x) \land 0(x) \to \neg S(x)) \land \textit{reverse-closed}(S)
\end{aligned}
$$

reverse-closed(S) oznacza: jeśli wierzchołek x to $\lor$ i ma przynajmniej jedno dziecko w S, to $x \in S$. Ponadto jeśli x to $\land$ to $x \in S \Leftrightarrow$ wszystkiego jego dzieci są w S.

Trzecie przybliżenie Alternująca osiągalność w grafach:

$$G = (V,E) \quad V = V_\forall \cup V_\exists$$

Zbiór grafów osiągalnych to najmniejszy taki zbiór $Z \subseteq V$, że $v \in Z$ jeśli:

$$
\begin{aligned}
&v \in V_\exists \text{ oraz } \exists_y E(x,y) \land y \in Z\\
&v \in V_\forall \text{ oraz } \forall_y E(x,y) \Rightarrow y \in Z
\end{aligned}
$$

Zauważmy, że wierzchołki $V_\forall$ bez żadnych krawędzi wychodzących należą do Z.

Napiszemy teraz formułę dla alt. osiągalności w grafach:

$$\begin{aligned}\phi(x) &= \forall_S(\textit{reverse-closed}(S) \Rightarrow S(x))\\ \textit{reverse-closed}(S) &=\\ &\forall_y(V_\exists(y) \wedge \exists_z E(y,z) \wedge S(z)) \Rightarrow S(y)\\ &\wedge \forall_y(V_\forall(y) \wedge \forall_z E(y,z) \Rightarrow S(z)) \Rightarrow S(y)\end{aligned}$$

$$\phi(x) \equiv \text{“x jest osiągalny”}$$

Oznaczenia:

$$\begin{aligned}Q &= \{q_1, \ldots, q_n\}\\ Q_i &= \{q_{n_i+1}, \ldots, q_{n_{i+1}}\}\end{aligned}$$

Prosty przypadek na początek

$$\begin{aligned}n_1 &= 0\\ n_{k+1} &= n\\ \forall_{S_{n_1+1},\ldots,S_{n_2}} \textit{reverse-closed}(S_{n_1+1}, \ldots, S_{n_2}) &\Rightarrow \exists_z \textit{root}(z) \wedge S_{n_1+1}(z)\\ \textit{reverse-closed}(S_{n_1+1}, \ldots, S_{n_2}) &= \bigwedge_{r\in\delta} \Psi_r\end{aligned}$$

Dla każdego przejścia z δ, mamy odpowiednią formułę Ψ_r. Np.

$$\begin{aligned}r : (q_u, a) &\rightarrow (q_r, \varepsilon)\\ \forall_x S_v(x) \wedge a(x) &\rightarrow S_u(x)\\ &\text{czyli jeśli akceptujemy z } v \text{ widząc } a,\\ &\text{to akceptujemy z tego samego miejsca z } u\end{aligned}$$

$$\begin{aligned}r : (q_u, a) &\rightarrow (q_v, \searrow)\\ \forall_{x,y} a(x) \wedge s_2(x,y) \wedge S_v(y) &\Rightarrow S_u(x)\\ &\text{ciekawszym przypadkiem jest branch}\\ (q_u, a) &\rightarrow ((q_v, q_w), \textit{branch})\\ \forall_{x,y,z} a(x) \wedge s_1(x,y) \wedge s_2(x,z) \wedge S_v(y) \wedge S_w(z) &\Rightarrow S_u(x)\end{aligned}$$

To było dla $k = 1$. Przejdźmy więc razem do ogólnego przypadku. No to hops:

$$\begin{aligned}\textit{reverse-closed}^{(i)}(x_1, \ldots, x_{i-1}, S_{n_1+1}, \ldots, S_{n_{i+1}}) &= \bigwedge_{r\in\delta_i} \Phi_r\\ \delta_i &= \{(a, \bar{b}, q_{(i)}, \ldots) \mid \ldots\}\end{aligned}$$

Jedyna istotna dla nas zmiana to kładzenie i podnoszenie kamienia. Warto jednak zauważyć, że teraz mamy wektor $\bar{b}$ i w każdym przejściu musimy sprawdzać czy aby na pewno $\bar{b}$ jest OK. Służą nam do tego zmienne $x_1, \ldots, x_{i-1}$.

Niech $pebbles_{\bar{b}}(x_1, \ldots, x_{i-1})$ mówi czy zmienne $x_1, \ldots, x_{i-1}$ zgadzają się z $\bar{b}$ oraz:

$$
\begin{aligned}
r : (q_u^{(i)}, a, \bar{b}) &\rightarrow (q_v^{i+1}, \textit{place-new-pebble}) \\
\forall_{x_i} a(x_i) \wedge pebbles_{\bar{b}}(x_1, \ldots, x_{i-1}) &\wedge \phi_v^{(i+1)}(x_1, \ldots, x_{i-1}, S_{n_1+1}, \ldots, S_{n_{i+1}}) \\
&\Rightarrow S_u(x_i)
\end{aligned}
$$

$$
\begin{aligned}
\text{gdzie } \phi_v^{(i+1)}(x_1, \ldots, x_{i-1}, S_{n_1+1}, \ldots, S_{n_{i+1}}) &= \forall_{S_{n_{i+1}+1}, \ldots, S_{n_{i+2}}} \textit{reverse-closed}(\bar{x}, S_1, \ldots, S_{n_{i+2}}) \\
&\Rightarrow S_v(x_i)
\end{aligned}
$$

$$
\begin{aligned}
r : (q_u^i, a, \bar{b}) &\rightarrow (q_v^{i-1}, \textit{pick-current-pebble}) \\
\forall_{x_i} a(x_i) \wedge pebbles_{\bar{b}}(x_i) \wedge S_v(\underbrace{x_{i-1}}_{\heartsuit}) &\rightarrow S_u(x_i)
\end{aligned}
$$

Zauważmy, że obecność $\heartsuit$ jest wymuszona przez interpretację: po podniesieniu kamienia teleportujemy się do poprzedniego kamienia.

Ogólna formuła na automat A:

$$\phi_A = \forall_{S_1, \ldots, S_{n_2}} \textit{reverse-closed}(S_1, \ldots, S_{n_2}) \Rightarrow \exists_z \textit{root}(z) \wedge S_1(z)$$

Dowód *typechecking*u przebiega następująco: Bierzemy $T^{-1}(M)$ - robimy *k-pebble tree automata* A; $L = L(C)$ z A robimy ϕ_A. Z ϕ_A robimy normalny automat B: Sprawdzamy, czy $L(C \cap \bar{B}) \neq \emptyset$.

$$
\begin{aligned}
t \in L(C) \wedge t \in L(\bar{B}) &\Rightarrow t \notin L(B) = T^{-1}(M) \\
&\text{czyli } T(t) \notin M
\end{aligned}
$$

11 Niepełne dane

Co tu?

12 Wymiana danych

12.1 Wstęp

Wymiana danych polega tym, że chcemy zorganizować interakcje pomiędzy dwoma bazami danych opartymi na XML. Organizacja danych w obu bazach może być zupełnie różna; naszym celem jest stworzenie takiego formalizmu, który uchwyciłby jak najszerszą klasę interakcji, a jednocześnie byłby w sensowny sposób obliczalny.

Każdą stronę wymiany będziemy przedstawiali sobie jako dokument XML z danymi z nałożonym schematem, danym przez automat. Dodatkowo definiujemy sobie język więzów, który będzie pomocny przy określaniu zależności, jakie mają zajść. Więz ma postać $\pi(\overline{x})$, gdzie $\overline{x}$ jest krotką danych. Powiedzmy, że krotka jest długości k. Wpierw zdefiniujemy więz logiką; ma on postać:

$$\pi(\overline{x}) = \exists_{v_1}\exists_{v_2}\ldots\exists_{v_l} \bigwedge \alpha(v_i) \wedge \bigwedge \beta(v_i, v_j).$$

Tutaj, $\alpha(v_i)$ to predykaty co się może dziać na kwantyfikowanym egzystencjalnie wierzchołku v_i. Mogą mówić, że:

- w v_i jest dana x;
- w v_i jest etykieta a.

Uwaga: wymagamy, by każda dana z zapytania była w jakimś wierzchołku.

$\beta(v_i, v_j)$ to predykaty, które mówią, co się dzieje pomiędzy dwoma wierzchołkami. Mogą mówić, że:

- pomiędzy v_i a v_j jest jakaś oś;
- dane w v_i oraz v_j są równe, lub różne.

Oczywiście drugi więz ma sens jedynie, gdy mamy kwantyfikowane zmienne, które nie mają wymuszonej danej.

Więz możemy sobie wyobrażać jako graf. Wierzchołki grafu to kwantyfikowane zmienne, które mogą być dodatkowo etykietowane wymuszonymi w nich danymi oraz wymuszonymi etykietami. Pomiędzy wierzchołkami mamy krawędzie mówiące bądź o wzajemnym położeniu poprzez zadanie osi, bądź o równości lub różności danych.

12.2 Tłumaczenie danych

Zastanowimy się teraz nad formalizacją zagadnienia przetłumaczenia danych z jednej bazy do drugiej. Od teraz zajmiemy się tylko płaskimi zapytaniami, to znaczy w słowie a nie w drzewie. Mamy więc jedną bazę ograniczoną schematem (automatem) S_1 oraz drugą, ograniczoną schematem (automatem) S_2. Chcemy, by tłumaczenie danych nie gubiło jakiś istotnych informacji. Dlatego nakładamy na tłumaczenie szereg ograniczeń postaci:

$$\forall_{\overline{x}}\forall_{\overline{y}}\ \pi_i(\overline{x}, \overline{y}) \ \longrightarrow\ \exists_{\overline{z}}\ \pi'_i(\overline{x}, \overline{z})$$

Przykładowo, jeśli w bazie danych hurtowni mamy opis produktu spożywczego wraz z ilością kartonów i takimi rzeczami, to po transferze do sklepu spożywczego chcemy mieć dane tego samego produktu, ale np. już z jego wartością odżywczą.

Zbiór naszych ograniczeń nazywamy Σ. Sumaryczną długość ograniczeń nazwiemy $||\Sigma||$. Mówimy teraz, że słowo z danymi w' jest rozwiązaniem dla słowa w, jeśli w' spełnia schemat S_2 oraz wszystkie ograniczenia z Σ są spełnione.

Teraz mamy następujący problem:

Translacja danych
Input: Schemat S_2, zbiór ograniczeń Σ, słowo z danymi w
Question: Czy istnieje słowo w', będące rozwiązaniem dla w względem S_2 oraz Σ?

Wpierw zastanowimy się, jak zrobić dla tego problemu brutalny algorytm działający w czasie $2^{|w|^{poly(||\Sigma||,|S_2|)}}$.

Wpierw zauważmy, że w czasie $|w|^{poly(||\Sigma||)}$ możemy brutalnie spróbować wyewaluować wszystkie lewe strony wszystkich ograniczeń na wszystkich możliwych krotkach wierzchołków w słowie w. W ten sposób dokładnie możemy znaleźć wszystkie więzy postaci $\exists_{\overline{z}}\ \pi'(\overline{a},\overline{z})$ dla ustalonych danych $\overline{a}$, które muszą być spełnione w rozwiązaniu w'. Niestety, a może stety, może takich więzów może być aż $|w|^{poly(||\Sigma||)}$. Oznaczmy ich liczbę przez N.

Niech w' będzie jakimkolwiek rozwiązaniem. Zobaczmy jak wygląda ewaluacja wszystkich potrzebnych więzów w w'. Zauważmy, że każdy więz dotyka co najwyżej $||\Sigma||$ wierzchołków, więc jeśli $|w'| > (|S_2|+1)(||\Sigma||+1)(N+1)$, to pomiędzy którymiś dwoma kolejnymi dotkniętymi wierzchołkami jest co najmniej $|S_2|+1$ niedotkniętych wierzchołków. Wtedy zauważmy, że możemy w tym fragmencie coś odpompować zachowując schemat S_2. Widzimy więc, że jeśli istnieje rozwiązanie to istnieje rozwiązanie długości $|w|^{poly(||\Sigma||,|S_2|)}$.

Teraz zauważmy, że mamy problem: dane ograniczenia postaci $\exists_{\overline{z}}\ \pi'(\overline{a},\overline{z})$ które mają być spełnione, oraz schemat S_2 i pytamy się, czy istnieje w' spełniające S_2 oraz wszystkie ograniczenia. Z powyższego argumentu wynika, że problem ten jest w NP: certyfikatem weryfikowalnym wielomianowo jest (wielomianowe od N) słowo w' oraz mapowanie zmiennych wszystkich ograniczeń w wierzchołki w'. Oznacza to, że umiemy rozwiązać problem w czasie $2^{|w|^{poly(||\Sigma||,|S_2|)}}$, poprzez po prostu zgadnięcie certyfikatu (przeglądnięcie wszystkich) i sprawdzenie go.

Teraz naszym celem jest ułożenie algorytmu o złożoności $|w|^{poly(||\Sigma||,|S_2|)}$. Zauważmy, że pierwszy krok, czyli znalezienie potrzebnych więzów $\exists_{\overline{z}}\ \pi'(\overline{a},\overline{z})$ możemy zrobić tak samo, bo to działało w małej złożoności. Uwaga: opisywany algorytm różni się w szczegółach od tego z wykładu, moim zdaniem tak jest bardziej naturalnie.

Problemem jest tak naprawdę to, jak poradzić sobie ze spełnianiem wszystkich więzów i jednoczesnym spełnianiem schematu S_2. Gdyby bowiem np. schemat S_2 akceptował każde słowo, to w czasie $|w|^{poly(||\Sigma||)}$ dla każdego więzu zgadlibyśmy słowo spełniające go, a potem wszystko sobie skonkatenowali. Trzeba jakoś tę intuicję przełożyć na to, że brutalna konkatenacja może nie być możliwa ze względu na schemat.

Rozpatrzmy sobie automat (deterministyczny, na słowie) definiujący schemat S_2. Automat ten można podzielić na silnie spójne składowe. Teraz wprowadzamy kluczową definicję. *Kindem* nazwiemy słowo postaci

$$w_1(...)^* w_2(...)^* w_3 \dots w_{k-1}(...)^* w_k,$$

gdzie w_i są słowami z danymi dodatkowo etykietowanymi stanami automatu definiującego S_2, zaś napisy $(...)^*$ są różnymi spójnymi składowymi automatu dodatkowo etykietowanymi stanem na końcu i stanem na początku. Od kindu wymagamy, żeby był poprawny w sensie biegu, to znaczy, że kolejne stany i litery są zgodne ze przejściami automatu. Zastanówmy się ile jest kindów długości L. Otóż w każdy polu kindu musimy zgadnąć jedną z $poly(|w|,|S_2|)$ wartości, więc w sumie kindów jest $|w|^{poly(L,|S_2|)}$. Powiemy, że słowo w spełniające S_2 *pasuje* do kindu, jeśli po wyetykietowaniu go biegiem automatu rozpoznającego S_2 można je dopasować do kindu, zwijając pewne fragmenty wewnątrz spójnych składowych do $(...)^*$.

Teraz powiemy, że kind ma marginesy długości K, jeśli każde ze słów w_i ma K pierwszych przejść należących do silnie spójnej składowej z poprzedniego $(...)^*$ oraz K ostatnich należących do następnej.

Lemat 1. *Dla każdej stałej K oraz słowa w akceptowanego przez schemat S_2, istnieje kind długości $O(K \cdot |S_2|)$ o marginesach K, do którego w pasuje.*

Dowód. Piszemy bieg automatu na w oraz zaznaczamy fragmenty zawarte w silnie spójnych składowych. Tych fragmentów jest nie więcej niż $|S_2|$ dla każdego fragmentu dłuższego niż $2K$ odcinamy po K marginesów z obu stron i środek zapisujemy jako $(...)^*$. Wówczas na każdy z $|S_2|$ fragmentów zużyliśmy co najwyżej $2K$ liter kindu i oczywiście kind ma marginesy długości K. □

Zauważmy, że z lematu wynika, że możemy sobie zgadnąć dokładnie kind o marginesach $2||\Sigma||$ do którego pasuje rozwiązanie (2 jest dla pewności) — gdyż istnieje taki kind nie dłuższy niż $O(||\Sigma|| \cdot |S_2|)$, a kindów takiej długości jest $|w|^{poly(||\Sigma||,|S_2|)}$. Więc zgadujemy i mamy już ustalony kind k do którego rozwiązanie ma pasować.

Teraz rozważmy jakiś więz $\exists_{\overline{z}}\pi'(\overline{a},\overline{z})$ i jakąś ewaluację jego wierzchołków w rozwiązaniu w'. Powiemy, że ewaluacja *zgadza* się z kindem k, jeśli na granicach pomiędzy silnie spójnymi składowymi a fragmentami wypisanymi explicite nie ma zmiennych powiązanych ze sobą relację następnika.

Lemat 2. *Jeśli słowo w' pasujące do k ma ewaluację więzów z Σ, to istnieje też takie słowo w'' pasujące do k, które też ma wszystkie ewaluacje, ale dodatkowo zgadzają się one z k.*

Dowód. Patrzymy na słowo w' i robimy następującą chirurgię na nim. Załóżmy, że mamy jakąś granicę pomiędzy słowem w_i oraz fragmentem $(...)^*$, bez utraty ogólności postaci $w_i(...)^*$. Teraz na początku fragmentu $(...)^*$ dopompowujemy następujący kawałek słowa. Ze stanu na początku $(...)^*$ na mocy silnej spójności idziemy jakkolwiek do stanu na początku marginesu w_i, a następnie przekopiowujemy cały margines, aż dojedziemy znowu do $(...)^*$. Twierdzimy, że teraz ewaluacje wszystkich więzów da się w taki sposób zmodyfikować do poprawnych ewaluacji w nowym słowie, żeby były zgodne z kindem k. Istotnie, każda ewaluacja dotyka co najwyżej $||\Sigma||$ zmiennych, więc mniej niż długość marginesu. Stąd w każdej ewaluacji istnieją w każdym marginesie (bez utraty ogólności postaci $w_i(...)^*$) dwie wyewaluowane zmienne, które nie są w relacji następnika. Teraz wszystkie ewaluacje zmiennych występujących po drugiej z tych zmiennych przenosimy do kopii marginesu wewnątrz $(...)^*$ (mówiąc obrazowo: rozciągamy sprężynkę). W ten sposób dostaniemy zawsze poprawną ewaluację, która już nie łamie granicy pomiędzy marginesem a silnie spójną składową relację następnika. □

Lemat 3. *Jeśli słowa w_1, w_2 pasujące go k mają odpowiednio ewaluacje więzów π_1, π_2 które są zgodne z k, to istnieje też słowo w' pasujące do k, które ma ewaluacje zarówno π_1 jak i π_2 zgodne z k.*

Dowód. Odpalamy ideę z konkatenacją: jeśli jakieś $(...)^*$ w w_1 jest słowem u_1, a w w_2 jest słowem u_2, to w w' pod $(...)^*$ podstawiamy $u_1 v u_2$, gdzie v jest dowolnym łącznikiem łączącym odpowiednie stany ze spójnej składowej. □

Z lematu 2 możemy zakładać, że szukamy ewaluacji, które są zgodne, a z lematu 3 wynika, że istnienie słowa posiadającego ewaluacje zgodne jest równoważne koniunkcji istnienia zgodnych ewaluacji dla każdego więzu z osobna. Oznacza to, że wystarczy nam następujący lemat do zakończenia sprawy.

Lemat 4. *Mając dany kind k długości L oraz więz π do spełnienia, w czasie $|w|^{poly(|L|,|S_2|,|\pi|)}$ da się sprawdzić istnienie słowa w pasującego do k mającego ewaluację π zgodną z k.*

Dowód. Robimy to tak jak na początku: zauważamy, że dotkniętych zmiennych wewnątrz $(...)^*$ jest jedynie $|\pi|$, więc można założyć, że każdy $(...)^*$ jest długości $O(|S_2| \cdot |\pi|)$. Jeśli więc istnieje rozwiązanie, to istnieje rozwiązanie długości nie większej niż $poly(|L|, |S_2|, |\pi|)$. W ten sposób możemy zgadnąć całe rozwiązanie, gdyż możliwych takich słów jest $|w|^{poly(|L|,|S_2|,|\pi|)}$ (a sprawdzenie pasowania do kindu możemy wykonać przy pomocy np. automatu). □

To kończy opis tego algorytmu. Dla drzew da się zrobić podobnie, tylko obrzydliwość szczegółów technicznych rośnie nieelementarnie. Ciekawe jest pytanie, czy zamiast algorytmu o złożoności $|w|^{poly(||\Sigma||,|S_2|)}$, który jest wielomianowy dla ustalonych $||\Sigma||, |S_2|$ (to tak zwany algorytm XP), da się zrobić algorytm o stałym wielomianie, w którym tylko stała zależy od $||\Sigma||, |S_2|$, to znaczy o złożoności $f(||\Sigma||, |S_2|)|w|^{O(1)}$ (tak zwany algorytm FPT). Okazuje się, że o ile zachodzą sensowne założenia teorii złożoności ($FPT \neq W[1]$), to nie da się tego zrobić. Jak będzie trochę czasu, to to dopiszę.

13 Wymiana danych — odpowiadanie na zapytania

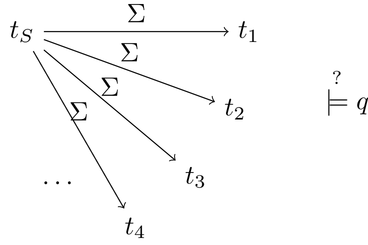
13.1 Definicja problemu

Tak jak w poprzednim rozdziale będziemy się zajmować wymianą danych, tylko w nieco innym kontekście. Rozważamy dwie bazy danych (źródłową i docelową) oparte na XML opisane schematami (odpowiednio D_S i D_T). Dostajemy drzewo w bazy źródłowej, $t_S \in L(D_S)$. Wiemy, że jest ono powiązane z drzewem z bazy docelowej $t_T \in L(D_T)$, którego nie znamy, więzami zdefiniowanymi w logice postaci

$$\forall_{\bar{x},\bar{y}}\ \Pi(\bar{x}, \bar{y}) \longrightarrow \exists_{\bar{z}}\ \Pi'(\bar{x}, \bar{z}),$$

które znaczą tyle, że jeśli lewa część implikacji spełniona jest w t_S, to prawa powinna być spełniona w t_T. Zbiór takich więzów nazywamy Σ. Zagadnieniem rozważanym podczas tego wykładu jest: co zwraca dane zapytanie q na drzewie t_T.

Problemem jaki napotykamy jest: ale przecież mogą być różne drzewa t_T z bazy docelowej, które spełniają więzy Σ. Te różne drzewa mogą dawać różne odpowiedzi na zapytanie q, co wtedy robimy?



Nim przejdziemy do dalszych rozważań spójrzmy na przykład.

Przykład Załóżmy, że D_S, D_T opisują drzewa wysokości dwa, a specyfikacja Σ zawiera jedną implikację mówiącą, że $//b(x) \to //a(x)$ zaś dane jest drzewo źródłowe

r

b(1) b(3)

Zastanówmy się jaką należy zwrócić odpowiedź na zapytanie $//a(15)$ do drzewa docelowego?

Istnieją różne drzewa docelowe spełniające schemat D_T i specyfikację Σ. Wszystkie one zawierają węzły postaci $a(1)$ i $a(3)$, a niektóre też $a(15)$, ale nie wszystkie. Co należy zwrócić w tej sytuacji?

13.2 Odpowiedzi pewne

Żeby rozwiązać jakoś ten problem umawiamy się, że mówimy wtedy, że $//a(15)$ nie jest spełnione, bo nie wszystkie drzewa docelowe je spełniają. Będziemy więc odpowiadać twierdząco wtedy, gdy wszystkie drzewa docelowe spełniają zapytanie. Dlatego technika, którą stosujemy nazywa się metodą odpowiedzi pewnych (ang. *certain answers*).

Formalnie mówimy, że $\mathcal{M} = (D_S, D_T, \Sigma)$ jest przekształceniem schematów, gdy D_S, D_T to schematy, a Σ to specyfikacja postaci $\forall_{\bar{x},\bar{y}}\ \Pi(\bar{x}, \bar{y}) \longrightarrow \exists_{\bar{z}}\ \Pi'(\bar{x}, \bar{z})$. Dla $T \in L(D_S)$ zbiór wszystkich drzew do niego pasujących oznaczamy

$$\mathcal{M}(T) = \{T' : T' \in L(D_T), (T, T') \models \Sigma\}.$$

Wówczas zbiór krotek, które na pewno zostaną zwrócone oznaczamy

$$\mathcal{M}(q, T) = \{\bar{a} : \forall_{T' \in \mathcal{M}}\ \bar{a} \in q(T')\},$$

gdzie przez $q(T')$ oznaczamy zbiór krotek, które zwraca zapytanie q na drzewie T'.

Zauważmy, że krotki mogą być też 0 elementowe, tak jak w przykładzie rozważanym wyżej $\text{certain}_{\mathcal{M}}(q, T) = \{\text{false}\}$.

13.3 Rozwiązanie uniwersalne

Chcielibyśmy teraz znaleźć algorytm, który dla danych $\mathcal{M}$, t, q oblicza $\text{certain}_{\mathcal{M}}(q, T)$. Pomysł jest taki, żeby obliczyć takie drzewo U, że $\text{certain}_{\mathcal{M}}(q, T)$ to są mniej więcej te krotki, które q zwróciłoby na U, czyli $q(U)$. Myśl jest taka, że wówczas U, to jest tak jakby najbardziej ogólne drzewo.

Przykład Niech schematy D_S i D_T opisują drzewa głębokości 2. Niech Σ składa się z jednego więzu:

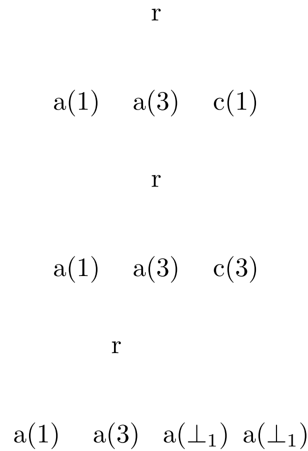
$$\forall_x b(x) \longrightarrow \exists_z a(x) \land c(z) \land a(z).$$

Rozważmy drzewo t_S:

r

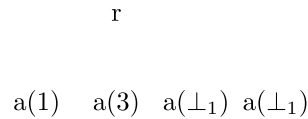
b(1) b(3)

Wówczas możliwymi drzewami t_T są na przykład:



gdzie $\perp_1$ jest pewną daną.

Wówczas mówimy, że rozwiązaniem uniwersalnym jest drzewo



bo jest jakby najbardziej ogólne. Dana $\perp_1$ może być równa zarówno 1 jak i 3 jak i cokolwiek innego. Za chwilę sformalizujemy definicję drzewa uniwersalnego (najbardziej ogólnego) w taki sposób w jaki zwykle definiuje się obiekty najbardziej ogólne. To znaczy powiemy, że to jest takie dobre drzewo t, że dla każdego innego dobrego drzewa t' istnieje homomorfizm z t w t'.

Ustalamy, że nasze drzewo uniwersalne oprócz normalnych danych może mieć w sobie dane nieokreślone, takie jak w naszym przykładzie $\perp_1$. Zbiór tych wartości będziemy oznaczać Null(U). Homomorfizm będzie je przenosił w prawdziwe dane innych drzew oznaczane Val lub ich dane nieokreślone Null.

Homomorfizm będzie się zatem składał tak naprawdę z dwóch homomorfizmów, h_{dom} przekształcającego wierzchołki drzewa U na wierzchołki drzewa docelowego oraz h_{val} przekształcającego dane nieokreślone U na dane (określone lub nieokreślone) drzewa docelowego.

Definicja rozwiązania uniwersalnego Powiemy, że U jest rozwiązaniem uniwersalnym dla drzewa T względem przekształcenia schematów $\mathcal{M}$, gdy

- $U \in \mathcal{M}(T)$, czyli U jest jednym z rozwiązań
- dla każdego $T' \in \mathcal{M}(T)$ istnieją homomorfizmy $h_{\mathrm{dom}} : \mathrm{dom}\, U \to \mathrm{dom}\, T'$ oraz $h_{\mathrm{val}} : \mathrm{Null}(U) \to \mathrm{Val}(T') \cup \mathrm{Null}(T')$
  - jeśli wierzchołek w jest dzieckiem wierzchołka v w U, to wierzchołek $h_{\mathrm{dom}}(w)$ jest dzieckiem wierzchołka $h_{\mathrm{dom}}(v)$ w T'
  - jeśli wierzchołek v w drzewie U zawiera σ dla danej σ, to wierzchołek $h_{\mathrm{dom}}(v)$ w T' też zawiera σ
  - jeśli wierzchołek v w drzewie U zawiera $\perp_i$ dla $\perp_i \in \mathrm{Null}(U)$, to wierzchołek $h_{\mathrm{dom}}(v)$ w T' zawiera $h_{\mathrm{val}}(\perp_i)$

Będziemy od tej pory zakładać, że D_S i D_T to DTD oraz Π i Π' używają jedynie /, //, *, =. Tak samo zapytania mogą używać tylko /, //, *, =.

Ilustracja Załóżmy, że U jest rozwiązaniem dla danego przekształcenia i ponadto jest to nasz *kandydat* na optymalne rozwiązanie.

$b(1)$

$a(\bot_3)$ $a(3)$

$c(\bot_1)$ $c(\bot_1)$

Zamieńmy go na wzorzec o zmiennych x_1, x_2 będący postaci $Q = \exists_{\bot_1,\bot_3}$

$b(x_1)$

$a(\bot_3)$ $a(x_2)$

$c(\bot_1)$ $c(\bot_1)$

Wtedy jeśli obliczymy Q na U to otrzymamy $Q(U) = \{(x_1 = 1, x_2 = 3)\}$. Czyli jeśli U ma faktycznie być zapytaniem optymalnym to na każdym innym drzewie T' będącym rozwiązaniem powinno zachodzić $Q(T') = Q(U) = \{(x_1 = 1, x_2 = 3)\}$. Czyli Q powinno zanurzać się w T' w taki sposób, że x_1 przechodzi na 1 a x_2 na 3. A to sprowadza się do stwierdzenia, że U zanurza się w T'.

Rozwiązanie uniwersalne jest bardzo przydatne, gdyż spełnia ono następujący fakt:

Fakt 1. *Dla dowolnego* $q \in \Pi(/, //, *, =)$ *zachodzi* $certain_{\mathcal{M}}(q, T) = q(U)|_{val}$*, gdzie przez* $|_{val}$ *rozumiemy obcięcie do tych krotek, które nie zawierają w ogóle Nulli.*

Dowód 1. *Udowodnimy zawieranie zbiorów w obie strony.*

Najpierw $\subseteq$*. Zupełnie jasne jest, że* $certain_{\mathcal{M}}(q, T) \subseteq q(U)$*, gdyż krotki, które siedzą we wszystkich* $q(T')$ *dla* $T' \in \mathcal{M}$ *w szczególności siedzą też w* $q(U)$*. Wystarczy więc jedynie uzasadnić, że wycinając krotki, które zawierają Nulle nie wytniemy żadnej krotki z* $certain_{\mathcal{M}}(q, T)$*. Zauważmy jednak, że w* $certain_{\mathcal{M}}(q, T)$ *są jedynie krotki, które nie zawierają Nulli. Przypuśćmy przeciwnie, niech pewna krotka zawiera Nulla* $\bot_i$*, musi on więc należeć też do* U*. Zauważmy, że skoro* U *należy do* $\mathcal{M}(T)$*, to* U'*, które różni się tylko tym od* U*, że w miejsce* $\bot_i$ *ma* $\bot'_i$ *też należy do* $\mathcal{M}(T)$*. Zatem* U' *nie zawiera* $\bot_i$*, sprzeczność z założeniem, że pewna krotka z* $certain_{\mathcal{M}}(q, T)$ *zawierała* $\bot_i$*.*

Teraz pokażemy zawieranie $\supseteq$*. Niech krotka* $\bar{a} \in q(U)|_{val}$*. Zatem* $U \models q(\bar{a})$*, czyli istnieje homomorfizm* f *z* $q(\bar{a})$ *w drzewo* U*. Gdy złożymy ten homomorfizm* f *z homomorfizmem* $h : U \to T'$ *(złożonym z* h_{dom} *i* h_{val}*), to dostaniemy homomorfizm* $f \circ h$ *idący z* $q(\bar{a})$ *w* T'*. Pokazuje on, że* $T' \models q(\bar{a})$*.*

13.4 Ograniczenia

Powyższe rozumowanie pokazuje, że o ile istnieje rozwiązanie uniwersalne U, to faktycznie wystarczy na nim ewaluować wszystkie zapytania. Niestety okazuje się, że dla tak ogólnych schematów i przekształceń jakie rozważaliśmy do tej pory, rozwiązanie uniwersalne nie musi istnieć. Dlatego też wprowadzimy dodatkowe ograniczenia.

Ograniczenie Σ Aby mieć szansę na sukces przyjmiemy ograniczenie, że w ramach specyfikacji przekształcenia Σ po stronie docelowej nie może być // ani $*$. Chodzi o to by uniknąć następującego problemu:

Rozważmy przekształcenie mówiące, że $r/a(x) \to r//b(x)$ i DTD docelowe D_t mówiące, że $r \to cd, c \to b, d \to b$. Wtedy dla drzewa wejściowego $r/a(7)$ wszystkie rozwiązania mają następujący kształt:

r

c d

b b

W szczególności rozwiązaniami są dwa drzewa: w jednym wartość 7 jest w lewym b a w drugim jest w prawym b. W takiej sytuacji nie da się znaleźć rozwiązania uniwersalnego.

Ograniczenie D_t Dodatkowo przyjmiemy, że schemat docelowy D_t jest tzw. *nested relational DTD*. Oznacza że:

- przejścia są postaci $\sigma_i \to \hat{\sigma_1}\hat{\sigma_2}\ldots\hat{\sigma_n}$ gdzie $\sigma_1, \sigma_2, \ldots, \sigma_n$ to alfabet zaś $\hat{\sigma_i}$ to jedna z możliwości: $\epsilon, \sigma_i, \sigma_i^*, \sigma_i^+, \sigma_i^?$.
- schemat jest nierekurencyjny.

Przyczyna dla której przyjmujemy to ograniczenie stanie się widoczna przy konstrukcji rozwiązania uniwersalnego.

13.5 Poprawianie schematu

Poniżej opisana jest procedura pozwalająca znaleźć rozwiązanie uniwersalne U o ile specyfikacja Σ i schemat D_t spełniają powyższe ograniczenia.

Najpierw stworzymy jakiś schemat uwzględniający różne wymagania co do rozwiązań a następnie będziemy go poprawiać by spełniał D_t i by był rozwiązaniem uniwersalnym.

Uzupełnianie Najpierw dla danego S_T i określonego schematu Π stworzymy $T' \in L(S_T)$ takie, że $T' \models \Pi$. Wtedy by stworzyć rozwiązanie T' dla specyfikacji Σ stworzymy drzewo łączące w korzeniu wszystkie Π_T które wymusza na nas Σ.

Dla danego S_T i Π postępujemy w dwóch krokach. Pierwszy z kroków zamienia dany schemat Π na drzewo oznaczane $cpl(\Pi)$. Drugi krok operuje na $cpl(\Pi)$ sklejając odpowiednie wierzchołki i tworzy drzewo oznaczone $wrg(cpl(\Pi))$.

Krok pierwszy: dodajemy do Π brakujące węzły w *minimalny* sposób.

Jeśli na przykład w S_T mamy $a \to b^+cd^?$ i dostajemy w Π gdzieś poddrzewo poniższej postaci to zamieniamy je od razu

a $\Rightarrow$ a

b b b b c

Drzewo powstałe w ramach powyższej procedury oznaczamy $cpl(\Pi)$. Pokażemy, że ma ono poniższą własność.

Lemat 5. *Dla każdego drzewa $T \in L(S_T)$ zachodzi*

$$T \models \Pi \Leftrightarrow T \models cpl(\Pi).$$

Dowód. Oczywiście implikacja w lewą stronę zachodzi, bo $cpl(\Pi)$ jest bogatszy niż Π. Jednocześnie dodatkowe wymagania nakładane przez $cpl(\Pi)$ wynikają z S_T, więc implikacja w drugą stronę też zachodzi. □

Krok drugi: sklejamy to co musi zostać sklejone.

Może się zdarzyć że mamy regułę $a \to bc^+$. Wtedy wykonujemy podmianę postaci

$a \qquad \Rightarrow \qquad a$

$b \quad b \quad c \quad c \qquad\qquad b \quad c \quad c$

$\Pi_1 \quad \Pi_2 \quad \Pi_3 \quad \Pi_4 \qquad \Pi_1 \quad \Pi_2 \quad \Pi_3 \quad \Pi_4$

Tutaj uwaga — przy powyższym sklejaniu uzgadniamy null'e! Tzn. jeśli sklejamy wierzchołek z wartością $\bot_i$ z wierzchołkiem o wartości $\bot_j$ to uzgadniamy $\bot_i = \bot_j$. Można to zrealizować na przykład zmieniając wszystkie wystąpienia $\bot_j$ na $\bot_i$.

Powyższą operację oznaczamy wrg. Na skutek złożenia dwóch opisanych powyżej operacji uzyskujemy drzewo $wrg(cpl(\Pi))$. Pokażemy, że ma ono poniższe własności.

Lemat 6. *Dla każdych S_T i Π zachodzi*

$$wrg(cpl(\Pi)) \equiv_{S_T} \Pi,$$

oraz

$$wrg(cpl(\Pi)) \models S_T.$$

Dowód. Co do pierwszej równoważności to korzystając z poprzedniego lematu wystarczy udowodnić równoważność $wrg(spl(\Pi))$ i $cpl(\Pi)$. Dowód jest identyczny jak poprzednio, tylko *na odwrót*: schemat $wrg(cpl(\Pi))$ jest uboższy niż $cpl(\Pi)$, ale wszystko co jest w nim sklejone zostało wymuszone przez S_T.

Druga własność wynika wprost z konstrukcji: w $wrg(cpl(\Pi))$ występują wszystkie litery które muszą występować w odpowiednich miejscach i to w dobrej kolejności. Dodatkowo ich liczba jest zgodna z S_T, bo tam gdzie było ich za dużo to skleiliśmy. □

Na skutek tej procedury stworzyliśmy drzewo $wrg(spl(\Pi))$ które spełnia S_T i jest równoważne Π (modulo S_T). Pozostaje teraz w tym drzewie zamienić zmienne na Null'e i w ten sposób otrzymujemy rozwiązanie uniwersalne U.