Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Automatyki i Informatyki Stosowanej

Rok akademicki 2015/2016

PRACA DYPLOMOWA INŻYNIERSKA

Michał Nowotnik

Prototyp dedukcyjnej bazy danych opartej na języku regułowym 4QL

Opiekun pracy:
dr inż. Mariusz Kamola

Ocena: .....

.....

Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego

Kierunek:	Informatyka
Specjalność:	Systemy Informacyjno-Decyzyjne
Data urodzenia:	30.06.1990
Data rozpoczęcia studiów:	01.10.2011

.....

podpis

EGZAMIN DYPLOMOWY

Złożył egzamin dyplomowy w dniu .....

z wynikiem .....

Ogólny wynik studiów: .....

Dodatkowe wnioski i uwagi komisji: .....

.....

Streszczenie

4QL to język deklaratywny programowania logicznego, wzorowany na języku Datalog, umożliwiający tworzenie zapytań do dedukcyjnych baz danych. Niniejsza praca opisuje algorytm do ewaluacji programów napisanych w tym języku i jego implementację o nazwie fovris. W ramach pracy zbadano różne podejścia do ewaluacji programów Datalog i wykorzystano je do wykonania wydajnej implementacji silnika 4QL.

Słowa kluczowe: Datalog, Programowanie logiczne, 4QL

Abstract

Title: Evaluation engine for the query language 4QL

4QL is a declarative programming language derived from Datalog. It is used to make queries to deductive databases. This work presents the algorithm for evaluation of 4QL programs and its implementation named fovris. Various approaches to Datalog evaluation were researched and used to create an efficient implementation of a 4QL engine.

Keywords: Datalog, Logic programming, 4QL

Spis treści

1 Wstęp

W dzisiejszych czasach sprawny dostęp do informacji jest ważniejszy, niż kiedykolwiek w dziejach ludzkości. Dzieje się tak, ponieważ informacje, przechowywane teraz głównie w formie cyfrowej w bazach danych, są dostępne w tak dużej ilości, że coraz trudniejsze staje się ich przetwarzanie i zarządzanie nimi. Kiedyś algebra relacyjna była uważana za standard, jeśli chodzi o język umożliwiający prace z bazami danych. Wkrótce jednak okazało się, że pewne proste operacje na danych są niemożliwe do wykonania jedynie za pomocą algebry relacyjnej [1, Rozdział 12].

Flagowym przykładem jest domknięcie przechodnie grafu. Weźmy za przykład system Veturilo w Warszawie. W bazie danych zdefiniowana jest relacja Połączenie(X,Y), która wyszczególnia bezpośrednie połączenia między punktami oznaczającymi stacje Veturilo. Relacja stanowi więc graf, gdzie krawędziami są bezpośrednie połączenia między stacjami a punktami stacje. Chcemy wiedzieć czy wszystkie stacje są możliwe do osiągnięcia, biorąc za punkt startowy Rondo Dmowskiego. Takie zapytanie jest niemożliwe do zrealizowania za pomocą algebry relacyjnej [1, Rozdział 17], a w konsekwencji za pomocą języka SQL (z wyjątkiem standardu SQL-3), który na algebrze relacyjnej jest oparty. Datalog pozwalający na rekursywne zapytania do relacyjnej bazy danych pozwala znaleźć odpowiedź na takie pytanie.

Datalog to deklaratywny język programowania logicznego opracowany w latach 80 ubiegłego wieku. W tamtym okresie zainteresowanie nim nie było duże z uwagi na brak widocznych zastosowań [7]. Dopiero w ostatnich latach pojawiły się nowe zastosowania w dziedzinach integracji danych, ekstrakcji informacji, analizy programów, bezpieczeństwa i obliczeń rozproszonych [7]. Datalog doczekał się wielu rozszerzeń i usprawnień, które sprawiają, że jest użyteczny w realnych zastosowaniach. Jednym z nich jest Datalog$^{\neg\neg}$, który umożliwia zdefiniowanie negacji w przesłankach i konkluzji reguł [1, Rozdział 14].

4QL to język, który buduje na bazie założeń Datalogu$^{\neg\neg}$. Elegancko radzi sobie z problemem niewiedzy w Datalogu oraz sprzecznościami w bazie wiedzy poprzez wprowadzenie czterowartościowej logiki. Obliczanie zapytań w tym języku ma złożoność wielomianową, może być więc używany w praktycznych zastosowaniach [8].

1.1 Cel pracy

Celem niniejszej pracy było stworzenie prototypu wydajnej implementacji silnika do ewaluacji języka 4QL* [8]. Aplikacja miała być szybsza, niż obecnie dostępne eksperymentalne implementacje tego silnika. Przyspieszenie obliczeń miało być osiągnięte przy wykorzystaniu jednostki GPU wspierającej bibliotekę CUDA. Implementacja samego silnika do obliczania programów 4QL okazała się nietrywialna, więc cel pracy został zmieniony i zrezygnowano z użycia GPU w aplikacji. Zamiast tego skupiono się na wykonaniu wydajnej implementacji na CPU, która zademonstruje możliwość zrównoleglenia albo rozproszenia obliczeń.

Język 4QL to rozszerzenie języka 4QL, ale dla wygody czytelnika, będzie nazywany w tej pracy po prostu 4QL.*

Założeniem pracy było zaimplementowanie funkcjonalności dostępnych w interpreterze Inter4QL 2 i rozszerzenie ich wedle uznania. Program miał wczytywać skrypty identyczne z tymi akceptowanymi przez Inter4QL i otrzymywać te same wyniki, o ile są one poprawne. Końcowym etapem pracy było zbadanie wydajności wykonanej aplikacji w stosunku do istniejących rozwiązań i przedyskutowanie wyników.

1.2 Istniejące rozwiązania

W tej sekcji skupiono się na programach, które umożliwiają pracę z językami 4QL i Datalog. Systemów wnioskujących do Dataloga jest dużo. Jako że Datalog istnieje od ponad 30 lat doczekał się bardzo wydajnych implementacji i rozszerzeń. Zostały zaprezentowane wybrane, które posłużyły do porównania wydajności opracowanego programu. Narzędzia do ewaluacji 4QL, które zostały wymienione niżej, są dostępne na stronie: `http://4ql.org`.

Inter4QL

Inter4QL to interpreter do ewaluacji programów języka 4QL. Wersja trzecia tego narzędzia, napisana przez Aleksandra Bulanowskiego, wyszła 30 marca 2016 roku. Rozszerza wersję drugą o dodatkowe elementy składni, poprawioną wydajność i usunięcie błędów w ewaluacji.

Aplikacja została napisana w języku C++11 i korzysta z bibliotek Flex i Bison do, odpowiednio, tokenizacji i parsowania skryptów. Po ewaluacji skryptu 4QL, wygenerowane fakty przetrzymywane są w pamięci komputera. Z poziomu interpretera użytkownik może wystosowywać zapytania do bazy o zdefiniowane relacje. Inter4QL jest dostępne w wersjach na system Microsoft Windows, kompilowaną przy użyciu środowiska MinGW, oraz Linux.

4QL Modeler

4QL Modeler to narzędzie napisane w języku C# działające na systemie Microsoft Windows. Najnowsza wersja to 4QL Modeler 2.0, która została wypuszczona 27 sierpnia 2014 roku. Autorem jej jest Maria Rekowska. Posiada interfejs graficzny do tworzenia zapytań i ich wizualizacji. Zezwala definiowanie w regułach kwantyfikatorów.

souffle

souffle to wynik pracy wielu badaczy pod kierownictwem Bernharda Scholza [**souffle**][**soufflePub**]. Jest to interpreter i translator do programów Datalog, który generuje wydajny kod w C++. Korzysta przy tym z wielu optymalizacji, w tym semi-naiwnej ewaluacji i paralelizacji. Głównym zastosowaniem souffle jest analiza statyczna kodu źródłowego programów.

2 Studia literaturowe

2.1 Datalog

Datalog to deklaratywny język zapytań do baz danych wywodzący się z programowania logicznego. Składnia klasycznego Dataloga stanowi podzbiór składni Prologa, a sam termin "Datalog" oznaczał początkowo po prostu elementy tej składni. Jest on wynikiem potrzeby integracji rozwiązań z zakresu sztucznej inteligencji i baz danych [**Ceri**]. Dokładniej, wprowadza on do relacyjnych baz rekurencję i umożliwia obliczenie domknięcia przechodniego. Mimo to, nie został ciepło przyjęty przez społeczność, a jego praktyczne zastosowanie długo było poddawane w wątpliwość [**Ceri**]. W dzisiejszych czasach Datalog jest używany w wielu niespodziewanych dziedzinach np. statyczna analiza kodu źródłowego [7]. Jest to możliwe dzięki wielu rozszerzeniom jakich doczekał się ten język.

2.1.1 Składnia

Datalog, jak i programowanie logiczne, wywodzi się z rachunku predykatów pierwszego rzędu. Zapytania w tym języku można więc wyrazić za pomocą notacji matematycznej. Programy Datalogu zawierają jedynie dwa typy wyrażeń: fakt i reguła. Reguła to alternatywa literałów z dokładniej jednym niezanegowanym literałem. Formuły takie nazywają się klauzalami Horna i umożliwiają dedukcję nowych faktów. Reguła może być zdefiniowana następująco:

$$H(t) \leftarrow R_1(t_1), ..., R_n(t_n),$$

gdzie $n \geq 1$, $R_1, ..., R_n$ to nazwy *relacji*, a $t_1, ..., t_n$ to krotki *termów*. $R(t)$ stanowi *literał*. Po prawej stronie strzałki stoi ciąg literałów tworzących *ciało* reguły, bądź *przesłanki* reguły. Po lewej stronie widoczna jest *głowa* reguły, która jest też nazywana *konkluzją*. Termy w regułach mogą zawierać *stałe* (np. stałe liczbowe: 1,2,3) lub *zmienne*.

Fakt jest to reguła, której przesłanki są zawsze prawdziwe:

$H(t) \leftarrow 1,$

gdzie $H(t)$ to ugruntowany literał zawierający termy, które mogą zawierać tylko stałe.

Składnia Dataloga odpowiada definicjom zapisanym powyżej, ale robi to w uproszczonej formie, która można być łatwo zapisana i odczytana przez aplikację. Przykładowo, zdanie "Kot mówi miau" można zapisać wprowadzając relację *mowi* z dwoma stałymi `kot` i `"miau"`:

```
mowi(kot,"miau").
```

Z kolei regułę, "Jan ma zwierzę X, które mówi miau", można zapisać w ten sposób:

```
ma(Jan,X) :- mowi(X,"miau").
```

Po dopasowaniu możliwych faktów do literałów regule można wywnioskować, że Jan ma kota.

2.1.2 Semantyka

Modelem programu Datalog P nazywamy taką *instancję* bazy, która spełnia wszystkie reguły w P. Istnieje wiele modeli spełniających reguły, ponieważ mają one formę implikacji - nawet jeśli przesłanki są fałszywe to głowa może być prawdziwa po podstawieniu stałych pod zmienne. Poszukiwane są jednak tylko te fakty, które wynikają logicznie z przesłanek. Wszystkie inne dodatkowe fakty są niepożądane, bo nie ma pewności, że są one prawdziwe. Model, który zawiera tylko fakty logicznie wynikające z reguł nazywany jest *modelem minimalnym* i jest on celem ewaluacji programów Datalog [1, Rozdział 12].

Podejściem najczęściej wykorzystywanym przy ewaluacji zakładającej wnioskowanie w przód jest *metoda punktu stałego*. Polega ona na dopasowywaniu faktów do reguł dla każdej reguły aż dwie kolejnej instancje bazy będą takie same. Nie ma potrzeby przeprowadzania kolejnych iteracji, ponieważ wnioskowanie jest *monotoniczne* - fakty raz wydedukowane nie mogą być usunięte.

2.1.3 Ewaluacja

Ewaluacja programów Datalogu jest realizowana zwykle dwa sposoby. Klauzule Horna umożliwiają dwie strategie wnioskowania: w przód (*modus ponens*) i w tył (*sld-rezolucja*). Wnioskowanie w tył zaczyna się od celu (literału), który chcemy udowodnić i polega na przeszukiwaniu różnych gałęzi rozwiązań, aż znajdziemy odpowiedni zestaw literałów, z którego wynika cel. Wnioskowanie w przód rozpoczyna się od bazowego zestawu faktów, który kolejno aplikuje się do ciał reguł, aż otrzymamy interesujący nas fakt. Zaletą wnioskowania w tył jest to, że przeszukuje tylko część drzewa rozwiązań i nie ma potrzeby obliczania wszystkich faktów. Wnioskowanie w przód nie jest skupione na celu, tylko na ewentualnym celu - obliczane są wszystkie możliwe fakty jakie wynikają z programu, dopóki nie znajdziemy celu. Ma jednak tę zaletę, że musi być przeprowadzone tylko raz. Omówiona zostanie tylko ewaluacja w kontekście wnioskowania w przód, ponieważ ta strategia użyta jest do ewaluacji programów 4QL.

Datalog jest to język, którego zadaniem miało być tworzenie zapytań do baz relacyjnych. Operuje on więc na relacjach. Deklaratywnym językiem, które pozwala definiowań operacje na relacjach jest *algebra relacyjna*. Każdą regułę można przetłumaczyć na równanie algebry relacyjnej **[Ceri]**. Do wyrażenia reguły wystarczą trzy operatory z algebry relacyjnej:

- **Selekcja**(σ) Selekcja jest używana przy dopasowywaniu relacji do pojedynczego literału w ciele reguły. Literał $R(1, X)$ można przetłumaczyć na: $\sigma_{1=1}(R)$. Dla przykładowej relacji R wynik tej operacji będzie następujący:

$$R = \{< 1, 2 >, < 2, 2 >, < 1, 3 >\}$$
$$\sigma_{col_1=1}(R) = \{< 1, 2 >, < 1, 3 >\}$$

- **Złączenie**($\bowtie$) Złączenie jest potrzebne do agregacji wyników dwóch kolejnych literałów. Mając w ciele reguły $H(X) \leftarrow R_1(X, Y), R_2(X, Z)$, trzeba złączyć obie relacje z warunkiem, że pierwszy parametr krotek jest taki sam. Przykła-

dowa ewaluacja ciała takiej reguły:

$$R_1 = \{< 1, 2 >, < 2, 2 >\}$$
$$R_2 = \{< 1, 3 >, < 2, 3 >\}$$
$$TMP(X, Y, Z) = R_1 \bowtie_{col_1 = col_1} R_2 = \{< 1, 2, 3 >\}$$

- **Projekcja**(Π)

Do wybrania tylko tych kolumn z relacji, które odpowiadają sygnaturze głowy reguły niezbędna jest projekcja. Punkt wyżej otrzymaliśmy tymczasową relację $TMP(X, Y, Z)$. Oryginalną regułę można zapisać uwzględniając złączenie: $H(X) \leftarrow TMP(X, Y, Z)$. Nie ma potrzeby dopasowywać relacji ani łączyć z innymi. Pozostała jedynie projekcja pierwszej kolumny - będzie to ostateczny wynik ewaluacji oryginalnej reguły:

$$TMP(X, Y, Z) = \{< 1, 2, 3 >\}$$
$$result = \Pi_{col1}(TMP) = \{< 1 >\}$$

Ewaluację programu Datalog można, więc zapisać jako wykonanie równań algebry relacyjnej na relacjach w bazie i dodawanie wyników do bazy. Należy powtarzać ten proces aż do osiągnięcia punktu stałego instancji. Jest to tzw. *naiwny* algorytm ewaluacji. Powoduje on obliczenie wszystkich faktów od nowa w każdej iteracji, nawet jeśli już występują w bazie. Ulepszeniem tego algorytmu jest algorytm *semi-naiwny*. Wymaga on rozbicia każdej reguły ze względu na predykaty *intensjonalne* (predykaty figurujące w głowach reguł). Niech r będzie regułą o budowie:

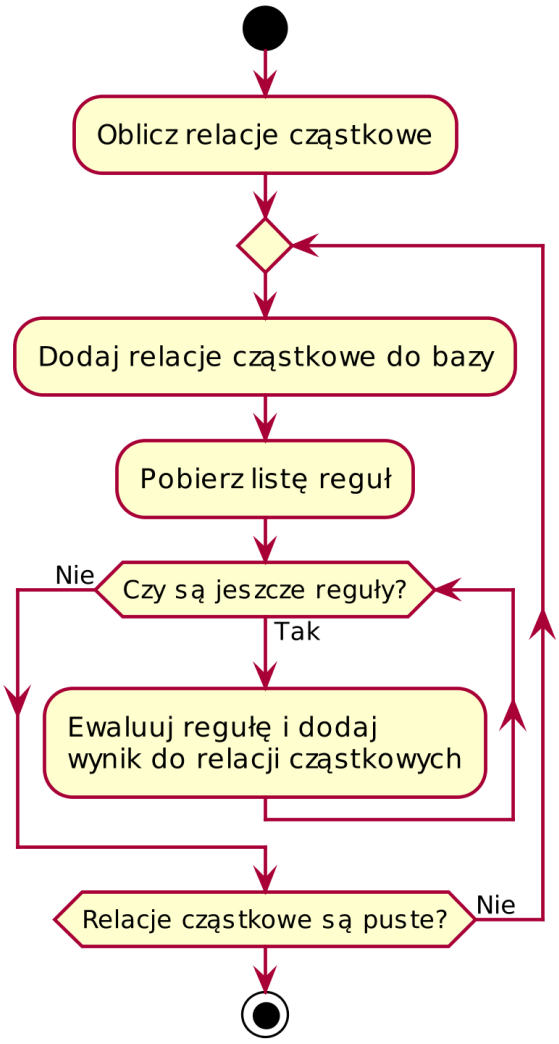
$$r = H(t) \leftarrow R_1(t_1), ..., R_n(t_n), S(s_1), ..., S_m(s_m),$$

gdzie $i \geq 1$, $m \geq 1$, $S_1, ..., S_m$ to relacje *ekstensjonalne* (relacje niewystępujące w głowach reguł), a $s_1, ..., s_m$ to krotki termów. Zdefiniujmy $\Delta^i_{R_n}$ jako zbiór ugruntowanych literałów należących do relacji R_n, które zostały dodane do bazy w trakcie iteracji $i - 1$. Rozbijmy r na n reguł w każdej iteracji według schematu:

$$r^i_j = H(t) \leftarrow R_1(t_1), ..., \Delta^i_{R_j}(t_j), R_n(t_n), S(s_1), ..., S_m(s_m),$$

gdzie $j \in [1, n]$, i. W celu obliczenia programu algorytmem semi-naiwnym trzeba naj-

pierw obliczyć relacje cząstkowe, $\Delta^1_{R_n}$, a później dopasowywać fakty do każdej reguły r^i_j aż do osiągnięcia punktu stałego. Pozwala to ominąć obliczanie wszystkie faktów od nowa, ponieważ brane są pod uwagę jedynie fakty wywnioskowane w poprzedniej iteracji. W zaprezentowanym algorytmie występuje jednak pewna redundancja obliczeń, którą można usunąć. Ulepszona wersja tego algorytmu jest w rozdziale 13 książki *Foundations of Databases: The Logical Level.*



Rysunek 1: Semi-naiwny algorytm do ewaluacji Dataloga

2.1.4 Negacja

Datalog przyjmuje *założenie zamkniętego świata* (z ang. Closed World Assumption) [1][Rozdział 12]. Postuluje ono, że każdy fakt, który nie jest aktualnie znany, musi być fałszywy. Wprowadzenie negacji w klasycznym Datalogu sprawia, że jest

on bardziej ekspresyjny, ale może łatwo doprowadzić do sytuacji, w której nie ma unikalnego minimalnego modelu. Rozważmy program pokazujący ten problem:

Wydruk 1: Przykład negacji bez minimalnego modelu

```
p :- r, !q.
q :- r, !p.
r.
```

Konsekwencją zdefiniowanych reguł jest to, że istnieją dwa minimalne modele. W zależności od której reguły zaczniemy ewaluację możemy otrzymać albo $\{r, p\}$ albo $\{r, q\}$. Taki niedeterministyczny wynik stoi w sprzeczności z tym czego oczekujemy po logicznym języku regułowym. Czemu tak się dzieje? Literały p i q należą do relacji intensjonalnych - ich zawartość może zmieniać się w trakcie obliczeń. Zakazanie negacji w literałach z predykatami intensjonalnymi pozwala na uniknięcie takich problemów. Odmiana Datalogu narzucająca to ograniczenie nazywa się Semipozytywny Datalog [1][Rozdział 15]. Nie jest jednak popularna, ponieważ to ograniczenie mocno zmniejsza użyteczność negacji. Innym sposobem na poradzenie sobie z negacją jest *stratyfikacja*. Polega na odczytywaniu reguł w taki sposób, żeby negacja relacji R była użyta dopiero po tym jak reguły z konkluzją w R zostały wykorzystane.

Wydruk 2: Przykładowy program przed stratyfikacją

```
r4 :- !r1.
r1 :- !r2.
r2 :- r3.
```

Wydruk 3: Przykładowy program po stratyfikacji

```
r2 :- r3.
r1 :- !r2.
r4 :- !r1.
```

Po lewej oryginalnym program, a po prawej poddany stratyfikacji. Ta operacja nie powoduje, że znajdziemy unikalny minimalny model. Modelów minimalnych może być cały czas wiele w zależności od której reguły zaczniemy ewaluację. Ta strategia wykonywania reguł jest intuicyjnie sensowna i nie doprowadzi do sprzeczności. Oryginalny program mógłby zwrócić błędny model, $\{r4, r1, r2\}$, który nie spełnia wszystkich reguł.

Stratyfikacja wymaga stworzenia grafu skierowanego reguł, która zaczyna się od reguł, które nie mają żadnej negacji w ciele. Dodawania kolejnych węzłów polega na znalezieniu reguł, które mają zanegowane predykaty obecne konkluzjach reguł

poprzednich węzłów. Strzałka z węzła R_1 do węzła R_2 oznacza, że R_2 posiada zanegowane predykat, który jest w konkluzji R_1. Odczytanie grafu w taki sposób, żeby zostało zachowane uporządkowanie definiowane przez krawędzie pozwala otrzymać stratyfikowany program [1][Rozdział 15]. Ta metoda ma jednak swoje ograniczenia. Istnieją programy, jak ten w wydruku 1, które są niestratyfikowalne. Prostym testem na sprawdzenie tego jest szukanie cykli w utworzonym grafie. Jeśli istnieją, programu nie da się ewaluować, wykorzystując tę semantykę. Ponadto

2.2 4QL

Negacja w językach programowania logicznego od zawsze sprawiła problemy. Opracowywane są specjalne semantyki, które radzą sobie z negatywną informacją przekazywaną w regułach. Szczególne problemy sprawia negacja w konkluzji reguł, która może prowadzić do sprzecznych wniosków **[SzalasZeszyt2013]**. 4QL wprowadza dodatkową wartość logiczną, *inconsistent*, która pozwala reprezentować taką informację. Język 4QL posługuje się 4-wartościową logiką:

- T - prawda (z ang. *true*)
- F - fałsz (z ang.*false*)
- U - brak informacji (z ang. *unknown*)
- I - sprzeczność (z ang. *inconsistent*)

4QL przyjmuje założenie otwartego świata. Brak wiedzy o jakimś fakcie nie oznacza, że można wywnioskować jego nieprawdziwość, jak to ma miejsce na przykład w relacyjnych bazach danych - jeśli nie ma wiersza z wartością 'Jan' w tabeli Pracownicy, to Jan nie jest pracownikiem firmy.

Ważną cechą jak różni 4QL od klasycznego Datalogu są własne matryce logiczne (Tabela 1). Powodują, że proste tożsamości logiczne znane z algebry Boole'a przestają obowiązywać. I tak, zamiana reguły:

$c \leftarrow (a \vee b)$ na $(c \leftarrow a) \vee (c \leftarrow b)$,

która jest naturalna w klasycznym Datalogu, w 4QL jest niepoprawna.

2.2.1 Składnia

Reguły w 4QL mogą być zdefiniowane jako klauzule Horna, ale generalna forma reguły już klauzulą Horna nie może być, ponieważ spójniki logiczne nie spełniają tożsamości algebry Boole'a - nie można rozbić reguły zawierającej alternatywę na dwie reguły. Zdefiniujmy więc generalną formę reguły jako:

$$H(t) \leftarrow (R_{1,1}(t_{1,1}) \wedge ... \wedge R_{1,n}(t_{1,n_1})) \vee ... \vee (R_{m,1}(t_{m,1}) \wedge ... \wedge R_{m,n_m}(t_{m,n_m})),$$

gdzie $n_i \geq 1$ dla każdego i, $m \geq 1$, a $R_{1,1}, ..., R_{m,n_m}$ to nazwy relacji z krotkami termów $t_{1,1}, ..., t_{m,n_m}$. Krotki termów mogą zawierać tylko zmienne i stałe. Krotki we fragmencie reguły tylko ze spójnikami koniunkcji, $t_{m,1}, ..., t_{m,n_m}$ muszą zawierać wszystkie zmienne jakie zawiera głowa reguły w t. Każdy literał L $(R_{m,n_m}(t_{m,n_m}))$ może pojawić się w regule w formie zanegowanej, $\neg L$, lub niezanegowanej. W ciele reguły mogą też występować specjalne literały zwane *zewnętrznymi*.

Reguły z pustymi przesłankami nazywane są, podobnie jak w Datalogu, *faktami*. Fakty mają wartość logiczną prawda, a zanegowane fakty - fałsz. Fakt jest sprzeczny, jeśli w bazie występuje fakt prawdziwy i fakt fałszywy jednocześnie.

Dodatkowymi elementami składni 4QL, wychodzącymi poza rachunek predykatów, są moduły i literały zewnętrzne. Reguły, relacje i fakty definiowane są w kontekście modułu. W ciele reguły można wystosowywać zapytania do innych modułów poprzez literały zewnętrzne zapisane jako:

$$nazwa\_modułu.nazwa\_relacji(parametry)$$

lub też

$$nazwa\_modułu.nazwa\_relacji(parametry)\, in\, S,$$

gdzie S to podzbiór wartości logicznych ze zbioru $\{F, U, I, T\}$. Ewaluacja literału zewnętrznego przebiega tak samo jak ewaluacja normalnego literału - fakty są dopasowywane, jeżeli spełniają warunki narzucone przez termy. Dodatkowym ograniczeniem może być podzbiór wartości logicznych, które mogą przyjmować fakty. Należy zauważyć, że wartość logiczna U w kontekście interpretacji oznacza brak ugruntowanego literału i jego negacji w zbiorze. Definiując wartość U w podzbiorze wartości logicznych

dla literału zewnętrznego, zadajemy pytanie o *brak informacji* - jest to odpowiednik negacji w Datalogu. Z tym, że architektura *warstwowa* narzucona przez moduły omija problem stratyfikacji reguł. Jedynie literał zewnętrzny może pytać moduły o negatywną informację, a moduły które pyta zostały już obliczone w momencie zapytania. Ponadto 4QL zabrania odnoszenia się w literałach zewnętrznych do modułów, które zostały zdefiniowane później, niż aktualny, co zapobiega cyklom w obliczeniach i gwarantuje deterministyczny wynik [**SzalasZeszyt2013**].

2.2.2 Semantyka

Semantyka 4QL opiera się logice czterowartościowej. Dodatkowe wartości logicznej są potrzebne w przypadkach, kiedy wiedza jest niepełna lub sprzeczna [8].

4QL ustala porządek wartości logicznych:

$$f < u < i < t$$

Zgodnie z tym porządkiem zdefiniowane są spójniki alternatywy i koniunkcji [8]:

$$a \vee b \stackrel{def}{=} max'(a, b)$$

$$a \wedge b \stackrel{def}{=} min'(a, b)$$

Tabela 1: Matryce logiczne dla spójników w 4QL

$\wedge$	f	u	i	t
f	f	f	f	f
u	f	u	u	u
i	f	u	i	i
t	f	u	i	t

$\vee$	f	u	i	t
f	f	u	i	t
u	u	u	i	t
i	i	i	i	t
t	t	t	t	t

$\rightarrow$	f	u	i	t
f	t	t	t	t
u	t	t	t	t
i	f	f	t	f
t	f	f	t	t

$\neg$	
f	t
u	u
i	i
t	f

Implikacja widoczna na Tabeli 1 wynika z założeń [**SzalasZeszyt2013**]:

- nie można wnioskować z fałszywych i niewiadomych przesłanek
- ze sprzeczności można wywnioskować jedynie sprzeczność
- przesłanki mogą prowadzić do wniosków prawdziwych albo sprzecznych. Ten drugi przypadek ma miejsce, kiedy w bazie wiedzy jest już obecny literał o przeciwnej wartości, niż otrzymana z konkluzji reguły

Istotnym pojęciem w 4QL jest *interpretacja*. Rozumiany jest przez nią zbiór literałów ugruntowanych, które mogą przyjmować wartości z algebry Boole'a : prawdę i fałsz. Logika czterowartościowa występuje dopiero jako ewaluacja literału w kontekście interpretacji. Niech I oznacza interpretację, a l literał o wartości logicznej prawda albo fałsz. Wartość literału l w kontekście I oznaczany jest notacją $I(l)$ i ma następującą definicję:

- $I(l) = T$ jeśli $l \in I \wedge \neg l \notin I$
- $I(l) = I$ jeśli $l \in I \wedge \neg l \in I$
- $I(l) = U$ jeśli $l \notin I \wedge \neg l \notin I$
- $I(l) = F$ jeśli $l \notin I \wedge \neg l \in I$

Poszukiwanym modelem dla programów 4QL nie jest model minimalny, jak w Datalogu, tylko *dobrze wspierany* [**SzalasZeszyt2013**]. Model jest dobrze wspierany, jeśli wartości logiczne wszystkich wygenerowanych literałów da się uzasadnić wnioskowaniem opartym na faktach [8]. W perspektywie ewaluacji programu oznacza to, że nie może mieć miejsca sytuacja, w której fakt jest prawdziwy, a jeden z faktów, który służył do jego dedukcji stał się sprzeczny. Potrzeba wprowadzić korekcje, aż wartości logiczne wszystkie faktów będą logiczną konkluzją reguł. Rozważmy program 4QL, gdzie każdy literał jest ugruntowany :

Wydruk 4: Przykład programu 4QL zaczerpnięty z „Partiality and Inconsistency in Agents' Belief Bases" [8]

```
w :- o | r.
r :- w.
!o :- r.
o.
```

Po wygenerowaniu modelu $\{o, \neg o, w, r\}$ moglibyśmy zakończyć proces obliczeń, ponieważ nasz model jest minimalny - kolejne iteracje wnioskowania w przód nie zmienią modelu. Jednak literał w przestała mieć oparcie w faktach. Trzeba więc dokonać korekty. Właściwym, dobrze wspieranym, modelem jest: $\{o, \neg o, w, \neg w, r, \neg r\}$.

Wydruk 5: Scalanie faktów dla jednej reguły

```
p(c) :- q(X).
q(a).
!q(a).
q(b).

p(c)?
--Result--
p(c): true
```

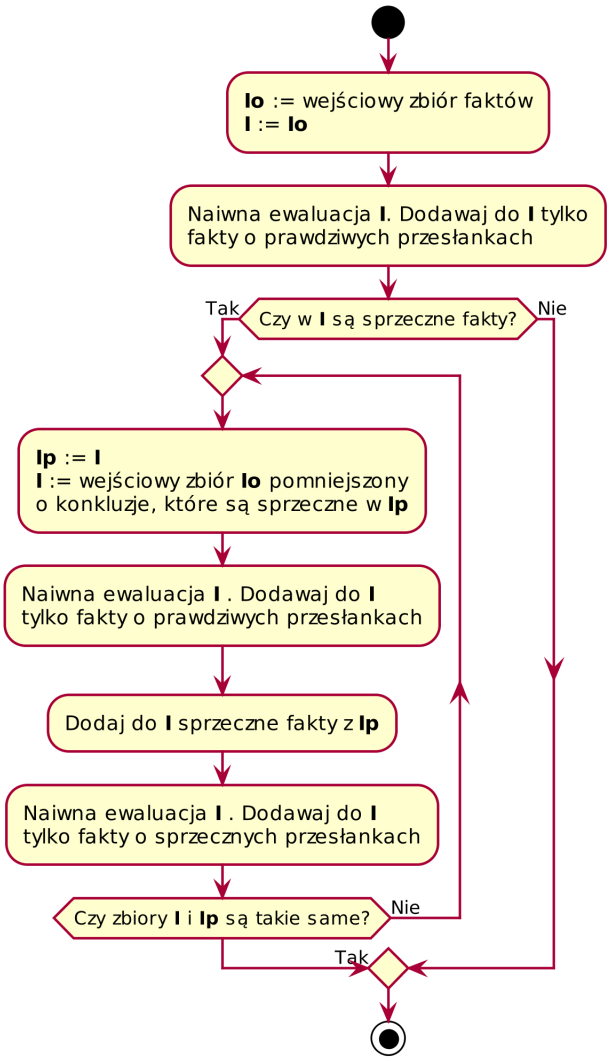
Wydruk 6: Scalanie faktów dla dwóch reguł

```
p(c) :- q(b).
p(c) :- q(a).
q(a).
!q(a).
q(b).

p(c)?
--Result--
p(c): inconsistent
```

Ważną właściwością języka 4QL jest różnica w traktowaniu wyników pojedynczej reguły i wielu reguł o tej samej konkluzji. Jeśli w wyniku ewaluacji tej samej reguły powstaną fakty o tych samych parametrach i odmiennych wartościach logicznych, to fakt, który jest sprzeczny, zostanie zignorowany w wynikach. Natomiast wyniki dwóch reguł, z których jedna produkuje fakt sprzeczny a druga prawdziwy zostaną sprowadzone do faktu sprzecznego. Przykład tej różnicy we wnioskowaniu jest podany w pseudokodzie na wydrukach 5 i 6.

2.2.3 Ewaluacja

Algorytm zdefiniowany w publikacji „Partiality and Inconsistency in Agents' Belief Bases" buduje na naiwnym algorytmie do ewaluacji Dataloga (Rys.2). Pierwszy krok to generacja wszystkich możliwych faktów o prawdziwych przesłankach aż do otrzymania punktu stałego interpretacji. Jeśli po tym kroku w bazie nie ma sprzecznych faktów to algorytm można zakończyć. W przeciwnym wypadku powtarzamy w pętli dwie naiwne ewaluacje aż do otrzymania punktu stałego. Pierwsza ewaluacja również generuje fakty tylko o prawdziwych przesłankach, ale zmniejsza zbiór literałów początkowych o te, których wartość stała się sprzeczna. Po tym kroku do modelu dodawane są sprzeczne fakty z poprzedniej interpretacji i następuje kolejna naiwna ewaluacja, która tym razem dodaje fakty tylko o sprzecznych przesłankach.



Rysunek 2: Algorytm ewaluacji programów 4QL [8]

3 fovris - baza dedukcyjna oparta na języku 4QL

fovris (od *Four-Valued Reasoning System*) to system zaimplementowany w ramach pracy. Składa się z dwóch modułów:

- Silnika bazy danych
- Interpretera

Silnik bazy danych udostępnia proste API do definiowania faktów i reguł z poziomu kodu. W tym module odbywa się wnioskowanie, przetwarzanie zapytań i zarządzanie bazą wiedzy przechowywaną w pamięci fizycznej komputera. Zawiera również parser do skryptów 4QL. Efektem kompilacji tego modułu jest biblioteka, którą można linkować dynamicznie.

Interpreter został napisany w oparciu o silnik bazy danych, który linkuje dynamicznie w czasie uruchamiania aplikacji. Umożliwia wczytywanie skryptów i wystosowywanie zapytań. Można go używać w trybie interaktywnym i nieinteraktywnym.

Ten rozdział opisuje architekturę, założenia projektowe i algorytm silnika oraz funkcjonalności interpretera. Najpierw zaprezentowany będzie interpreter. Jak z niego korzystać, jakie dane wczytuje i zwraca oraz jakich intuicyjnie oczekiwać wyników. Następnie zostanie omówiony silnik bazy i zasady jego działania.

3.1 Wykorzystane technologie

Aplikacja w całości jest napisana w języku C++11. Do budowania aplikacji użyto popularnego systemu *cmake* [4], który potrafi generować skrypty wejściowe dla innych systemów budowania i kompilatorów. Test jednostkowe są uruchamiane za pomocą biblioteki *Catch* [5]. Do testów funkcjonalnych został napisany skrypt w jęzku *python3*. Interfejs programistyczny zapewniany przez standard OpenMP został wykorzystany do zrównoleglenia obliczeń. Z tego powodu wsparcie dla wielowątkowości w fovris jest tylko dostępne za pomocą kompilatorów posiadających implementację OpenMP. Dodatkowo użyte zostały następujące biblioteki i narzędzia:

- re2c - generuje bardzo szybkie tokenizatory [13]. W projekcie generuje kod do tokenizowania skryptów 4ql.

- lemon - narzędzie w do generacji wydajnych parserów. Składnia skryptów do lemon jest bardziej odporna na błędy, niż yacc i bison. Zawiera nieterminalne destruktory, które ułatwiają pisanie parserów bez wycieków pamięci [9]. Generuje kod do parsowania skryptów 4ql i obsługi błędów w składni.

- boost - znany zestaw wydajnych i przetestowanych bibliotek do C++. W projekcie używane są biblioteki nagłówkowe: Boost.Functional/Hash,Boost.MultiIndex i Boost.Optional.

- linenoise-ng - biblioteka zapewniające podstawowe funkcje dostępne w terminalach systemów Unix: historię poleceń i autouzupełnianie [10].

- optionparser - biblioteka służąca do obsługi argumentów lini komend [12].

- date - biblioteka do obsługi dat i czasu, której brakuje w bibliotece standardowej [6].

3.2 Wspierane systemy

Aplikacja wspiera jedynie system Linux, ale prawdopodobnie wspieranie innych systemów typu Unix wymagałoby tylko drobnych zmian. Trudniejsze byłoby dostosowanie fovris do kompilatora Microsoft Visual Studio, ponieważ jego wsparcie dla OpenMP jest niepełne. Dzięki wykorzystaniu systemu do budowania projektów cmake, dodawanie wsparcia dla budowania na innych systemach operacyjnych nie jest problemem. Szczególnie przydatną możliwością jaką posiada cmake jest generacja projektów dla Visual Studio w formacie *sln*.

3.3 Interpreter

3.3.1 Format skryptów 4ql

Interpreter fovris wczytuje skrypty w takim samym formacie, jaki akceptuje Inter4QL 2. Składnia ta w formie notacji Backusa-Naura dostępna jest online w dokumencie opracowanym przez Łukasza Białka [2]. Poniżej zaprezentowano schemat struktury typowego programu:

Wydruk 7: Struktura bazowego skryptu

```
module <nazwa_modulu>:
   domains:
     <typ_domeny> <alias_dla_domeny>.
     ...
   relations:
     <nazwa_relacji> <typ_domeny>.
     ...
   rules:
     <relacja>(<param1>,<param2>,...) :- <relacja>(<param>,...) ,
<relacja>(<param>,...), ... .
   facts:
     <relacja>(<param>,...).
     ...
   end.

...
```

W wydruku 7 dane uzupełniane przez użytkownika między trójkątnymi klamrami $< >$. Trzy kropki po elemencie oznaczają, że dany element może pojawić się więcej, niż jeden raz. Ten schemat nie wyczerpuje składni akceptowanych skryptów, ale jest widoczne podobieństwo ze skryptami Datalog.

Jak widać na wydruku 7, najwyższą jednostką organizacyjną w skryptach są moduły, które mogą być zdefiniowane wielokrotnie w jednym programie. Każdy moduł musi mieć unikalną nazwę. W ciele modułu mogą figurować 4 sekcje lub też żadna z nich.

Sekcja `domains`, pozwala na zdefiniowanie aliasów domen wartości, które nie pełnią żadnej roli podczas ewaluacji skryptu. Poprawiają jedynie czytelność kodu.

Sekcja `relations` jest obowiązkowa, jeśli w module mają być zdefiniowane reguły lub fakty. W tej sekcji definiowane są wszystkie relacje, których elementy mogą figurować w sekcji `facts` oraz jedynie te relacje mogą być w konkluzjach reguł. W fovris, podobnie jak w Inter4QL, parametry w relacji mogą należeć do jednego z 8 typów danych:

- `integer` - liczba całkowita
- `real` - liczba rzeczywista
- `logic` - wartość logiczna, jedna z czterech: unknown,inconsistent,true,false
- `string` - zmienna tekstowa, tekst otoczony cudzysłowami
- `literal` - identyfikator, który ma takie same ograniczenia jak zmienne w C++ i może zaczynać się tylko od małej litery
- `date` - data w formacie ISO 8601 - YYYY-MM-DD
- `datetime` - data i czas w formacie ISO 8601 - YYYY-MM-DDTHH:MM:SS

Sekcja `rules` zawiera listę reguł składających się z konkluzji (głowa reguły) i przesłanek (ciało reguły). Reguły mają identyczną składnie jak opisana w rozdziale 2.2.

Sekcja `facts` to lista faktów, których wartość logiczna jest znana przed uruchomieniem programu. Fakt może być:

- Prawdziwy: `p(a).`
- Fałszywy: `!p(a).`
- Sprzeczny:

```
p(a).
!p(a).
```

Interpreter fovris wczytuje skrypty akceptowane przez Inter4QL 2 (z pewnymi wyjątkami) oraz rozszerza składnie o nowe elementy.

Wydruk 8: Przykładowy program prezentujący różnice w składni

```
module a:
    relations:
        p(literal).
```

```
    facts:
        p(e).
end.

module b:
    relations:
        p(literal).
        r(literal).
    rules:
        r(X) :- p(X), X != f, a.p(X) = unknown. // /1
    facts:
        p(e).
        p(w).
end.

b.r(X)? // zapytanie w kodzie programu /2
```

Wydruk 8 pokazuje przykładowy program wczytywany przez interpreter fovris. Linijki z komentarzami zawierają dodatkowe elementy składni wprowadzone przez fovris. Linijka z numerem jeden pokazuje uproszczoną składnie dla literałów zewnętrznych oraz jeden z wbudowanych operatorów binarnych. Zamiast za każdym razem definiować jednoelementowy podzbiór wartości logicznych, można się posłużyć operatorem równości = lub nierówności ! =, po którego prawej stronie stoi jedna z wartości logicznych. Natomiast operatory binarne porównujące zmienne to standardowe rozszerzenie Dataloga. Dostępne są następujące operatory: =, ! =, <, >, <=, >=.

3.3.2 Ograniczenia w definicji reguł

fovris pozwala na definiowanie tylko *bezpiecznych* reguł. Zasady definiowania są podobne jak w Datalogu.

- Każdy zmienna w konkluzji reguły musi pojawić się w literale produkującym skończoną relację w ciele reguły

  Niedozwolona reguła: `p(X,Y) :- r(X).`

  Dozwolona reguła: `p(X,Y) :- r(X), q(Y).`

- Każdy zmienna w literale w przesłankach reguły, z którego można otrzymać nieskończoną relację, musi się pojawić w literale produkującym skończoną relację

Niedozwolone reguły:

```
p(X,Y) :- X>Y.
```

```
p(X,Y) :- m.r(X,Y) = unknown.
```

Dozwolone reguły:

```
p(X,Y) :- X>Y, r(X), q(Y).
```

```
p(X,Y) :- m.r(X,Y) = unknown, w(X,Y).
```

Te ograniczenia są ważne, gdyż sprawiają, że wynik ewaluacji reguły jest skończony i niezależny od dziedziny parametrów literału.

Dodatkowym ograniczeniem narzucanym przez sam język jest zakaz odwoływania się do kolejnych modułów (wg kolejności definiowania) przez poprzednie moduły. Jest to spowodowane warstwową architekturą języka [8].

3.3.3 Tryb nieinteraktywny

Program z wydruku 8 można wczytać poleceniem:

```
fovris -i script.4ql
```

Po zakończeniu obliczeń na ekranie pojawi się wynik zapytania podanego na końcu skryptu:

```
#b.r(X)                 // zapytanie do bazy
r(w)            : true  // wyniki zapytania
```

W wynikach po lewej stronie pojawią się ugruntowane literały, a po prawej ich wartość logiczna, jaka wynikła z procesu wnioskowania.

3.3.4 Tryb interaktywny

Tryb interaktywny można rozpoznać po pojawieniu się znaku zachęty:

```
fovris>
```

Jeśli wydruk 8 nie miał na końcu podanego zapytania, to interpreter przeszedłby w tryb interaktywny i uruchomił wbudowany terminal. Terminal zapamiętuję historię wpisywanych poleceń, które można przeszukiwać standardowo strzałkami góra, dół i za pomocą inkrementalnego przeszukiwania uruchamianego Ctrl-R. Obsługiwane jest także autouzupełnianie poleceń i zapytań o zdefiniowane moduły i relację klawiszem TAB.

Wydruk 9: Wynik komendy help

```
fovris> help
Available commands:
   <query>
      where <query> has structure module.predicate(term,...)
   import  </path/to/program.4ql>
      import and evaluate 4ql program
   help
      print this
   clear
      clear modules from memory
   modules
      print a list of modules
   exit
      exit the program
fovris>
```

Na wydruku 9 widnieją dostępne w trybie interaktywnym polecenia. Jest ich niewiele i bliźniaczo przypominają te w Inter4QL tylko bez wymagań na kropkę na końcu polecenia i cudzysłowy.

3.3.5 Inne opcje

Interpreter pozwala na wybranie różnych algorytmów ewaluacji. Dostępne są dwa: naiwny i semi-nawiny; opisane poniżej. Można też zdefiniować dodatkowe zapytanie flagą `--query`, które zapobiega wejściu w tryb interaktywny. Wyniki wyświetlane są domyślnie w takim samym formacie jak w Inter4QL, ale dostępne są też formaty `csv` i `json`. Specjalna flaga `--threads` służy do ograniczenia liczby wątków używanych do obliczeń, ale nie ma ona efektu, jeśli kompilator nie wspiera interfejsu OpenMP.

3.4 Parser

Wydruk 10: Przykład użycia parsera

```
#include "QlDeserializer.h"

int main() {
    fovris::QlDeserializer sp;
    sp.parse("module p:\
                    relations:\
                        parent(literal,literal).\
                        ancestor(literal,literal).\
                    rules:\
                        ancestor(X,Y):- ancestor(X,Z),parent(Z,Y).\
                        ancestor(X,Y):- parent(X,Y).\
                    facts:\
                        parent(alice,bob).\
                        parent(alice,bill).\
                        parent(bob,carol).\
                        parent(carol,dennis).\
                        parent(carol,david).\
                end.");
    std::vector<fovris::Module> modules = sp.getModules();
}
```

Klasa `QlDeserializer` umożliwia parsowanie skryptów *4ql*. Akceptuje standardowe zmienne tekstowe typu `std::string` zawierające skrypt. W przypadku wystąpienia błędu w składni wyrzuca wyjątek `ParsingError`, z którego można odczytać pozycję,w której wystąpił błąd(numer linii, numer znaku) oraz komunikat błędu. Standardowe użycie z obsługą błędów w aplikacji użytkownika wygląda następująco:

Wydruk 11: Przykład użycia parsera z obsługą błędów

```
fovris::QlDeserializer sp;
try{
    sp.parse(readScript());
} catch (fovris::ParsingError &e) {
    std::cerr << "error:" << e.getLine() << ':' << e.getPos() << ": "
            << e.what() << std::endl;
    return 1;
}
std::vector<fovris::Module> modules = sp.getModules();
std::vector<fovris::Query> queries = sp.getQueries();
```

Po udanym parsowaniu dostępne są obiekty API wymagane przez silnik bazy.

W skryptach mogą wystąpić dwa główne konstrukty: zapytania i moduły. Mapują się one logicznie do klas typu `fovris::Query` i `fovris::Module`.

3.5 Interfejs programistyczny biblioteki

fovris zapewnia zestaw klas przechowujących dane i umożliwiających pracę z silnikiem bazy.

Moduł

Wydruk 12: Dane składowe klasy Module

```
class Module {
    std::string name;
    std::vector<RelationDef> relations;
    std::vector<Rule> rules;
    std::vector<GroundLiteral> facts;
};
```

Klasa `fovris::Module` logicznie odpowiada modułom w skryptach. Zawiera więc listę relacji, reguł i faktów. Domeny zostały pominięte, bo nie wpływają na wynik obliczeń.

Relacja

Wydruk 13: Dane składowe klasy RelationDef

```
class RelationDef {
   std::string predicateSymbol;
   std::vector<TermType> types;
};
```

Klasa `fovris::RelationDef` przechowuje nazwę relacji i listę typów domen parametrów tej relacji.

Reguła

W programach 4QL reguły składaja się z konkluzji i przesłanek. O ile konkluzja bezpośrednio odpowiada składowej `head`, o tyle w przypadku przesłanek zdecydowano się wyodrębnić dodatkową klasę. Klasa ta, `RuleDisjunct`, odpowiada formułom zawierającym tylko operatory koniunkcji. Koniunkcja ma wyższy priorytet, niż alternatywa,

dlatego jest ewaluowana pierwsza.

Wydruk 14: Dane składowe klasy RelationDef

```
class Rule {
    Literal head;
    std::vector<RuleDisjunct> body;
};
```

Klasa `RuleDisjunct` zawiera już tylko listę literałów połączonych spójnikiem koniunkcji. Będą one później przetwarzane po kolei w trakcie ewaluacji reguły.

Wydruk 15: Dane składowe klasy RuleDisjunct

```
class RuleDisjunct {
    std::vector<RuleLiteral> literals;
};
```

Literał

Klasa `Literal` przechowuje podstawowe informacje o literałach, które występują w konkluzjach reguł i sekcji faktów. Niezbędna jest zatem wiedza o tym, czy literał jest zanegowany, jaki jest symbol jego relacji i lista termów, które są jego argumentami.

Wydruk 16: Dane składowe klasy Literal

```
class Literal {
    bool isTrue_;
    std::string predicateSymbol;
    std::vector<Term> terms;
};
```

Pomocnicza klasa `GroundLiteral` odpowiada ugruntowanym literałom, czyli faktom. Od `Literal` różni się jedynie sprawdzaniem czy żaden z termów nie jest zmienną.

Literał w regule

`RuleLiteral` rozszerza `Literal` o możliwość przechowywania nazwy zewnętrznego modułu, predefiniowanej funkcji i ograniczenie na zbiór wartości logicznych. Dodatkowe składowe klasy odpowiadają różnym typom literałów, jakie mogą występować w regule:

- Zwykły literał. np.: `p(X,2)`
- Zewnętrzny literał. np.: `m.p(X,2)`
- Zewnętrzny literał z ograniczeniem zbioru wartości logicznych. np.: `m.p(X,2) in {true,false}`
- Wbudowany predykat albo funkcja użytkownika. np.: `X>2`

Wydruk 17: Dane składowe klasy RuleLiteral

```
class RuleLiteral : public Literal {
   PredicateFunction predicateFunction;
   bool isBuiltinPredicateFunction = false;
   boost::optional<std::string> externalModule;
   TruthValueSet logicConstraint;
};
```

Taki projekt klasy (Wydruk 17) jest łatwy w użyciu, ale trudny w utrzymaniu i rozwijaniu. Zakłada on wykorzystanie jako kilka różnych typów w zależności od tego czy członki klasy zostaną zainicjalizowane. Np. `predicateFunction` nie wskazujący na żadną funkcję oznacza, że klasa nie prezentuje predykatu funkcyjnego. Jeśli aplikacja miałaby być ulepszana, w miejsce tej klasa powinna być wprowadzona hierarchia klas.

Term

Klasa `Term` widoczna na Wydruku 18 odpowiada termom, które są argumentami literałów. Może być albo zmienną albo stałą. Zaimplementowany jest jako "otagowana unia", czyli klasa z unią i tagiem, który określa wartość w unii.

Wydruk 18: Dane składowe klasy Term

```
class Term {
  protected:
    union Val_ {
        double d_;
        int i_;
        TruthValue t_;
        std::string *s_;
        DateTime dateTime_;
        Date date_;
    } val_; // unia
```

```
    TermType type; // tag
};
```

Tabela 2: Zestawienie typów wartości w skrypcie i API

[!h] Skrypt	TermType	Typ Danych
integer	Integer	int
real	Real	double
literal	Id	std::string
string	String	std::string
zmienna	Var	std::string
datetime	DateTime	DateTime
date	Date	Date

Tabela 2 przedstawia jak mapuje się typ termu w skrypcie do klas API. Wszystkie typy poza `Id`,`String` i `Var` przechowywane są w różnych typach danych. Z tego powodu wyrażenie `Term{"X"}` nie pozwala stwierdzić czy term jest typu Var czy String. Wymusza to jednoznacznie definiowanie przez użytkownika termów o tych typach danych.

Wydruk 19: Błędna definicja reguły

```
Rule rule = {{"ancestor", {"X", "Y"}}, {{"parent", {"X", "Y"}}}};
```

Reguła w Wydruku 19 nie może być więc zdefiniowana za pomocą wspieranych domniemanych konstruktorów, bo nigdzie nie są zawarte informacje czy termy `X` i `Y` są zmiennymi czy stałymi tekstowymi. **Domyślnie** program przyjmuje, że są to zmienne tekstowe. Poprawnie zdefiniowana reguła wygląda następująco:

Wydruk 20: Poprawna definicja reguły

```
Rule rule = {{"ancestor", {Var{"X"}, Var{"Y"}}}, {{"parent", {Var{"X"},
   Var{"Y"}}}}};
```

`Var` to klasa, która jest wprowadzona dla wygody i jest tożsama z definicją `Term{"nazwa_zmiennej",TermType::Var}`.

Inaczej sytuacji wygląda dla faktów modułu, gdzie zmienne nie mogą być zdefiniowane. fovris zakłada, że typy parametrów faktu odpowiadają tym zdefiniowanym

w definicji relacji, do której fakt należy. Niejako sztuczny podział na typy `literal` i `string` traci sens w kontekście API biblioteki. Dla przykładu, jeśli definicja relacji `r` wygląda następująco `r(literal)`, to akceptowany będzie literał `r(a)`, jak i niepoprawny w kontekście skryptu `r("Abc")`.

3.5.1 Predykaty funkcyjne

Wcześniej omówiona klasa `RuleLiteral` może zawierać `PredicateFunction`, która reprezentuje predykat funkcyjny (predykat oznaczający funkcję a nie relację).

Wydruk 21: Definicja PredicateFunction

```
typedef std::function<bool(const Array<Term> &)> PredicateFunction;
```

Wydruk 21 pokazuje definicję typu `PredicateFunction`. Jest to instancja szablonu klasy `std::function`, która zwraca wartość `bool` a jako argument bierze tablicę obiektów `Term`. `std::function` jest dostępne wraz ze standardem C++11 i może przechowywać funkcje dowolnych typów: wskaźniki do funkcji, funktory, wyrażenia lambda. Minusem tego rozwiązania jest brak możliwości sprawdzenia zgodności typów w trakcie kompilacji. Od użytkownika zależy czy upewni się, że do funkcji jest przekazywana odpowiednia ilość termów o odpowiednich typach.

Wydruk 22: Przykład użycia PredicateFunction

```
PredicateFunction pred = [](const Array<Term>&terms){
    return terms[0].get<int>() > 10;
};
```

`PredicateFunction` pozwala na elastyczne tworzenie funkcji przez użytkownika za pomocą wyrażeń lambda. W wydruku 22 term musi być większy od 10, żeby fakt go zawierający został wybrany podczas przetwarzania reguły.

3.5.2 Interfejs do silnika bazy

Klasa `Program` stanowi punkt dostępu do silnika bazy. Za pomocą obiektów tej klasy można dodawać nowe moduły, ewaluować obecne i tworzyć zapytania. Jest zaimplementowana z użyciem wzorca *pimpl* (z ang. *Pointer To Implementation*) w

celu ukrycia detalów implementacji przed użytkownikiem i skrócenia czasu kompilacji.

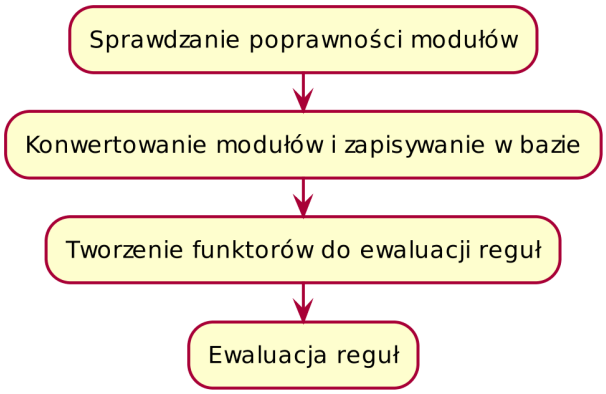
Wydruk 23: Deklaracja klasy Program

```
enum Algorithm { Naive, Seminaive };

class Program {
    class ProgramImpl;
    std::unique_ptr<ProgramImpl> impl;

  public:
    Program(Algorithm algorithm);
    void addModule(const Module &module);
    void addModules(const std::vector<Module> &modules);

    std::vector<std::string> listModules() const;
    std::vector<RelationDef> listRelations(const std::string &module)
   const;

    void evaluate();

    void clear();

    QueryResult query(const std::string &module, const std::string
   &predicate, const std::vector<Term> &terms) const;
};
```

Wydruk 39 w Dodatku A przedstawia implementację programu z Wydruku 11, gdzie był widoczny w formie skryptu. Najpierw definiowany jest cały moduł `m` z sekcjami odpowiednio od góry: definicje relacji, reguły, fakty. Następnie tworzony jest obiekt klasy `Program` zapewniający dostęp do bazy z wybranym algorytmem naiwnym do ewaluacji. Moduł jest ewaluowany a do bazy wystosowane zapytanie o relację `ancestor`. Na końcu wyniki wypisywane są na ekranie.

3.6 Architektura silnika bazy

Rola silnika bazy zaczyna się kiedy pierwszy raz dodawane są moduły metodą `addModule`. Moduły są sprawdzane i zamieniane na wewnątrzną reprezentację w bazie. W kolejnym kroku po wywołaniu metody `Program::evaluate()` następuje ewaluacja wprowadzonych modułów aż do obliczenia punktu stałego modelu. Ewaluację i



Rysunek 3: Schemat działania silnika

wprowadzanie modułów można podzielić na kilka faz wyszczególnionych na rysunku 3.

3.6.1 Sprawdzanie poprawności modułów

To pierwszy etap podczas, którego sprawdzana jest poprawność definicji reguł. Sprawdzane są:

- Zgodność typów stałych z deklaracjami relacji
- Krotność argumentów literałów z deklaracjami relacji
- Czy reguły są bezpieczne

Jeśli któryś z warunków zawiedzie, wyrzucany jest wyjątek `InvalidInputException` z opisem miejsca w skrypcie, które spowodowało problem.

3.6.2 Przetwarzanie i przechowywanie faktów

Podczas przetwarzania modułów obiekty klas przedstawionych w poprzednich akapitach zamieniane są na bardziej kompaktową reprezentację.

Wydruk 24: Dane składowe klas KBTerm i KBGroundTerm

```
class KBTerm {
    unsigned val_;
    bool isVar_;
};
```

```
class KBGroundTerm {
    unsigned val_;
};
```

Wszystkie stałe w literałach są zapamiętywane w specjalnej strukturze danych i zastępowane wartością typu `unsigned`, która pozwala na wyciągnięcie stałej z tej struktury. Klasa `Term` jest mapowana do `KBTerm` dla ogólnego przypadku lub do `KBGroundTerm` dla termów występujących w faktach.

Wydruk 25: Deklaracja klasy TermMapper

```
class TermMapper {
    DataPool<double> dblTermPool_;
    DataPool<Date> dateTermPool_;
    DataPool<DateTime> dateTimeTermPool_;
    DataPool<std::string> strTermPool_;

  public:
    unsigned internTerm(const Term &term);

    unsigned queryTermId(const Term &term) const throw(std::out_of_range);

    Term queryTerm(unsigned id, TermType type) const
   throw(std::out_of_range);
};
```

Za mapowanie indeksów termów do ich wartości odpowiada klasa `TermMapper`. Widocznie w jej ciele są 4 odrębne obiekty `DataPool`, które mapują wartości do indeksów typu `unsigned` i na odwrót. Zaimplementowana w ramach pracy klasa `DataPool` zapewnia podwójne indeksowanie tak samo jak `FactsPool` i nie została zastąpiona przez instancję szablonu z biblioteki Boost.MultiIndex jedynie z powodu ograniczeń czasowych. Nie jest to problemem, bo `DataPool` jest równie szybka jak `FactsPool`, a jedynie trochę mniej wydajna w kontekście złożoności pamięciowej.

Metoda `TermMapper::internTerm` odpowiada za "internalizację" termów, czyli dodawaniem termów do indeksowanych pul danych i zastępowanie ich przypisanymi indeksami.

Wydruk 26: Dane składowe klasy KBTuple

```
class KBTuple {
    Array<KBGroundTerm> terms;
```

```
    TruthValue value;
};
```

Pojedynczy fakt w bazie wiedzy reprezentowany jest przez klasę `KBTuple` (Wydruk 26). Klasa ta zawiera dynamicznie zaalokowaną tablicę termów i wartość logiczną. Rozwiązanie takie powoduje niestety wolniejsze działanie programu poprzez koszt czasowy alokacji i narzutu pamięci spowodowane dynamiczną alokacją wielu małych obiektów.

Wydruk 27: Struktura przechowująca fakty

```
using FactsPool = boost::multi_index::multi_index_container<
    KBTuple,
    boost::multi_index::indexed_by<
        boost::multi_index::random_access<>, // 1
        boost::multi_index::hashed_unique<  //2
            boost::multi_index::const_mem_fun<
               KBTuple, const Array<KBGroundTerm> &,
   &KBTuple::getTerms>>>>;
```

Fakty należące do relacji trzymane są w podwójnie indeksowanej strukturze danych zapewnianej przez bibliotekę Boost.MultiIndex. Na wydruku 27 pokazana jest definicja tej struktury. Indeks losowego dostępu (linijka z numerem 1) z biblioteki Boost.MultiIndex jest niezbędny, żeby można było łatwo dzielić fakty w bazie na zakresy, które później delegowane są do oddzielnych wątków. Udostępnia on operator dostępu `[]` indeksowany wartościami całkowitoliczbowymi oraz nie powoduje realokacji pamięci kiedy zwiększa swoją maksymalną pojemność [3]. Oznacza to, że wskaźniki i iteratory wskazujące na elementy kontenera z tym indeksem nigdy nie zostaną unieważnione, jak to ma miejsce w przypadku klasy `std::vector`. Drugi indeks potrzebny jest do uzyskania unikalności danych i sprecyzowania jaki element indeksowanej klasy ma być unikalny (linijka z numerem 2).

Wydruk 28: Dane składowe klasy KBRelation i metoda addTuple

```
class KBRelation {
  private:
    FactsPool facts;
    unsigned inconsFactsNumber_ = 0;

  public:
```

```
    KBRelation(unsigned id, std::vector<TermType> colDomains);

    const unsigned id;
    const std::vector<TermType> colDomains;
};
```

Klasa reprezentujące relacje, `KBRelation`, zawiera podany wyżej kontener faktów, unikalny numer id generowany na podstawie jej nazwy i modułu, definicje typów domen oraz liczbę sprzecznych faktów. Ta ostatnia wartość pozwala dowiedzieć się czy dodanie faktu do relacji spowodowało jakąś zmianę w instancji bazy.

Relacje nie są pogrupowane w moduły, jako można by przypuszczać, tylko są bezpośrednio identyfikowane globalnie unikalnym numerem `id`, który służy do indeksowania kontenera, `std::deque<KBRelation>`, zawierającego wszystkie relacje w programie. Typ `std::deque` użyty jest tu ze względu na to, żeby nie następowała kosztowna realokacja pamięci i unieważnianie wskaźników do relacji podczas zwiększania pojemności kontenera.

3.6.3 Przetwarzanie reguł

Odwrotnie sytuacja wygląda dla reguł i literałów w regułach. Reguł w typowym programie 4QL czy Datalog jest znacznie mniej niż faktów, a potrzebna jest dokładna wiedza na temat ich zawartości. Literały w wewnętrznej reprezentacji zawierają dużo metadanych na swój temat przygotowanych jeszcze przed ewaluacją. Ich stosunkowo mała ilość powoduje, że narzut pamięci jest pomijalny. Obliczany i przechowywany jest zbiór zmiennych w literale(`varSet`) oraz indeksy pod jakimi występują poszczególne zmienne (`varIndices`). Redundancja w informacji trzymanej w obu tych obiektach jest dopuszczalna z powodów wymienionych powyżej.

Wydruk 29: Dane składowe klasy KBLiteral

```
    unsigned relId;
    std::map<KBTerm, std::vector<unsigned>> varIndices;
    std::vector<KBTerm> varSet;
    std::vector<KBTerm> terms;
```

Klasy dziedziczące z `KBLiteral` to `KBHeadLiteral` odpowiadające głowie reguły,

`KBRuleLiteral` odpowiadający literałowi w regule i `KBRuleFunction` odpowiadający predykatowi funkcyjnemu.

Wydruk 30: Dane składowe klas dziedziczących z KBLiteral

```
class KBRuleFunction : public KBLiteral {
    PredicateFunction predicateFunction;
    std::vector<TermType> termTypes;
};

class KBRuleLiteral : public KBLiteral {
    TruthValueSet logicSet;
};

class KBHeadLiteral : public KBLiteral {
    bool negated = false;
};
```

`KBRuleLiteral` nie posiada członka `negated`, gdyż negacja literału jest tłumaczona jako dodanie wartości `TruthValue::False` do obiektu klasy `logicSet`, a negacja zbioru wartości jest traktowana jako jego komplement i też dołączana do `logicSet`. Dla przykładu:

`!m.a(X) in {false}` jest konwertowane do `m.a(X) in {true,incons,unknown}`.

Wydruk 31: Dane składowe klas KBRule i KBRuleDisjunct

```
class KBRule {
    KBHeadLiteral head;
    std::vector<KBRuleDisjunct> disjuncts;
};
class KBRuleDisjunct {
    std::vector<KBRuleLiteral> body;
    std::vector<KBRuleFunction> functions;
};
```

Deklaracje klas na Wydruku 31 odpowiadają przedtem wspomnianym klasom API: `Rule` i `RuleDisjunct`. Części reguł zawierające tylko operatory koniunkcji (`KBRuleDisjunct`) zostały podzielone na dwie części: literałów z predykatami relacyjnymi i literałów z predykatami funkcyjnymi. Literały relacyjne są sortowane w konstruktorze klasy tak, żeby bezpieczne literały były z przodu a te produkujące potencjalnie nieograniczone wyniki z tyłu. Na przykład reguła:

`w(X,Z) :- m.a(X,Y) in {unknown}, b(Y), c(X), d(Z).`

jest sortowana, żeby wyglądać następująco:

`w(X) :- b(Y), c(X), m.a(X,Y) in {unknown}, d(Z).`

Jest to niezbędne, ponieważ literały są ewaluowane od lewej do prawej, a w celu uzyskania różnicy trzeba najpierw obliczyć lewy ze zbiorów będący złączeniem relacji `b(Y)` i `c(X)`:

$\Pi_{col_2,col_1}((b(Y) \bowtie c(X))) - m.a(X,Y)$

3.6.4 Przechowywanie modułów

Klasa `KBModule` (Wydruk 32) potrzebna jest do logicznego pogrupowania relacji i reguł, które są zdefiniowane w danym module. Dzięki temu można operować na pojedynczych modułach. Członek `id` indeksuje moduły w kolejności w jakiej zostały dodane. Członek `relationIds` został dodany dla wygody.

Wydruk 32: Dane składowe klasy KBModule

```
struct KBModule {
    const std::string name;
    unsigned id;
    std::vector<KBRule> rules;
    std::vector<unsigned> relationIds;
    std::vector<std::reference_wrapper<KBRelation>> relations;
};
```

3.6.5 Konwersja reguł do funktorów

Tworzenie funktorów do ewaluacji reguł odbywa się po wywołaniu `Program::evaluate` na poziomie pojedynczych algorytmów. Wydruk 33 przedstawia deklarację funktora służącego do ewaluacji reguły, czyli generacji nowych faktów na podstawie reguły i wejściowych relacji. Funktory przyjmują tylko jeden argument, wektor referencji do kontenerów z faktami. Zwracają nowo wygenerowane fakty, które **mogą** zawierać duplikaty faktów. Eliminacja duplikatów odbywa się dopiero przy dodawaniu faktów do relacji albo podczas agregacji wyników. Do stworzenia instancji takiego funktora

Wydruk 33: Nagłowek pliku Eval/EvalRule.h

```
class EvalRule {
  public:
    EvalRule() {}
    virtual std::vector<KBTuple> operator()(
        std::vector<std::reference_wrapper<const FactsPool>> relations) =
   0;
    virtual const KBRule& getRule() const = 0;
    virtual ~EvalRule() {}
};

std::unique_ptr<EvalRule> CreateEvalRule(KBRule rule,const TermMapper&
   termMapper={});
```

Wydruk 34: Dane składowe klasy EvalRuleWithoutDisj

```
class EvaluateRuleWithoutDisjunction : public EvalRule {
    std::vector<EvalSubgoalPtr> subgoalEvals;
    KBRuleDisjunct ruleDisjunct_;
    KBHeadLiteral head_;
    KBRule rule_;
    KBRuleLiteral resultRelation_;
    std::reference_wrapper<const TermMapper> termMapper_;
};
```

używana jest funkcja `CreateEvalRule`.

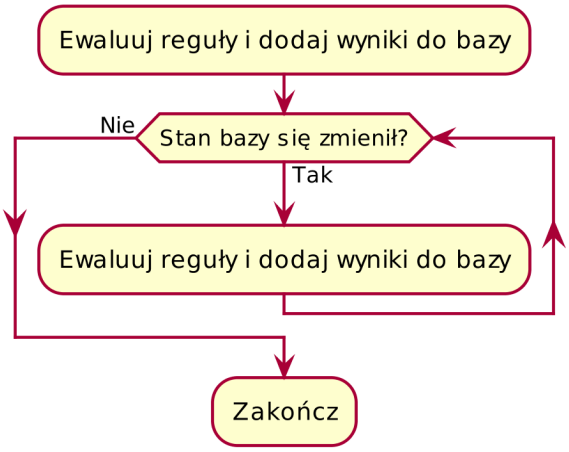
Fragmenty reguł bez spójnika alternatywy są ewaluowane w pierwszej kolejności przez funktor `EvalRuleWithoutDisj` (Wydruk 34). Przechowuje on funktory do ewaluacji pojedynczych literałów ze swojego fragmentu, główę reguły (bądź pożądany wynik ewaluacji) w członku `head`, regułę (`rule_`) , sygnaturę zwracanej relacji (`resultRelation_`) oraz obiekt do mapowania indeksów termów do ich wartości `TermMapper`.

Klasa do ewaluacji reguł z alternatywą, `EvalRuleWithDisj` (Wydruk 35), oblicza wynik na dwóch poziomach. Traktuje fragmenty reguły bez spójnika alternatywy jako pojedyncze reguły i trzyma funktory do ich ewaluacji w wektorze `disjunctEvaluations`. Jednocześnie do ewaluacji alternatywy wyników otrzymanych z tych fragmentów służą funktory w wektorze `disjunctResultEvals`.

Wydruk 35: Dane składowe klasy EvalRuleWithDisj

```
class EvalRuleWithDisj : public EvalRule {
    KBRule rule_;
    std::vector<EvaluateRuleWithoutDisjunction> disjunctEvaluations;
    std::vector<EvalSubgoalPtr> disjunctResultEvals;
    std::reference_wrapper<const TermMapper> termMapper_;
};
```

3.6.6 Generacja faktów



Rysunek 4: Obliczanie nowych faktów - generalnych schemat

Reguły ewaluowane są przy pomocy funktorów reguł do czasu aż zostanie osiągnięty punkt stały instancji bazy, czyli stan bazy przestanie się zmieniać. To znaczy, że kolejne iteracje nie generują już nowych faktów ani nie zmieniają wartości logicznej obecnych.

3.7 Tryby ewaluacji

W ramach pracy zaimplementowano dwa tryby ewaluacji. Oba tryby bazują na algorytmie z pracy „Partiality and Inconsistency in Agents' Belief Bases" [8] opisanym w sekcji 2.2.3. Algorytm naiwny nie różni się od tego opisanego w publikacji. Natomiast algorytm semi-naiwny korzysta z semi-naiwnego algorytmu do ewaluacji Dataloga, kiedy tylko jest to możliwe. Wykorzystany jest więc w pierwszej pętli ewaluacji, która generuje tylko fakty na podstawie prawdziwych przesłanek. Jeśli istnieją

sprzeczne fakty rozpoczyna się pętla kolejno dwóch ewaluacji. Pierwsza z nich również generuje fakty na podstawie prawdziwych przesłanek, więc może być tam wykorzystany algorytm semi-naiwny. Natomiast druga dodaje tylko wyniki, które są sprzeczne. Jako że wyniki z jednej reguły są sumowane względem wartości logicznych to muszą być uwzględniane zawsze wszystkie fakty z relacji w obliczaniu sumy logicznej. Jeśli pominiemy jakiś fakt, to wynikowy literał może się zmienić na sprzeczny. Podobnie zachowuje się spójnik alternatywa w regułach. Tu też zawsze muszą być uwzględniane wszystkie fakty i nie można użyć relacji cząstkowych.

3.8 Reguły 4QL jako równania algebry relacyjnej

Jak wspomniano w rozdziale 2.1, programy Datalog można przetłumaczyć na zapytania w języku algebry relacyjnej. W pracy również zastosowano takie podejście do ewaluacji reguł 4QL. Nie można po prostu przetłumaczyć reguł 4QL na operacje algebry relacyjnej tak samo jak Datalog, ponieważ krotki w relacjach zawierają poza parametrami również dodatkowy atrybut określający wartość logiczną: prawda, fałsz albo sprzeczność.

Uwaga, notacja $operator_{col_i}(relacja)$ jest używana, żeby określić i-tą kolumnę relację w nawiasach, na której ma być przeprowadzana operacja.

W zobrazowaniu tej różnicy pomoże prosty przykład. W bazie zdefiniowana jest relacja Osoba(Imię,Nazwisko,Wiek,Miasto) oraz relacja Mieszka(Imię,Nazwisko,Miasto). Stan relacji *Osoba* prezentuje Tabela 3.

Tabela 3: Zawartość relacji *Osoba*

Imię	**Nazwisko**	**Wiek**	**Miasto**	**Wartość**
Jan	Kowalski	30	Warszawa	true
Alicja	Marzec	31	Kraków	false

Niech w przetwarzanym programie znajduje się reguła:

```
Mieszka(X,Z) :- Osoba(X,Y,Z,M).
```

Regułę można przetłumaczyć na równanie algebry relacyjnej:

$\Pi_{\text{Imię,Miasto}}(Osoba)$,

które zwróci relację zawierającą

Tabela 4: Zawartość relacji *Mieszka*

Imię	**Miasto**
Jan	Warszawa
Alicja	Kraków

Wynik w Tabeli 4 jest poprawny dla programu napisanego w języku Datalog. 4QL wprowadza założenie otwartego świata i uwzględnia również wartość logiczną faktów. W szczególności nie pozwala na wnioskowanie na podstawie nieznanych i fałszywych przesłanek [8]. Regułę powyżej można przetłumaczyć na równanie w algebrze relacyjnej, które zwróci poprawny wynik dla 4QL:

$\Pi_{\text{Imię,Miasto}}(\sigma_{\text{Wartość}\neq'\text{false}'}(Osoba))$

Podobnie można przetłumaczyć literały zewnętrzne nie ograniczone do wartości *unknown*. Na przykład regułę:

```
r(X) :- m.a(X,Y) in {true,false}
```

można zamienić na:

$\Pi_{col_1}(\sigma_{\text{Wartość}\in\{true,false\}}(m.a))$

Literały zawierające stałe i nakładające ograniczenia na zmienne tłumaczone są tak samo jak dla Dataloga:

`q(X) :- r(X,X,'foo')` -> $\Pi_X(\sigma_{\text{Wartość}\neq false \wedge col_1=col_2 \wedge col_3='foo'}(r))$

Złączenia w regułach niemających alternatyw również można prosto przetłumaczyć (zakładamy, że relacje zostały już poddane selekcji):

```
r(X) :- a(X,Y), b(Y,Z).
```

na

$\Pi_{col_1}(a \bowtie b)$

Uwaga, koniunkcja literałów w regule oznacza, że wartości logiczne krotek w relacjach również muszą być poddane koniunkcji a wynik przypisany do wynikowej krotki zgodnie z matrycą koniunkcji języka 4QL zdefiniowaną w rozdziale 2.2. Dla wygody, ten krok został pominięty z tłumaczeń koniunkcji w regułach na złączenia w algebrze relacyjnej.

Jeśli reguła ma więcej, niż dwa literały potrzeba zagnieździć operacje złączenia:

```
r(X,V) :- a(X,Y), b(Y,Z), c(Z,V).
```

tłumaczone jest na

$\Pi_{col_1,col_4}((a \bowtie b) \bowtie c)$

Złączenia produkują tymczasową relacje, która agreguje powstałe dotychczas wyniki. Na końcu ta tymczasowa relacja jest rzutowana do ostatecznego wyniku, który odpowiada głowie reguły. Etapy dla powyższej reguły mogłyby wyglądać następująco:

1. `r(X,V) :- a(X,Y), b(Y,Z), c(Z,V).` - złączenie a i b
2. `r(X,V) :- tmp1(X,Y,Z), c(Z,V).` - złączenie *tmp1* i c
3. `r(X,V) :- tmp2(X,Y,Z,V).` - projekcja X,V z *tmp2*

Kolejne kroki ewaluacji reguły wygląda podobnie jak na powyższej liście z tymże zbędne zmienne (w tym wypadku Y i Z) są od razu usuwane poprzez odpowiednie projekcje, jak tylko przestaną być potrzebne.

Translacja reguł i przygotowywanie planu ewaluacji reguł odbywa się przed samą ewaluacją podczas konwersji reguł do funktorów. W podanych wyżej przypadkach nie jest tworzone drzewo ekspresji jak możnaby pomyśleć, ale przyjęto prostszy schemat ewaluacji, w którym każdy literał w regule odpowiada operacjom selekcji i projekcji na predykacie literału, a każdy dodatkowy predykat literału jest łączony z poprzednim aż wyczerpią się wolne literały.

Negatywna informacja

W 4QL negatywna informacja jest definiowana w regułach za pomocą literałów

zewnętrznych z ograniczeniem na wartość logiczną: *nieznany* (z ang. *unknown*). Odpowiada ona operatorowi negacji w stratyfikowanym Datalogu i również może być przetłumaczona na różnicę relacji pod warunkiem, że reguła z literałem zewnętrznym jest "bezpieczna". Dla przykładu, regułę

```
r(X) :- m.a(X,Y) in {unkown}, b(X,Y).
```

można przetłumaczyć na:

$\Pi_{col_1}(b - m.a)$

Z powodu ograniczeń nakładanych na różnicę relacji (krotność i parametry muszą być identyczne) można skonstruować reguły, które trudniej przetłumaczyć na zapytanie w algebrze relacyjnej. Na szczęście implementacja tego rozwiązania nie jest trudna:

```
r(X,Y) :- m.a(X) in {unkown}, b(X,Y).
```

można przetłumaczyć na

$b \bowtie (\Pi_{col_1}(b) - m.a)$

Szczególnym przypadkiem jest pojawienie się innej wartości logicznej oprócz *nieznany* w zbiorze ograniczającym literał zewnętrzny:

```
r(X,Y) :- m.a(X) in {unkown,true}, b(X,Y).
```

to w algebrze relacyjnej

$b \bowtie_{m.a_{\text{Wartość}}=true} m.a \cup (b \bowtie (\Pi_{col_1}(b) - m.a))$

Implentacja tych rozwiązań może wydawać się skomplikowana, ale język programowania C++ jest dużo bardziej ekspresyjny od algebry relacyjnej, która jest ograniczona i musi trzymać się pewnych zasad (np. w kwestii różnicy relacji).

Predykaty funkcyjne

Predykaty funkcyjne dodawane są jako warunki selekcji na końcu ewaluacji reguły. Nie jest to najlepsze rozwiązanie, ponieważ zmienne w literałach, które są wykorzystywane tylko w danym predykacie funkcyjnym, nie mogą być usuniętę poprzez projekcje jak tylko przestaną być potrzebne. Skracanie tymczasowych relacji poprzez

selekcję powoduje mniejsze nakłady obliczeniowe, ponieważ złożoność obliczeniowa operacji selekcji i złączenia zależy od rozmiaru relacji wejściowych.

Na przykład, reguła:

```
r(X) :- a(X,Y), X > 2, b(Y).
```

Będzie przetwarzana w następujących krokach:

1. `r(X) :- a(X,Y), X > 2, b(Y).` - złączenie relacji: $a \bowtie b$
2. `r(X) :- tmp(X,Y), X > 2.` - selekcja: $\sigma_{col1>2}(tmp)$
3. `r(X) :- tmp2(X,Y).` - projekcja do konkluzji: $\Pi_{col_1}(tmp2)$

Alternatywa

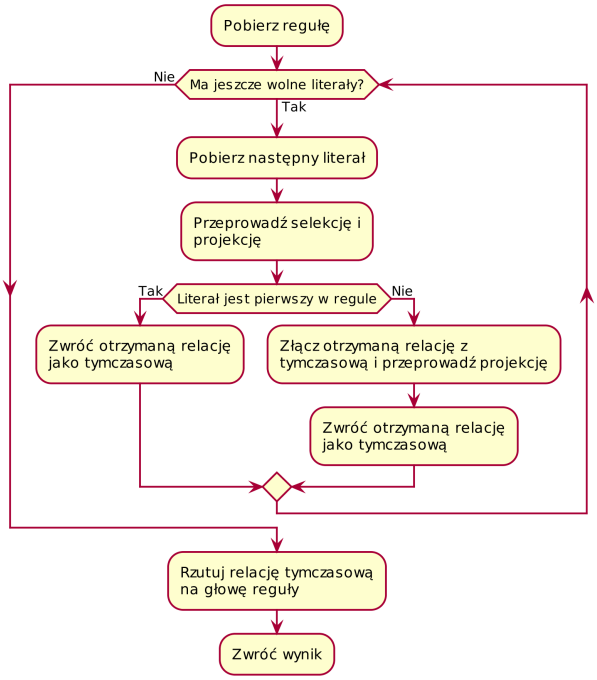
Spójnik alternatywy tłumaczony jest na *pełne złączenie zewnętrzne* (z ang. Full Outer Join). Operacja ta wymagała wprowadzenie dodatkowej wartości, *null*, oznaczającej brak jakiejkolwiek wartości. Tłumaczenie na operacje algebry relacyjnej jest identyczne jak w przypadku koniunkcji, tylko ma niższy priorytet wykonania i wymaga przeprowadzenia sumy logicznej na wartościach zamiast koniunkcji.

3.9 Ewaluacja reguły

Na Rysunku 5 przedstawiono obliczanie wyniku reguły. Najpierw wykonywana jest selekcja i projekcja, a potem złączenie z relacją tymczasową (o ile przetwarzany literał nie jest pierwszym w ciele reguły) i projekcja do wyniku. Po selekcji zawsze jest wykonywana projekcja, żeby usunąć zbędne kolumny. W fovris selekcja i projekcja są połączone i wykonywane podczas tego samego skanu tabeli z relacją. Podobnie złączenie i projekcja do wyniku również są scalone w jedną operację.

Selekcja

Selekcja jest zaimplementowana jako standardowa operacja skanowania o złożoności *O(N)*. Wybierane są literały spełniające warunki selekcji i ograniczenia narzucone na wartości logiczne. Istnieje też wielowątkowa wersja tej operacji, która się uaktywnia, jeśli komputer ma więcej, niż 2 procesory i zostanie przekroczony pewien próg faktów w wejściowej relacji. Warunek progu faktów jest potrzebny, ponieważ



Rysunek 5: Schemat ewaluacji reguły w silniku

małe tablice danych zostaną szybciej przetworzone przez jednej wątek, niż przez wiele wątków, które mają swój dodatkowy narzut operacyjny. Ograniczenie na liczbę procesorów jest spowodowane samym algorytmem. Trzeba przeskanować dwa razy tablicę. Pierwszy raz, żeby znaleźć wielkość wynikowej relacji i wynikowych partycji dla każdego wątku. Drugi raz, żeby przepisać literały do tych partycji. Możliwe przyspieszenie w wykonaniu funkcji wynosi maksymalnie $T/2$, gdzie T to liczba wątków. Jeśli można wykonać równolegle tylko 2 wątki naraz, przyspieszenie wyniesie 1.

Złączenia

Złączenie i pełne złączenie zewnętrzne zostały zaimplementowane w oparciu o algorytm złączenia sortowanego (z ang.Sort Merge Join). Jest to jeden z algorytmów złączenia, który najwydajniej przyspieszyć za pomocą paralelizacji obliczeń. Główny koszt w algorytmie to sortowanie obu relacji, których koszt można rozłożyć na dodatkowe wątki. Czas wykonania takiego złączenia jest więc proporcjonalny do $NlogN/T + MlogM/T + N + M$, gdzie N i M to wejściowe relacje, a T to liczba wątków.

3.10 Różnice między fovris a Inter4QL

W tej sekcji opisano istotne różnice między interpreterami Inter4QL i fovris.

3.10.1 Inne wyniki

W trakcie pracy starano się zreplikować funkcjonalność Inter4QL. Oba programy obliczają jednak różne modele, jeśli reguły zawierają literały zewnętrzne z ograniczeniem na wartość logiczną U. Jak wspomniano wcześniej, problem negacji w Datalogu rozwiązuje się typowo poprzez zastosowanie ograniczeń na definiowane reguły. Jest to tzw. Bezpieczny Datalog. Inną możliwością jest generowanie faktów na podstawie wszystkich wartości dziedzin parametrów literału. Tak właśnie problem ten jest rozwiązany w Inter4QL.

fovris przyjmuje takie same założenia jak Bezpieczny Datalog. Każda zmienna, która występuje w literale ze zmiennymi z nieograniczoną dziedziną (np. `X>Y` lub `m.r(X) = unknown`), musi znajdować pokrycie w literale z ograniczoną dziedziną.

3.10.2 Wbudowane operatory i nowe elementy składni

Operatory binarne

fovris rozszerza składnie o dwuargumentowe operatory porównań, które narzucają ograniczenia na wartości przyjmowane przez zmienne w regułach.

Uproszczona składnia dla literałów zewnętrznych

Zamiast `m.r(X) in {unknown}` można napisać `m.r(X) = unknown`.

Definiowanie zapytań w kodzie programu

W skryptach akceptowanych przez interpreter fovris można zawrzeć zapytania o aktualny stan bazy, które zostaną natychmiast obliczone i wypisane na standardowe wyjście po wczytaniu modułów.

3.10.3 Brak możliwości dodawania predefiniowanych predykatów

Inter4QL umożliwia użytkownikowi dodanie własnych funkcji boolowskich poprzez przeszukiwanie katalogu *plugins* w celu znalezienia plików wykonywalnych. Pliki są uruchamiane z poziomu kodu aplikacji, a jako argumenty wejściowe przekazywane są aktualnie przetwarzane wartości parametrów literału.

Mimo że interpreter dostarczany z fovris natywnie nie pozwala na dodawanie własnych funkcji, to interfejs użytkownika silnika bazy wspiera taką możliwość. Łatwo więc dodać nowe funkcje w module interpretera lub dołączyć do fovris dynamiczny interpreter języka skryptowego, który pozwoli na dodawanie funkcji bez rekompilacji. W ramach pracy testowano interpreter LuaJIT, który wczytuje skrypty napisane w języku lua. Zakładając że dodawane przez użytkownika funkcje są stosunkowo proste i niewymagające obliczeniowo, podejście to jest wielokrotnie szybsze od uruchamiania oddzielnie nowego procesu.

4 Analiza wydajności programu

Test wydajnościowe zostały przeprowadzone w następujących warunkach:

- **Komputer**. Intel i5-4690 3.50 GHz (4 rdzenie). Dostępna pamięć RAM DDR3: 7GB
- **System**. Ubuntu 14.04 LTS 64-bit i Windows 7 64-bit
- **Kompilator**. gcc wersja 5.0

Do porównania zostały wybrane programy: Interl4QL 3.1, souffle.

4.1 Rodzaje testów

Przy doborze testów kierowano się tymi opisanymi w „A Datalog Engine for GPUs" [11]. Testy to programy Datalog, które wykonano dla różnej ilości faktów i zmierzono czasy wykonania programu.

Ta sama generacja

Znany program Datalog o nazwie *Same-Generation*, który generuje mnóstwo krotek w każdej iteracji. Zaczerpnięty z *Foundations of Databases: The Logical Level* [1][Rozdział 13]. Fakty wejściowe wygenerowano następującymi równaniami:

$$up = \{(a, b_i)|i \in [1, n]\} \cup \{(b_i, c_j)|i, j \in [1, n]\}$$
$$flat = \{(c_i, d_j)|i, j \in [1, n]\}$$
$$down = \{(d_i, e_j)|i, j \in [1, n]\} \cup \{(e_i, f)|i \in [1, n]\}$$

Reguły testowego programu są widoczne na Wydruku 36.

Wydruk 36: Reguły do testowego programu sg

```
sg(X,Y) :- flat(X,Y).
sg(X,Y) :- up(X,X1), sg(X1,Y1), down(Y1,Y).
```

Domknięcie przechodnie graphu

Również znany program do obliczania domknięcia przechodniego grafu, czyli sprawdzaniu czy węzły na grafie są połączone.

Fakty generowane są równaniem:

$$edge = \{(a_i, b_j)|i, j \in [1, n]\}$$

,gdzie pod a_i i b_j wstawiane są losowe liczby z przediału $[1, n]$.

Wydruk 37: Reguły do testowego programu tc

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- path(X,Z), path(Z,Y).
```

Złączenie tabel

Proste złączenie dużych tabel, bez rekurencji, w celu przetestowania wydajności operacji selekcji i złączeń.

Wydruk 38: Reguły do testowego programu join

```
join(X,Y) :- tab(X,Z),tab2(Z,3),tab3(Y,Z)
```

Fakty generowane są równaniem:

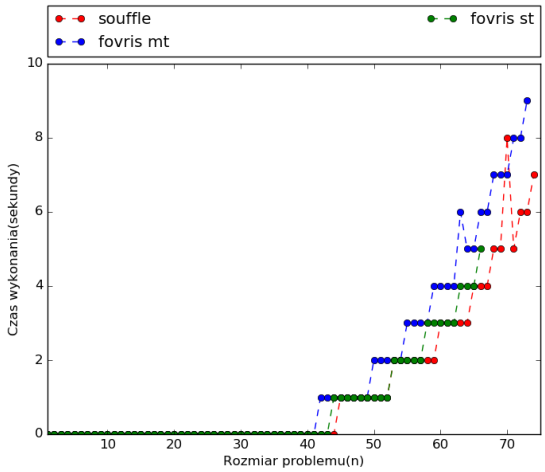
$$tab = \{(a_i, b_j)|i, j \in [1, n]\}$$
$$tab2 = \{(c_i, 3)|i \in [1, n]\}$$
$$tab3 = \{(d_i, e_j)|i, j \in [1, n]\}$$

,gdzie pod zmienne a_i, b_j, c_i, d_i, e_j podstawiane są liczby wygenerowane losowo z przedziału $[1, n]$.

4.2 Wyniki

Ta sama generacja

Z Wykresu 6 wynika, że dla programu same-generation fovris ma podobne wyniki do souffle. Wersja wielowątkowa (fovris_mt) ma gorsze wyniki, niż jednowątkowa, co

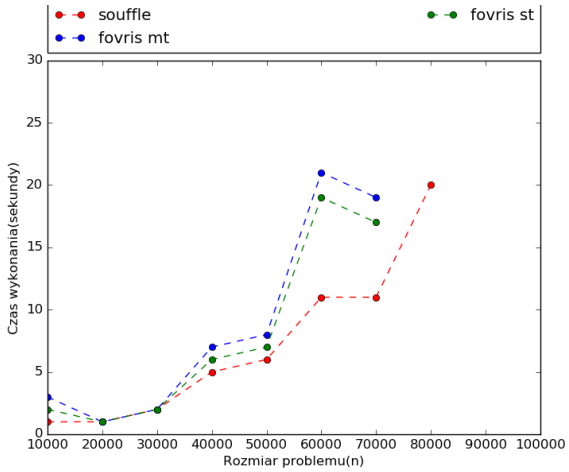


Rysunek 6: Wyniki testów dla programu sg

może wynikać z dodatkowe narzutu jakiego wymaga wielowątkowy algorytm sortujący.

Domknięcie przechodnie graphu

W tym wypadku fovris radzi sobie zdecydowanie gorzej, niż souffle w obliczaniu domknięcia przechodzniego grafu. Program ten wymaga dużo rekurencyjnych operacji dodających w każdej iteracji stosunkowo niewielką ilość faktów. fovris został stworzony z myślą o dużych relacjach, więc wynik porównania nie zaskakuje. Wersja wielowątkowa fovris w dalszym ciągu ma gorszy czas pracy o stały odstęp.

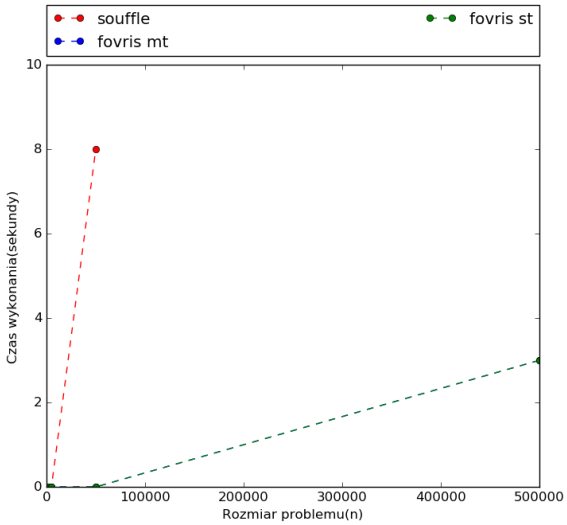


Rysunek 7: Wyniki testów dla programu tc

Złączenie tabel

W złączeniu tabel fovris radzi sobie zdecydowanie lepiej, niż souffle, które prawdopodobnie ma jakiś defekt, bo nie jest w stanie policzyć większych programów *join*

w rozsądnym czasie.



Rysunek 8: Wyniki testów dla programu join

5 Dyskusja wyników i propozycje rozwoju

Wstępne testy wskazują, że silnik fovris nie odstaje daleko od rozwiązań do ewaluacji Dataloga, jeśli chodzi o typowe problemy w tym języku. Potrzebne są porównania z innymi programami poza souffle, żeby wycenić możliwą poprawę w wydajności, jaką można osiągnąć. Wersja wielowątkowa miała gorsze wyniki na programach intensywnie wykorzystujących rekurencję, ale może to być spowodowane zbyt niskim pułapem aktywacji wielowątkowej wersji funkcji do selekcji i złączeń.

Inter4QL został pominięty w wynikach z uwagi na zbyt wolną generację wyników. Dla $n = 5000$ w programie *join* Inter4QL zajmował prawie całą dostępną pamięć. Dzieje się tak najprawdopodobniej z powodu generacji wszystkich niewiadomych faktów na podstawie domeny. Dla 3 kolumn daje to n^3 faktów. Ilość faktów do wygenerowania rośnie więc w wysokim tempie wraz z rozmiarem problemu.

5.1 Dalsze prace

Mniejsze zużycie pamięci

fovris ma dużą złożoność pamieciową spowodowaną wymaganiami nakładanymi na przechowywanie faktów. Możliwe, że inne struktury danych zapewniające unikalność elementów zbioru i losowy dostęp, zajmowałyby mniej pamięci na jeden element zbioru, a spadek szybkości byłby kosztem do zaakceptowania.

Innym powodem dużej złożoności pamięciowej, który łatwiej wyeliminować, jest dynamiczna alokacja faktów. Gdyby ugruntowane literały o tej samej krotności przechowywać w dynamicznych blokach pamięci spadłby nie tylko czas wymagany na alokacje i dealokacje, z powodu mniejszej fragmentacji pamięci, ale również zmniejszyłoby się całkowite zużycie pamięci. Dynamiczna alokacja pojedynczych obiektów niesie ze sobą dodatkowy koszt w postacie narzutu pamięci dla różnych metadanych o zaalokowanym bloku pamięci. Zwłaszcza dla małych bloków pamięci narzut ten może sięgać 100% wielkości zaalokowanej pamięci, co dzieje się dla faktów z małą ilością parametrów. Zaradzić temu problemowi mogą dedykowane alokatory pamięci, alokujące całe bloki pamięci jak `boost::pool_allocator`. Mankamentem tego rozwiązania

jest to, że fakty o każdej ilości parametrów musiałyby mieć oddzielny alokator.

Paralelizacja na GPU

Jak zademonstrował Martin Angelez, przetwarzanie programów Datalog może być efektywnie sparalelizowane w obrębie operatów algebry relacyjnej [11]. W tej pracy również wykorzystano operacje algebry relacyjnej : selekcja, projekcja, różnica i złączenie. Wprowadzenie do fovris biblioteki CUDA i przyśpieszenie za pomocą niej tych operacji może być przynieść dobre efekty, ale wiązałoby się z gruntowną refaktoryzacją aplikacji.

Rozszerzenia składni

fovris może być rozszerzeny o dodatkowe konstrukty wprowadzone przez Inter4QL 3 pozwalające na grupowanie modułów i łączenie wiedzy z modułów w grupach.

Inne przydatne rozszerzenia spotykane w systemach Datalog:

- Funkcje agregujące jak `min`,`count`,`average`
- Symbol wieloznaczny `_` oznaczający brak ograniczenia dla parametru literału
- Rozbudowana składnia zapytań do bazy umożliwiająca większą kontrolę nad otrzymywanymi wynikami. Np. możliwość porządkowania wyników.
- Złożone obiekty jako termy w ugruntowanych literałach

Dodatkowe optymalizacje

Zaimplementowany algorytm semi-naiwny można jeszcze ulepszyć w taki sam sposób, jak dla Dataloga. Aktualny algorytm generuje niepotrzebne fakty w przypadku reguł z koniunkcją w ciele zawierających więcej niż jeden literał z tej samej relacji.

Stosunkowo prosta w realizacji jest również optymalizacja reguł polegająca na usunięciu zbędnych literałów. Przykład:

Z reguły `r(X,Y) :- q(X,Y), p(Z,W).` można usunąć literał `p(Z,W).` bez zmiany wyniku obliczeń.

Jedną z optymalizacji dla semi-naiwnego algorytmu jest zbadanie czy dla literału, do którego akurat dopasowywana jest delta relacji da się znaleźć co najmniej jedną pasującą krotkę. Jeśli się nie da, to ewaluacje reguły można przerwać. Aktualnie nie jest to sprawdzane i powoduje niepotrzebne tworzenie tymczasowych relacji, który mogą nigdy nie zostać wykorzystane.

6 Wnioski

Wynikiem pracy jest aplikacja i biblioteka *fovris*. Interfejs programistyczny biblioteki pozwala na definiowanie relacji, faktów, reguł i grupowania ich w moduły, co odpowiada składni 4QL. Interpreter oparty na tej bibliotece zapewnia analogiczne funkcjonalności co Inter4QL i rozszerza je o tryb nieinteraktywny. fovris zostało przetestowane dużym zestawem skryptów i testów jednostkowych. Implementacja ta otwiera pole do dalszych prac i udoskonaleń. Osiągnięto cel zakładany w pracy, jakim było uzyskanie większej wydajności w stosunku do istniejących rozwiązań. W pracy pokazano, że programy 4QL można, przynajmniej częściowo, tłumaczyć na równania algebry relacyjnej i w ten sposób ewaluować zapytania w tym języku do bazy.

7 Bibliografia

Bibliografia

[1] Serge Abiteboul, Richard Hull i Victor Vianu, red. *Foundations of Databases: The Logical Level*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201537710.

[2] Łukasz Białek. *The inter4ql Interpreter*. 2013. URL: http://4ql.org/wp-content/uploads/2013/10/inter4ql.pdf (term. wiz. 25. 08. 2016).

[3] *BoostMultiIndex Tutorial: Index types*. URL: http://www.boost.org/doc/libs/1_61_0/libs/multi_index/doc/tutorial/indices.html (term. wiz. 29. 08. 2016).

[4] *CMake*. URL: https://cmake.org/ (term. wiz. 26. 08. 2016).

[5] *CMake*. URL: https://github.com/philsquared/Catch (term. wiz. 26. 08. 2016).

[6] *date*. URL: https://howardhinnant.github.io/date/date.html (term. wiz. 26. 08. 2016).

[7] Shan Shan Huang, Todd Jeffrey Green i Boon Thau Loo. „Datalog and Emerging Applications: An Interactive Tutorial". W: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD '11. Athens, Greece: ACM, 2011, s. 1213–1216. ISBN: 978-1-4503-0661-4. DOI: 10.1145/1989323.1989456. URL: http://doi.acm.org/10.1145/1989323.1989456.

[8] Małuszyński Jan i Szałas Andrzej. „Partiality and Inconsistency in Agents' Belief Bases". W: *Frontiers in Artificial Intelligence and Applications* 252.Advanced Methods and Technologies for Agent and Multi-Agent Systems (2013), s. 3–17. ISSN: 0922-6389. DOI: 10.3233/978-1-61499-254-7-3. URL: http://doi.org/10.3233/978-1-61499-254-7-3.

[9] *lemon*. URL: http://www.hwaci.com/sw/lemon/ (term. wiz. 26. 08. 2016).

[10] *linenoise-ng*. URL: https://github.com/arangodb/linenoise-ng (term. wiz. 26. 08. 2016).

[11] Carlos Alberto Martínez-Angeles i in. „A Datalog Engine for GPUs". W: *Declarative Programming and Knowledge Management: Declarative Programming*

Days, KDPD 2013, Unifying INAP, WFLP, and WLP, Kiel, Germany, September 11-13, 2013, Revised Selected Papers. Wyed. Michael Hanus i Ricardo Rocha. Cham: Springer International Publishing, 2014, s. 152–168. ISBN: 978-3-319-08909-6. DOI: 10.1007/978-3-319-08909-6_10. URL: http://dx.doi.org/10.1007/978-3-319-08909-6_10.

[12] *optionparser*. URL: http://optionparser.sourceforge.net/ (term. wiz. 26. 08. 2016).

[13] *re2c*. URL: http://re2c.org/ (term. wiz. 26. 08. 2016).

8 Spis rysunków

Spis rysunków

9 Spis tabel

Spis tablic

10 Spis wydruków

Spis wydruków

Dodatek A Przykład wykorzystania API

Wydruk 39: Przykład wykorzystania API fovris

```
#include "Module.h"
#include "Program.h"
#include <iostream>

int main() {
    using namespace fovris;
    Module m("m",
             {
                 {"parent", {TermType::Id, TermType::Id}},
                 {"ancestor", {TermType::Id, TermType::Id}}
             },
             {
                 {{"ancestor", {Var{"X"}, Var{"Y"}}}, {{"ancestor",
  {Var{"X"}, Var{"Z"}}}, {"parent", {Var{"Z"}, Var{"Y"}}}}},
                 {{"ancestor", {Var{"X"}, Var{"Y"}}}, {{"parent",
  {Var{"X"}, Var{"Y"}}}}}
             },
             {
                 {"parent", {"alice", "bob"}},
                 {"parent", {"alice", "bill"}},
                 {"parent", {"bob", "carol"}},
                 {"parent", {"carol", "dennis"}},
                 {"parent", {"carol", "david"}}
             });
    Program program(Algorithm::Naive);
    program.addModule(m);
    program.evaluate();
    auto results = program.query("m","ancestor",{Var{"X"},Var{"Y"}});
    std::cout << "Results:" << std::endl;
    for(auto& result: results){
        std::cout << result << std::endl;
    }
}
```