Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Maksymilian Humpich
Nr albumu: 209206

Implementacja kontroli typów w systemie LoXiM

Praca magisterska
na kierunku INFORMATYKA
w zakresie BAZY DANYCH

Praca wykonana pod kierunkiem
dra hab. Krzysztofa Stencla
Instytut Informatyki

Marzec 2008

Oświadczenie kierującego pracą

Oświadczam, że niniejsza praca została przygotowana pod moim kierunkiem i stwierdzam, że spełnia ona warunki do przedstawienia jej w postpowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

W pracy prezentuję system półmocnej kontroli typów, który zaimplementowałem w systemie LoXiM - obiektowej, półstrukturalnej bazie danych z językiem zapytań SBQL (Stack Based Query Language). Taka kontrola pozwala w czasie kompilacji zapytania wskazać możliwe do wykrycia błędy typologiczne. Do czasu wykonania przenosi natomiast te sprawdzenia, które są statycznie niewykonalne. Przedstawiam problemy, z jakimi zmierzyłem się wdrażając moduł kontroli typów, moją koncepcję realizacji tablic decyzyjnych, sposób zapewnienia spójności schematu danych oraz rozwiązania dotyczące typów referencyjnych.

Słowa kluczowe

statyczna kontrola typów, obiektowe bazy danych, SBQL, poprawność zapytań, metabaza, schemat danych

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

H. Information Systems
H.2. Database Management
H.2.8. Database Applications

Spis treści

Wprowadzenie

W ramach seminarium z Obiektowych Baz Danych i Systemów Zarządzania Bazami Danych pod okiem dra Krzysztofa Stencla powstaje implementacja obiektowej, półstrukturalnej bazy danych o nazwie LoXiM, z obiektowym językiem zapytań SBQL (Stack Based Query Langauge). System w tej chwili jest już dość zaawansowany (zapewnia m.in. transakcyjność, odbudowanie po awarii, indeksy, statyczną optymalizacje zapytań, model danych został rozszerzony o klasy i dziedziczenie, a także o operacje imperatywne i procedury) i wciąż jest rozbudowywany. Do niedawna jednak LoXiM nie zapewniał żadnej kontroli typologicznej na etapie kompilacji zapytań. Jedyna weryfikacja miała miejsce w czasie wykonania. Moim zadaniem było stworzenie systemu tzw. półmocnej kontroli typów. Zakłada ona przeniesienie dużej części kontroli poprawności zapytania do fazy kompilacji, odkładając na czas wykonania jedynie te sprawdzenia, których nie można wykonać na wcześniejszym etapie. Jest to możliwe dzięki odpowiedniemu podejściu do typów oraz dzięki precyzyjnej semantyce operacyjnej języka SBQL, wynikającej z zastosowania podejścia stosowego - tak powszechnego w popularnych językach programowania. Zależało nam nie tylko na zaimplementowaniu konkretnych reguł kontroli i wnioskowania o typach, ale stworzeniu przejrzystego mechanizmu, w którym te reguły można modyfikować. Taki system kontroli typów udało mi sie wdrożyć w LoXiM dla tzw. modelu M0. Opracowałem nowy fragment składni pozwalający deklarować schemat danych zapewniając jego spójność i kompletność. Zrealizowałem swoją koncepcję przejrzystych tablic decyzyjnych, które wykorzystałem do wnioskowania o typach w głównym przebiegu kontroli zapytań oraz przy wznawianiu procesu kontroli po błędzie. Rozwiązałem również kwestię obsługi typów referencyjnych oraz przetwarzania sygnatur wariantowych. Celem niniejszej pracy jest przedstawienie rozwiązania, które zaimplementowałem.

Praca składa się z czterech rozdziałów. W pierwszym przedstawiam koncepcję półmocnej kontroli typów, jej założenie i cele, oraz zarys logicznej architektury aparatu kontroli typów. Drugi rozdział poświęcony jest metabazie - ważnemu elementowi statycznej kontroli typów. Trzeci rozdział zawiera opis implementacji kontroli typów w systemie LoXiM. Pokazuję jak zrealizowałem założenia, wskazuję problemy, przed jakimi stanąłem podczas pracy. W ostatnim rozdziale określam jak ustawiać i manipulować modułem kontroli typów, jak go zmieniać i jak rozszerzać. Takie informacje są istotne z uwagi na nieustanny rozwój systemu LoXiM. Na końcu zamieszczam dwa dodatki. Jeden z przykładowym schematem danych, zaś w drugim na konkretnej serii zapytań opartych na zaproponowanym schemacie prezentuję działanie kontroli typów. Przykłady zawierają zarówno zapytania poprawne, jak i błędne.

Rozdział 1

Koncepcja półmocnej kontroli typów

1.1. Statyczna i dynamiczna kontrola typów

Kontrola typów jest istotnym elementem języków zapytań i programowania. Pozornie ograniczając programistę eliminuje wiele jego błędów oraz pozwala utrzymać dane w takiej formie, w jakiej zostały zadeklarowane. Warto rozróżnić statyczną kontrolę typów od dynamicznej. Pierwsza wykony-wana jest w trakcie kompilacji zapytania (programu), druga zaś - w czasie jego wykonania. Dynamiczna kontrola jest skuteczna, ponieważ operuje na rzeczywistych danych, a przy pełnej informacji można wykryć wszelkie błędy. Ma jednak dwie podstawowe wady, które powodują, że stawia się na statyczną kontrolę.

- Czas wykonania zostaje obciążony sprawdzaniem poprawności zapytania. Zdarza się, że proste zapytanie operuje na bardzo dużej ilości podobnych danych. W takim przypadku zbędne jest powtarzanie kontroli przy każdym przetwarzanym obiekcie, a narzut nieproporcjonalnie duży.
- Zapytania, które nie są wykonywane 'ad-hoc', a mają być składowane w postaci skompilowanej i uruchamiane w innym momencie - są sprawdzane dopiero w chwili ich wykonania. To oznacza, że programista traci kontrolę nad jakością programu, taka sytuacja jest oczywiście niepożądana.

Zatem dlaczego nie zrealizować "prawdziwej, mocnej", całkowicie statycznej kontroli typów? Okazuje się to niemożliwe przede wszystkim z uwagi na nieregularność danych (mamy do czynienia z bazą semistrukturalną), a także takie cechy języka zapytań jak elipsy, czy koercje dynamiczne. Spójrzmy na przykładowe zapytanie:

$$Student\ \textbf{where}\ Nick = "Tlusty"; \tag{1.1}$$

Jeśli o każdym studencie wiemy, że ma dokładnie jedną ksywkę, to nie ma problemu. Jeśli jednak uznamy, że przezwisko jest atrybutem opcjonalnym, co więcej - że można mieć ich kilka - wtedy takie zapytanie trzeba by uznać za niepoprawne. To jednak byłoby poważnym ograniczeniem dla programisty. Rozwiązaniem jest więc przeniesienie części kontroli do czasu wykonania - ustawienie dynamicznej koercji, która sprawdzi, czy dany obiekt faktycznie ma pojedynczy podobiekt *Nick*. Stąd koncepcja półmocnej kontroli typów, która jest w swojej istocie statyczna, jednak pozostawia otwartą furtkę do kontroli podczas ewaluacji zapytania. Staramy się wykryć maksymalnie wiele błędów w czasie kompilacji, obciążając czas wykonania wyłącznie niezbędnymi sprawdzeniami.

1.2. Typy i sygnatury

Potraktujmy typy możliwie prosto. Uznajmy, że są napisami, ew. strukturami napisów przypisanymi do obiektu (czy ogólniej - bytu bazy danych, bo może to równie dobrze być klasa, procedura, perspektywa), które określają dany byt. Nie nadajemy im żadnej głębszej semantyki. W naszym rozumieniu typ jest więc opisem struktury i właściwości pewnego bytu. Typ można obiektom nadać bezpośrednio, lub za pomocą wcześniej zdefiniowanego **typu nazwanego**, który należy potraktować jak rozwijalną makrodefinicję. Gdybyśmy chcieli zadeklarować obiekt *Address*, jako nazwę ulicy i numer domu moglibyśmy napisać:

```
Address_1 :(Street: string, HouseNo: integer);
```

Taką definicję uznamy za równoważną następującej, wykorzystującej typ nazwany *AddressType*:

```
typedef AddressType = (Street: string, HouseNo: integer);
Address_2 : AddressType;
```

Porównując definicje dwóch obiektów badać można strukturalną zgodność ich typów. W wypadku struktur jest to więc sprawdzenie rekurencyjne. Definiując typ nazwany możemy jednak zaznaczyć, że jego nazwa jest istotna i w jego przypadku należy zastosować zgodność nazwową. Taki typ nazwiemy **rozróżnionym**, a porównanie obiektu opisanego przez taki typ z dowolnym innym daje wynik pozytywny tylko wtedy, gdy drugi obiekt zadeklarowany jest przez ten sam typ.

Kontrola poprawności typologicznej dowolnego zapytania będzie polegała na symulowaniu ewaluacji tego zapytania, gdzie - zamiast wartości faktycznie przetwarzanych podczas wykonania - operuje się na ich statycznych odpowiednikach, tzw. **sygnaturach**. Sygnatura jest opisem, definicją wartości, które występują na stosie wyników podczas ewaluacji zapytania. Dla dowolnego bytu bazy danych jego typ reprezentowany jest więc przez pewną sygnaturę. Każda sygnatura składa się z podstawy oraz pewnej liczby atrybutów. Podstawa sygnatury może przyjąć następujące formy:

Typ elementarny - *"string", "integer", "double"*, lub dowolna inna nazwa typu elementarnego, który został przyjęty w konkretnym języku.

Typ referencyjny - **ref**(id) - zawiera identyfikator definicji wskazywanego obiektu.[1]

Binder statyczny - **n**(S), gdzie **n** jest nazwą, a S - dowolną sygnaturą. Odpowiada binderowi z czasu wykonania.

Struktura - **struct**$(\ s_1, s_2, \ldots s_n\ )$, gdzie s_i są dowolnymi sygnaturami. Taka struktura powstaje np. w wyniku zastosowania operatora **join**.

Wariant - **variant**$(\ s_1, s_2, \ldots s_n\ )$ to sygnatura wyniku, który podczas wykonania może przyjąć postać dowolnej sygnatury spośród $s_1 \ldots s_2$.

Każda sygnatura może jeszcze mieć szereg atrybutów. W systemie opisywanym w tej pracy, występują następujące atrybuty:

card (liczebność) Opisuje dopuszczalną wielkość kolekcji określonej przez daną sygnaturę. Przyjmuje wartości: "0..0", "0..1", "1..1", "0..*", "1..*". Taki zapis wskazuje dolną i górną granicę liczebności, przy czym '*' oznacza 'dowolnie wiele'. Domyślną wartością jest "1..1".

[1] Tak naprawdę jest to identyfikator dowolnego wierzchołka metabazy, o której mowa w rozdziale 2.

collectionKind (rodzaj kolekcji) Kolekcje mogą być sekwencjami, zbiorami, wielozbiorami, słownikami, etc. W rozwiązaniu wdrożonym w LoXiM występują jedynie listy i wielozbiory (bag, sequence). Są one wystarczające, aby pokazać ideę kontroli typów. Nic nie stoi na przeszkodzie, aby wprowadzić nowe rodzaje kolekcji, definiując dla nich odpowiednio semantykę wszystkich operatorów. Jeśli '*' jest górną granicy liczebności, domyślnie przyjmowana jest wartość "*bag*", w przeciwnym razie atrybut pozostaje pusty.

typeName (nazwa typu) Sygnatura, która posiada taki atrybut, jest niezgodna z każdą sygnaturą, której nazwa typu jest inna lub pusta. Przy braku nazw typów brana jest pod uwagę strukturalna zgodność sygnatur. Domyślnie nazwa typu jest pusta, chyba, że sygnatura określa obiekt zadeklarowany za pomocą tzw. typu wyróżnionego (czyli w przypadku naszego systemu - zdefiniowanego ze słowem kluczowym **distinct**).

mutability (zmienność) Przyjmuje wartość **true** dla tych sygnatur, które definiują obiekty, które mogą zostać zmienione, tzn. np. być podmiotem dla operatorów **delete**, **assign**, **insert**. Będą to zawsze identyfikatory wierzchołków metabazy, dlatego w odróżnieniu od innych atrybutów - ten występuje implicite.

ref (flaga zakazu dereferencji) Atrybut ustawiany poprzez specjalny operator unarny **ref()**. Jeśli ta flaga ma wartość **true**, dana sygnatura nie podlega automatycznym dereferencjom.

1.3. Aparat statycznej kontroli typów

1.3.1. Podstawowe struktury statyczne

Półmocna kontrola typów polega na symulacji wykonania zapytania, zatem system kontroli typów musi korzystać ze statycznych struktur, które odzwierciedlają te wykorzystywane podczas ewaluacji. Podczas wykonywania dowolnego zapytania SBQL aparat wykonawczy przechodzi jego drzewo syntaktyczne, korzystając ze stosu środowisk, na którym wiązane są wszystkie nazwy występujące w zapytaniu oraz ze stosu wyników, gdzie odkładane są rezultaty kolejnych zagnieżdżonych operatorów. Naturalnie potrzebny jest również dostęp do składu danych, aby operować na obiektach przechowywanych w bazie. Każda z tych struktur ma swój statyczny odpowiednik:

Schemat danych Inaczej metabaza - to struktura odpowiadająca składowisku danych. Tu zamiast realnych danych trzymane są ich definicje. Powstaje w wyniku deklarowania obiektów, jakie mają występować w bazie. Podobnie jak sam skład danych - wyróżnione są obiekty korzeniowe (bazowe). Dzięki tej strukturze znając nazwę obiektu można określić jego strukturę i powiązania z innymi obiektami. W sensie logicznym schemat danych jest grafem, w którym węzłami są definicje bytów bazodanowych (obiektów, procedur, klas), a krawędziami - związki między tymi bytami (np. referencje przy obiektach wskaźnikowych, połączenia obiektu z podobiektami). Szczegółowy opis realizacji schematu danych w systemie LoXiM znajduje się w kolejnym rozdziale.

Statyczny stos środowisk Jest to analogia stosu środowisk czasu wykonania. Elementami tego stosu są sekcje zawierające kolekcje binderów statycznych - sygnatur wiążących identyfikatory wierzchołków metabazy z nazwami obiektów, jakie definiują. Do sekcji bazowej stosu, przed rozpoczęciem kontroli wstawiane są bindery statyczne z identyfikatorami wszystkich obiektów korzeniowych w schemacie.

Statyczny stos wyników Odpowiednik stosu wyników czasu wykonania. Jego elementami są sygnatury, powstałe w rezultacie działania operatorów na sygnaturach wejściowych.

Drzewo syntaktyczne zapytania Ta sama struktura, na której wykonywana jest ewaluacja. Istotny jest fakt, że statyczna kontrola odbywa się nie tylko na takim samym, ale dokładnie **tym samym** drzewie składni abstrakcyjnej, które będzie przekazane do czasu wykonania. Ma to znaczenie, ponieważ podczas kontroli typologicznej drzewo składni abstrakcyjnej zapytania może zostać zmienione - zaopatrzone w dodatkowe węzły (dynamiczne koercje, elipsy, automatyczne dereferencje), które powinny zostać przetworzone przez aparat wykonawczy.

1.3.2. Tablice decyzyjne

Poza wyżej wymienionymi elementami moduł kontroli typów ma jeszcze do dyspozycji zbiór **tablic decyzyjnych**. Stanowią one mechanizm, który dla każdego operatora pozwala określić rezultat jego działania na dowolnym zestawie sygnatur. Dla danych argumentów możliwe sa trzy rodzaje decyzji:

1. **Sukces** - Argumenty są poprawne dla danego operatora, zostaje wskazana sygnatura wynikowa.

2. **Błąd** - Argumenty są niepoprawne, zgłaszany jest błąd kontroli typów. Nie oznacza to przerwania kontroli - aby wykryć w jednym przebiegu maksymalnie wiele błędów, podejmowana jest próba wznowienia procesu kontroli.

3. **Kontrola dynamiczna** - Na etapie kompilacji niemożliwe jest stwierdzenie, czy argumenty są poprawne. Jakaś część kontroli musi zostać przeniesiona do czasu wykonania zapytania. W tym miejscu wskazywana jest akcja, jaką należy podjąć podczas ewaluacji (zazwyczaj oznacza to dodanie do drzewa zapytania dodatkowego węzła koercji dynamicznej).

Dla każdego operatora należy podjąć taką decyzję zarówno wobec podstaw sygnatur (próbując ustalić podstawę sygnatury wynikowej), jak i wobec każdego atrybutu. Okazuje się, że w znakomitej większości są operatory, które cechuje pełna ortogonalność ze względu na przetwarzanie podstawy sygnatury oraz jej atrybutów. W praktyce oznacza to, że wyprowadzenie podstawy sygnatury wynikowej może odbywać się niezależnie od ustalenia jej liczebności, nazwy typu czy rodzaju kolekcji. Niezależność tych wyprowadzeń będę nazywał **zasadą ortogonalności**. Dla zdecydowanej większości operatorów ta zasada jest spełniona, dlatego każdy operator pociąga za sobą kilka tablic decyzyjnych (tak naprawdę kilka zestawów reguł - po jednym dla każdego atrybutu oraz jeden dla podstawy). Brak reguł dla danego atrybutu oznacza przyjęcie jego domyślnej wartości, natomiast w przypadku podstawy - ponieważ nie istnieje wartość domyślna - brak reguł oznacza, że operator nie jest rozpoznawany przez system kontroli typów.

Wybór konkretnych reguł to szereg dość arbitralnych decyzji, które nie wynikają z żadnej teorii i nie powinny być narzucone twórcom języka zapytań. Czy dodając napis do liczby powinniśmy zgłosić błąd, potraktować liczbę jako jej zapis i dokonać konkatenacji, czy też spróbować konwersji napisu do liczby, aby wykonać działanie arytmetyczne? Tego typu decyzje powinny być podjęte na podstawie zdrowej intuicji programistycznej oraz poprawiane, gdy doświadczenie wskazuje, że są niepraktyczne. Naturalnie w dużej mierze te reguły wynikną z semantyki operatorów, i nie mogą istnieć w oderwaniu od samego aparatu wykonawczego. Traci bowiem sens reguła pozwalająca dodawać wartości logiczne, czyli sygnatury typu

"boolean" (w zamierzeniu - działająca tak jak logiczny operator 'OR'), jeśli takiej możliwości nie przewiduje moduł ewaluacji zapytań. Poniżej przedstawiam przykładowe tablice dla dwóch wybranych operatorów: dodawania oraz konstruktora struktury. Analizę tablic wielu z pozostałych operatorów można znaleźć w [Stencel06].

Operacja dodawania

Podobnie jak przy innych operacjach arytmetycznych, dodawanie można uznać za poprawne nie tylko wtedy, gdy argumentami są liczby. Przyjrzyjmy się poniższej tablicy decyzyjnej.

Podstawa sygnatury			
q_1	q_2	$q_1 + q_2$	Koercje
integer	integer	integer	
double	double	double	
string	string	string	
integer	double	double	Koercja q_1 do double
double	integer	double	Koercja q_2 do double
string	integer	string	Koercja q_2 do string
integer	string	integer	Koercja q_2 do integer z kontrolą dyn.
Inne		Błąd	
Atrybut **card**			
1..1	1..1	1..1	
x..y	1..1	1..1	Koercja q_1 do 1..1 z kontrolą dyn.
1..1	x..y	1..1	Koercja q_2 do 1..1 z kontrolą dyn.
Inne		1..1	Koercja q_1 i q_2 do 1..1 z kontrolą dyn.
Atrybut **typeName**			
–	–	–	
Inne		Błąd	

Pierwsze wiersze tablicy są zrozumiałe - dodawanie zgodnych typów elementarnych jest poprawne, kontrola kończy się sukcesem, można też łatwo wskazać typ wyniku. Kolejne cztery reguły to zalecenia koercji, tzn. próby zrzutowania argumentu na inny typ. Wskazany typ wyniku to ten, który otrzymamy przy założeniu, że koercja się powiedzie. Zwróćmy uwagę na to, że pierwsze trzy koercje zawsze zakończą się sukcesem, dlatego będziemy je nazywać statycznymi. Czwarta natomiast jest próbą zrzutowania napisu na liczbę. Powodzenie takiej operacji można sprawdzić dopiero podczas wykonania zapytania, mając do dyspozycji konkretny napis. Takie koercje nazwiemy dynamicznymi - kontrola poprawności zapytania zostaje przeniesiona do czasu wykonania. Podkreślę jeszcze raz - taki wybór reguł jest tylko przykładem, istnieje wiele alternatywnych rozwiązań. Trudno na gruncie jakiejkolwiek teorii typów wykazywać prymat jednego wyboru nad innym - takiej oceny można dokonać na podstawie doświadczeń, intuicji, prób i błędów. Nas interesuje sam mechanizm oraz wykorzystanie tablic decyzyjnych. Przy ich konstrukcji natomiast - istotne jest, aby były w prosty sposób konfigurowalne.

Oba argumenty dodawania muszą być pojedynczymi elementami. W przypadku, gdy któraś liczebność jest inna niż 1..1, dodawana jest koercja dynamiczna - sprawdzenie, czy dany

argument podczas wykonania faktycznie reprezentuje jedną wartość.

Nie ma potrzeby rozpatrywania atrybutu **collectionKind**, bo liczebność wyniku zawsze jest 1..1. Natomiast dla atrybutu **typeName** sprawa jest dość prosta: powinien on być pusty w obu atrybutach. W przeciwnym przypadku zgłaszany jest błąd kontroli typów. Taka reguła wynika z zasady, że obiekty zadeklarowane za pomocą typów rozróżnionych można porównywać (o ile nazwy typów się zgadzają), ale nie wykonywać na nich żadnych operacji arytmetycznych. Przyjęcie takiej zasady może wydawać się mało elastyczne. Z pomocą programiście przychodzi jednak jawne rzutowanie, dzięki któremu każda operacja jest wykonalna. Jeśli np. pensja pracownika nie jest zadeklarowana jako liczba, ale jako *PLN* - nazwany typ rozróżniony, którego podstawą jest *integer*, wtedy śmiało można pensje dodawać, ale dopiero po zrzutowaniu na *integer*.

Konstruktor struktury

Wynikiem tego operatora jest wielozbiór struktur - wszystkie możliwe kombinacje składników lewego argumentu ze składnikami prawego. Spójrzmy jak przedstawia się tablica decyzyjna konstruktora struktury:

Podstawa sygnatury			
q_1	q_2	(q_1, q_2)	Koercje
S_1	S_2	$(S_1[\mathbf{card} = 1..1], S_2[\mathbf{card} = 1..1])$	
Atrybut **card**			
$x_1..y_1$	$x_2..y_2$	$x_1 * x_2$ .. $y_1 * y_2$	
Atrybut **typeName**			
Cokolwiek		–	
Atrybut **collectionKind**			
–	–	–	
–	*bag*	*bag*	
bag	–	*bag*	
bag	*bag*	*bag*	
Inne		Błąd	

W tej tablicy występują tzw. *metazmienne*, tzn. formalne symbole, które oznaczają dowolną podstawę/atrybut sygnatury, które pokazują, w jaki sposób wynik operacji konstruowany jest na podstawie argumentów. Zauważmy, że - w odróżnieniu od tablicy dla dodawania - rezultaty reguł nie są konkretnymi podstawami sygnatur lub wartościami atrybutów, ale przepisami na skonstruowanie wyniku. Takie *generatory* wyników mogą być dowolnie skomplikowane i zależą od semantyki operatora. W przypadku struktur występuje generator podstawy oraz liczebności. Zobaczmy, podstawa wynikowej sygnatury zależy zarówno od podstaw, jak i atrybutów sygnatur argumentów. To oznacza złamanie zasady ortogonalności. Jak widać nie stanowi to jednak problemu dla aparatu kontroli typów, o ile tablice decyzyjne są właściwie zrealizowane i umożliwiają definiowanie generatorów na podstawie pełnych sygnatur wejściowych.

Liczebność wyniku konstruktora struktury jest iloczynem liczebności argumentów, co zostało odzwierciedlone w regule atrybutu **card**. Mnożąc dolne i górne ograniczenia wystarczy

przyjąć, że gwiazdka (symbolizująca "dowolnie wiele") pomnożona przez cokolwiek daje w wyniku również gwiazdkę.

Atrybut **typeName** struktury jest zawsze pusty, bez względu na nazwy typów argumentów. Rodzaj kolekcji natomiast to wielozbiór. Zgodnie z powyższą tabelą błędem zakończy się próba utworzenia struktury na podstawie list, natomiast poprawnie zostaną wykonane struktury, gdy argumenty są pojedynczymi elementami lub wielozbiorami.

1.4. Algorytm typeCheck

Mając do dyspozycji struktury opisane w poprzedniej sekcji oraz drzewo syntaktyczne, aparat kontroli typów wykonuje algorytm *typeCheck* - przeprowadza symulację wykonania zapytania, przy każdym operatorze weryfikując poprawność jego argumentów i wyliczając jego wynik. Zalecenia koercji, które występują w tablicach decyzyjnych powodują wzbogacenie drzewa zapytania o odpowiednie węzły lub flagi koercji. W przypadku wykrycia błędu można by zakończyć tę procedurę, uznając, że dalsza kontrola nie ma sensu, skoro zapytanie jest niepoprawne. To jest jednak mało efektywne, szczególnie w przypadku skomplikowanych zapytań lub dłuższych programów, dlatego warto zarejestrować wystąpienie błędu kontroli typów, a następnie podjąć próbę odbudowania procesu i kontynuowania kontroli, aby w pojedynczym przebiegu wykryć maksymalnie wiele pomyłek. Tylko w przypadku nieudanej próby wznowienia procesu przerywana jest kontrola i zgłaszany ostateczny błąd.

Zanim algorytm zostanie uruchomiony inicjowane są jego pomocnicze struktury danych, m.in. zostaje utworzona sekcja bazowa na statycznym stosie środowisk, zawierająca statyczne bindery do obiektów korzeniowych. *typeCheck* przechodzi drzewo składni abstrakcyjnej zapytania rekurencyjnie, wykonując następujące operacje, w zależności od rodzaju węzła:

- **Literał** - Najprostszy przypadek, na statyczny stos wyników wkładana jest sygnatura odpowiedniego typu elementarnego
- **Nazwa** n - Następuje próba wiązania nazwy n na statycznym stosie środowisk. Stos przeszukiwany jest od wierzchołka w dół, w poszukiwaniu bindera $n(S)$, gdzie S jest dowolną sygnaturą. Poprawne wiązanie kończy się włożeniem na stos wyników sygnatury S. Jeśli nazwa nie została znaleziona, następuje próba rozwinięcia elipsy (opisanego w 1.5). Jeśli i to nie skutkuje - zgłaszany jest błąd kontroli.
- **Operator Unarny** $\Delta(q)$ - Sprawdzane jest rekurencyjnie podzapytanie q, a jego sygnatura pobierana jest ze stosu wyników. Na podstawie otrzymanej sygnatury oraz tablicy przypisanej operatorowi Δ podejmowana jest decyzja. Jeśli jest pozytywna - na stos wkładany jest sygnatura skonstruowana na podstawie odpowiedniej reguły w tablicy. W przypadku błędu podjęta jest próba dokonania automatycznej dereferencji (patrz sekcja 1.5), a jeśli i to nie zmienia rezultatu, zgłaszany jest błąd kontroli. Następnie podjęta zostaje próba naprawienia procesu. W przypadku zalecenia koercji w tablicy, drzewo zapytania uaktualniane jest o odpowiednie węzły lub flagi koercji.
- **Operator Algebraiczny** $q_1 \Delta q_2$ - Rekurencyjnie wykonywany jest *typeCheck* na zapytaniu q_1, a następnie na q_2. Wyniki tych kontroli pobierane są ze statycznego stosu wyników. Dalej schemat kontroli jest podobny jak w przypadku operatorów unarnych. Wybierana jest tabela decyzyjna dla danego operatora, wynikowa sygnatura zostaje odłożona na stosie. Zalecenia koercji przekładają się na operacje na drzewie zapytania, po błędzie następuje próba automatycznej dereferencji, ewentualnie zgłaszany jest błąd kontroli.

- **Operator Niealgebraiczny** $q_1 \Delta q_2$ - Kontrola takich operatorów jest podobna do kontroli operatorów algebraicznych, z tą różnicą, że po wykonaniu *typeCheck* na lewym podzapytaniu zmieniany jest stos środowisk. Dodana zostaje sekcja zawierająca wynik funkcji *static_nested* (statycznego odpowiednika *nested* z czasu wykonania) na wyniku lewego podzapytania. W takim zmodyfikowanym kontekście wykonywana jest kontrola prawego poddrzewa, po której odpowiednia sekcja jest zdejmowana ze stosu środowisk. Dalszy ciąg operacji jest identyczny jak dla operatorów algebraicznych.
- **Operator imperatywny** - Operacje imperatywne, w zależności od ich semantyki podlegają pod jeden z wyżej wymienionych schematów, nie stanowią wyzwania koncepcyjnego, ich wynikiem - przy pozytywnej weryfikacji argumentów - są zazwyczaj puste sygnatury. Niektóre z nich (operatory: przypisania **:=**, usuwania **delete**, wstawienia :<, tworzenia **create**) wymagają jednak skomplikowanych sprawdzeń, łamiących zasadę ortogonalności. Takie operatory można podporządkować powyższym schematom, przerzucając ciężar kontroli na ich tablice decyzyjne. Właśnie takie podejście zastosowałem w rozwiązaniu opisywanym w tej pracy. Szczegóły w rozdziale 3.6

1.5. Metody uzupełniające

Elipsy

Przyjrzyjmy się poniższemu przykładowi zapytania, opartemu na schemacie danych z dodatku A:

$$Student \textbf{ where } ToughtBy.Title = "AssociateProfessor"; \tag{1.2}$$

Takie zapytanie nazwiemy eliptycznym, ponieważ jest niepełną, skróconą formą zapytania

$$Student \textbf{ where } ToughtBy.Professor.Title = "AssociateProfessor"; \tag{1.3}$$

Zauważmy, że bez żadnych dodatkowych operacji w zapytaniu 1.2 nazwa *Title* nie zostanie poprawnie związana. W momencie jej wiązania na statycznym stosie środowisk znajduje się jedynie sekcja bazowa, wynik *stat_nested* zastosowanej do *Student*, oraz do obiektu *ToughtBy*, który jest referencją do *Professor*. Zatem na wierzchołku nie ma wnętrza obiektu *Professor*. Taka skrócona forma jest jednak jasna i intuicyjna i można pokusić się o uznanie jej za poprawną, jeśli tylko aparat statycznej kontroli poradzi sobie z rozwinięciem jej do pełnej formy 1.3. Taka operacja jest możliwa - wystarczy przejrzeć stos od góry rozwijając każdą sygnaturę referencyjną za pomocą funkcji *stat_nested* a wśród wyników szukać zagubionej nazwy (w naszym przykładzie - nazwy *Title*. Jeśli uda się ją znaleźć, nazwa bindera rozwiniętej sygnatury referencyjnej posłuży do uzupełnienia zapytania do pełnej formy.

Automatyczne dereferencje

Szczególnym przykładem elips (stosowania skróconych zapytań) jest pominięcie jawnej dereferencji przy odwoływaniu się do nazw obiektów o sygnaturac elementarnych.

$$Student \textbf{ where } Name = "Christine"; \tag{1.4}$$

W powyższym zapytaniu *Name* zwraca sygnaturę referencyjną, zaś "Christine" - sygnaturę typu elementarnego *string*. Aby porównanie mogło dojść do skutku, *Name* powinno zostać opatrzone dereferencją. W takiej sytuacji tablica decyzyjna dla operatora = zgłosi błąd, co poprowadzi właśnie do próby automatycznej dereferencji. Oba argumenty równości zostają poddane dereferencji, w efekcie lewa sygnatura staje się sygnaturą elementarną *string*,

a prawa pozostaje bez zmian. Tablica decyzyjna porównania zostanie znów wykorzystana, wskazując, że nowe sygnatury są poprawnymi argumentami operatora =. W takiej sytuacji drzewo zapytania zostanie uzupełnione do postaci:

$$Student \textbf{ where deref}(Name) = "Christine"; \tag{1.5}$$

Naprawa i wznowienie procesu

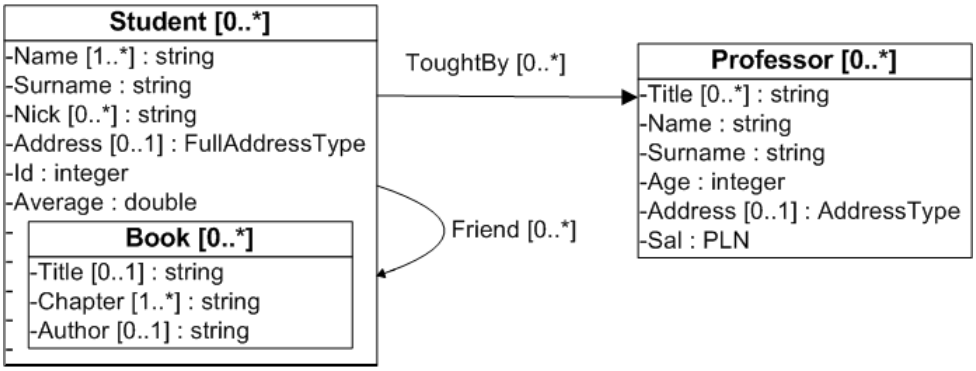
Aby wykryć więcej niż jeden błąd typologiczny podczas pojedynczego przebiegu kontroli, podejmowana jest próba naprawy procesu. Taki zabieg polega na określeniu najbardziej prawdopodobnej sygnatury wynikowej w miejscu, gdzie nastąpił błąd.

Jeśli był on spowodowany nieprawidłową nazwą, należy odszukać na statycznym stosie środowisk możliwie bliskiej nazwy. Trudno podać przepis na "najbliższą" nazwę. Można w minimalizować "odległość edycyjną" od szukanej nazwy w całym stosie. Już samo określenie miary odległości edycyjnej można zrealizować na wiele sposobów. Być może warto jednak przyjąć, że im głębiej na stosie, tym mniej prawdopodobny jest zamiennik, nadając tym samym odpowiednie wagi kolejnym sekcjom. Dobór algorytmu, podobnie jak w przypadku reguł tablic decyzyjnych, powinien być poparty doświadczeniami i zdrową intuicją.

Jeśli niepoprawne są argumenty jakiegoś operatora, trzeba podjąć decyzję odpowiadając na pytanie: Jaki byłby wynik, gdyby argumenty były poprawne? Można to zrealizować na różne sposoby. W rozwiązaniu, jakie przyjąłem w LoXiM zastosowałem mechanizm uproszczonych tablic decyzyjnych, bardzo podobnych do tych, wykorzystywanych przez algorytm *typeCheck*. Te tablice w zależności od wejściowych sygnatur zwracają domyślny wynik, lub komunikują ostateczny błąd kontroli typów (skutkiem którego jest przerwanie kontroli).

1.6. Przykład działania algorytmu

Prześledźmy działanie algorytmu *typeCheck* na przykładzie poniższego zapytania do składu danych o schemacie danych opisanym w dodatku A. Rysunek 1.1 przedstawia odpowiedni fragment schematu, niezbędny do realizacji tego zapytania.

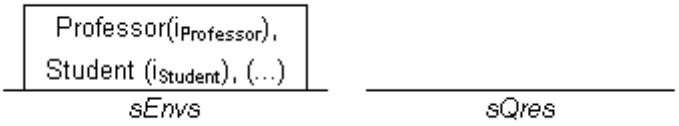


Rysunek 1.1: Fragment schematu danych odpowiadający zapytaniu 1.6

$$(Student \textbf{ where } Nick="Wscibski") \textbf{ join } (ToughtBy.Sal) \tag{1.6}$$

Powyższe zapytanie zwraca zbiór par identyfikatorów, z których pierwszy odnosi sie do studentów, a drugi wskazuje pensje ich nauczycieli. Zanim kontrola się rozpocznie inicjowane są statyczne stosy. Na stosie środowisk (*sEnvs*) są bindery obiektów korzeniowych (*Student*,

Professor, *College*, etc.), natomiast statyczny stos wyników (*sQres*) jest pusty, jak pokazuje rysunek 1.2. Dla zwiększenia przejrzystości stos środowisk będzie przedstawiany w uproszczony sposób - niektóre, nieistotne w rozważanym zapytaniu, elementy będą pomijane. Na rysunku 1.3 widać jak zmieniają się stosy w kolejnych krokach.



Rysunek 1.2: Stan stosów przed rozpoczęciem kontroli.

1. Schodząc rekurencyjnie po drzewie zapytania, *typeCheck* rozpoczyna kontrolę od nazwy *Student*. Zostaje ona pomyślnie związana na stosie *sEnvs*, a rezultat tego wiązania - sygnatura referencyjna $\mathbf{ref}(i_{Student})$ trafia na stos *sQres*.

2. Następnie analizowany jest operator niealgebraiczny **where**. Jego lewym podzapytaniem była rozpatrzona w poprzednim kroku nazwa. Przed analizą prawego podzapytania - na stos *sEnvs* wkładana jest sekcja wnętrza obiektu *Student*, czyli bindery $Name(i_{Name})$, $Nick(i_{Nick})$, $Id(i_{Id})$, $ToughtBy(i_{ToughtBy})$ i wszystkie pozostałe podobiekty *Student*. W takim zmienionym środowisku przebiegnie analiza prawego podzapytania: $Nick =$ "*Wscibski*".

3. Przechodzimy do analizy Nazwy *Nick*. Zostaje związana w górnej sekcji stosu i wynik wiązania: $\mathbf{ref}(i_{Nick})$ wkładany jest na statyczny stos wyników.

4. Dalej mamy literał "*Wscibski*" - sygnatura *string* zostaje umieszczona na stosie *sQres*.

5. Teraz przystępujemy do analizy operatora porównania =. Zdejmujemy dwie sygnatury z *sQres* i w tablicy decyzyjnej dla = szukamy reguły odpowiadającej argumentom $\mathbf{ref}(i_{Nick})$ oraz *string*. Takie sygnatury nie podlegają porównaniom - dlatego z tablicy odczytamy błąd kontroli - niepoprawne podstawy sygnatur.

6. Wobec błędnych podstaw sygnatur następuje automatyczna dereferencja lewej sygnatury, wynikiem której jest sygnatura *string*. Nowe sygnatury wejściowe znów przykładamy do tej samej tablicy decyzyjnej. Porównanie napisów kończy się sukcesem, wynikiem jest sygnatura *boolean*. Tablica decyzyjna nakaże jednak koercję ze względu na liczebności argumentów. U studenta *Nick* ma liczebność 0..*, zaś wymaga się liczebności 1..1. Dlatego zostaje dodana koercja **element()** wokół lewego podzapytania, a na stos *sQres* trafia sygnatura wynikowa *boolean*. Po tym kroku zapytanie zostało uaktualnione do postaci:

$$(Student\ \mathbf{where}\ \mathbf{element}(\mathbf{deref}(Nick)) = "Wscibski")\ \mathbf{join}\ (ToughtBy.Sal) \quad (1.7)$$

7. Tym sposobem zakończyliśmy kontrolę prawego podzapytania operatora **where**. Ze stosu wyników zdejmujemy dwie sygnatury: $\mathbf{ref}(i_{Student})$ oraz *boolean*. Tablica decyzyjna operatora **where** przyjmuje takie argumenty jako poprawne i jako wynik zwraca sygnaturę lewego argumentu, czyli $\mathbf{ref}(i_{Student})$, która następnie trafia na *sQres*.

8. Kontrola prowadzi dalej do operatora **join**. Jest to również operator niealgebraiczny, dlatego na stos środowisk ponownie trafia sekcja z binderami otrzymanymi w wyniku zastosowania funkcji *stated_nested* dla sygnatury $\mathbf{ref}(i_{Student})$. Stos *sEnvs* wygląda zatem tak samo, jak po kroku drugim.

9. Kolejną operacją jest związanie nazwy *ToughtBy* na stosie środowisk i umieszczenie $\mathbf{ref}(i_{ToughtBy})$ na stosie wyników.

10. Dalej niealgebraiczny operator złączenia nawigacyjnego. Na *sEnvs* wkładany jest binder $Professor(i_{Professor})$ jako wynik funkcji *stated_nested* na sygnaturze referencyjnej z wierzchołka stosu *sQres*. W tym kontekście rozpatrzone będzie prawe podzapytanie kropki.

11. Następuje próba związania nazwy *Sal* na statycznym stosie środowisk. Próba kończy się niepowodzeniem, ponieważ odpowiedni binder nie znajduje się w żadnej z sekcji. W takiej sytuacji aparat kontroli zakłada, że może mieć do czynienia z elipsą, więc próbuje ją rozwinąć. Najwyższa sekcja *sEnvs* zawiera binder $Professor(i_{Professor})$ wskazujący definicję obiektu *Professor*, którego jeden z podobiektów ma szukaną nazwę *Sal*. Elipsa może zatem zostać rozwinięta. Na stos wyników wkładana jest sygnatura $\mathbf{ref}(i_{Student})$, a odpowiedni fragment drzewa zapytania zostaje uzupełniony:

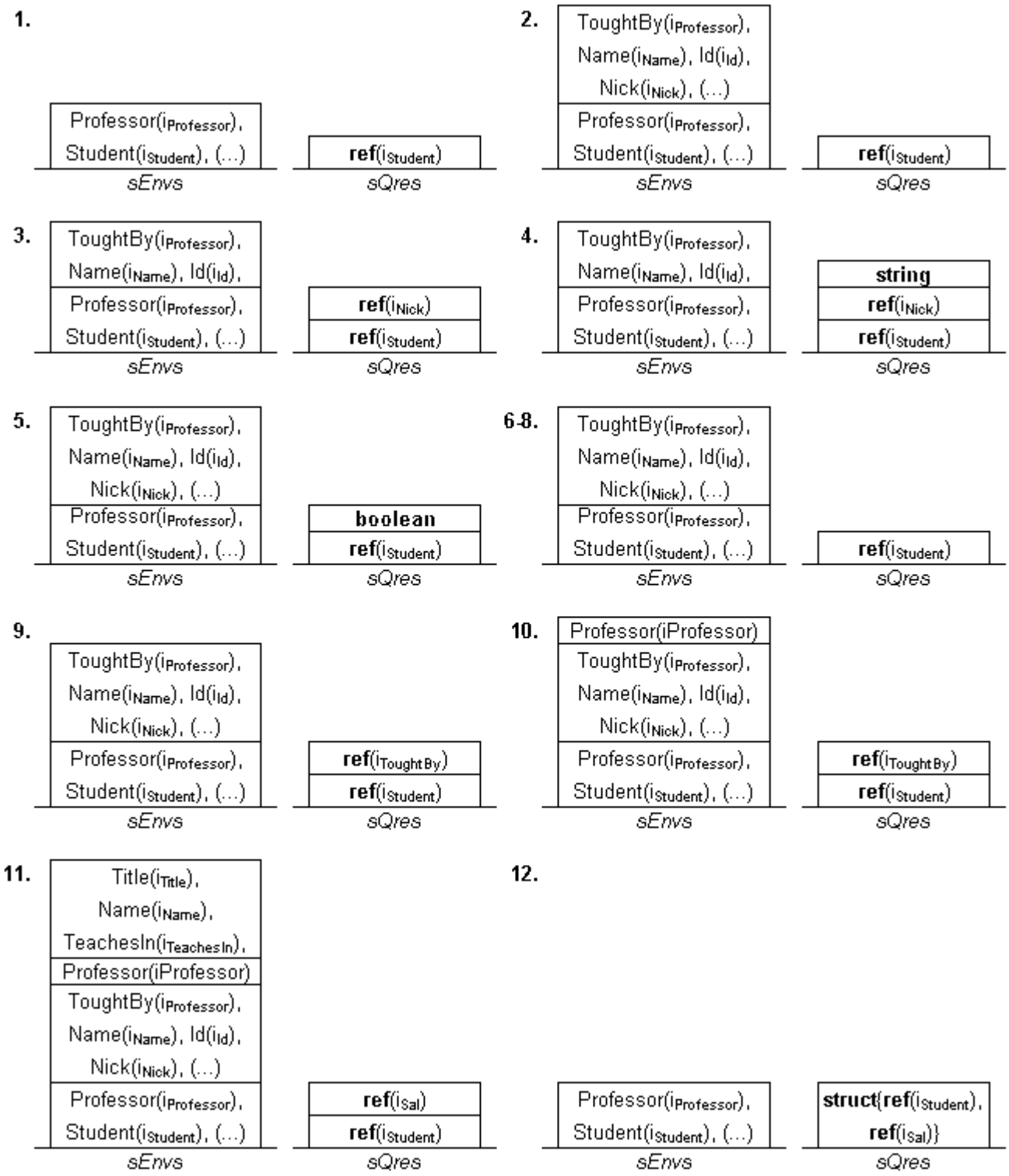
$$(\ldots)\ \mathbf{join}\ (ToughtBy.\mathbf{Professor.}Sal) \tag{1.8}$$

12. Po analizie obu argumentów wracamy do operatora **join**. Na podstawie tablicy decyzyjnej argumenty zostają uznane za poprawne i konstruowana jest wynikowa sygnatura struktury złożonej z dwóch argumentów (sygnatur z wierzchu stostu *sQres*). Ma ona postać:

$$\mathbf{struct}[0..*,\ \mathrm{bag}]\{\mathbf{ref}(i_{Student})[1..1],\ \mathbf{ref}(i_{Sal})[1..1]\}$$

Statyczna kontrola typologiczna zapytania została więc pomyślnie zakończona. Pamiętajmy jednak, że dodana została koercja dynamiczna **element()**, zatem pełna poprawność zostanie zweryfikowana dopiero podczas ewaluacji zapytania. Do aparatu wykonawczego przekazywane jest zmodyfikowane drzewo składni abstrakcyjnej zapytania:

$$\begin{array}{cc} (Student & \mathbf{where}\quad \mathbf{element}(\mathbf{deref}(Nick)) = \text{"}Wscibski\text{"}) \\ & \mathbf{join}\quad (ToughtBy.\mathbf{Professor.}Sal) \end{array}$$



Rysunek 1.3: Stany stosów po każdym kroku kontroli typologicznej przykładowego zapytania.

Rozdział 2

Realizacja schematu danych

2.1. Przechowywanie danych

Metabaza przechowywana jest w taki sam sposób jak pozostałe dane. Jest zbiorem obiektów złożonych o ściśle określonym formacie, który umożliwia systemowi wczytanie schematu danych do odpowiednich struktur w pamięci podręcznej. Format metabazy w LoXiM opracowaliśmy już wcześniej na potrzeby statycznej ewaluacji zapytań. Został opisany w [Sitek07]. Odpowiednie struktury danych również zostały wtedy zaimplementowane. W ramach wdrażania systemu kontroli typów stanąłem jednak przed koniecznością ich rozbudowy, aby uwzględniały nie tylko bezpośrednie deklaracje obiektów, ale również definicje typów nazwanych oraz deklaracje obiektów za ich pomocą. Dodałem również fragment nowej składni, który pozwala w prosty sposób deklarować schemat danych, pilnując jednocześnie odpowiednich więzów spójności. W dodatku A znajduje się przykład schematu zdefiniowanego za pomocą nowej składni.

Poniżej prezentuję format danych, nadany obiektom metabazy w modelu *M0*. Format ten różni się w zależności od tego, jaki rodzaj obiektu jest definiowany (czy jest to obiekt atomowy, wskaźnikowy, złożony, czy zadeklarowany przy użyciu typu nazwanego). Deklaracje obiektów atomowych, referencyjnych oraz złożonych mają niezmieniony format w stosunku do tego opisanego w [Sitek07], natomiast format definicji typów nazwanych oraz obiektów zadeklarowanych przy pomocy takich typów został dodany przeze mnie na potrzeby modułu kontroli typów.

Każdy element metabazy - zarówno definicja typu jak i deklaracja obiektu - jest obiektem złożonym o ustalonej nazwie (*__MDNT__* dla typów nazwanych oraz *__MDN__* w pozostałych przypadkach) i ustalonych podobiektach. Wspólne podobiekty, występujące w każdym elemencie schematu danych, to obiekty atomowe o nazwach: *name*, czyli nazwa opisywanego obiektu oraz *kind* - rodzaj obiektu. *kind* przyjmuje wartości: *atomic*, *link*, *complex* oraz *defined_type*, które oznaczają odpowiednio obiekty atomowe, wskaźnikowe, złożone oraz zdefiniowane poprzez typ nazwany. Deklaracje obiektów zawierają również podobiekt *card* - oznaczający liczebność i przyjmujący takie wartości, jak atrybut *card* w sygnaturach, opisany w sekcji 1.2. Deklaracje obiektów innych niż korzeniowe zawierają dodatkowo podobiekt referencyjny o nazwie *owner*, wskazujący nadobiekt, czyli taki obiekt złożony, dla którego dany obiekt jest podobiektem. Ponadto każdy rodzaj definiowanego bytu charakteryzują dodatkowe podobiekty.

- **Obiekt atomowy** jest definiowany przez podobiekt *type* określający konkretny typ

atomowy. Przyjmuje wartości *string*, *int*, *double*. Przyjrzyjmy się przykładowej deklaracji obiektu atomowego:

```
<id1, __MDN__ ,
  {
  <id2, name, "Location">        /* nazwa opisywanego obiektu */
  <id3, kind, "atomic">          /* rodzaj obiektu - atomowy */
  <id4, type, "string">          /* typ - napis */
  <id5, card, "1..1">            /* pojedynczy obiekt */
  <id6, owner, id18>             /* ref. do nadobiektu */
  }
>
```

- **Obiekt wskaźnikowy** jest określany przez podobiekt referencyjny *target*, zawierający identyfikator wskazywanego obiektu. Przykład deklaracji obiektu wskaźnikowego:

```
<id1, __MDN__ ,
  {
  <id2, name, "PointsTo">
  <id3, kind, "link">            /* rodzaj obiektu - wskaznikowy */
  <id4, card, "1..*">
  <id5, target, id20>            /* referencja do wskazywanego obiektu. */
  <id6, owner, id18>
  }
>
```

- **Obiekt złożony** jest opisany rekurencyjnie. Jego deklaracja zawiera listę podobiektów referencyjnych o nazwie *subobject*, z których każdy odpowiada jednemu z podobiektów danego obiektu złożonego. Wskazywane elementy są zatem także deklaracjami obiektów, zbudowanymi zgodnie z opisywanym formatem.

```
<id1, __MDN__ ,
  {
  <id2, name, "Student">          /* nazwa opisywanego obiektu */
  <id3, kind, "complex">         /* rodzaj obiektu - zlozony */
  <id4, card, "0..*">            /* liczebność */
  <id6, owner, id18>             /* ref. do nadobiektu*/
      <id6, subobject, id30>     /* referencje do podobiektow */
      <id7, subobject, id36>     /* ... */
      <id8, subobject, id22>
      ...
  }
>
```

- **Obiekt zadeklarowany za pomocą typu nazwanego** zawiera nazwę tego typu w podobiekcie *typeName*. Dzięki unikalności nazw typów (por. rozdział 2.2, pkt. 1) taka informacja jest wystarczająca, aby odtworzyć budowę danego obiektu. Przykładowa deklaracja:

```
<id1, __MDN__ ,
  {
  <id2, name, "Address">          /* nazwa opisywanego obiektu */
  <id3, kind, "defined_type">    /* obiekt opisany przez typ nazwany */
  <id4, typeName, "AdressType">  /* nazwa typu, który opisuje dany obiekt */
  <id5, card, "1..1">            /* liczebnosc */
  <id6, owner, id18>             /* ref. do nadobiektu */
  }
>
```

- **Typ nazwany**
  Definicje typów są trzymane w bazie w niemal identyczny sposób, jak deklaracje obiektów. Przyjmują więc jedną z czterech postaci wymienionych powyżej. Jedyne różnice to brak podobiektu *card* oraz obecność nowego podobiektu *isDistinct* określającego, czy do danego typu stosuje się zgodność nazwową. Dodatkowo nazwa obiektów definiujących typy różni się od odpowiedniej nazwy dla deklaracji obiektów. (*__MDNT__* zamiast *__MDN__*);

```
<id1, __MDNT__ ,
  { ...
    ...
   <id5, isDistinct, "1">        /* flaga typu rozróżnionego */
  }
>
```

2.2. Spójność schematu

Poprawny schemat danych musi spełniać określone warunki, aby aparat kontroli typów mógł z niego korzystać. Dotychczas nie istniały żadne mechanizmy, które tego pilnowały. Poniżej opisuję jak je zrealizowałem. Zwróćmy uwagę na fakt, że część kontroli poprawności schematu jest przeprowadzana podczas jego deklaracji, aby uniemożliwić rozspójnienie schematu przy jego tworzeniu. Niektóre sprawdzenia jednak są wykonywane dopiero przy inicjacji odpowiednich struktur danych realizujących metabazę w pamięci. Takie podejście wynika między innymi z konieczności dopuszczenia wzajemnych odwołań referencyjnych.

1. **Unikalne nazwy**

   W systemie LoXiM przyjęliśmy, że nazwy typów oraz obiektów w bazie danych muszą być unikalne. Dopuszczenie powtarzalnych nazw znacznie utrudniłoby proces kontroli typologicznej. Przy deklarowaniu dowolnego obiektu lub definiowaniu typu istnieje zatem potrzeba sprawdzenia, że jego nazwa nie występuje już w metabazie jako nazwa typu nazwanego lub obiektu korzeniowego.

   Jest to zrealizowane za pomocą prostego wektora wykorzystanych nazw, trzymanego w podręcznych strukturach metabazy w pamięci. Ta kolekcja jest sprawdzana i uzupełniana przy każdej nowej deklaracji.

2. **Wzajemne odwołania typów**

Dowolny obiekt możemy zdefiniować poprzez pewien typ nazwany. Podobnie typ nazwany - może odwoływać się do innego typu. Występuje tu zagrożenie zapętlenia definicji, weźmy przykład:

```
typedef A = (atr_1: string, atr_2 : B);
typedef B = (atr_3: string, atr_4 : A);
```

Taką sytuację uznajemy za nieprawidłową, ponieważ próba rozwinięcia powyższych definicji prowadzi do nieskończonej rekurencji. Należy zatem zabronić tego typu wzajemnych odwołań typów.

Odpowiednią kontrolę zapewnia graf zależności między typami, zrealizowana jako słownik (obiekt klasy *std::map*), który każdej nazwie typu przypisuje kolekcję typów bezpośrednio od niego zależnych. Jest on uaktualniany przy każdej definicji typu nazwanego. Proste przeszukiwanie grafu w poszukiwaniu potencjalnej pętli pozwala zwrócić błąd deklaracji przy próbie zdefiniowania wzajemnie odwołujących się do siebie typów.

3. **Wzajemne odwołania referencyjne**

Należy odróżnić odwołania opisane powyżej od referencji do obiektów bądź innych typów. W przypadku referencji - pętle są dozwolone i nie prowadzą do nieskończonej rekurencji. Zabronienie takich konstrukcji byłoby niedopuszczalnym ograniczeniem. Uniemożliwiałoby tworzenie wielu struktur takich jak listy wskaźnikowe. Dopuszczamy zatem takie deklaracje. Można je znaleźć w przykładowym schemacie danych w dodatku A. Poniżej fragmenty odpowiednich konstrukcji:

```
Student : (Name : string, (...), Friend[0..*] : ref Student);
typedef StListType = (Id : integer, next[0..1] : ref StListType);
```

Pierwsza konstrukcja nie powinna budzić wątpliwości, nie może tu być bowiem mowy o nieskończonej rekurencji. W sygnaturze obiektu *Student* podobiekt *Friend* będzie zwykłą sygnaturą referencyjną, wskazującą identyfikator $i_{Student}$. W przypadku drugiej konstrukcji jednak mogą rodzić się zastrzeżenia - skoro typ nazwany traktujemy jak rozwijalne makro, taką sytuację należałoby uznać za niepoprawną.

Aby rozwiązać problem wzajemnych odwołań typów referencyjnych, przyjąłem, że jest on traktowany bardzo podobnie jak referencja do obiektu. Zatem w sygnaturze dowolnego obiektu zadeklarowanego za pomocą typu *StListType* podobiekt *next* jest również zwykłą sygnaturą referencyjną. Identyfikator celu to w tym przypadku $i_{StListType}$, czyli identyfikator typu. Na etapie deklaracji schematu jest to bardzo wygodne, ponieważ rozwiązuje problem wzajemnych odwołań. Podczas kontroli zapytania natomiast rodzi się potrzeba rozróżniania sygnatur referencyjnych wskazujących na typy od tych wskazujących na obiekty. Okazuje się, że jest to dość łatwo osiągalne, ponieważ dla dowolnego identyfikatora wierzchołka metabazy potrafimy powiedzieć, czy jest to obiekt korzeniowy, zagnieżdżony, czy może właśnie typ nazwany.

4. **Kompletność metabazy**

Kiedy w deklaracji obiektu lub definicji typu pojawia się odwołanie do innego bytu, nie wymaga się wcześniejszego zadeklarowania tego bytu. Takie wymaganie utrudniałoby

pracę programisty, a co gorsze - uniemożliwiałoby wzajemne odwołania referencyjne, opisane powyżej. Skoro jednak takiego wymagania nie ma, powstaje problem pełności metabazy, czyli zapewnienia, że w momencie korzystania ze schematu danych przez aparat kontroli typów jest on kompletny, czyli nie istnieją puste odwołania (do nieistniejących typów lub obiektów).

Struktury metabazy w pamięci mają dwie flagi: aktualności oraz kompletności. Kiedy flaga aktualności ma wartość negatywną, oznacza to, że zostały dodane do schematu danych pewne deklaracje, które mogły zaburzyć jego kompletność i należy ją ponownie zweryfikować, aby poprawność typologiczna dowolnego zapytania mogła zostać sprawdzona. Flaga kompletności określa, czy w danym momencie schemat danych jest kompletny i jest uaktualniana przy odświeżaniu podręcznych struktur metabazy. W tym celu trzymany jest słownik brakujących obiektów i typów. Każdej nazwie brakującego bytu przypisany jest jego rodzaj (obecnie rodzaj wskazuje jedną z dwóch możliwości: *typ nazwany* albo *dowolny obiekt*). Takie rozróżnienie jest niezbędne, aby móc zapobiec takim sytuacjom:

```
typedef A = (atr_1: string, atr_2 : B);
B: string;
```

Zauważmy, że po pierwszej deklaracji schemat danych staje się niekompletny, brakuje bytu o nazwie B. Jednak odwołanie do niego wskazuje, że musi to być typ nazwany, a nie obiekt. Dlatego Następna deklaracja jest nieprawidłowa. Takie błędy są wykrywane przy deklaracji właśnie dzięki wyżej opisanemu słownikowi.

Warto wspomnieć w tym miejscu, że kompletność metabazy nie jest sprawdzana i uaktualniana po każdej deklaracji. Istnieje specjalne zapytanie (polecenie) *reloadScheme*, które nakazuje, aby odświeżyć te struktury, a tym samym zweryfikować pełność metabazy.

Rozdział 3

Implementacja w LoXiM

Wdrażając w LoXiM kontrolę typologiczną stworzyłem nowy moduł *TypeCheck*, w którym zamyka się większość elementów, które składają się na system kontroli. Tutaj znajduje się implementacja tablic decyzyjnych i reguł, całego algorytmu *typeCheck*, w tym również algorytmu odbudowy procesu po błędzie. Poza modułem znalazły się elementy kontroli dynamicznej. Jako właściwe aparatowi wykonawczemu, są zaimplementowane w module *QueryExecutor*. Niektóre struktury, z których korzysta mechanizm kontroli typów, były już wcześniej zrealizowane na potrzeby statycznej ewaluacji zapytań, wykorzystywanej przy optymalizacji zapytań. Do tych struktur zaliczam statyczne stosy (środowisk i wyników), sygnatury, schemat danych oraz drzewa składni abstrakcyjnej. Schemat danych został opisany w poprzednim rozdziale, poniżej przedstawiam pozostałe elementy. Stosy nie wymagały praktycznie żadnych zmian, podobnie drzewa syntaktyczne zapytań - tu zmiany były znikome, dlatego skoncentruję się przede wszystkim na sygnaturach, tablicach decyzyjnych i realizacji samego algorytmu kontroli.

3.1. Stosy statyczne

Stos środowisk

Statyczny stos środowisk realizowany jest przez klasę *StatEnvStack*. Elementami tego stosu są sekcje, które z kolei są kolekcjami binderów statycznych. Sekcji odpowiada klasa *StatEnvSection*, zaś kolekcji binderów - klasa *BinderWrap*. Niestety projektując struktury stosów popełniliśmy błędy, które zaowocowały dość skomplikowanymi konstrukcjami, trudnymi do opanowania i rozwijania. Elementy stosu (sekcje) tworzą listę łączoną, podobnie jak kolekcje binderów w sekcjach. Co więcej, typowe operacje stosowe zostały zaimplementowane w tych samych klasach, które realizują metody charakterystyczne dla statycznej ewaluacji lub kontroli typologicznej zapytania. Należało skorzystać z gotowych rozwiązań zapewniających funkcjonalność stosu, natomiast implementację operacji takich jak wiązanie nazw, lub rozwijanie elips oddzielić, stosując rozsądne wzorce projektowe (np. dekorator). Powyższe uwagi dotyczą implementacji obu statycznych stosów (środowisk oraz wyników), a rozszerzoną dyskusję na ten temat można znaleźć w [Sitek07]. Poza zwykłymi operacjami stosowymi klasa *StatEnvStack* udostępnia metody *bindName(**string** name)* (wiązanie nazwy *name* na stosie), oraz *bindNameEllipsis(**string** name)* (próba rozwinięcia elipsy).

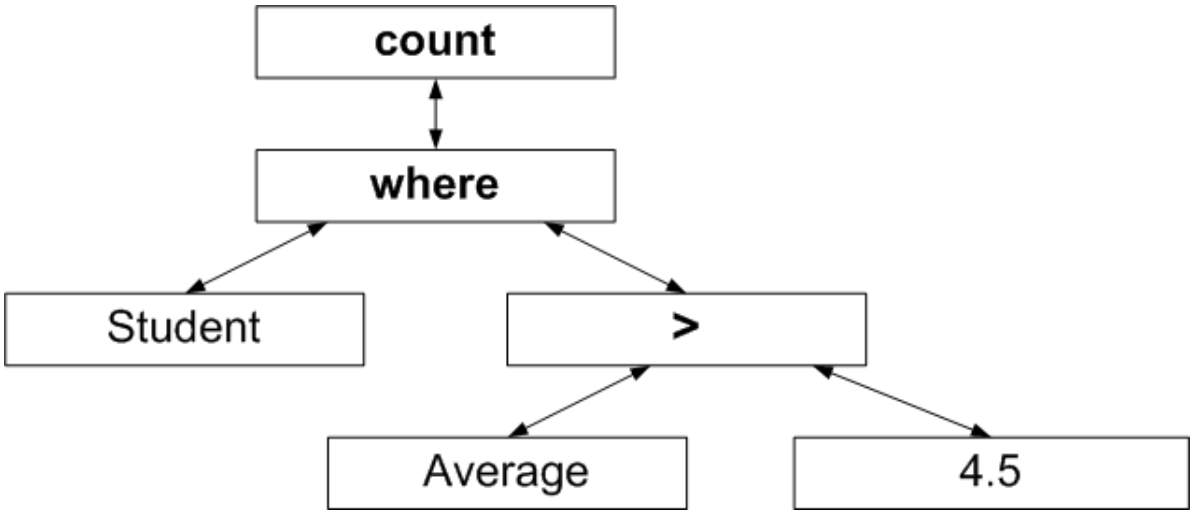
Stos wyników

Statyczny stos wyników realizuje klasa *StatQResStack*, a jego elementami są opisane niżej sygnatury - elementy jednej z podklas klasy *Signature*. Podobnie jak stos środowisk, również *StatQResStack* został zaimplementowany jako lista wskaźnikowa elementów. Stanowi on prostszą konstrukcję niż *StatEnvsStack*, gdyż udostępnia jedynie standardowy interfejs stosu o elementach *Signature*.

3.2. Drzewa składni abstrakcyjnej

Analizator składniowy (realizowany przez moduł *QueryParser*) przekształca tekstową reprezentację zapytania do postaci drzewa składni abstrakcyjnej. Dopiero na takiej strukturze operuje aparat kontroli typów, jak również optymalizator oraz sam aparat wykonawczy. Rysunek 3.1 przedstawia drzewo przykładowego zapytania zliczającego studentów o wysokiej średniej:

$$count(Student\ \textbf{where}\ Average > 4.5) \tag{3.1}$$



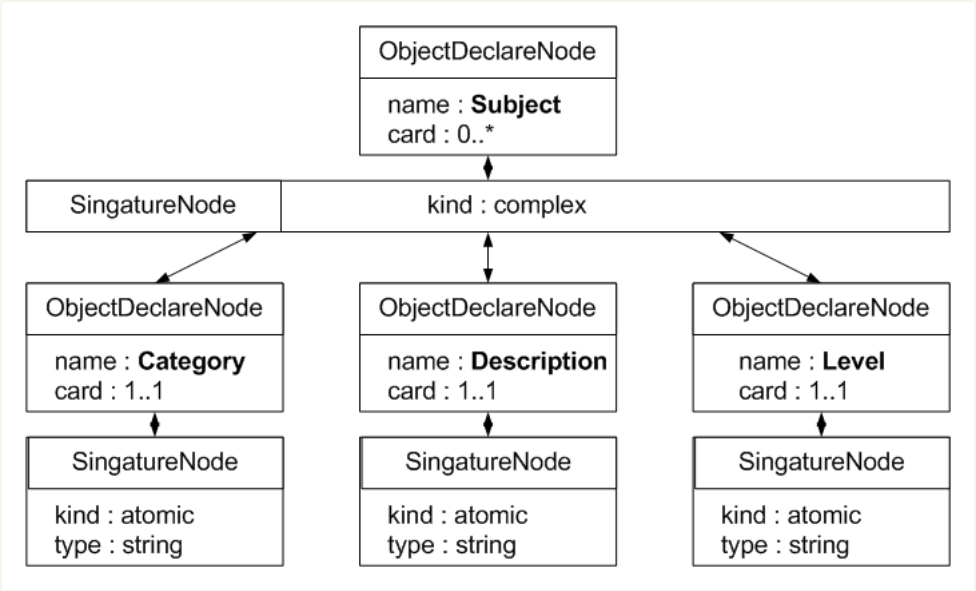
Rysunek 3.1: Drzewo składni abstrakcyjnej zapytania 3.1

Drzewa składni abstrakcyjnej są realizowane przez hierarchię klas o wspólnej abstrakcyjnej nadklasie *TreeNode*. Poszczególne podklasy implementują węzły odpowiadające operatorom SBQL. Niektóre węzły są dodatkowo parametryzowane i ich klasy odpowiadają całym grupom operatorów, np. klasa *NonAlgOpNode* realizuje wszystkie operatory niealgebraiczne SBQL. Powiązania między węzłami drzewa są realizowane poprzez wskaźniki do poddrzew. Liście, czyli np. węzły odpowiadające literałom lub nazwom (*ValueNode*, *NameNode*) nie posiadają oczywiście takich powiązań, natomiast węzły operatorów, w zależności od liczby argumentów operatora, mają jedno poddrzewo (wtedy dziedziczą po *UnOpNode*), dwa (podklasy *TwoArgsNode*), lub całą kolekcję (*VectorNode*). Wśród atrybutów *TreeNode*, dostępnych w każdej z podklas jest również wskaźnik do "nadwęzła" - rodzica w drzewie zapytania (pusty tylko w korzeniu). Każda podklasa *TreeNode* implementuje wirtualną metodę ***int*** *type()*, która pozwala zidentyfikować daną podklasę, oraz wybrany zestaw z innych metod, wprowadzonych na potrzeby poszczególnych modułów. Dzięki polimorfizmowi takie rozwiązanie pozwala w prosty sposób korzystać z funkcjonalności udostępnianej przez obiekty *TreeNode*, jednak niesie za sobą również niekorzystne konsekwencje. Węzły drzewa syntaktycznego implementują

funkcjonalności typowe dla różnych modułów, co czyni kod mało przejrzystym i utrudnia jego rozwijanie. Można by temu zapobiec stosując typowy dla takich konstrukcji wzorzec *Wizytor*, umożliwiający przechodzenie drzewa zapytania i wykonywanie na nim dowolnych operacji charakterystycznych dla wybranego modułu.

W ramach realizacji modułu kontroli typów rozszerzyłem istniejącą hierarchię o kilka nowych klas. Mechanizm półmocnej kontroli wymaga wzbogacanie drzewa zapytania o różne koercje, dlatego stworzyłem nową klasę *CoerceNode*, parametryzowaną rodzajem koercji. Koercje można uznać za nową klasę operatorów jednoargumentowych - wymagają atrybutu wskazującego podzapytanie, wokół którego koercja została dodana. Wdrażając kontrolę typów dodałem również nowe fragmenty składni do języka, pozwalające na deklaracje obiektów i definicje typów nazwanych. To wymagało oczywiście dodania nowych rodzajów węzłów, stąd hierarchia podklas *TreeNode* wzbogaciła się o następujące klasy: *SignatureNode*, *StructureTypeNode*, *DeclareDefineNode*, *ObjectDeclareNode*, *TypeDefNode*. Poniżej przykład deklaracji z dodatku A oraz odpowiadające mu drzewo syntaktyczne, zbudowane z obiektów nowych klas (rys. 3.2):

```
Subject[0..*]: (Category: string, Description: string, Level: integer);
```



Rysunek 3.2: Schemat drzewa syntaktycznego przykładowej deklaracji obiektu.

3.3. Sygnatury

Sygnatury w LoXiM zostały zaimplementowane już wcześniej, wymagały kilku dodatkowych elementów związanych z modułem *TypeCheck*. Każdemu rodzajowi sygnatury odpowiada pewna podklasa abstrakcyjnej klasy *Signature*, a publiczna metoda ***int*** *type()* pozwala je rozróżnić. We wspólnej nadklasie znajdują się między innymi atrybuty sygnatur takie jak liczebność (*card*), nazwa typu (*typeName*), rodzaj kolekcji (*collectionKind*), flaga zakazu dereferencji (*ref*). Odpowiednie podklasy *Signature*:

SigAtomType Odpowiada sygnaturom elementarnym. Atrybut *atomType* określa konkretny rodzaj sygnatury (czy jest to napis, czy liczba, czy np. wartość logiczna).

SigRef Sygnatura wskaźnikowa, jednym z elementów tej klasy jest *refNo* - identyfikator wskazywanego wierzchołka schematu danych.

SigBinder Klasa bindera statycznego, zawiera atrybuty: *name* (nazwa bindera), oraz *value* (sygnaturę, której ta nazwa została nadana).

SigVoid Oznacza pustą sygnaturę, wprowadzona na potrzeby operatorów imperatywnych, aby mogły zwracać pusty wynik.

SigStruct Klasa modelująca sygnaturę struktury. Głównym atrybutem tej klasy jest kolekcja sygnatur elementów struktury. Niestety została zaimplementowana przez nas na początku istnienia systemu, jako lista wskaźnikowa. Był to nasz błąd projektowy, który potęgował przy operowaniu na tych strukturach. Nie ma to przełożenia na czas wykonywanych operacji, ponieważ te listy są zawsze bardzo krótkie, a większość operacji i tak wymaga przejrzenia całej kolekcji. Niemniej jednak programista, który korzysta z tych struktur, ma utrudnione zadanie. Przy dalszym rozwijaniu modułu warto rozważyć reimplementację tej klasy (podobnie jak klas stosów statycznych, które również zostały zrealizowane za pomocą list wskaźnikowych).

SigVariant Klasa sygnatury wariantowej. Budową bardzo przypomina wyżej opisaną klasę *SigStruct* - strukturę cechuje lista jej elementów (sygnatur), natomiast do wariantu przypisana jest lista sygnatur, jakie może on przyjąć.

Metody klasy *Signature* (przedefiniowane w podklasach), które zasługują na uwagę to:

- `virtual Signature *deref();`

  Zwraca wynik dereferencji danej sygnatury. W przypadku klas *SigStruct* i *SigVariant* sprowadza się do prostego wywołania rekurencyjnego, natomiast dla sygnatury referencyjnej *SigRef* - metoda ta sięga do metabazy, aby skonstruować sygnaturę obiektu o wskazywanym identyfikatorze *refNo*.

- `virtual bool isStructurallyEqualTo(Signature *sig)`

  Ta metoda bada strukturalną zgodność podstaw dwóch sygnatur, ich atrybuty (np. *card*, *typeName*) nie są uwzględniane. Metoda sprawdza, że są to sygnatury tego samego rodzaju (określonego przez *type*()), oraz nakłada różne dodatkowe warunki w każdej z podklas. Typy elementarne muszą być takie same (pole *atomType*), w sygnaturach referencyjnych muszą zgadzać się identyfikatory obiektu wskazywanego, natomiast w przypadku *SigBinder* oraz *SigStruct* sprawdzenie jest rekurencyjne.

- `virtual int compareNamedSigCrt(Signature *flatSig, bool needTName);`

  Jest to metoda bada zgodność sygnatur, jaka jest potrzebna przy kontroli operatorów **create**, **insert** oraz **assign**. Parametr *needTName* określa, czy powinna być badana zgodność nazwowa. Ta metoda stanowi rygorystyczne sprawdzenie zgodności strukturalnej. Jest niesymetryczna w przypadku sygnatur strukturalnych, ponieważ fakt, że sygnaturę S_1 można np. przypisać na sygnaturę S_2 nie oznacza, że odwrotna operacja jest poprawna (jest to związane z liczebnościami podobiektów). Warto omówić realizację tej metody dla struktur oraz referencji - w pozostałych przypadkach są to proste sprawdzenia, podobne jak przy badaniu zgodności strukturalnej. Sygnatury referencyjne są interesujące ze względu na istnienie zarówno referencji do typów, jak i referencji do obiektów, oraz na konieczność porównywania ich. O rozwiązaniu tej kwestii piszę w rozdziale 3.6. Poniżej opisuję realizację tej metody w klasie *SigStruct*

Niesymetryczne przyrównywanie struktur

Metoda ta, wywołana na sygnaturze *Sig*, ma sprawdzić, czy można na nią przypisać drugą sygnaturę *flatSig*. Jest również wykorzystywana przy wstawianiu złożonych obiektów oraz przy ich tworzeniu. Zwracana wartość to 0, w przypadku pozytywnego wyniku, lub odpowiedni kod błędu w przeciwnym wypadku. Jest jednak jeszcze trzecia możliwość - sygnatury mogą do siebie "pasować" w sensie statycznym, natomiast wymagać dodatkowej kontroli dynamicznej. W takim przypadku zwracany jest szczególny, wyróżniony kod błędu *ESigCdDynamic*. Przykład takiej sytuacji można znaleźć w dodatku B (zapytanie 15).

Przebieg metody:

- Sprawdzamy podstawową zgodność typów (czy *flatSig* jest też strukturą), oraz - o ile na to wskazuje parametr *needTName* - zgodność nazwowa.
- Inicjujemy słownik *subMap*, który każdej nazwie podobiektu przypisuje sumaryczną dolną i górną granicę liczebności.
- W pętli przebiegamy wszystkie podobiekty *flatSig*. Dla każdego podobiektu *subObject* wykonujemy następujące kroki:
  - Wyszukujemy w bazowej sygnaturze *Sig* podobiektu o tej samej nazwie. W przypadku braku takiego podobiektu - zgłaszany jest błąd (niezgodność sygnatur).
  - rekurencyjnie wywołujemy metodę na znalezionym obiekcie, podając *subObject* jako parametr.
  - Jeśli rekurencyjne porównanie zwróciło błąd inny niż *ESigCdDynamic*, zwracamy ten błąd.
  - Wyszukujemy w *subMap* nazwy *subObject*. Jeśli taki klucz nie istnieje, wstawiamy go inicjując wartość na $(0, 0)$. Następnie uaktualniamy wartość dodając odpowiednio dolną i górną granicę liczebności podobiektu *subObject*.
- Jeśli pomyślnie przeszliśmy przez całą pętlę, oznacza to, że *flatSig* nie ma żadnych podobiektów, które nie mają swoich odpowiedników w bazowej sygnaturze *Sig*. Może się jednak zdarzyć, że pewnych podobiektów brakuje, lub jest za dużo. Do w pętli przebiegamy wszystkie podobiekty *Sig*, a dla każdego wyszukujemy w słowniku *subMap* jego nazwy i wykonujemy następujące sprawdzenia:
  - Jeśli odpowiedni wpis w słowniku nie istnieje, uzupełniamy odpowiednie miejsce zerami.
  - Jeśli dolna granica liczebności podobiektu przekracza górną granicę ze słownika, zwracamy błąd - brakuje wymaganych podobiektów.
  - Jeśli dolna granica ze słownika przekracza górną granicę podobiektu, to również zwracamy błąd - zbyt dużo podobiektów.
  - Jeśli żadna z powyższych sytuacji nie zachodzi, ale sumaryczny przedział liczebności w słowniku jest szerszy, niż ten określony dla podobiektu, zwracana jest wyróżniona wartość *ESigCdDynamic*.

3.4. Tablice Decyzyjne

Podstawowym kryterium, jakim się kierowałem przy implementacji tablic decyzyjnych, była elastyczność w ustalaniu konkretnych reguł. Uważam, że ten zamysł się udał. W rozdziale

4.1 opisuję jak korzystać z tych tablic. Poniżej przybliżam ich implementację.

Tablice decyzyjne są strukturami statycznymi. Nie zmieniają się w trakcie działania aplikacji i wystarczy, aby jedna kopia każdej tablicy była trzymana w pamięci. Dlatego zastosowałem wzorzec singletonu, stworzyłem klasę *DecisionTableHandler*, która dba o zachowanie pojedynczej kopii każdej tablicy w systemie. Dla każdego operatora zwraca odpowiadającą mu tablicę decyzyjną, inicjując ją przy pierwszym odwołaniu.

Same tablice są realizowane przez dwie klasy: *UnOpDecisionTable* oraz *DecisionTable*, odpowiednio dla operatorów jedno- oraz dwu-argumentowych, ze wspólną abstrakcyjną nadklasą *DTable*. Do każdego operatora przypisana jest jedna tablica decyzyjna. Każda zawiera kilka wektorów (standardowe klasy *std::vector*) reguł - po jednym dla każdego atrybutu sygnatury oraz jeden dla podstawy sygnatury. Taka konstrukcja obrazuje ortogonalność kontroli typów ze względu na poszczególne atrybuty. Można powiedzieć, że jest to dość wierne odzwierciedlenie logicznej idei tablicy decyzyjnej. Każdemu wierszowi odpowiada pojedyncza reguła, a zestawy wierszy dotyczące danego atrybutu tworzą wektory reguł. Spójrzmy, jak wygląda inicjacja reguł dodawania i porównajmy ten fragment kodu z odpowiednią tablicą logiczną przedstawioną w rozdziale 1.3.2:

Przykład 3.1.

```
//Operator ' + '
/*BASE*/
addTcRule(        "integer",  "integer",  "integer");
addTcRule(        "double",   "double",   "double");
addTcRule(        "string",   "string",   "string");
addTcRule(        "integer",  "double",   "double");
addTcRule(        "double",   "integer",  "double");
addTcRule(        "integer",  "string",   "integer",  BS_TO_INT, RIGHT, DYNAMIC);
addTcRule(        "string",   "integer",  "string",   BS_TO_STR, RIGHT, STATIC);
addTcRule(        "double",   "string",   "double",   BS_TO_DBL, RIGHT, DYNAMIC);
addTcRule(        "string",   "double",   "string",   BS_TO_STR, RIGHT, STATIC);
addTcRule(M_ELSE,                         TC_RS_ERROR);
/*CARD*/
addCdRule(        "1..1",     "1..1",     "1..1");
addCdRule(M_L,    "x..y",     "1..1",     "1..1",     CARD_TO_11, LEFT, DYNAMIC);
addCdRule(M_R,    "1..1",     "x..y",     "1..1",     CARD_TO_11, RIGHT, DYNAMIC);
addCdRule(M_B,    "",         "",         "1..1",     CARD_TO_11, BOTH, DYNAMIC);
/*T-NAME*/
addTnRule(M_B,    M_0,        M_0,        "");
addTnRule(M_ELSE,                         TC_RS_ERROR);
```

Odczytanie decyzji

Aparat kontroli typów korzysta z tablic decyzyjnych w jeden sposób - dla podanych sygnatur argumentów żąda podjęcia odpowiedniej decyzji. Z punktu widzenia algorytmu *typeCheck* tablice udostępniają więc jedną metodę *getResult*:

```
int getResult(TypeCheckResult &finalResult, Signature *lSig, Signature *rSig);
```

Pierwszy parametr, przekazany przez referencję służy do zebrania informacji o podjętej decyzji, kolejne dwa to sygnatury argumentów, dla których te decyzję należy podjąć. W przypadku tablic operatorów unarnych jest więc o jeden parametr mniej.

W ramach tej metody wszystkie wektory reguł zostają "zaaplikowane" do podanych argumentów, zaczynając od reguł dotyczących podstaw sygnatur. Jeśli wynikiem jest błąd metoda zostaje zakończona, w przeciwnym wypadku - aplikowane są kolejne wektory. Rezultaty są zapisywane w przekazywanym parametrze *finalResult* klasy *TypeCheckResult* opisanej poniżej, w sekcji 3.4.2. Aplikacja wektora reguł polega na odnalezieniu reguły pasującej do podanych argumentów oraz zastosowaniu jej. Są one przeszukiwane zgodnie z kolejnością, w jakiej występują w wektorze, która jest z kolei zachowywana przy inicjacji tablic i wywoływaniu odpowiednich metod *addRule*. Jest to dość istotna własność - w tablicach niektórych operatorów kolejność stosowania reguł ma duże znaczenie. Pozostaje określić, czym są reguły, jak dopasowujemy je do argumentów oraz w jaki sposób rejestrowana jest decyzja. Temu poświęcone są kolejne dwie sekcje.

3.4.1. Reguły

Reguły są realizowane przez klasę *Rule* i jej podklasy *UnOpRule* oraz *TCRule*, odpowiednio dla unarnych i binarnych operatorów. Wśród atrybutów tych klas występują:

specialArg Parametr sterujący dopasowaniem reguły do argumentów, dzięki któremu mogą funkcjonować metareguły. Wskazuje, który z argumentów reguły jest metazmienną, nakazuje równość argumentów lub ich brak. Przyjmowane wartości: *M_L*, *M_R*, *M_B* - wskazany argument (odpowiednio lewy, prawy, oba) jest metazmienną, *M_EQ* - zgodne argumenty, *M_EX* - przyrównanie do któregokolwiek argumentu (tzn. reguła będzie dopasowana, o ile przynajmniej jeden z argumentów jest równy wartości *leftArg*, opisanej niżej). *specialArg* może również przyjąć wartość *M_ELSE*, która dopasowuje daną regułę do dowolnego zestawu argumentów.

result Wynik. Może określać wynikową sygnaturę, lub przyjąć jedną z wyróżnionych wartości, wskazujących błąd kontroli lub nakazanie koercji.

sigGen, attrGen Identyfikatory generatorów wyniku, dla podstawy sygnatury lub dla dowolnego atrybutu. Generatory stosowane są tam, gdzie wynik nie jest sztywno określony, lecz należy go skonstruować na podstawie argumentów.

action Identyfikator wybranej koercji (o ile dana reguła zaleca jakąkolwiek koercję).

leftArg, rightArg Wartości lub metazmienne, do jakich będą porównywane argumenty sygnatur wejściowych. Czy traktować je jako metazmienne, czy próbować dokładnie dopasować - o tym decyduje opisane wyżej pole *specialArg*.

Dopasowanie reguły do danych sygnatur polega na przyrównaniu ich podstaw lub atrybutów do argumentów reguły (*leftArg* oraz *rightArg*), uwzględniając parametr sterujący *specialArg*. Odpowiada za to metoda *appliesTo*(*string lArg*, *string rArg*). W przypadku pozytywnego dopasowania generowana jest decyzja, czyli wynik zastosowania reguły do sygnatur wejściowych. Wynik może być jawnie wskazany w regule, lecz może też wymagać tzw. generatora. Każdy taki generator to metoda zaimplementowana w jednej z podklas *DTable*, zwracająca wynikową sygnaturę lub jej atrybut na podstawie argumentów, której został przypisany pewien identyfikator. Dzięki temu reguła znając identyfikator generatora, może skonstruować wynik.

Przyjrzyjmy się wybranej regule z przykładu 3.1:

```
addCdRule(M_R,   "1..1",   "x..y",   "1..1",   CARD_TO_11, RIGHT, DYNAMIC);
```

Wywoływany jest tu konstruktor reguły, która zostanie dodana do reguł liczebności danej tablicy decyzyjnej. Pierwszy parametr (*M_R*, jak *META_RIGHT*) definiuje atrybut *specialArg*, dwa następne to kolejno *leftArg* oraz *rightArg*. Parametr sterujący *M_R* wskazuje, że lewy argument należy dopasować dokładnie, natomiast prawy jest tylko symbolem formalnym. Czwarty parametr to wynik (*result*) - jeśli ta reguła zostanie pomyślnie dopasowana, zwróci w wyniku *1..1*. Atrybut *sigGen* pozostanie pusty, gdyż tworzona reguła dotyczy liczebności, a nie podstawy sygnatury. Również *attrGen* nie zostanie jednak wykorzystany, ponieważ nie potrzebny nam w tym wypadku generator wyniku, skoro wynik można jawnie wskazać. Ostatnie trzy parametry powyższej metody dotyczą zalecanej koercji. *CARD_TO_11* to identyfikator akcji **element()** sprawdzającej, że dana kolekcja ma jeden element. Kolejne dwa to elementy pomocnicze, pierwszy nakazuje zastosować koercję tylko do prawego podzapytania, zaś drugi zaznacza, że kontrola przenoszona jest do czasu wykonania.

3.4.2. Decyzje

Klasa *TypeCheckResult* służy do konstruowania i przechowywania decyzji dotyczącej wyniku kontroli dowolnego operatora. Obiekt tej klasy przekazywany jest przez referencję do metody *getResult* w dowolnej tablicy decyzyjnej. Aby go właściwie uzupełnić, każdy zestaw reguł "dodaje" swój wynik. W ramach decyzji (obiektu klasy *TypeCheckResult*) zapisywany jest rodzaj całościowego rezultatu kontroli danego operatora (jedna z trzech wartości: *SUCCESS*, *ERROR* lub *COERCE*) oraz - w przypadku sukcesu lub zalecenia koercji - wynikowa sygnatura. W pierwszej kolejności aplikowane są reguły dotyczące podstawy sygnatury. Wtedy zostaje utworzona sygnatura wynikowa, jako obiekt odpowiedniej podklasy *Signature*. Aplikacje kolejnych zestawów reguł uzupełniają kolejne atrybuty tej sygnatury.

Zauważmy, że każdy atrybut może zawierać osobne zalecenie koercji, lub zgłosić błąd. Żeby nie stracić żadnej informacji, obiekty klasy *TypeCheckResult* przechowują zestaw atrybutów, które spowodowały błąd (*vector¡string¿ errorParts*) oraz zestaw zaleconych koercji (*vector¡ActionStruct¿ actionIds*). Koercje zapamiętywane są w postaci struktury *ActionStruct*, która przechowuje identyfikator koercji (podobnie jak generatory wyników w tablicach decyzyjnych, tak wszystkie koercje są "rejestrowane" w aparacie kontroli typów i mają swoje identyfikatory) oraz specjalny parametr. Określa on, które argumenty operatora (prawy, lewy, czy może oba) powinny podlegać danej koercji. Może też przyjąć wyróżnioną wartość *MARK_NODE*, która nakazuje nie dodawać nowego węzła koercji nad żadnym z zapytań, ale oflagować węzeł samego identyfikatora. Dotyczy to nielicznych operatorów (np. operator wstawiania, przypisania), które należy potraktować jako wyjątkowe. Uzasadnienie takiego rozwiązania można znaleźć w rozdziale 3.6, w sekcji **Koercje, kontrola dynamiczna**. Na potrzeby tych kilku operatorów zapisywana jest również sygnatura koercji. Jest to sygnatura, która będzie przekazana aparatowi wykonawczemu, aby mógł zbadać zgodność otrzymanych rezultatów z ich deklaracją w schemacie danych.

3.5. Algorytm typeCheck

Algorytm realizowany jest przez rekurencyjną metodę *typeCheck* zaimplementowaną w klasie *TypeChecker*. Ta klasa odpowiada całemu aparatowi kontroli typów - jej atrybutami są

wszystkie niezbędne struktury statyczne oraz drzewo zapytania. Do tablic decyzyjnych dostęp jest poprzez *DecisionTableHandler*, jak opisałem w sekcji 3.4. Metoda *typeCheck* jest wierną realizacją algorytmu przedstawionego w 1.4, zgodnie z opisaną tam procedurą podejmuje odpowiednie operacje w zależności od węzła w drzewie zapytania. Warto przyjrzeć się realizacji metod pomocniczych, uzupełniających drzewo składni oraz przypadkowi, gdy wynik wskazany przez tablicę decyzyjną to błąd.

Błąd kontroli

Aparat kontroli typów (obiekt klasy *TypeChecker*) w momencie wykrycia błędu powinien to zarejestrować i podjąć próbę naprawy procesu. Zarejestrowanie błędu polega na utworzeniu obiektu *TCError* na podstawie decyzji uzyskanej z tablicy oraz dodaniu go do globalnego wyniku (obiektu klasy *TCGlobalResult*). Globalny wynik przechowuje wektor wszystkich zgłoszonych błędów. Po skończonej kontroli - o ile jakiekolwiek błędy miały miejsce - serializuje się ten obiekt i wysyła klientowi jako spis błędów kontroli typów.

Uzupełnianie drzewa zapytań

Podczas wykonywania metody *typeCheck* drzewo zapytań może zostać zmienione w skutek poniższych operacji:

Automatyczna dereferencja: *augmentTreeDeref*() Gdy tablica decyzyjna zgłosi błąd niepoprawnych podstaw argumentów następuje próba dokonania automatycznej dereferencji. Odpowiada za to metoda *augmentTreeDeref()*. Parametry, które przyjmuje to:

- Drzewo zapytania - aby móc je w razie konieczności uaktualnić.
- Sygnatury wejściowe - na nich będzie dokonywana ewentualna dereferencja
- Obiekt Decyzji (*TypeCheckResult*) - przekazany przez referencję, aby mógł być uaktualniony w razie powodzenia operacji dereferencji.

Ta metoda sprawdza najpierw, czy wejściowe sygnatury powinny podlegać dereferencji (np. czy nie mają ustawionej flagi *ref*), następnie wykonuje na każdej z sygnatur odpowiednią dla niej metodę *deref*(). Jeśli którakolwiek z sygnatur zmieniła się w wyniku tej operacji - na podstawie otrzymanej tablicy decyzyjnej uzupełniana jest ponownie decyzja. Jeśli nowa decyzja wskazuje na sukces, wtedy uzupełniane jest drzewo zapytania. W zależności od tego, które sygnatury uległy zmianie w wyniku dereferencji, do lewego lub prawego podzapytania aktualnego węzła zostaje wstawiony węzeł **deref**.

Rozwinięcie elipsy: *augmentTreeEllipsis*() Jeśli jakaś nazwa *name* nie została poprawnie związana na statycznym stosie środowisk, może to oznaczać zastosowanie przez programisty skróconej wersji zapytania, tzw. elipsy. Aby spróbować ją rozwinąć, stos przeszukiwany jest podobnie jak przy zwykłym wiązaniu nazwy, jednak napotkane sygnatury referencyjne zostają rozwinięte (poddane metodzie *stat_nested*) i dopiero wśród wyników tej metody wyszukiwana jest nazwa *name*. Jeśli zostanie znaleziona, to nazwa rozwiniętej sygnatury uaktualni drzewo zapytania. W drzewie zapytania pobierany jest rodzic aktualnego węzła z nazwą i zamiast tej nazwy wstawiany jest nowy węzeł operatora nawigacji, gdzie prawym podzapytaniem jest właśnie *name*, natomiast lewym - nazwa znalezionej, rozwiniętej sygnatury.

Koercje: *augmentTreeCoerce*() Jeśli tablica decyzyjna zaleci koercje, w obiekcie Decyzji zapisuje listę identyfikatorów koercji do wykonania. *TypeChecker* dla każdego identyfikatora dobiera odpowiedni rodzaj węzła koercji i wstawia go w miejsce odpowiedniego podzapytania. W niektórych przypadkach zamiast wstawiać nowy węzeł - w węźle operatora ustawia flagę koercji oraz zapisuje w nim sygnaturę koercji (zazwyczaj jest to któraś z wejściowych sygnatur). Przetwarzanie koercji odbywa się w czasie ewaluacji zapytania, zatem jest właściwe dla aparatu wykonawczego. Dlatego implementacja wykonania konkretnych koercji została umieszczona w module *QueryExecutor*. Są to zazwyczaj proste przekształcenia lub sprawdzenia.

Wznowienie Procesu

Celem algorytmu naprawy procesu po błędzie jest znalezienie możliwie prawdopodobnego wyniku dla danego operatora. Przy niektórych operatorach nie nastręcza to żadnego problemu. Można na przykład uznać, że wynik porównania to sygnatura elementarna *boolean*. Nie zawsze jednak wynik jest tak oczywisty. Sposób wyboru optymalnego wyniku można opracować zapewne na wiele sposobów. W LoXiM zaimplementowałem algorytm oparty na uproszczonych tablicach decyzyjnych: *RDecisionTables*. Są to podklasy odpowiednich tablic decyzyjnych używanych podczas głównego przebiegu kontroli typów. Jedynymi przedefiniowanymi metodami są te związane z inicjacją tablic, czyli wstawianiem właściwych reguł. W nowych tablicach nie pojawiają się koercje i dużo rzadziej zgłaszane są błędy. Uważam, że takie rozwiązanie - korzystające znów z tablic decyzyjnych - jest bardzo elastyczne. Daje bowiem możliwość manipulowania samymi regułami i jest bardzo przejrzyste. Ma te same zalety, co wykorzystanie tablic w samym algorytmie *typeCheck*. Dobór konkretnych reguł i wynikowych sygnatur jest znów - jak w przypadku zwykłych tablic decyzyjnych - kwestią arbitralnych decyzji projektanta języka. Sam uzupełniając te tablice kierowałem się intuicją oraz doświadczeniami zdobytymi podczas testowania systemu kontroli typów.

3.6. Problemy i rozwiązania

Trudne operatory

Nie wszystkie operatory cechuje pełna ortogonalność ze względu na podstawę i atrybuty sygnatur argumentów. Co więcej, niektóre wymagają skomplikowanych sprawdzeń, które trudno, lub wręcz nie da się wyrazić prostym mechanizmem. Dlatego niezbędne jest rozwiązanie, które pozwoli na obsługę dowolnie skomplikowanych semantycznie operatorów.

Rozwiązanie, jakie przyjąłem, aby uwzględniać niestandardowe operatory, polega na przeciążeniu generatorów podstaw sygnatur wynikowych w tablicach decyzyjnych. Te metody, które domyślnie wykonują proste operacje na argumentach, aby utworzyć z nich wynikową sygnaturę, dostają dostęp do pełnych sygnatur (nie tylko do ich podstaw). Co więcej, jako parametr mogą również otrzymać przekazany przez referencję obiekt *TypeCheckResult*, czyli dowolnie modyfikować Decyzję - zalecać dowolne koercje, zgłosić błąd, ustawić dowolne atrybuty w wynikowej sygnaturze. Przykładem takiego generatora jest:

```
Signature *insertCheck(Signature *lSig, Signature *rSig, TypeCheckResult &ret);
```

Jest to dość skomplikowany generator wyniku dla operacji wstawienia. Widać zatem, że jest to pewne "nadużycie", gdyż jest to operator imperatywny, który zwraca pustą sygnaturę *void*. Generator w tym przypadku służy do sprawdzenia poprawności obu sygnatur, nie do

wygenerowania wyniku. Takie rozwiązanie jest moim zdaniem korzystne, dlatego że pozwala niestandardowym operatorom przechodzić dokładnie taki sam schemat kontroli, co inne operatory.

Typy referencyjne do typów nazwanych

Typy referencyjne do typów nazwanych nie mogą być traktowane jako zwykła makrodefinicja, którą rozwija się przy powoływaniu odpowiedniej sygnatury, jak pokazałem w rozdziale 2.2. Przyjąłem zatem, że jest to zwykła sygnatura referencyjna, przy czym wskazywany wierzchołek metabazy nie definiuje obiektu, ale typ nazwany. To z kolei prowadzi do problemów w dwóch sytuacjach - przy wiązaniu nazw na stosie środowisk oraz przy porównywaniu sygnatur referencyjnych.

- W wyniku zastosowania funkcji *stated_nested* taka sygnatura może trafić na stos środowisk. Nie chcemy jednak, by nazwa typu była wiązana tak jak nazwy obiektów. Dlatego sygnatura opatrzona jest specjalną flagą *typedef*, dzięki której nie zostanie znaleziona na stosie przy próbie odwołania się do nazwy typu. Przyjrzyjmy się przykładowi zapytania dla schematu danych z dodatku A, znajdującego liczbę członków komisji, do której należy profesor:

  ```
  Professor.BelongsTo.CommitteeType.MemberNo;
  ```

  Jest to niepoprawne zapytanie, ponieważ *CommitteeType* jest typem, a nie obiektem. Jednak po analizie przez algorytm *typeCheck* fragmentu *Professor.BelongsTo.* na stosie środowisk faktycznie znajdzie się binder $CommitteeType(i_{CommitteeType})$. Dzięki opisanemu wyżej zabiegowi, nazwa typu nie zostanie jednak związana i zgłoszony będzie błąd kontroli typów. Pozostaje pytanie jak poprawnie odczytać liczbę członków komisji profesora. Odpowiedź jest prosta - wystarczy jawna dereferencja wokół *BelongsTo*, która włoży na stos między innymi binder $MembersNo(i_{MembersNo})$. Przykład, który ilustruje poprawne odwołanie poprzez dereferencję znajduje się w dodatku B (przykład 6).

- Strukturalne porównywanie typów obiektów złożonych jest procedurą rekurencyjną. Przy zwykłych (nie referencyjnych) odwołaniach do typów nazwanych nie ma zagrożenia nieskończonej rekurencji, gdyż wzajemne odwołania, które prowadziłyby do zapętlenia procesu są niemożliwe, jak opisałem w sekcji 2.2, w punkcie 2. Podobne ograniczenia nie mają jednak miejsca w przypadku sygnatur referencyjnych. Nie wystarczy również porównać identyfikatorów wskazywanych obiektów. Ograniczyłoby to znacznie możliwości wykorzystania typów wskaźnikowych.

  Porównując typ referencyjny z drugim (lub z obiektem wskaźnikowym) postępujemy zatem rekurencyjnie - sprawdzamy zgodność wskazywanych obiektów / typów. Aby uniknąć zapętlenia tego procesu utrzymujemy pomocniczą kolekcję, której elementami są pary porównywanych typów. Na jej podstawie wykrywamy cykle wzajemnych odwołań i przerywamy rekurencyjne rozwijanie definicji typów, uznając dwa cykle za zgodne, o ile nie wykryliśmy żadnych konfliktów typów. Zastosowanie tych konstrukcji jest zobrazowane przykładami 16 - 19 z dodatku B.

Koercje, kontrola dynamiczna

- **Dodatkowy węzeł**
  Dodanie koercji zazwyczaj polega na wstawieniu węzła koercji między węzłem danego

operatora (którego tablica decyzyjna zaleca koercję) a pewnym jego podzapytaniem (czasem wstawia się po jednym węźle nad każdym podzapytaniem). Są jednak sytuacje, gdzie takie rozwiązanie nie jest wystarczające. Jeśli do czasu wykonania przenosimy sprawdzenie, czy argumenty dwuargumentowego operatora "pasują do siebie", wtedy węzeł koercji musiałby być wspólnym nadwęzłem poddrzew odpowiadających obu podzapytaniom. To oznacza konieczność połączenia węzła koercji z węzłem operatora, ponieważ oba potrzebują dwóch argumentów. Zatem w niektórych sytuacjach zamiast wstawiać nowy węzeł do drzewa zapytania, dekorujemy węzeł odpowiedniego operatora flagą koercji, wskazującą jakie sprawdzenie należy wykonać. Przykładem operatora, który wymaga takiego zabiegu jest operator wstawienia **insert**.

- **Sygnatury w kontroli dynamicznej**
  Dynamiczna kontrola polega zazwyczaj na dokonaniu pewnych sprawdzeń lub przekształceń na wyniku pewnego podzapytania, przy czym w węźle koercji są tylko elementarne parametry pozwalające wskazać którą z akcji należy wykonać. Są jednak sytuacje, gdzie wyniki podzapytań nie są wystarczającą informacją - zdarza się, że podczas dynamicznej kontroli trzeba sięgnąć do schematu danych, a czasem nawet do dowolnej sygnatury otrzymanej w czasie statycznej kontroli. Przykładami takich wyjątkowych operatorów są znów operator tworzenia obiektu **create** oraz wstawienia - **insert**. Oba mogą wymagać (w przypadk tworzenia / wstawiania złożonych obiektów) porównania wyniku podzapytania z pewną sygnaturą.

  Rozwiązaniem jest uzupełnienie węzła koercji (podczas statycznej kontroli) o odpowiednie sygnatury. Wśród istniejących w LoXiM operatorów żaden nie wymaga więcej niż jednej takiej sygnatury, dlatego klasę *CoerceNode* wzbogaciłem o dodatkowy opcjonalny atrybut będący wskaźnikiem na pojedynczą sygnaturę. Oczywiście gdyby zaistniała taka potrzeba, można tu wstawić kolekcję sygnatur, nie stanowi to żadnego wyzwania koncepcyjnego. Zwróćmy jednak uwagę na to, że nie tylko węzły koercji potrzebują takiego atrybutu. Ze względu na sytuacje opisane w poprzednim punkcie - węzeł dowolnego operatora może być oznaczony flagą dynamicznej kontroli. Z tego powodu nie tylko *CoerceNode*, ale nadklasa *TreeNode* została zaopatrzona w atrybut sygnatury koercyjnej.

- **Statyczne koercje**
  Podczas kontroli typów część sprawdzeń lub rzutowań przenoszona jest do czasu wykonania. W tym sensie wszystkie dodane koercje są dynamiczne. Nie zawsze jednak oznaczają przeniesienie kontroli poprawności typologicznej do czasu wykonania. Niektóre operacje (takie jak rzutowanie liczby na napis) nigdy nie zgłoszą błędu kontroli. Takie koercje nazwijmy umownie statycznymi. W tablicach decyzyjnych zostają one oznaczone jako *STATIC*, w odróżnieniu od tych oznaczonych *DYNAMIC*, które faktycznie mogą w czasie wykonania zakończyć się poprawnie lub zgłosić błąd. Takie rozróżnienie pozostaje w obecnym systemie bez dalszych konsekwencji, ponieważ wynik kontroli typologicznej jest ignorowany, o ile nie jest to błąd. Można by jednak rozbudować moduł *TypeCheck*, aby rejestrował wyniki kontroli każdego zapytania. W takim przypadku warto rozróżnić koercje, które jedynie przekształcają wynik od tych, które wprowadzają elementy kontroli dynamicznej.

Sygnatury wariantowe i rzutowanie

Istnienie sygnatur wariantowych może bardzo komplikować proces kontroli typów, powodując konieczność rozpatrywania drzewa alternatywnych kombinacji sygnatur oraz odpowiedniego

scalania wyników kontroli. Aby pozbyć się tego problemu można uznać, że suma mnogościowa może dotyczyć wyłącznie elementów o zgodnych sygnaturach. Rozwiązanie, które przyjąłem w LoXiM jest bardziej elastyczne.

Dopuszczam sumę mnogościową różnych sygnatur, a co za tym idzie - powstawanie sygnatur wariantowych. Nadmiernych komplikacji z tym związanych udało się uniknąć dzięki założeniu, że przy większości operatorów taka sygnatura jako argument powoduje błąd kontroli typów. (Wyjątkami są na przykład suma mnogościowa lub dereferencja, które w wyniku również zwracają wariant). Aby "skonsumować" sygnaturę wariantową i wykorzystać jako argument dowolnego operatora należy ją najpierw odpowiednio zrzutować. W tym celu rozszerzyłem semantykę istniejącego już operatora rzutowania klas ($cast(\ldots to \ldots)$). Po zmianach dowolne zapytanie można rzutować na wybrany typ lub sygnaturę. Przykłady wykorzystujące sygnatury wariantowe i rzutowanie znajdują się w dodatku B, zapytania 10, 11, 12. Rzutowanie na typy okazuje się również pożyteczne w innych przypadkach, jak pokazuje przykład 3 z tego dodatku.

Dodanie nowego znaczenia operatorowi rzutowania wymagało również określenia dla niego reguł kontroli typów. Pierwszym pomysłem było traktowanie dowolnych rzutowań jako poprawne. Aparat wykonawczy przecież odrzucałby te elementy zbioru rezultatów, które nie podlegają danemu rzutowaniu. Zauważyłem jednak, że część tej kontroli można uczynić statyczną. Sygnatura wariantowa przechowuje bowiem kolekcję sygnatur, jakie może przyjmować. We wdrożonym rozwiązaniu zrealizowałem następujące założenie: jeśli podstawa żadnej z tych sygnatur nie jest zgodna z docelową (tą, na którą rzutujemy), rzutowanie uznane jest za niepoprawne i zgłaszany jest błąd statycznej kontroli typów. W przeciwnym razie operator rzutowania nie zgłasza błędu, a jego wynikiem jest docelowa sygnatura.

Aktywacja i dezaktywacja kontroli typów

Kontrola typów może być "niewygodna", nawet jeśli jest zrealizowana tak jak w LoXiM - uwzględniając półstrukturalny charakter danych i różne nietypowe konstrukcje językowe. Można chcieć umożliwiać dowolne manipulowanie danymi i tworzenie obiektów, które nie były wcześniej deklarowane. Można również uznać, że kontrola jest konieczna, ale chcieć - np. dla celów testowych - wykonać kilka zapytań niezgodnych z obowiązującym schematem danych. Takie założenie skłoniło mnie do stworzenia mechanizmów aktywujących i dezaktywujących tę kontrolę zarówno trwale (na czas działania serwera bazy danych) jak i tymczasowo, lokalnie (w pojedynczej sesji klienckiej, na potrzebę wykonania kilku zapytań).

- Inicjując pracę serwera LoXiM można wyłączyć kontrolę typologiczną manipulując plikiem konfiguracyjnym i flagą *type_checking*. Wyłączenie tej opcji powoduje, że zapytania nie są poddawane żadnej kontroli statycznej - przekazywane są bezpośrednio do aparatu wykonawczego i dopiero tam dokonywane są ewentualne sprawdzenia. Żadne zapytanie nie będzie zatem również wzbogacone o automatyczne dereferencje ani rozwinięcia elips.

- Przy włączonej opcji *type_checking* każdy klient może we własnej sesji czasowo wyłączyć kontrolę typów. Polecenie *tcOFF* ustawia odpowiednią flagę w obiekcie analizatora składniowego przydzielonego danemu klientowi, która tymczasowo dezaktywuje kontrolę. Aby przywrócić aktywność modułu *TypeCheck* należy użyć zapytania *tcON*. Te dwa polecenia są bezskuteczne, gdy plik konfiguracyjny nie zezwala na kontrolę typów.

Rozdział 4

Zmiany i rozszerzenia

4.1. Manipulowanie regułami

Wdrażając system kontroli typów w LoXiM uzupełniłem tablice decyzyjne regułami, które uznałem za zgodne ze zdrowym rozsądkiem, albo przydatne do zobrazowania możliwości półmocnej kontroli. Nie jest jednak moim zamierzeniem pokazanie wyższości tych decyzji nad alternatywnymi, ale stworzenie takiego mechanizmu, który pozwoli w łatwy sposób podmieniać i dostosowywać te reguły. Uważam, że ten cel został osiągnięty. Zauważmy, że przy znakomitej większości operatorów można dodać/usunąć dowolną regułę, ewentualnie podmienić wynik, lub dodać koercję. Kod dotyczący inicjacji tablic (cały zawarty w pliku *DecisionTable.cpp*) jest według mnie bardzo intuicyjny - każdemu wierszowi w tablicy (czyli każdej regule) odpowiada wiersz kodu - konstruktor reguły. Szczegóły opisałem w rozdziale 3.4.1. Zatem nie trzeba wczytywać się w kod i zmieniać zawartości żadnych metod, należy tylko zrozumieć jak dopasowywane są reguły do argumentów, jak wykorzystywane są metareguły, aby móc zmieniać rezultaty kontroli dla wybranego operatora. Zobaczmy, jak można zmienić reguły dotyczące przywoływanego już przykładu dodawania (przykład 3.1). Chcemy zmienić tylko reguły dla podstawy sygnatury. Przyjrzyjmy się tablicy przed zmianami oraz po (zmiany zaznaczone symbolem (!)):

Przykład 4.1.

```
//Operator ' + '
/*BASE*/
addTcRule(      "integer",  "integer",  "integer");
addTcRule(      "double",   "double",   "double");
addTcRule(      "string",   "string",   "string");
addTcRule(      "integer",  "string",   "integer",  BS_TO_INT, RIGHT, DYNAMIC);
addTcRule(      "string",   "integer",  "string",   BS_TO_STR, RIGHT, STATIC);
addTcRule(      "double",   "string",   "double",   BS_TO_DBL, RIGHT, DYNAMIC);
addTcRule(      "string",   "double",   "string",   BS_TO_STR, RIGHT, STATIC);
addTcRule(M_ELSE,                       TC_RS_ERROR);

Po zmianach:

addTcRule(      "integer",  "integer",  "integer");
addTcRule(      "double",   "double",   "double");
addTcRule(      "string",   "string",   "string");
```

```
addTcRule(      "boolean",  "boolean",  "boolean"); //(!) nowa reguła
addTcRule(      "integer",  "string",   "string",    BS_TO_STR, LEFT, STATIC); //(!)
addTcRule(      "string",   "integer",  "string",    BS_TO_STR, RIGHT, STATIC);
addTcRule(      "double",   "string",   "string",    BS_TO_STR, LEFT, STATIC); //(!)
addTcRule(      "string",   "double",   "string",    BS_TO_STR, RIGHT, STATIC);
addTcRule(M_ELSE,                       TC_RS_ERROR);
```

Dodałem jedną regułę, która pozwala dodawać wartości logiczne (zakładając, że aparat wykonawczy zrozumie to jako alternatywę tych wartości). Co więcej zmieniły się reguły dodawania, gdy jednym z argumentów był napis. Teraz bez względu na to, czy występuje on jako lewej, czy po prawej stronie operatora '+', drugi argument rzutowany jest na napis, a wynikiem dodawania jest konkatenacja tak otrzymanych argumentów. Zwróćmy uwagę na to, jakich zmian w kodzie wymagała taka zmiana koncepcji. Zmieniliśmy dwie linijki (odpowiadające dwóm regułom). W każdej wynik przyjął wartość *string*, oraz zmieniły się parametry koercji - *BS_TO_STR* zastąpił rzutowania na liczby.

W powyższym przykładzie cztery reguły dodające koercję można by chcieć zastąpić jedną. Przy tak ustalonych zasadach dodawania, intuicyjne wydaje się bowiem zdanie: "Jeśli jednym z argumentów jest napis, pozostały argument należy zrzutować na napis, a wynikiem jest również napis". Moduł *TypeCheck* pozwala zapisać to w postaci reguły:

```
addTcRule(M_EX,   "string",   "",   "string",   BS_TO_STR, BOTH, STATIC);
```

Atrybut sterujący *M_EX* (ang: *EXISTS*) sprawia, że dana reguła będzie dopasowana do dwóch sygnatur, jeśli którakolwiek jest równa pierwszemu argumentowi reguły (w tym przypadku: *"string"*, czyli napis). Zauważmy jednak, że prawdopodobnie wcale nie chcielibyśmy, żeby móc dodawać napis do struktury albo referencji, ale jedynie napis do dowolnego typu elementarnego. Wystarczy więc wcześniej dodać reguły zgłaszające błąd przy próbie dodawania innych sygnatur. Pamiętajmy, że kolejność wpisywanych reguł ma znaczenie, dlatego muszą one wystąpić przed wyżej wymienioną regułą dla napisów. Ostatecznie nasza tablica mogłaby wyglądać następująco:

```
//Operator ' + '
/*BASE*/
addTcRule(        "integer",  "integer",  "integer");
addTcRule(        "double",   "double",   "double");
addTcRule(        "string",   "string",   "string");
addTcRule(        "boolean",  "boolean",  "boolean"); //nowa reguła !
addTcRule(M_EX,   "variant",  "",         TC_RS_ERROR);
addTcRule(M_EX,   "struct",   "",         TC_RS_ERROR);
addTcRule(M_EX,   "ref",      "",         TC_RS_ERROR);
addTcRule(M_EX,   "string",   "",         "string",   BS_TO_STR, BOTH, STATIC);
addTcRule(M_ELSE,                         TC_RS_ERROR);
```

Oczywiście nie w każdym przypadku wyprowadzanie wyniku mieści się w tym samym schemacie. Semantyka niektórych operatorów jest bardziej skomplikowana, niekiedy tracona jest np. ortogonalność atrybutów, lub po prostu wymagane są zaawansowane sprawdzenia. Dlatego zmian mogą wymagać niektóre metody (generatory podstaw i atrybutów sygnatur) z pliku *DecisionTable.cpp*. Chcąc na przykład dokonać zmian w przetwarzaniu operatora wstawienia, trzeba niestety wczytać się w kod generatora wynikowej struktury. Moim zdaniem jednak takie sytuacje są nie do uniknięcia - nie można skomplikowanego procesu modelować zupełnie prostym schematem. Za sukces uznaję zminimalizowanie liczby tych operatorów,

dla których kontrola jest nietypowa. Co więcej, nawet te operatory również przechodzą przez standardowy schemat kontroli, a ich wyjątkowość przekłada się jedynie na rozbudowane metody generujące wynik, w miejscu prostych generatorów.

4.2. Modyfikowanie algorytmu *RestoreProcess*

Algorytm naprawy procesu kontroli typów po błędzie można realizować na różne sposoby. Ten, który zaimplementowałem w LoXiM jest moim zdaniem elastyczny - oparty na tablicach decyzyjnych pozwala na tak samo proste manipulacje jak te opisane w poprzednim rozdziale (4.1). Moim zadaniem było jednak stworzenie środowiska, w którym taki algorytm będzie można w prosty sposób podmienić. W tym celu zastosowałem wzorzec strategii - stworzyłem abstrakcyjną klasę *RestoreAlgorithm*, której interfejs odpowiada algorytmowi naprawy procesu kontroli. Kontekstem dla tej klasy stał się w naturalny sposób obiekt *TypeChecker*, z którego jest dostęp do wszystkich struktur niezbędnych podczas statycznej kontroli typów. Aby wdrożyć nowy algorytm, należy zmodyfikowac plik *RestoreAlgorithm.cpp*, dodając nową podklasę *RestoreAlgorithm* (istniejąca implementacja realizowana jest przez klasę *DTCRestoreAlgorithm*), zaimplementować metody *restoreOnMisspelledName()* oraz *restoreOnBadArg()*, oraz podmienić inicjację atrybutu *restoreAlgorithm* w konstruktorze obiektu *TypeChecker*, odpowiedzialnego za realizację kontroli typów.

4.3. Rozszerzenie kontroli typologicznej

W tym rozdziale opisuję jakie kroki należy podjąć przy rozszerzaniu modułu *TypeCheck*. Najpierw omówię sytuację, gdy do języka dodany jest nowy operator, a następnie przedyskutuję możliwość rozciągnięcia kontroli typów aby obejmowała kolejne modele danych. Należy jednak w tym miejscu zaznaczyć, że jest to jedynie wskazanie kierunku działań, a nie szczegółowy opis jak zrealizować rozszerzoną kontrolę. Chcących dowiedzieć się więcej na ten temat odsyłam do [Stencel06, rozdz. 10].

Nowy operator

Pojawienie się nowego operatora oznacza konieczność dodania nowej tablicy decyzyjnej. W przeciwnym razie nowy element będzie nierozpoznawalny przez moduł *TypeCheck*, zatem zapytania zawierające dany operator będą przekazywane do aparatu wykonawczego bez właściwej kontroli. Dodanie tablicy decyzyjnej w module *TypeCheck* to uzupełnienie jednej z metod inicjacyjnych (*initUnOpRules()* lub *initOtherBinaryRules()*) zestawem konstruktorów odpowiednich reguł, analogicznie jak w przykładzie 3.1. Należy również zadbać o obsługę tego operatora w głównej metodzie algorytmu kontroli *TypeChecker::typeCheck()*. Jeśli nie należy on do żadnej ze zdefiniowanych grup (np. operatory algebraiczne / niealgebraiczne - dla takich operatorów żadna dodatkowa czynność nie jest konieczna), należy dopisać dla niego fragment metody, analogiczny do już istniejących, zgodnie z opisami w rozdziałach 1.4 i 3.5.

W najprostszym przypadku to już wystarczy, aby nowy operator był poprawnie obsługiwany przez kontrolę typologiczną. Może się jednak okazać, że posiada on skomplikowaną semantykę i wymaga bardziej skomplikowanych sprawdzeń, niż te występujące w istniejącym schemacie. W takim przypadku należy postąpić tak, jak opisałem w rozdziale 3.6, sekcji **Trudne operatory**. Należy zaimplementować nowy generator podstawy sygnatury wyniku, który

wykona odpowiednie sprawdzenia, a następnie zarejestrować go, dodając nowy identyfikator w kolekcji *DecisionTable::ResultGenerator* oraz przypisując go (poprzez dodanie wpisu w metodzie *DecisionTable::doSig()*) utworzonemu generatorowi. Przy dodaniu nowego operatora zmian w module kontroli typów wymagają zatem pliki *DecisionTable.cpp* i *TypeChecker.cpp*.

Rozszerzony model danych

Rozciągnięcie kontroli typologicznej na kolejne modele danych z pewnością nie będzie prostym zabiegiem, a podczas jego realizacji niemal pewne jest, że pojawią się nowe problemy do rozwiązania. Jednak schemat kontroli pozostanie bardzo podobny do tego obowiązującego dla modelu *M0*. Dlatego istniejący moduł *TypeCheck* stanowi dobrą podstawę dla takich rozszerzeń. Zobaczmy jakie elementy na pewno będą zmienione.

Przede wszystkim należy zadbać o odwzorowanie nowego modelu danych w metabazie. Pojawienie się nowych bytów wiąże się z koniecznością rozszerzenia formatu metabazy i mechanizmów zapewniających jej spójność (zmiany mogą więc dotyczyć aparatu wykonawczego - *QueryExecutor.cpp*). To z kolei prowadzi do przebudowy struktur przechowujących schemat danych w pamięci, w tym klas *DataObjectDef* oraz *DataScheme*, a także metod wczytujących i obsługujących te struktury, zawartych w plikach *DataRead.cpp*, *DataRead.h* modułu *QueryParser*. Zmiany schematu mogą polegać na uwzględnianiu nowych bytów i zależności, ale mogą również być bardziej skomplikowane. Jak opisano w [Stencel06], rozdz. 10, kontrola programów (procedur, metod) wymaga na przykład wprowadzenia tzw. metabazy lokalnej (na potrzeby zmiennych i obiektów lokalnych). Zatem nawet wobec opracowanych już rodzajów bytów bazodanowych niezbędne są pewne zmiany.

Zmiany w schemacie danych odbiją się prawdopodobnie na sygnaturach. Może się to wiązać z dodaniem nowych podklas do hierarchii *Signature*. Możliwe też, że w istniejące sygnatury będą wzbogacone o nowe atrybuty, dla których konieczna będzie kontrola typologiczna. Dodanie nowego, ortogonalnego atrybutu do sygnatur wymusza szereg zmian w module *TypeCheck*. Przede wszystkim w klasach tablic decyzyjnych (*DecisionTable*, *UnOpDecisionTable*) należy dodać deklarację nowej kolekcji reguł (przypomnijmy, że podstawie i każdemu z atrybutów odpowiada jeden taki zestaw). Następnie metoda *getResult()*, opisana w rozdziale 3.4, musi zostać zmodyfikowana, aby uwzględnić nowy atrybut. Na końcu dla każdej tablicy decyzyjnej należy dodać odpowiedni zestaw konstruktorów w metodach inicjacyjnych. Już na takim

Najbardziej bezpośrednim i najprostszym przykładem rozszerzenia wydaje się uwzględnienie klas i dziedziczenia. Jakie kroki należy podjąć, aby to zrealizować? Najpierw schemat danych - należy go uzupełnić tak, aby przechowywał definicje klas i ich metod, relacje dziedziczenia między klasami, a także powiązania między obiektami a klasami (relacja bycia obiektem pewnej klasy). Sygnatury i tablice decyzyjne prawdopodobnie nie będą wymagały dużych zmian. Również sam algorytm *typeCheck* pozostanie bez znacznych modyfikacji. Jedynie przy przetwarzaniu operatorów niealgebraicznych należy zadbać o uzupełnianie statycznego stosu środowisk o sekcje odpowiednich klas. W tym celu wystarczy zmodyfikować pomocnicze metody *openScope* i *closeScope* (w pliku *TypeChecker.cpp*), aby - na podstawie rozszerzonego schematu danych - uwzględniały również sekcje klas i nadklas obiektu o wskazanej sygnaturze. Wydaje się zatem, że jedyna trudność może tkwić w rozbudowie i utrzymaniu spójności schematu danych. Cała reszta wdrożonego systemu kontroli typów powinna się poprawnie aplikować do modelu danych zawierającego klasy i dziedziczenie.

Podsumowanie

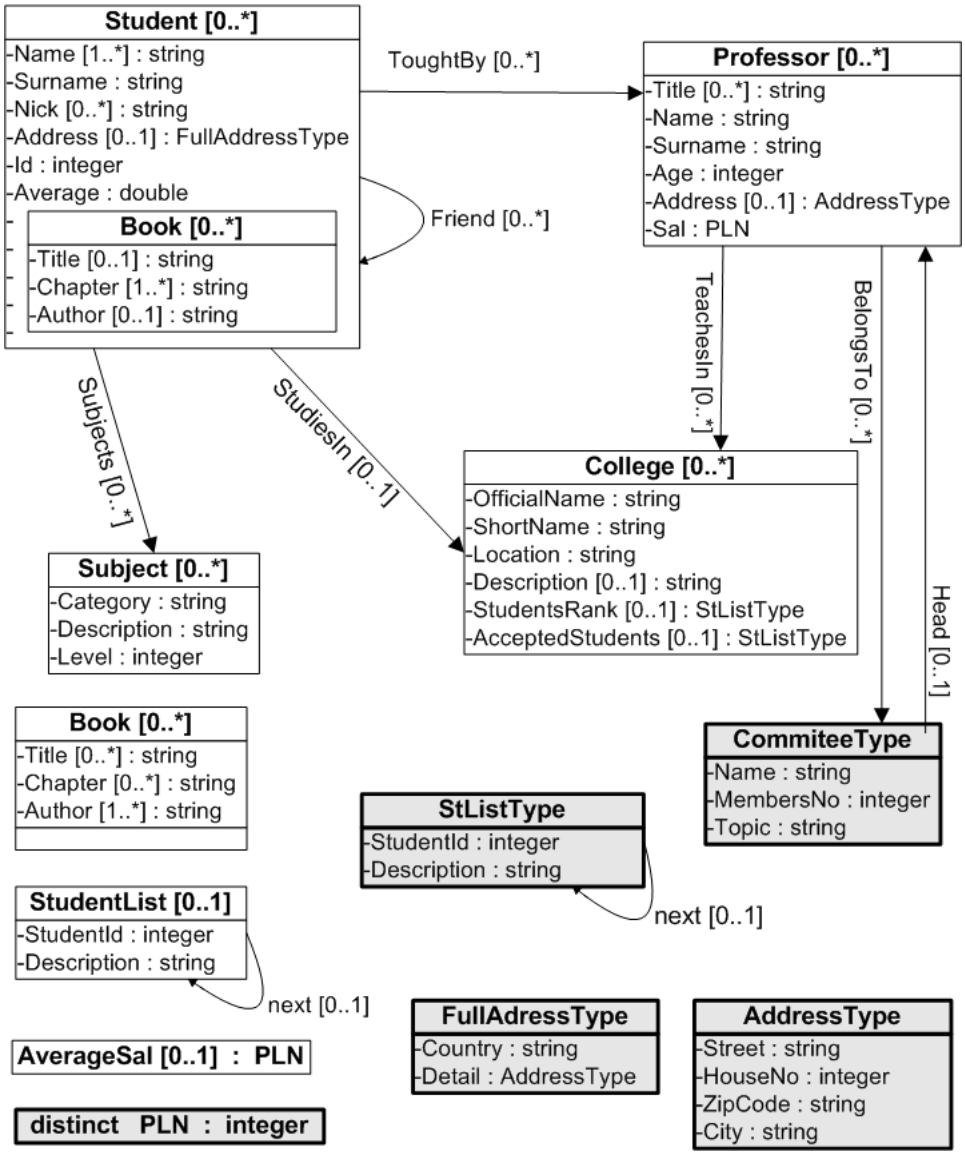
Wdrożenie systemu kontroli typów w LoXiM należy moim zdaniem uznać za udane. Przede wszystkim dlatego, że ze wstępnych testów wynika, że działa poprawnie, co więcej - zazwyczaj działa zgodnie z oczekiwaniami i intuicją programisty. Pierwsze doświadczenia pokazały, że ilekroć wynik kontroli odbiegał od intuicji, w łatwy sposób można to było poprawić manipulując odpowiednio regułami w tablicach decyzyjnych. Ta implementacja dowodzi więc, że idea półmocnej kontroli typów nie jest jedynie teorią, ale praktyczną i spójną wizją, którą można zaimplementować i wdrożyć w wybranym systemie zarządzania obiektową bazą danych, opartą o język zapytań typu SBQL.

Ponadto warto zauważyć, że udało się stworzyć nie tylko implementację konkretnych zasad kontroli typów, ale mechanizm, który - m.in. dzięki przejrzystej realizacji tablic decyzyjnych - umożliwia w miarę proste sterowanie regułami kontroli. Daje twórcy języka możliwość swobodnego decydowania jakie zapytania uznawać za błędne, jakie za poprawne, przy jakich przenosić kontrolę do czasu wykonania, dla każdego operatora zdefiniować zachowanie w zależności od sygnatur argumentów.

W następnych krokach warto rozszerzyć tę kontrolę, aby obejmowała model danych, który już jest dostępny w systemie LoXiM, w tym klasy, metody, procedury i perspektywy.

Dodatek A

Przykład Schematu danych



Rysunek A.1: Przykładowy schemat danych

Poniżej deklaracja schematu danych pokazanego na rysunku A.1, zgodna z nową składnią deklaracji typów SBQL. Na rysunku A.1 definicje typów zostały wyróżnione (w stosunku do deklaracji obiektów) grubszymi konturami i ciemniejszym tłem. Wszystkie przykłady występujące w dodatku B oraz większość przykładów w niniejszej pracy dotyczy danych opisanych przez ten schemat.

```
Student[0..*]: (
      Name[1..*]: string,
      Surname: string,
      Nick[0..*]: string,
      Address[0..1]: FullAddressType,
      Id : integer,
      Average[0..1]: double,
      Book[0..*]:(Title[0..1]:string, Chapter[1..*]:string, Author[0..1]:string),
      Subjects[0..*]: ref Subject,
      StudiesIn[0..1]: ref College,
      ToughtBy[0..*]: ref Professor,
      Friend[0..*] : ref Student
      );
/
Book[0..*]:(Title[0..*]:string, Chapter[0..*]:string, Author[1..*]:string);
/
Professor[0..*]: (
      Title[0..*]: string,
      Name: string,
      Surname: string,
      Age: integer,
      Address[0..1]: AddressType,
      Sal: PLN,
      TeachesIn[0..*]: ref College,
BelongsTo[0..*]: ref CommitteeType);
/
Subject[0..*]: (Category: string, Description: string, Level: integer);
/
College[0..*]:(OfficialName: string,
      ShortName: string,
      Location: string,
      Description[0..1]: string,
      StudentsRank[0..1]: StListType,
      AcceptedStudents[0..1]: StListType);
/
StudentList[0..*]:(StudentId:integer, Desc: string, next[0..1] : ref StudentList);
/
typedef StListType = (StudentId: integer, Desc: string, next[0..1] : ref StListType);
/
typedef FullAddressType = (Country: string, Detail: AddressType);
/
typedef AddressType = (Street: string,
                       HouseNo: integer,
```

```
                        ZipCode: string,
                        City: string);
/
typedef CommitteeType = (Name: string,
                          Head[0..1]: ref Professor,
                          MembersNo: integer,
                          Topic: string);
/
typedef distinct PLN = integer;
/
AverageSal[0..1] : PLN;
/
reloadScheme;
/
```

Dodatek B

Wyniki kontroli typów na przykładach

Poniżej znajduje sie seria zapytań opartych na schemacie danych z dodatku A, wraz z wynikami kontroli typów. Testowe zapytania zostały tak dobrane, aby pokazać różne możliwości i funkcje aparatu półmocnej kontroli typów. Niektóre zostały opatrzone krótkim komentarzem. W kilku przypadkach zamieściłem również (zaznaczając symbolem '#') postać zapytania przekazywanego do aparatu wykonawczego, po przejściu kontroli, z dodanymi dereferencjami, koercjami lub rozwiniętymi elipsami.

1. ```
   Student.Sal;
   ----------------------------
    TypeCheck result: ERROR
   * [error: Bad Name] : Name 'Sal' not found.
   ```

2. ```
   Professor where Sal = 2000;
   ---------------------------
    TypeCheck result: ERROR
   * [error: Bad Args] : typeName of sigs: (ref->78,[1..1]), (integer,[1..1])
   (derefed to : (integer,PLN[1..1]), (integer,[1..1])
   does not match operator ' = '.
   ```

   Atrybut *Sal* jest wyróżnionego typu nazwanego *PLN*, dlatego może być porównywany jedynie z obiektami takiego samego typu. Nie można go porównywać z dowolną liczbą. Następny przykład pokazuje, jak można wymusić takie porównanie.

3. ```
   Professor where Sal = cast (2000 to PLN);
   -----------------------------------------
    TypeCheck result: SUCCESS
   ```

4. ```
   Professor where TeachesIn.ShortName = "UW";
   -------------------------------------------
   # Professor where deref(TeachesIn.College.ShortName) = "UW";
   ----------------------------------------------------------
    TypeCheck result: SUCCESS
   ```

   Zwróćmy uwagę, że nastąpiło rozwinięcie elipsy i dodanie automatycznej dereferencji.

5.
```
create ("Jan" as Name,
        "Kowalski" as Surname,
        35 as Age, 3000 as Sal) as Professor;
-----------------------------------------
 TypeCheck result: SUCCESS
```

6.
```
Professor where deref(BelongsTo).MembersNo > 15;
-----------------------------------------------
# Professor where element(deref(BelongsTo).MembersNo) > 15;
-----------------------------------------------------------
 TypeCheck result: DYNAMIC COERCE
 Error: Query Executor: Coerce failed, set is empty.
```

   Przy tym zapytaniu nastąpiło dodanie koercji liczebności do *1..1* z kontrolą dynamiczną. Został zgłoszony błąd czasu wykonania, ponieważ w utworzonym obiekcie *Professor* nie ma żadnego elementu *BelongsTo*. Zwróćmy uwagę na sposób odwoływania się do podobiektów komisji, do których należy profesor. Związek *BelongsTo* został zadeklarowany jako referencja do typu. W takiej sytuacji nieprawidłowe jest zapytanie *BelongsTo.CommitteeType.MembersNo*, ponieważ *CommitteeType* nie jest obiektem. Stąd jawna dereferencja wokół *BelongsTo*.

7.
```
(("100" - "19") / "100") * Sal;
-------------------------------
 TypeCheck result: ERROR
* [error: Bad Args] : base  of args: (string,[1..1]) , (string,[1..1])
        invalid for operator  '  -  '.
* [error: Bad Args] : base  of args: (string,[1..1]) , (string,[1..1])
        invalid for operator  '  /  '.
* [error: Bad Name] :  Name "Sal" not found.
```

   W powyższym przykładzie udało się wykryć kilka błędów przy pojedynczym przebiegu kontroli dzięki skutecznemu wznawianiu procesu po błędzie. Prześledźmy sygnatury, jakie wygeneruje *TypeCheck* przy wznowieniach procesu. Dla wszystkich trzech operatorów algebraicznych występujących powyżej ('-', '+', '/') w tablicach decyzyjnych używanych przez *RestoreAlgorithm* zadziała reguła, która wskazuje napis jako najbardziej prawdopodobny wynik, o ile którykolwiek argument jest napisem. Zatem przy każdym wznowieniu procesu domyślnym wynikiem staje się sygnatura typu *string*, co powoduje, że argumentami kolejnych operacji są cały czas dwa napisy.

8.
```
"3pi.14" - 34.3;
----------------
# toDouble("3pi.14") - 34.3;
----------------------------
 Error: Query Executor: Coerce failed: Cannot coerce to Double
```

9.
```
("123.14" - 3.14) - "20";
-------------------------
# (toDouble("123.14") - 3.14) - toDouble("20");
-----------------------------------------------
 TypeCheck result: SUCCESS
```

   W tym zapytaniu oba rzutowania zakończą się sukcesem, a wynikiem odejmowania dwóch argumentów typu *double* jest również liczba typu *double*.

10.
```
(Professor.Title union Professor.Age) + 30;
-------------------------------------------
 TypeCheck result: ERROR
* [error: Bad Args] : base of sigs: variant(...), (integer)
                      invalid for operator ' + ';
```

   W wyniku sumy mnogościowej może powstać sygnatura wariantowa. Tak jest w powyższym przykładzie, ponieważ *Age* jest liczbą, natomiast *Title* to napis. Takie sygnatury są błędnym argumentem dla większości operatorów. Aby móc je dalej przetwarzać, trzeba wykonać rzutowanie, jak pokazano w poniższych dwóch przykładach. W przykładzie 12. rzutowanie na napis (*string*) nie powiedzie się, ponieważ już na etapie kompilacji wiadomo, że ten wariant w żadnym przypadku nie będzie napisem.

11.
```
cast ((Professor.Title union Professor.Age) to integer) + 30;
-------------------------------------------------------------
 TypeCheck result: SUCCESS
```

12.
```
cast ((Professor.Age union Professor.Address) to string) + "10";
----------------------------------------------------------------
 TypeCheck result: ERROR
* [error: Bad Args] : base of sigs: variant(...), (integer)
                      invalid for operator CAST;
```

13.
```
create ("Jan" as Name, "Kac" as Surname, 16384 as Id) as Student;
-----------------------------------------------------------------
 TypeCheck result: SUCCESS
```

14.
```
create ("3MSc" as Title, "All41" as Title,
        "Ch 1" as Chapter, "Dumas" as Author) as Book;
-------------------------------------------------------------------
 TypeCheck result: SUCCESS
```

15.
```
(Student where Id = 16384) :< Book;
-----------------------------------
 TypeCheck result: DYNAMIC COERCE
 Error: Query Executor: Coerce dynamic check failed:
                        Cards of subobjects do not match.
```

   Powyższe zapytanie zwróci błąd czasu wykonania jeśli zostało wykonane w następstwie poprzednich dwóch przykładów. Obiekty *Book* oraz *Student.Book* są tak zadeklarowane, że w czasie kompilacji nie jest zgłaszany żaden błąd, jednak zaznaczana jest flaga koercji przy operatorze :<. Gdyby w obiekcie *Book* był tylko jeden podobiekt *Title* - wówczas wstawienie zostałoby poprawnie wykonane.

16.
```
create ("University of Warsaw" as OfficialName,
        "UW" as ShortName,
        "Warsaw" as Location) as College;
create (512 as Id, "Winner" as Desc) as StudentList;
create (1024 as Id, "Follow up" as Desc) as StudentList;
-----------------------------------------------------
 3 x TypeCheck result: SUCCESS
```

Trzy zapytania powyżej są poprawne, a posłużą mi do zobrazowania możliwości, jakie dają typy referencyjne. *College.StudentsRank* jest zdefiniowany poprzez typ *StListType*, będący listą wskaźnikową. Poniżej dwa zapytania pokazujące poprawną konstrukcje listy rankingowej studentów, oraz trzecie - błędne.

17.
```
College :< (128 as Id, "Champion" as Desc) as StudentsRank;
-----------------------------------------------------------
 TypeCheck result: SUCCESS
```

18.
```
College.StudentsRank :< ref(StudentList where Id = 512) as next;
----------------------------------------------------------------
 TypeCheck result: SUCCESS
```

19.
```
(StudentList where Id = 1024) :< ref(College.StudentsRank) as next;
-------------------------------------------------------------------
 TypeCheck result: ERROR
* [error: Bad Args] : Type Checker: Error while comparing signatures:
                      miss-matched types.
       bases  of args: (ref->78,[1..1]) , next,[1..1]((ref->74,[1..1]) )
       invalid for operator  '  :<  '.
```

Wskaźnik *next* w obiektach *StudentList* jest referencją właśnie do elementów *StudentList*, a nie do dowolnego elementu o zgodnym typie. W powyższym przykładzie próbujemy natomiast wstawić w to miejsce obiekt *College.StudentsRank*. Stąd - zgodnie z oczekiwaniami - błąd kontroli typów.

Bibliografia

[HLSS05] Rafał Hryniów, Michał Lentner, Krzysztof Stencel, Kazimierz Subieta, *Types and type checking in stack-based query languages*, Instytut Podstaw Informatyki PAN, Warszawa, 2005

[Plodzien00] Jacek Płodzień, *Optimization methods in Object Query Languages*, Warszawa, 2000

[Sitek07] Jakub Sitek, *Implementacja optymalizacji zapytań w systemie LoXiM*, Warszawa, 2007

[Stencel06] Krzysztof Stencel, *Półmocna kontrola typów w językach programowania baz danych*, Wydawnictwo PJWSTK, Warszawa 2006

[Subieta04] Kazimierz Subieta, *Teoria i konstrukcja obiektowych języków zapytań*, Wydawnictwo PJWSTK, Warszawa 2004