Object Query Language

Typing

Arūnas Janeliūnas

Object Databases

Name resolution

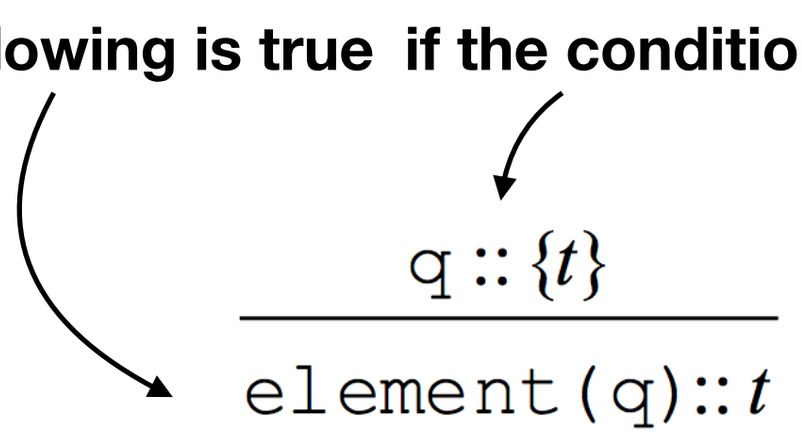
Name resolution order:

1. variable;
2. property;
3. query name;
4. name from the schema.

```
class Person {
 . . .
};

class Car {
 person: Person;
 . . .
};

name Persons : set <Person>;

define person as . . . ;

select . . .
  from person in Persons,
       auto in MyCVars
 where auto.person = . . .
```

Typing rules

Rules are read like this:



Rule examples

Casting:

$$\frac{\mathtt{q} :: c' \, , \, c \leq c' \text{ arba } c \geq c'}{(\mathtt{c})\,\mathtt{q} :: c}$$

Property extraction:

$$\frac{\mathtt{q} :: t \, , \, t \leq [\mathtt{a} : t']}{\mathtt{q.a} :: t'}$$

Sum:

$$\frac{\mathtt{q}_1 :: int \, , \, \mathtt{q}_2 :: int}{\mathtt{q}_1 + \mathtt{q}_2 :: int}$$

Rule examples

Another sum:

$$\frac{\mathtt{q}_1 :: t_1 \ , \mathtt{q}_2 :: t_2 \ , \{real\} \subseteq \{t_1\} \cup \{t_2\} \subseteq \{int, real\}}{\mathtt{q}_1 \ \mathtt{+} \ \mathtt{q}_2 :: real}$$

Calling a method:

$$\frac{\mathtt{q} :: c \ , \quad \mathtt{m} : (c, t_1 \ , \ldots , t_n) \rightarrow t \ , \quad \forall i = \overline{1,n} \quad \mathtt{q}_\mathtt{i} :: t'_i \ , t'_i \leq t_i}{\mathtt{q.m(q}_1\mathtt{,\ldots,q}_\mathtt{n}\mathtt{)} :: t}$$

Structure constructor:

$$\frac{\forall i = \overline{1,n} \quad \mathtt{q}_\mathtt{i} :: t_i}{\mathtt{struct(a}_1 \mathtt{:q}_1\mathtt{,\ldots,a}_\mathtt{n} \mathtt{:q}_\mathtt{n}\mathtt{)} :: \mathtt{[a}_1 \mathtt{:}t_1\mathtt{,\ldots,a}_\mathtt{n} \mathtt{:}t_n\mathtt{]}}$$

Rule examples

Iterator:

$$\mathtt{q}_1 :: col(t_1),\ \mathtt{q}_2 :: [t_1] \to col(t_2),\ \ldots,\quad \mathtt{q}_n :: [t_1,\ldots,t_{n-1}] \to col(t_n),$$

$$\mathtt{p} :: [t_1,\ldots,t_n] \to bool\ ,\quad \mathtt{q} :: [t_1,\ldots,t_n] \to t$$

$$\mathtt{x}_1 :: t_1\ ,\ldots,\mathtt{x}_n :: t_n\ ,$$

$$\left(\begin{array}{l}\texttt{select q[x}_1\texttt{,...,x}_n\texttt{]}\\ \quad\texttt{from x}_1\texttt{ in q}_1\texttt{, x}_2\texttt{ in q}_2\texttt{[x}_1\texttt{],...,x}_n\texttt{ in q}_n\texttt{[x}_1\texttt{,...,x}_{n-1}\texttt{]}\\ \texttt{where p[x}_1\texttt{,...,x}_n\texttt{]}\end{array}\right) :: \{\{t\}\}$$

Typing tree

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\ldots :: t_1}\quad \dfrac{}{\ldots :: t_2}}{\ldots :: t}\quad \dfrac{}{\ldots :: t_3}}{\ldots :: t_1}\quad \dfrac{}{\ldots :: t_2}}{\ldots :: t}\quad \dfrac{}{\ldots :: t_2}}{\ldots :: t}\quad \dfrac{}{\ldots}$$

$$5 :: t_3 \qquad t_3 \equiv int$$

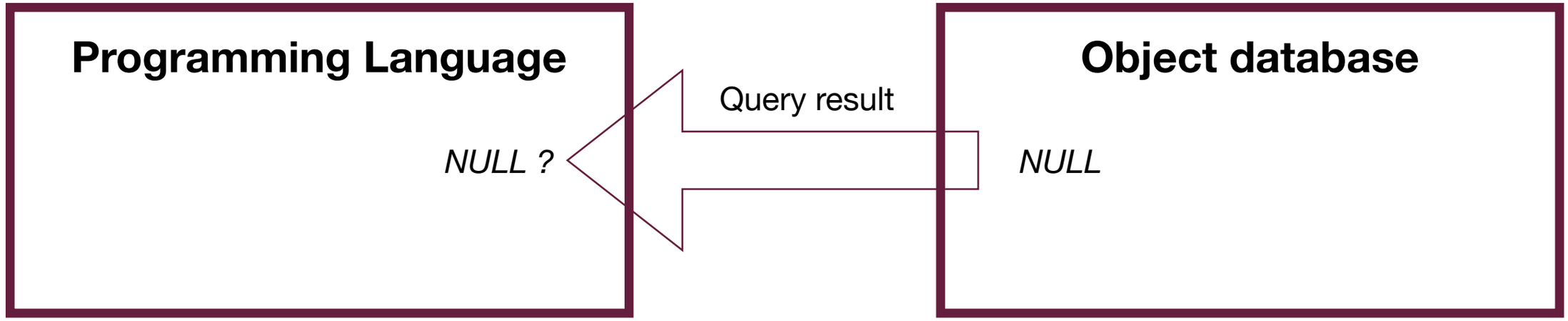
Run-time typing errors

Regardless query typing, run-time errors still exists:

- operations **min**, **max**, **avg**, **sum**, etc with empty set;
- operation **element** with a set having more than one element;
- division by zero;
- index out of bounds with arrays, lists, strings…;
- wrong casting;
- accessing properties of *nil*.

NULL value issue

Is there a *NULL* value in the Programming Language (*NULL* != *nil*)



Attribute specialisation issue

$$(\forall\, i \in [1, n], n \le m : t_i \le t'_i) \;\Rightarrow\; [a_1 : t_1 , \ldots , a_m : t_m] \le [a_1 : t'_1 , \ldots , a_n : t'_n]$$

```
class A {
  x: { myBool: boolean }
};

method set_x_to_true:boolean in class A {
    if(this.x.myBool == False) {
        this.x = {myBool:true};
        return true;
    }
    else return false;
};

class B extends A {
  x: { myBool: boolean, myInt: int }
};


method read_x_int:integer in class B {
    return this.x.myInt
};
```

All is well here…

… but this query breaks it all:

```
select b.read_x_int()
  from b in Some-B-Objects-Set
 where b.set_x_to_true();
```

So no attribute specialisation is allowed in practice

Covariation vs. Contrvariation

```
class Point {
 x: real,
 y: real
};

class ColorPoint extends Point {
 c: string  // x and y inherited from Point
};

method equal (p:Point):boolean
                    in class Point
{
  return ((this.x == p.x) && (this.y == p.y));
};

method equal (p:ColorPoint):boolean
                    in class ColorPoint
{
    return ( (this.x == p.x) &&
             (this.y == p.y) &&
             (this.c == p.c) )
};
```

All is well here…

… but then let's add this:

```
method break_it (p:Point):boolean
                            in class Point
{
   return p.equal (this)
};
```

… and write a query:

```
(new Point()).break_it(new ColorPoint())
```

Matematikos ir informatikos fakultetas

Arūnas Janeliūnas

Covariation vs. Contravariation

Class **C** method **m : A → B**

∨

Class **c** method **m : a → b**

It is safe to use method **c:m** instead of **C:m**, if:

- **c:m** can accept same arguments as **C:m** (**a** ≥ **A**)
- any code expecting results from **C:m** will accept results from **c:m** (**B** ≥ **b**)

Class **C** method **m : A → B**

∨ ∧I ∨V

Class **c** method **m : a → b**

that's *Contravariation* rule

Covariation vs. Contravariation

type-safe

Class **C** method **m : A → B**

V ΛI IV

Class **c** method **m : a → b**

Contravariation

practical

Class **C** method **m : A → B**

V IV IV

Class **c** method **m : a → b**

Covariation

Matematikos
ir informatikos
fakultetas

Arūnas Janeliūnas