Leveraging the Expressivity of Grounded Conjunctive Query Languages

Alissa Kaplunova, Ralf Möller,

Michael Wessel

Hamburg University of Technology (TUHH)

SSWS 07, November 27, 2007

TUHH
Technische Universität Hamburg-Harburg
TONES
boemie
Bootstrapping Ontology Evolution with Multimedia Information Extraction
Racer Systems

Background

- Grounded conjunctive query languages for the SemWeb are well established
  - no or only shallow reasoning:
    - e.g., RDF(S): RQL, RDQL, SPARQL, ...
  - more reasoning: DL & OWL QIs:
    - e.g., nRQL, SPARQL DL, SWRQL, ...
    - also consider inferred (axiomatic) „triples“
  - „grounded“ easier to implement than full (unrestricted) conjunctive queries
    - QA systems for unrestricted conjunctive queries exist (QuOnto), but for less expressive DLs
  - > focus on GCQs in RacerPro (nRQL)
    - nRQL offers more, but irrelevant for this talk

Simple Example Queries

- From the well-known university domain
  - retrieve all **student X course** pairs

$$ans(x,y) \leftarrow Student(x), takesCourse(x,y)$$

```
(retrieve (?x ?y)
   (and (?x Student) (?x ?y takesCourse)))
(retrieve (?x)
   (and (?x Student) (?x ?y takesCourse)))
```

- Semantics of CQs:

$$\{ (i_1, \ldots, i_m) \mid \exists \alpha : X \mapsto (i_1, \ldots, i_m), (i_1, \ldots, i_m) \in \text{inds}(\mathcal{O})^m, \mathcal{O} \models \exists Z . \alpha(atom_1) \wedge \cdots \wedge \alpha(atom_n) \}.$$

- For GCQs: remove $\exists Z.$, change to $\exists \alpha : Y$

TUHH
Technische Universität Hamburg-Harburg

Statement & Motivation

- Many practically important features still missing in available SemWeb QA systems
  - SQL-like aggregation operators: `count, sum, max, min, avg, ...`
    - many more imaginable
    - `group-by, order-by` needed? complicated...
  - queries with constraints on datatype values
    - often „ad hoc“ filter predicates in queries needed
    - predicate description language needed
    - problem: predicates often fixed (OWL 1.0)
  - (open world) reasoning with such extensions may be very difficult or even undecidable
    - but pragmatic solutions in practice needed

A General Purpose „Solution“

- Add a procedural extension / functional expression language to address these problems („Mini Lisp“)
  - concise **ad hoc specification** of arbitrary aggregation operators and filter predicates inline within the queries **-> flexibility**
  - termination-safe (no „unsafe queries“)
- Drawbacks of the approach:
  - filter predicates: no true concrete domain reasoning (or use CD of Racer -> true CD reasoning, but set of CD predicates is fixed)
  - aggregation operators: work on named ontology individuals only (OWA vs. CWA)

TUHH
Technische Universität Hamburg-Harburg

Examples in the University Domain

- Simple Aggregation
  - *how many courses does each student take?*
  - how many hours does a professor teach?
- Ad hoc filter
  - *which students take courses whose names contain the substring „42“ ;-)*
- Basic idea is simple:
  - allow **lambda expressions** as terms in ans predicate or `retrieve` head, resp.
  - lambdas are applied and their results included at that position in the answer tuple

TUHH
Technische Universität Hamburg-Harburg

Reminder: Lambda Expressions

- Formulation

  $\lambda\,(x_1, \ldots, x_n) \bullet body$

  – formal parameters: $x_1, \ldots, x_n$

- Application

  $((\lambda\,(x_1, \ldots, x_n) \bullet body)\, i_1, \ldots, i_n)$

  – applied to actual arguments: $i_1, \ldots, i_n$

- Reduction example

  $((\lambda\,(x,y) \bullet x+y)\; 3,4) \rightarrow 3+4 \rightarrow 7$

Lambda Expressions in MiniLisp

- Formulation
  `(lambda (x1 ... xn) body)`
- Application
  ```
  ((lambda (x1 ... xn) body)
     i1 ... in))
  ```
- Reduction example
  ```
  ((lambda (x y) (+ x y)) 3 4)
  -> (+ 3 4) -> 7
  ```
- Lambda filter: return ⊥ = `:reject`
- Aggregation: construct & pose subqueries

UD Filter Example

- All pairs with a course containing 42 in its name are rejected:

```
(retrieve (?x ((lambda (course)
                 (let ((cn
                   (first (datatype-fillers x #!:name))))
                    (if (search "42" cn)
                         cn
                       :reject)))
               ?y))

           (and (?x #!:Student)
                (?x ?y #!:takesCourse))
```

TUHH
Technische Universität Hamburg-Harburg

UD Aggregation Example

- Naive solution: for each student, a subquery is constructed and executed which retrieves the students courses:

```
(retrieve (?x
            ((lambda (student)
               (let ((courses
                      (retrieve '(?c)
                       `(,student ?c #!:takesCourse)))))
                 `(?num-courses ,(length courses))))
               ?x)))

        (?x #!:Student))
```

TUHH
Technische Universität Hamburg-Harburg

Semantics of GCQs with Lambdas

$$
\begin{aligned}
\{\, (j_1, \ldots, j_m) \mid \exists \alpha : \boldsymbol{Y} \mapsto (i_1, \ldots, i_k), (i_1, \ldots, i_k) \in \mathsf{inds}(\mathcal{O})^k, \\
\mathcal{O} \models \alpha(atom_1), \ldots \mathcal{O} \models \alpha(atom_n), \\
\text{such that for all } l \in 1 \ldots m\text{:} \\
j_l = \alpha(h_l) \quad \text{if } h_l \text{ is a variable,} \\
j_l = ((\lambda(v_1, \ldots, v_p) \bullet \ldots)\ \alpha(y_1), \ldots, \alpha(y_p)) \\
\text{if } j_l = ((\lambda(v_1, \ldots, v_p) \bullet \ \ldots)\ y_1, \ldots, y_p) \\
\text{and } j_l \neq \bot \,\}.
\end{aligned}
$$

TUHH
Technische Universität Hamburg-Harburg

MiniLisp in a Nutshell

- numbers, strings, symbols, lists
- cond. evaluation, file IO (HTML, XML)
- structure mapping and finite loops
- many of the standard Common Lisp functions for the supported datatypes
- access to all RacerPro API functions
- it is termination-safe, because
  - no infinite loops or lists
  - no **`defun`**, no **`setq`**
  - lambdas not first class, but special forms

    ~~**`((lambda (Y Y) (Y Y)) (lambda (Y Y) (Y Y))`**~~

TUHH
Technische Universität Hamburg-Harburg

Notes on Performance

- The analog of what MiniLisp is doing could also be implemented in a RacerPro client (e.g.) in Java, but
  - MiniLisp is efficently **executed on the RacerPro server**
    - no TCP socket communication latency / overhead, no string parsing and construction
  - dedicated optimizations (see below)
    - special precompilation optimization for subqueries being called from MiniLisp, so-called „promises“
  - next: simple benchmarks illustrating these issues

UD Filter Example

- Test with 1 LUBM university
  - 17174 individuals, 51207 concept / class assertions, 49336 role / property assertions
  - `(retrieve (?x) (?x Student))`
    7790 tuples, 5 seconds
  - ```
    (retrieve (?x ?y) (and (?x Student)
                           (?x ?y takesCourse)))
    ```
    21489 tuples, 5 seconds
- Filter („42“)
  - 432 tuples
  - MiniLisp: 6.4 (then 1.8) seconds
  - external Lisp: 38 (then 23) seconds
  - approx. 6 times faster

TUHH
Technische Universität Hamburg-Harburg
    ```

UD Aggregation Example

- Naive aggregation (number of courses):
  - 7790 tuples
  - MiniLisp: 26 (then 22) seconds
  - external Java / Lisp: Ctrl-c after 3 minutes
- MiniLisp is much faster, but there are still problems:
  - 7790 subqueries have to be parsed, optimized, compiled -> time and memory consuming!
  - nRQL maintains queries as objects; but even if the subqueries are immediately deleted, 7790 subqueries are constructed

A Special Optimization - Promises

Basic idea: replace the runtime query construction in the outer query

```
(... (retrieve '(?c)
              `(,student ?c #!:takesCourse))
.... )
```

with something like

```
(prepare-abox-query (?z)
                    (?x ?z #!:takesCourse)
                    :id :num-courses))

(.... (execute-query :num-courses) ... )
```

TUHH
Technische Universität Hamburg-Harburg

Promises Explained

Problem:?x can neither be treated as individual nor variable by the compiler:

```
(prepare-abox-query (?z)
                     (?x ?z #!:takesCourse)
                     :id :num-courses))
```

- not a variable (?x will be bound by outer query)
- not an individual (since ?x will change)
- the optimizer may treat ?x as an individual if we „promise“ that a binding for ?x will be supplied before execution

Aggregation Query with Promise

```
(with-future-bindings (?x)
   (prepare-abox-query (?z)
      (?x ?z #!:takesCourse))
      :id :num-courses))

(retrieve (?x
             ((lambda (x)
                 (with-nrql-settings (:bindings `((?x ,x)))
                   `(?num-courses
                     ,(length
                        (execute-or-reexecute-query
                                       :num-courses)))))
             ?x))
      (?x #!:Student))
```

Technische Universität Hamburg-Harburg

Effectiveness of Promises

- Naive aggregation without promise:
  - 7790 tuples
  - MiniLisp: 26 (then 22) seconds
  - external Java / Lisp: Ctrl-c after 3 minutes
- Naive aggregation with promise:
  - 2.5 seconds
  - speed up: approx. 10
  - the bigger the intermediate result sets, the more time you save

Conclusion

- MiniLisp is very flexible and handy
  - solves practical relevant problems
  - ad hoc solutions possible (no precompilation of „plugins“ for the the query engine required)
  - concise and (almost) declarative
  - **`lisp-to-xml, xml-to-lisp`**
- Aggregations have to be computed on the server („move the query, not the data“)
- The ideas could be applied in other query engines
  - but engine must offer query life cycle managment, optimization and compilation

Thanks!

λ

TUHH
Technische Universität Hamburg-Harburg

TONES

boemie
Bootstrapping Ontology Evolution with Multimedia Information Extraction

Racer Systems