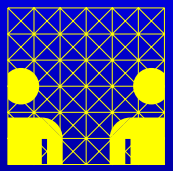
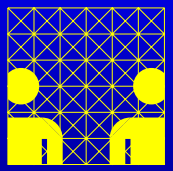


Extended Query Facilities for Racer and an Application to Software-Engineering Problems

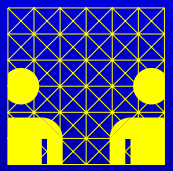
Volker Haarslev, Ralf Möller,
Ragnhild Van Der Straeten and Michael Wessel

Cognitive Systems Group
University of Hamburg
Germany



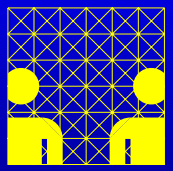
A New Pragmatic ABox Query Language for Racer

- Racer
  - Volker's & Ralf's ABox DL reasoner for $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$



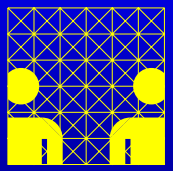
A New Pragmatic ABox Query Language for Racer

- Racer
  - Volker's & Ralf's ABox DL reasoner for $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$
  - offers ABox retrieval functions



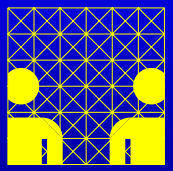
A New Pragmatic ABox Query Language for Racer

- Racer
  - Volker's & Ralf's ABox DL reasoner for $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$
  - offers ABox retrieval functions
    - `(concept-instances C)`



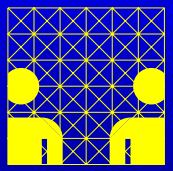
A New Pragmatic ABox Query Language for Racer

- Racer
  - Volker's & Ralf's ABox DL reasoner for $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$
  - offers ABox retrieval functions
    - `(concept-instances C)`
    - `(individual-fillers i R)`



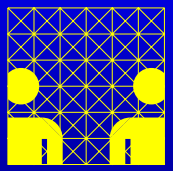
A New Pragmatic ABox Query Language for Racer

- Racer
  - Volker's & Ralf's ABox DL reasoner for $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$
  - offers ABox retrieval functions
    - `(concept-instances C)`
    - `(individual-fillers i R)`
    - . . . a few dozens more (see Racer manual)



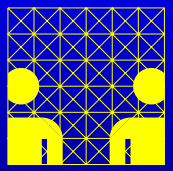
A New Pragmatic ABox Query Language for Racer

- Racer
  - Volker's & Ralf's ABox DL reasoner for $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$
  - offers ABox retrieval functions
  - Trend: recently, users create really big ABoxes …



A New Pragmatic ABox Query Language for Racer

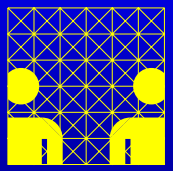
- Racer
  - Volker's & Ralf's ABox DL reasoner for $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$
  - offers ABox retrieval functions
  - Trend: recently, users create really big ABoxes …
  - … or want to navigate through the relational structure of an ABox (Ragnhild, DL 2003)



A New Pragmatic ABox Query Language for Racer

- Racer
  - Volker's & Ralf's ABox DL reasoner for $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$
  - offers ABox retrieval functions
  - Trend: recently, users create really big ABoxes ...
  - ... or want to navigate through the relational structure of an ABox (Ragnhild, DL 2003)

  $\Rightarrow$ more sophisticated "LOOM-like" queries demanded by users

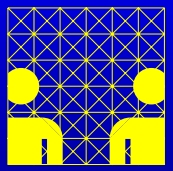


A New Pragmatic ABox Query Language for Racer

- Racer
  - Volker's & Ralf's ABox DL reasoner for $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$
  - offers ABox retrieval functions
  - Trend: recently, users create really big ABoxes ...
  - ... or want to navigate through the relational structure of an ABox (Ragnhild, DL 2003)

  $\Rightarrow$ more sophisticated "LOOM-like" queries demanded by users

$\Rightarrow$ RQL: Racer Query Language

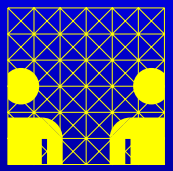


A New Pragmatic ABox Query Language for Racer

- Racer
  - Volker's & Ralf's ABox DL reasoner for $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$
  - offers ABox retrieval functions
  - Trend: recently, users create really big ABoxes ...
  - ... or want to navigate through the relational structure of an ABox (Ragnhild, DL 2003)

  $\Rightarrow$ more sophisticated "LOOM-like" queries demanded by users

$\Rightarrow$ nRQL: new Racer Query Language



A New Pragmatic ABox Query Language for Racer

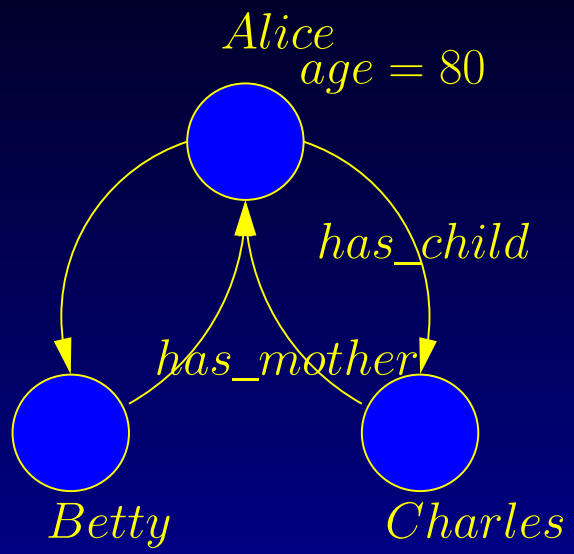
- Racer
  - Volker's & Ralf's ABox DL reasoner for $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$
  - offers ABox retrieval functions
  - Trend: recently, users create really big ABoxes ...
  - ... or want to navigate through the relational structure of an ABox (Ragnhild, DL 2003)

  $\Rightarrow$ more sophisticated "LOOM-like" queries demanded by users

$\Rightarrow$ nRQL: new Racer Query Language

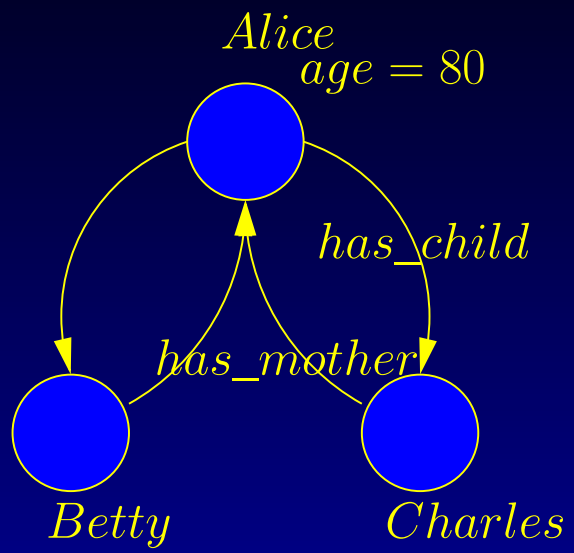
- pragmatic approach, straightforward combination of Racer's ABox API functions

Motivating Simple Example



$$mother(alice),\ age(alice) = 80,$$
$$has\_mother(betty, alice),$$
$$has\_mother(charles, alice),$$
$$mother(betty),\ mother(betty)$$

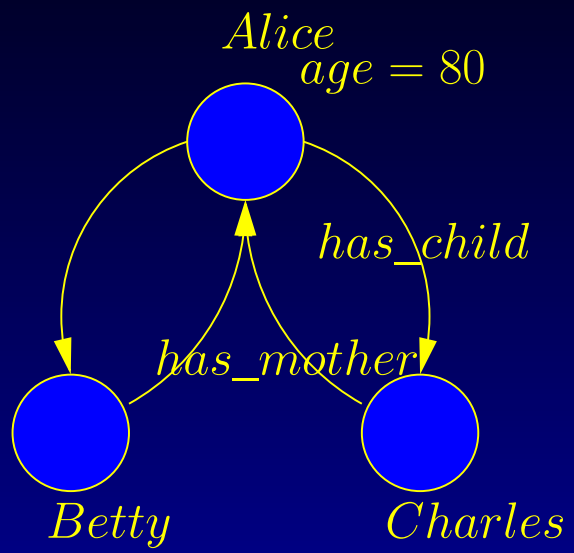
Motivating Simple Example



$mother(alice)$, $age(alice) = 80$,
$has\_mother(betty, alice)$,
$has\_mother(charles, alice)$,
$mother(betty)$, $mother(betty)$

- How do we retrieve pairs of siblings (e.g., $\{(betty, charles)\}$)?

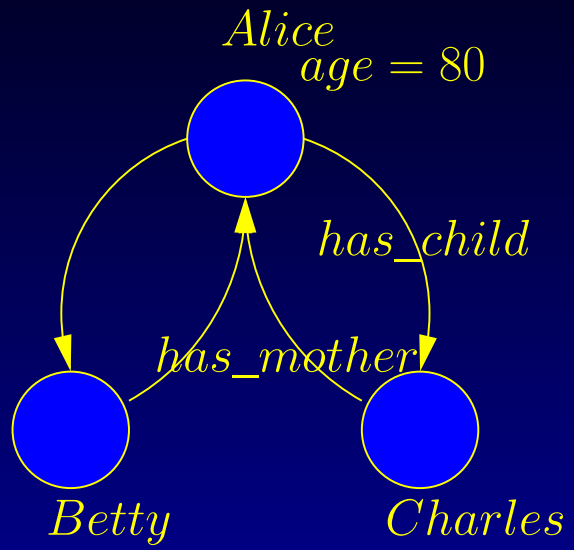
Motivating Simple Example



$mother(alice)$, $age(alice) = 80$,
$has\_mother(betty, alice)$,
$has\_mother(charles, alice)$,
$mother(betty)$, $mother(betty)$

- How do we retrieve pairs of siblings (e.g., $\{(betty, charles)\}$)?
- ⊖ write a “search program” (not declarative)

Motivating Simple Example

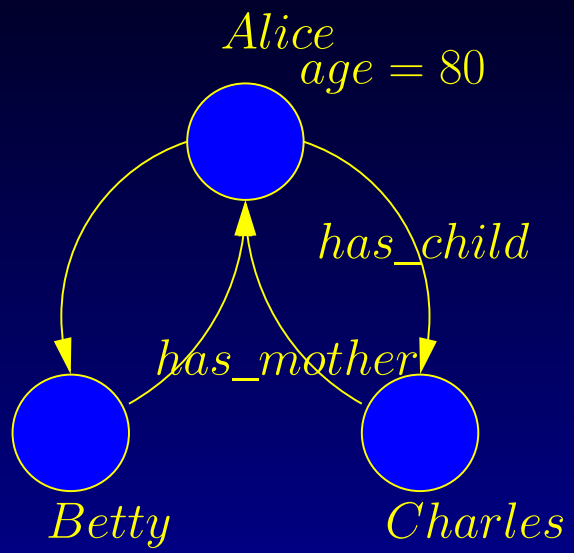


$mother(alice)$, $age(alice) = 80$,
$has\_mother(betty, alice)$,
$has\_mother(charles, alice)$,
$mother(betty)$, $mother(betty)$

- How do we retrieve pairs of siblings (e.g., $\{(betty, charles)\}$)?
- ⊖ write a “search program” (not declarative)
- ⊕ use nRQL:

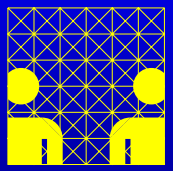
```
(retrieve (?x ?y)
          (and (has-child ?z ?x)
               (has-child ?z ?y)))
```

Motivating Simple Example



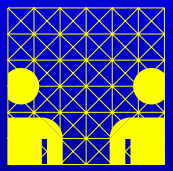
$mother(alice)$, $age(alice) = 80$,
$has\_mother(betty, alice)$,
$has\_mother(charles, alice)$,
$mother(betty)$, $mother(betty)$

- How do we retrieve pairs of siblings (e.g., $\{(betty, charles)\}$)?
  - ⊖ write a "search program" (not declarative)
  - ⊕ use nRQL:
    - ⇒
```
(((?X CHARLES) (?Y BETTY))
 ((?X BETTY) (?Y CHARLES)))
```



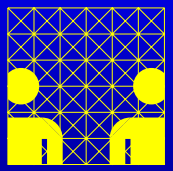
Motivating Example

- Work of Ragnhild v.d. Straeten et al.



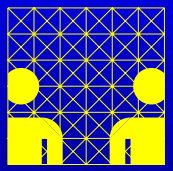
Motivating Example

- Work of Ragnhild v.d. Straeten et al.
  - checking for inconsistencies in UML diagrams



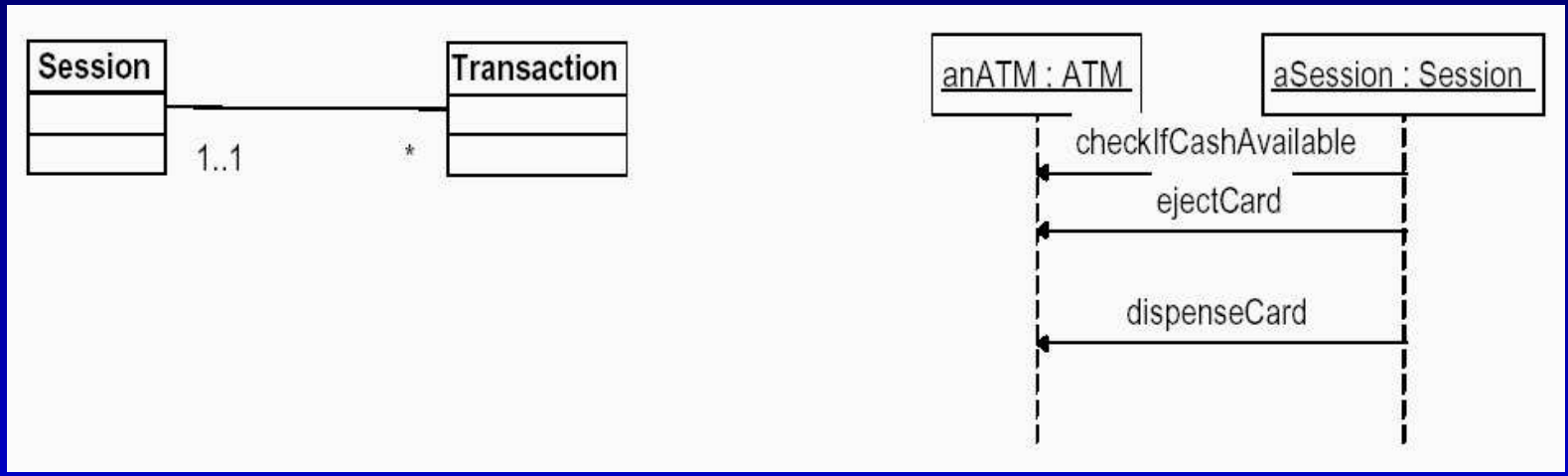
Motivating Example

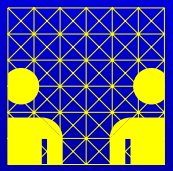
- Work of Ragnhild v.d. Straeten et al.
  - checking for inconsistencies in UML diagrams
  - e.g., are there instances of classes for which no class has been defined? (CWA!)



Motivating Example

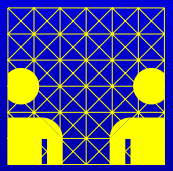
- Work of Ragnhild v.d. Straeten et al.
  - checking for inconsistencies in UML diagrams
  - e.g., are there instances of classes for which no class has been defined? (CWA!)





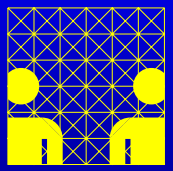
Motivating Example

- Work of Ragnhild v.d. Straeten et al.
  - checking for inconsistencies in UML diagrams
  - e.g., are there instances of classes for which no class has been defined? (CWA!)
  - UML diagrams are represented as ABoxes



Motivating Example

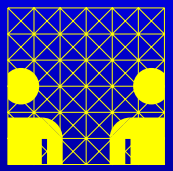
- Work of Ragnhild v.d. Straeten et al.
  - checking for inconsistencies in UML diagrams
  - e.g., are there instances of classes for which no class has been defined? (CWA!)
  - UML diagrams are represented as ABoxes
  - used LOOM, but wants Racer (DL ’03)



Motivating Example

- Work of Ragnhild v.d. Straeten et al.
  - checking for inconsistencies in UML diagrams
  - e.g., are there instances of classes for which no class has been defined? (CWA!)
  - UML diagrams are represented as ABoxes
  - used LOOM, but wants Racer (DL ’03)
- LOOM:

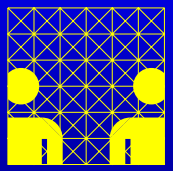
```
(do-retrieve (?object ?class)
   (:and (Object ?object)
         (Instance-of-class ?object ?class)
         (has-classmodel ?class NIL)))
```



Motivating Example

- Work of Ragnhild v.d. Straeten et al.
  - checking for inconsistencies in UML diagrams
  - e.g., are there instances of classes for which no class has been defined? (CWA!)
  - UML diagrams are represented as ABoxes
  - used LOOM, but wants Racer (DL ’03)
- Racer without nRQL:

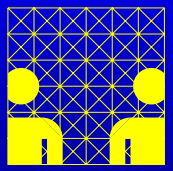
```
(loop for ?o in (concept-instances object) do
  (loop for ?c in
        (ret.-ind.-fillers ?o 'instance-of-class)
    when (null (retr.-ind.-fillers
                ?c 'has-classmodel)) do
      (format t ''Classless instance'')))
```



Motivating Example

- Work of Ragnhild v.d. Straeten et al.
  - checking for inconsistencies in UML diagrams
  - e.g., are there instances of classes for which no class has been defined? (CWA!)
  - UML diagrams are represented as ABoxes
  - used LOOM, but wants Racer (DL ’03)
- Racer with nRQL:

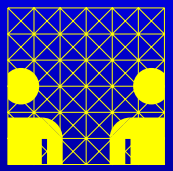
```
(retrieve (?o ?c)
    (and (?c class) (?o object)
         (?o ?c instance-of)
         (not (?c (:has-known-successor
                   has-classmodel)))))
```



Motivating Example

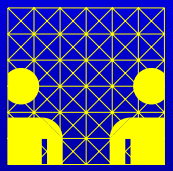
- Work of Ragnhild v.d. Straeten et al.
  - checking for inconsistencies in UML diagrams
  - e.g., are there instances of classes for which no class has been defined? (CWA!)
  - UML diagrams are represented as ABoxes
  - used LOOM, but wants Racer (DL ’03)
- Racer with nRQL:

```
(retrieve (?o ?c)
    (and (?c class) (?o object)
         (?o ?c instance-of)
         (?c NIL has-classmodel)))
```



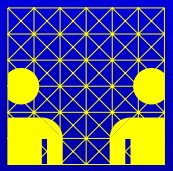
Pragmatic Design Decisions

- Variables range over ABox individuals only



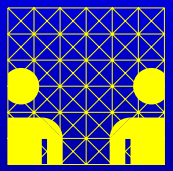
Pragmatic Design Decisions

- Variables range over ABox individuals only
  - no distinction of must-bind, may-bind and do-not-bind variables (OWL-QL)
  - variables are always distinguished resp. must-bind



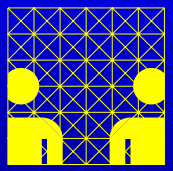
Pragmatic Design Decisions

- Variables range over ABox individuals only
  - no distinction of must-bind, may-bind and do-not-bind variables (OWL-QL)
  - variables are always distinguished resp. must-bind
  - no automatic “rolling up”, e.g. if `?y` is not distinguished (no bindings wanted!)
    `(?x ?y R)` $\Rightarrow$ `(?x (some R top))`



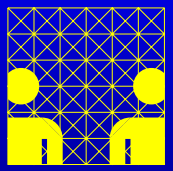
Pragmatic Design Decisions

- Variables range over ABox individuals only
  - no distinction of must-bind, may-bind and do-not-bind variables (OWL-QL)
  - variables are always distinguished resp. must-bind
  - no automatic “rolling up”, e.g. if `?y` is not distinguished (no bindings wanted!)
    `(?x ?y R)` $\Rightarrow$ `(?x (some R top))`
- Support for “Negation as Failure”



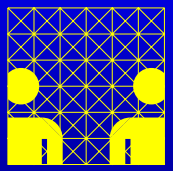
Pragmatic Design Decisions

- Variables range over ABox individuals only
  - no distinction of must-bind, may-bind and do-not-bind variables (OWL-QL)
  - variables are always distinguished resp. must-bind
  - no automatic “rolling up”, e.g. if `?y` is not distinguished (no bindings wanted!)
    `(?x ?y R)` $\Rightarrow$ `(?x (some R top))`
- Support for “Negation as Failure”
  - `(retrieve (?x) (?x (not woman)))`
  - `(retrieve (?x) (not (?x woman)))`
- Support for disjunctive queries



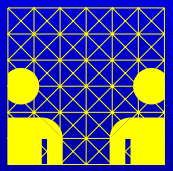
Pragmatic Design Decisions

- Variables range over ABox individuals only
  - no distinction of must-bind, may-bind and do-not-bind variables (OWL-QL)
  - variables are always distinguished resp. must-bind
  - no automatic "rolling up", e.g. if `?y` is not distinguished (no bindings wanted!)
    `(?x ?y R)` $\Rightarrow$ `(?x (some R top))`
- Support for "Negation as Failure"
  - `(retrieve (?x) (?x (not woman)))`
  - `(retrieve (?x) (not (?x woman)))`
- Support for disjunctive queries
- Support for concrete domain predicates



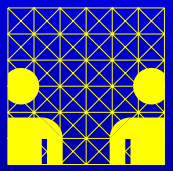
nRQL Syntax - Query Atoms

- Unary Atoms



nRQL Syntax - Query Atoms

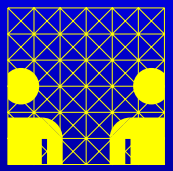
- Unary Atoms
  - Concept query atoms: $woman(alice)$



nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $woman(alice)$

```
(retrieve () (alice woman))
⇒ T
```

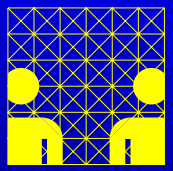


nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $woman(alice)$

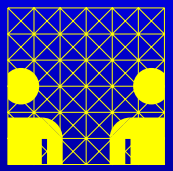
```
(retrieve () (alice woman))
⇒ T

(individual-instance? alice woman)
⇒ T
```



nRQL Syntax - Query Atoms

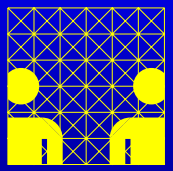
- Unary Atoms
  - Concept query atoms: $woman(x)$



nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $woman(x)$

```
(retrieve (?x) (?x woman))
⇒ (((?x alice)) ((?x betty)))
```

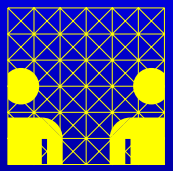


nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $woman(x)$

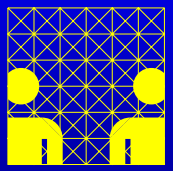
```
(retrieve (?x) (?x woman))
⇒ (((?x alice)) ((?x betty)))

(concept-instances woman)
⇒ (betty alice)
```



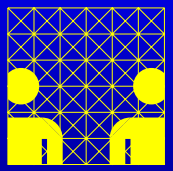
nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$



nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(alice, has\_child)$

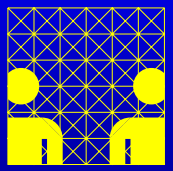


nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(alice, has\_child)$

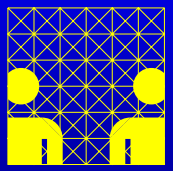
```
(retrieve ()
    (alice (:has-known-successor
                has-child)))
```

$\Rightarrow$ `T`



nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(x, has\_child)$

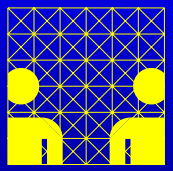


nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(x, has\_child)$

```
(retrieve ()
    (?x (:has-known-successor
             has-child)))
```

$\Rightarrow$ `(((?x alice)))`



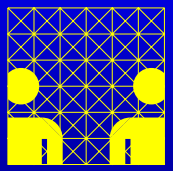
nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(x, has\_child)$

```
(retrieve ()
    (?x (:has-known-successor
             has-child)))
```

$\Rightarrow$ `(((?x alice)))`

  - What about Betty? ABox: $mother(betty)$



nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(x, has\_child)$

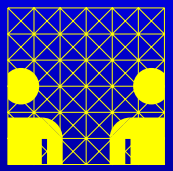
```
(retrieve ()
    (?x (:has-known-successor
             has-child)))
```

$\Rightarrow$ `(((?x alice)))`

  - What about Betty? ABox: $mother(betty)$
  - `(retrieve (?x)`

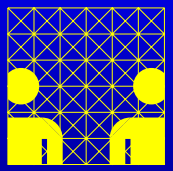
```
    (?x (some has-child top)))
```

$\Rightarrow$ `(((?x alice)) ((?x betty)))`



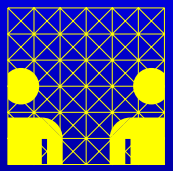
nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$



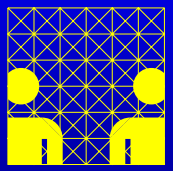
nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms



nRQL Syntax - Query Atoms

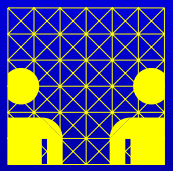
- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $has\_child(alice, charles)$



nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $has\_child(alice, charles)$

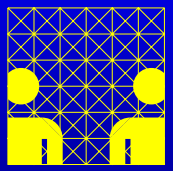
```
(retrieve ()
    (alice charles has-child))
⇒ T
```



nRQL Syntax - Query Atoms

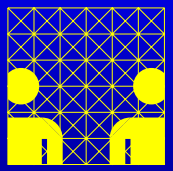
- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $has\_child(alice, charles)$

```
(retrieve ()
    (alice charles has-child))
⇒ T
(individuals-related? alice charles
                      has-child)
⇒ T
```



nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $has\_child(alice, y)$

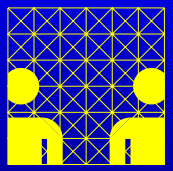


nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $has\_child(alice, y)$

```
(retrieve (?y) (alice ?y has-child))
```

$\Rightarrow$ `(((?y betty)) ((?y charles)))`

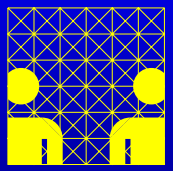


nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $has\_child(alice, y)$

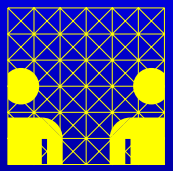
```
(retrieve (?y) (alice ?y has-child))
⇒ (((?y betty)) ((?y charles)))

(individual-fillers alice has-child)
⇒ (betty charles)
```



nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $has\_child(x, y)$



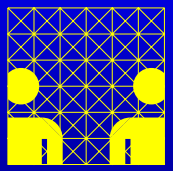
nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $has\_child(x, y)$

```
(retrieve (?x ?y) (?x ?y has-child))
```

$\Rightarrow$

```
(((?x alice) (?y betty))
 ((?x alice) (?y charles)))
```

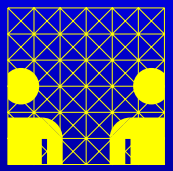


nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $has\_child(x, y)$

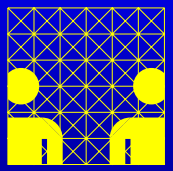
```
(retrieve (?x ?y) (?x ?y has-child))
⇒ (((?x alice) (?y betty))
   ((?x alice) (?y charles)))

(related-individuals has-child)
⇒ ((alice betty) (alice charles))
```



nRQL Syntax - Query Atoms

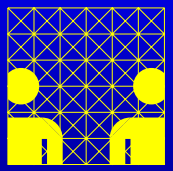
- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $R(a, b)$



nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $R(a, b)$
  - Constraint query atoms:

    $= (has\_mother \circ age(x), has\_mother \circ age(y))$

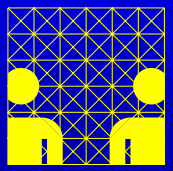


nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $R(a, b)$
  - Constraint query atoms:

    $= (has\_mother \circ age(x), has\_mother \circ age(y))$

```
(retrieve (?x ?y)
    (?x ?y (:constraint (h-mother age)
                        (h-mother age) =)))
```



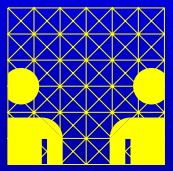
nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $R(a, b)$
  - Constraint query atoms:

    $= (has\_mother \circ age(x), has\_mother \circ age(y))$

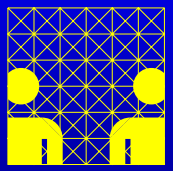
```
(retrieve (?x ?y)
    (?x ?y (:constraint (h-mother age)
                        (h-mother age) =)))
```

$\Rightarrow$ `( ((?x alice) (?y betty)) ...)`



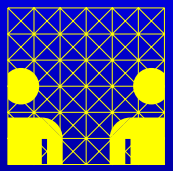
nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $R(a, b)$
  - Constraint query atoms: $P(f_1 \circ \cdots \circ f_n(a), g_1 \circ \cdots \circ g_m(b))$



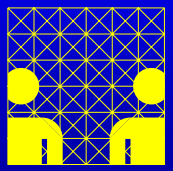
nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $R(a, b)$
  - Constraint query atoms: $P(f_1 \circ \cdots \circ f_n(a), g_1 \circ \cdots \circ g_m(b))$
- "Negated" Atoms



nRQL Syntax - Query Atoms

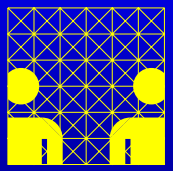
- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $R(a, b)$
  - Constraint query atoms: $P(f_1 \circ \cdots \circ f_n(a), g_1 \circ \cdots \circ g_m(b))$
- "Negated" Atoms
  - $\backslash(C(a))$



nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $R(a, b)$
  - Constraint query atoms: $P(f_1 \circ \cdots \circ f_n(a), g_1 \circ \cdots \circ g_m(b))$
- "Negated" Atoms
  - $\backslash(C(a))$

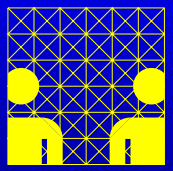
```
(retrieve () (charles woman))
⇒ NIL
```



nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $R(a, b)$
  - Constraint query atoms: $P(f_1 \circ \cdots \circ f_n(a), g_1 \circ \cdots \circ g_m(b))$
- "Negated" Atoms
  - $\backslash(C(a))$

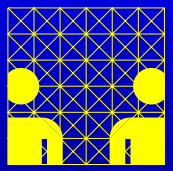
```
(retrieve () (not (charles woman)))
⇒ T
```



nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $R(a, b)$
  - Constraint query atoms: $P(f_1 \circ \cdots \circ f_n(a), g_1 \circ \cdots \circ g_m(b))$
- "Negated" Atoms
  - $\backslash(C(a))$

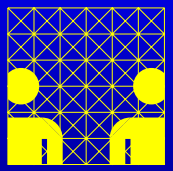
```
(retrieve () (charles (not woman)))
⇒ NIL
```



nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $R(a, b)$
  - Constraint query atoms: $P(f_1 \circ \cdots \circ f_n(a), g_1 \circ \cdots \circ g_m(b))$
- "Negated" Atoms
  - $\backslash(C(a))$

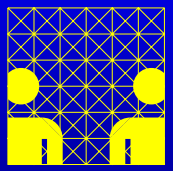
```
(retrieve (?x) (?x woman))
⇒ (((?x betty)) ((?x alice)))
```



nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $R(a, b)$
  - Constraint query atoms: $P(f_1 \circ \cdots \circ f_n(a), g_1 \circ \cdots \circ g_m(b))$
- "Negated" Atoms
  - $\backslash(C(a))$

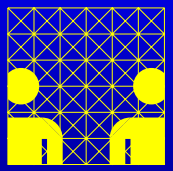
```
(retrieve (?x) (not (?x woman)))
⇒ (((?x charles)))
```



nRQL Syntax - Query Atoms

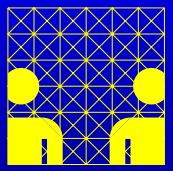
- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $R(a, b)$
  - Constraint query atoms: $P(f_1 \circ \cdots \circ f_n(a), g_1 \circ \cdots \circ g_m(b))$
- "Negated" Atoms
  - $\backslash(C(a))$

```
(retrieve (?x) (?x (not woman)))
⇒ ()
```



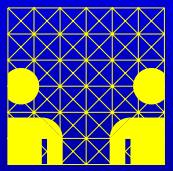
nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $R(a, b)$
  - Constraint query atoms: $P(f_1 \circ \cdots \circ f_n(a), g_1 \circ \cdots \circ g_m(b))$
- "Negated" Atoms
  - $\backslash(C(a))$
  - $\backslash(has\_known\_successor(a, R))$



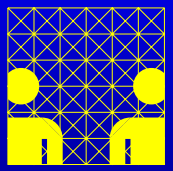
nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $R(a, b)$
  - Constraint query atoms: $P(f_1 \circ \cdots \circ f_n(a), g_1 \circ \cdots \circ g_m(b))$
- "Negated" Atoms
  - $\backslash(C(a))$
  - $\backslash(has\_known\_successor(a, R))$
    LOOM: `(retrieve (?x) (?x NIL R))`



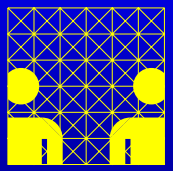
nRQL Syntax - Query Atoms

- Unary Atoms
  - Concept query atoms: $C(a)$
  - "Has known successor" atoms: $has\_known\_successor(a, R)$
- Binary Atoms
  - Role query atoms: $R(a, b)$
  - Constraint query atoms: $P(f_1 \circ \cdots \circ f_n(a), g_1 \circ \cdots \circ g_m(b))$
- "Negated" Atoms
  - $\backslash(C(a))$
  - $\backslash(has\_known\_successor(a, R))$
  - $\backslash R(a, b)$, $\backslash P(f_1 \circ \cdots \circ f_n(a), g_1 \circ \cdots \circ g_m(b))$



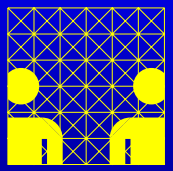
Semantics for Query Atoms

- Atoms use variables and/or individuals



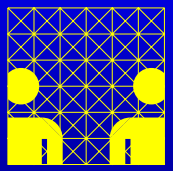
Semantics for Query Atoms

- Atoms use variables and/or individuals
- Variables are always “distinguished”



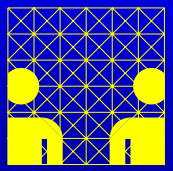
Semantics for Query Atoms

- Atoms use variables and/or individuals
- Variables are always “distinguished”
- Default: UNA for variables (non-UNA available)



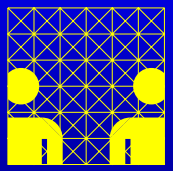
Semantics for Query Atoms

- Atoms use variables and/or individuals
- Variables are always “distinguished”
- Default: UNA for variables (non-UNA available)
- “Negation as Failure” / complement with “\”



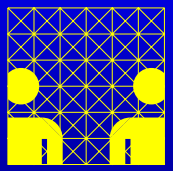
Semantics for Query Atoms

- Atoms use variables and/or individuals
- Variables are always “distinguished”
- Default: UNA for variables (non-UNA available)
- “Negation as Failure” / complement with “$\setminus$”
- For unary atoms $C(a)$



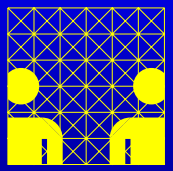
Semantics for Query Atoms

- Atoms use variables and/or individuals
- Variables are always “distinguished”
- Default: UNA for variables (non-UNA available)
- “Negation as Failure” / complement with “$\setminus$”
- For unary atoms $C(a)$
  - $\mathsf{inds}(\mathcal{A}) = \mathsf{answer}(C(a)) \cup \mathsf{answer}(\setminus C(a))$
  - $\mathsf{answer}(C(a)) \cap \mathsf{answer}(\setminus C(a)) = \emptyset$



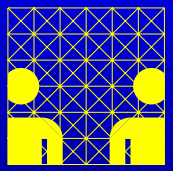
Semantics for Query Atoms

- Atoms use variables and/or individuals
- Variables are always “distinguished”
- Default: UNA for variables (non-UNA available)
- “Negation as Failure” / complement with “$\backslash$”
- For unary atoms $C(a)$
  - $\mathsf{inds}(\mathcal{A}) = \mathsf{answer}(C(a)) \cup \mathsf{answer}(\backslash C(a))$
  - $\mathsf{answer}(C(a)) \cap \mathsf{answer}(\backslash C(a)) = \emptyset$
- For binary atoms $R(a, b)$



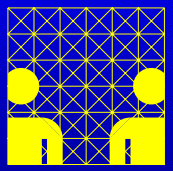
Semantics for Query Atoms

- Atoms use variables and/or individuals
- Variables are always “distinguished”
- Default: UNA for variables (non-UNA available)
- “Negation as Failure” / complement with “$\backslash$”
- For unary atoms $C(a)$
  - $\mathsf{inds}(\mathcal{A}) = \mathsf{answer}(C(a)) \cup \mathsf{answer}(\backslash C(a))$
  - $\mathsf{answer}(C(a)) \cap \mathsf{answer}(\backslash C(a)) = \emptyset$
- For binary atoms $R(a, b)$
  - $\mathsf{inds}(\mathcal{A})^2 =$
    $\mathsf{answer}(R(a, b)) \cup \mathsf{answer}(\backslash R(a, b)) \cup \mathsf{Id}$



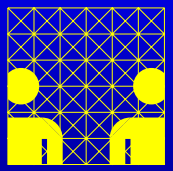
Semantics for Query Atoms

- Atoms use variables and/or individuals
- Variables are always “distinguished”
- Default: UNA for variables (non-UNA available)
- “Negation as Failure” / complement with “$\backslash$”
- For unary atoms $C(a)$
  - $\mathsf{inds}(\mathcal{A}) = \mathsf{answer}(C(a)) \cup \mathsf{answer}(\backslash C(a))$
  - $\mathsf{answer}(C(a)) \cap \mathsf{answer}(\backslash C(a)) = \emptyset$
- For binary atoms $R(a, b)$
  - $\mathsf{inds}(\mathcal{A})^2 =$
    $\mathsf{answer}(R(a, b)) \cup \mathsf{answer}(\backslash R(a, b)) \cup \mathsf{Id}$
  - $\mathsf{answer}(R(a, b)) \cap \mathsf{answer}(\backslash R(a, b)) = \emptyset$



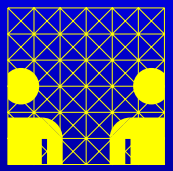
Neg. Atoms with Individuals?

- $atom$ and $\backslash(atom)$ are always complementary



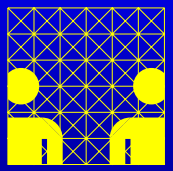
Neg. Atoms with Individuals?

- $atom$ and $\backslash(atom)$ are always complementary
- `(retrieve () (betty woman))`
  $\Rightarrow$ `T`



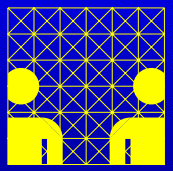
Neg. Atoms with Individuals?

- $atom$ and $\backslash(atom)$ are always complementary
- `(retrieve (betty) (betty woman))`
  $\Rightarrow$ `(((betty betty)))`



Neg. Atoms with Individuals?

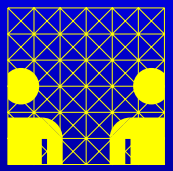
- $atom$ and $\backslash(atom)$ are always complementary
- `(retrieve (betty) (betty man))`
  $\Rightarrow$ `()`



Neg. Atoms with Individuals?

- $atom$ and $\backslash(atom)$ are always complementary
- `(retrieve (betty) (not (betty man)))`

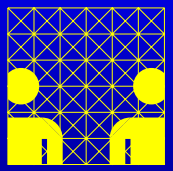
  $\Rightarrow$ `(((betty charles) (betty alice)`
  `(betty betty)))`



Neg. Atoms with Individuals?

- $atom$ and $\backslash(atom)$ are always complementary
- `(retrieve (?x) (not (?x man)))`

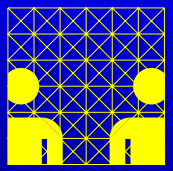
  $\Rightarrow$ `(((?x charles) (?x alice)`
  `(?x betty)))`



Neg. Atoms with Individuals?

- $atom$ and $\backslash(atom)$ are always complementary
- `(retrieve (betty) (not (betty man)))`

  $\Rightarrow$ `(((betty charles) (betty alice)`
  `(betty betty)))`

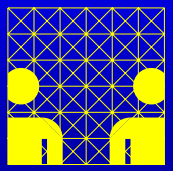


Neg. Atoms with Individuals?

- $atom$ and $\backslash(atom)$ are always complementary
- `(retrieve (betty) (not (betty man)))`

  $\Rightarrow$ `(((betty charles) (betty alice)`
  `(betty betty)))`

$\Rightarrow$ "negated individuals" turn into variables



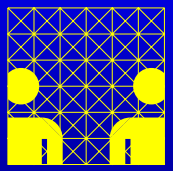
Neg. Atoms with Individuals?

- $atom$ and $\backslash(atom)$ are always complementary
- `(retrieve (betty) (not (betty man)))`

  $\Rightarrow$ `(((betty charles) (betty alice)`
  `(betty betty)))`

$\Rightarrow$ "negated individuals" turn into variables

- Add a `bind-individual` conjunct



Neg. Atoms with Individuals?

- $atom$ and $\backslash(atom)$ are always complementary
- `(retrieve (betty) (not (betty man)))`

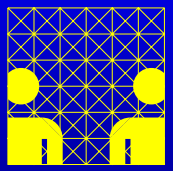
  $\Rightarrow$ `(((betty charles) (betty alice)`
  `(betty betty)))`

$\Rightarrow$ "negated individuals" turn into variables

- Add a `bind-individual` conjunct

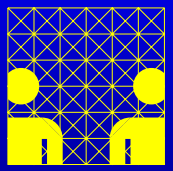
```
(retrieve (betty)
    (and (not (betty man))
         (bind-individual betty)))
```

$\Rightarrow$ `(((betty betty)))`



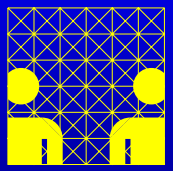
Query Bodies

- Each query atom is a body



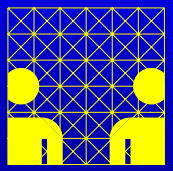
Query Bodies

- Each query atom is a body
- If b_i are query bodies, then also



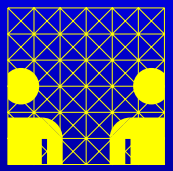
Query Bodies

- Each query atom is a body
- If b_i are query bodies, then also
  - $\backslash b_1, b_1 \wedge \cdots \wedge b_n, b_1 \vee \cdots \vee b_n$



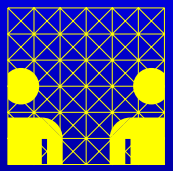
Query Bodies

- Each query atom is a body
- If b_i are query bodies, then also
  - $\backslash b_1, b_1 \wedge \cdots \wedge b_n, b_1 \vee \cdots \vee b_n$
- Meaning of “or”
  - TBox: $\top \Rightarrow woman \sqcup man$



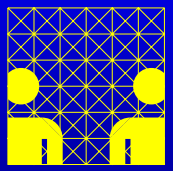
Query Bodies

- Each query atom is a body
- If b_i are query bodies, then also
  - $\backslash b_1, b_1 \wedge \cdots \wedge b_n, b_1 \vee \cdots \vee b_n$
- Meaning of "or"
  - TBox: $\top \Rightarrow woman \sqcup man$
  - `(retrieve (?x) (?x top))`
    $\Rightarrow$ `(((?x alice)) ((?x betty)) ((?x charles)))`



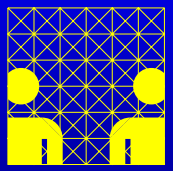
Query Bodies

- Each query atom is a body
- If b_i are query bodies, then also
  - $\backslash b_1, b_1 \wedge \cdots \wedge b_n, b_1 \vee \cdots \vee b_n$
- Meaning of "or"
  - TBox: $\top \Rightarrow woman \sqcup man$
  - `(retrieve (?x) (?x (or woman man)))`
    $\Rightarrow$ `(((?x alice)) ((?x betty)) ((?x charles)))`



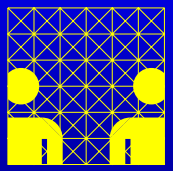
Query Bodies

- Each query atom is a body
- If b_i are query bodies, then also
  - $\backslash b_1, b_1 \wedge \cdots \wedge b_n, b_1 \vee \cdots \vee b_n$
- Meaning of "or"
  - TBox: $\top \Rightarrow woman \sqcup man$
  - `(retrieve (?x) (?x (or woman man)))`
    $\Rightarrow$ `(((?x alice)) ((?x betty)) ((?x charles)))`
  - `(retrieve (?x) (or (?x woman)`
    `(?x man)))`
    $\Rightarrow$ `(((?x alice)) ((?x betty)))`



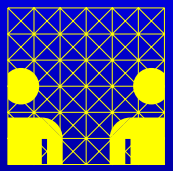
Query Bodies

- Each query atom is a body
- If b_i are query bodies, then also
  - $\backslash b_1, b_1 \wedge \cdots \wedge b_n, b_1 \vee \cdots \vee b_n$
- Meaning of "or"
  - TBox: $\top \Rightarrow woman \sqcup man$
  - `(retrieve (?x) (?x (or woman man)))`
    $\Rightarrow$ `(((?x alice)) ((?x betty)) ((?x charles)))`
  - `(retrieve (?x) (or (?x woman)`
    `(?x man)))`
    $\Rightarrow$ `(((?x alice)) ((?x betty)))`
  - `(concept-instances woman)` $\cup$
    `(concept-instances man)`
    $\subset$ `(concept-instances (or woman man))`



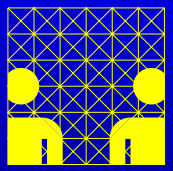
Arity of Query Bodies

- Arity of `(and (?x C) (?y D))`: 2



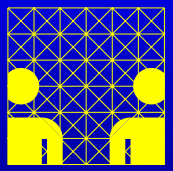
Arity of Query Bodies

- Arity of `(and (?x C) (?y D))`: 2
- Arity of `(or (?x C) (?y D))`: ?



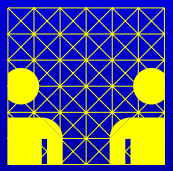
Arity of Query Bodies

- Arity of `(and (?x C) (?y D))`: 2
- Arity of `(or (?x C) (?y D))`: ?
- We want the NNF:



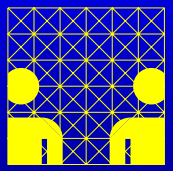
Arity of Query Bodies

- Arity of `(and (?x C) (?y D))`: 2
- Arity of `(or (?x C) (?y D))`: ?
- We want the NNF:
  - `(not (and (not (?x C))`
`(not (?y D))))`



Arity of Query Bodies

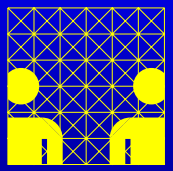
- Arity of `(and (?x C) (?y D))`: 2
- Arity of `(or (?x C) (?y D))`: ?
- We want the NNF:
  - `(not (and (not (?x C))`
`(not (?y D))))`
  - `NOT` preserves arity



Arity of Query Bodies

- Arity of `(and (?x C) (?y D))`: 2
- Arity of `(or (?x C) (?y D))`: ?
- We want the NNF:
  - `(not (and (not (?x C))`
`(not (?y D))))`
  - `NOT` preserves arity

  $\Rightarrow$ Arity of `(or (?x C) (?y D))`: 2

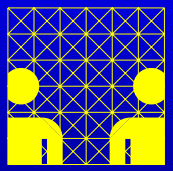


Arity of Query Bodies

- Arity of `(and (?x C) (?y D))`: 2
- Arity of `(or (?x C) (?y D))`: ?
- We want the NNF:
  - `(not (and (not (?x C))`
`(not (?y D))))`
  - `NOT` preserves arity

  $\Rightarrow$ Arity of `(or (?x C) (?y D))`: 2
- Internally rewritten into

```
(or (and (?x C) (?y TOP))
    (?y D) (?x TOP))
```



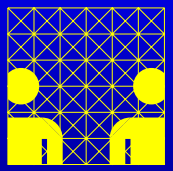
Arity of Query Bodies

- Arity of `(and (?x C) (?y D))`: 2
- Arity of `(or (?x C) (?y D))`: ?
- We want the NNF:
  - `(not (and (not (?x C))`
    `(not (?y D))))`
  - `NOT` preserves arity

  $\Rightarrow$ Arity of `(or (?x C) (?y D))`: 2
- Internally rewritten into

```
(or (and (?x C) (?y TOP))
    (?y D) (?x TOP))
```

$\Rightarrow$ If C as well as D instances are present, then



Arity of Query Bodies

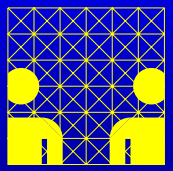
- Arity of `(and (?x C) (?y D))`: 2
- Arity of `(or (?x C) (?y D))`: ?
- We want the NNF:
  - `(not (and (not (?x C))`
    `(not (?y D))))`
  - `NOT` preserves arity

  $\Rightarrow$ Arity of `(or (?x C) (?y D))`: 2
- Internally rewritten into

```
(or (and (?x C) (?y TOP))
         (?y D) (?x TOP))
```

$\Rightarrow$ If C as well as D instances are present, then

```
(retrieve (?x) (or (?x C) (?y D)))
```



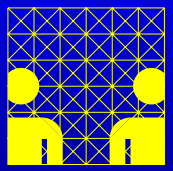
Arity of Query Bodies

- Arity of `(and (?x C) (?y D))`: 2
- Arity of `(or (?x C) (?y D))`: ?
- We want the NNF:
  - `(not (and (not (?x C))`
    `(not (?y D))))`
  - `NOT` preserves arity

  $\Rightarrow$ Arity of `(or (?x C) (?y D))`: 2
- Internally rewritten into

```
(or (and (?x C) (?y TOP))
    (?y D) (?x TOP))
```

$\Rightarrow$ If C as well as D instances are present, then is equivalent to



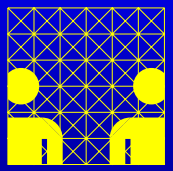
Arity of Query Bodies

- Arity of `(and (?x C) (?y D))`: 2
- Arity of `(or (?x C) (?y D))`: ?
- We want the NNF:
  - `(not (and (not (?x C)) (not (?y D))))`
  - `NOT` preserves arity

  $\Rightarrow$ Arity of `(or (?x C) (?y D))`: 2
- Internally rewritten into

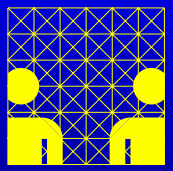
```
(or (and (?x C) (?y TOP))
         (?y D) (?x TOP))
```

$\Rightarrow$ If C as well as D instances are present, then `(retrieve (?x) (?x TOP))`



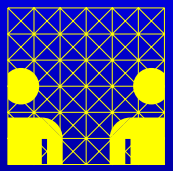
nRQL Query Processing Chain

- Replace syntactic sugar



nRQL Query Processing Chain

- Replace syntactic sugar
  - `(?x NIL R)`
    $\Rightarrow$ `(not (?x (:has-known-successor R)))`



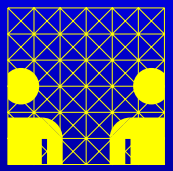
nRQL Query Processing Chain

- Replace syntactic sugar
  - `(?x NIL R)`
    $\Rightarrow$ `(not (?x (:has-known-successor R)))`
  - Break up feature chains

```
(?x ?y (:constraint (has-father age)
                    (has-mother age) =))
```

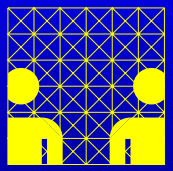
$\Rightarrow$

```
(and (?x $?x-father has-father)
     (?y $?y-mother has-mother)
     ($?x-father $?y-mother
                 (:constraint age age =)))
```



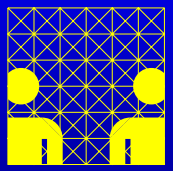
nRQL Query Processing Chain

- Replace syntactic sugar
- Bring query into DNF



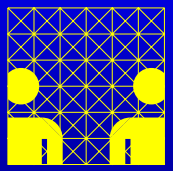
nRQL Query Processing Chain

- Replace syntactic sugar
- Bring query into DNF
- Reasoning (incomplete in general, but useful!)



nRQL Query Processing Chain

- Replace syntactic sugar
- Bring query into DNF
- Reasoning (incomplete in general, but useful!)
  - `(and (?x woman)`
    `(?x ?y has-child)`
    `(?x (not mother)))`

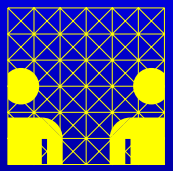


nRQL Query Processing Chain

- Replace syntactic sugar
- Bring query into DNF
- Reasoning (incomplete in general, but useful!)
  - `(and (?x woman)`
    `(?x ?y has-child)`
    `(?x (not mother)))`

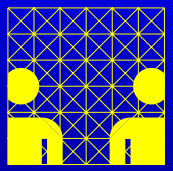
  $\Rightarrow$ inconsistent
  - `(and (?x woman) (?y mother)`
    `(?x ?y has-child))`

  $\Rightarrow$ `(and (?x grandmother) (?y mother)`
    `(?x ?y has-daughter))`



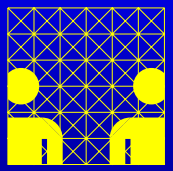
nRQL Query Processing Chain

- Replace syntactic sugar
- Bring query into DNF
- Reasoning (incomplete in general, but useful!)
- Heuristic Optimization



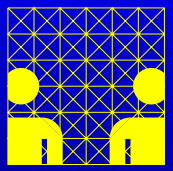
nRQL Query Processing Chain

- Replace syntactic sugar
- Bring query into DNF
- Reasoning (incomplete in general, but useful!)
- Heuristic Optimization
  - `(and (?x woman) (?y man)`
    `(?x ?y has-child))`



nRQL Query Processing Chain

- Replace syntactic sugar
- Bring query into DNF
- Reasoning (incomplete in general, but useful!)
- Heuristic Optimization
  - `(and (?x woman) (?y man)`
`(?x ?y has-child))`
  - $3! = 6$ orderings of conjuncts



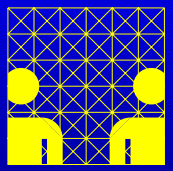
nRQL Query Processing Chain

- Replace syntactic sugar
- Bring query into DNF
- Reasoning (incomplete in general, but useful!)
- Heuristic Optimization
  - `(and (?x woman) (?y man)`
    `(?x ?y has-child))`
  - $3! = 6$ orderings of conjuncts

⊖ `(?x woman) (?y man) (?x ?y has-child)`

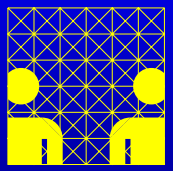
⊕ `(?x woman) (?x ?y has-child) (?y man)`

⊕⊕ `(?y man) (?y ?x child-of) (?x woman)`



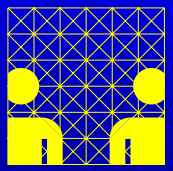
nRQL Query Processing Chain

- Replace syntactic sugar
- Bring query into DNF
- Reasoning (incomplete in general, but useful!)
- Heuristic Optimization
- Execution: find solutions of a finite domain CSP



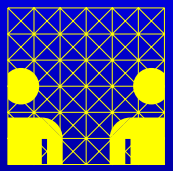
nRQL Query Processing Chain

- Replace syntactic sugar
- Bring query into DNF
- Reasoning (incomplete in general, but useful!)
- Heuristic Optimization
- Execution: find solutions of a finite domain CSP
  - Compilation of a LISP program from query
  - Backtracking Search



nRQL Query Processing Chain

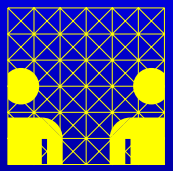
- Replace syntactic sugar
- Bring query into DNF
- Reasoning (incomplete in general, but useful!)
- Heuristic Optimization
- Execution: find solutions of a finite domain CSP
- Construction of result tuples from body bindings



nRQL Query Processing Chain

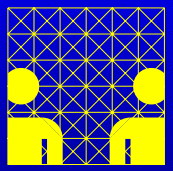
- Replace syntactic sugar
- Bring query into DNF
- Reasoning (incomplete in general, but useful!)
- Heuristic Optimization
- Execution: find solutions of a finite domain CSP
- Construction of result tuples from body bindings
  - Duplications and reorderings possible, e.g.

```
(retrieve (?x ?y ?x)
      (and (?y ...) (?x ...) ...))
```



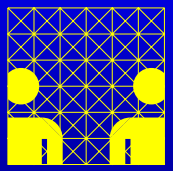
Future Work

- Reasoning with queries



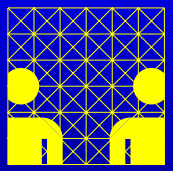
Future Work

- Reasoning with queries
  - $\Rightarrow$ Functions for Racer's API



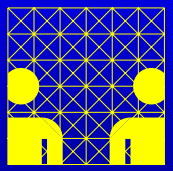
Future Work

- Reasoning with queries
  - $\Rightarrow$ Functions for Racer's API
- Server Continuations



Future Work

- Reasoning with queries
  - ⇒ Functions for Racer's API
- Server Continuations
  - "Tuple at a time"
  - "Set at a time"
  - ⇒ Functions / Switches for Racer's API



Future Work

- Reasoning with queries

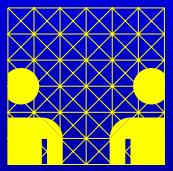
  $\Rightarrow$ Functions for Racer's API

- Server Continuations
  - "Tuple at a time"
  - "Set at a time"

  $\Rightarrow$ Functions / Switches for Racer's API

- Complex constraint expressions:

```
(?x ?y (:constraint
                (>= (- (father age) 10)
                    (mother age))))
```



Future Work

- Reasoning with queries

  $\Rightarrow$ Functions for Racer's API

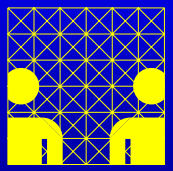
- Server Continuations
  - "Tuple at a time"
  - "Set at a time"

  $\Rightarrow$ Functions / Switches for Racer's API

- Complex constraint expressions:

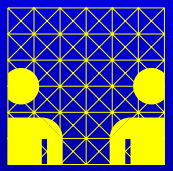
```
(?x ?y (:constraint
                (>= (- (father age) 10)
                    (mother age))))
```

- Enhance efficency



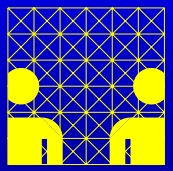
Limitations of nRQL

- Consider the ABox $\mathcal{A} =_{def} \{(\exists R.C)(i)\}$



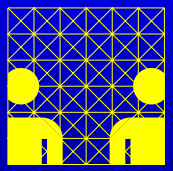
Limitations of nRQL

- Consider the ABox $\mathcal{A} =_{def} \{(\exists R.C)(i)\}$
- Logic tells us $\mathcal{A} \models \exists x.C(x)$



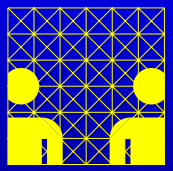
Limitations of nRQL

- Consider the ABox $\mathcal{A} =_{def} \{(\exists R.C)(i)\}$
- Logic tells us $\mathcal{A} \models \exists x.C(x)$
- But there is no $j \in \mathsf{individuals}(\mathcal{A})$ with $j^{\mathcal{I}} \in C^{\mathcal{I}}$ in all $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ with $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}) \models \mathcal{A}$



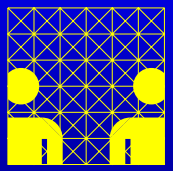
Limitations of nRQL

- Consider the ABox $\mathcal{A} =_{def} \{(\exists R.C)(i)\}$
- Logic tells us $\mathcal{A} \models \exists x.C(x)$
- But there is no $j \in \mathsf{individuals}(\mathcal{A})$ with $j^{\mathcal{I}} \in C^{\mathcal{I}}$ in all $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ with $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}) \models \mathcal{A}$
- `(retrieve (?x) (?x C))` $\Rightarrow$ `NIL`



Limitations of nRQL

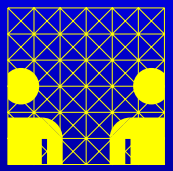
- Consider the ABox $\mathcal{A} =_{def} \{(\exists R.C)(i)\}$
- Logic tells us $\mathcal{A} \models \exists x.C(x)$
- But there is no $j \in \mathsf{individuals}(\mathcal{A})$ with $j^{\mathcal{I}} \in C^{\mathcal{I}}$ in all $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ with $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}) \models \mathcal{A}$
- `(retrieve (?x) (?x C))` $\Rightarrow$ `NIL`
- Reason: `(concept-instances C)` $\Rightarrow$ `NIL`



Limitations of nRQL

- Consider the ABox $\mathcal{A} =_{def} \{(\exists R.C)(i)\}$
- Logic tells us $\mathcal{A} \models \exists x.C(x)$
- But there is no $j \in \mathsf{individuals}(\mathcal{A})$ with $j^{\mathcal{I}} \in C^{\mathcal{I}}$ in all $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ with $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}) \models \mathcal{A}$
- `(retrieve (?x) (?x C))` $\Rightarrow$ `NIL`
- Reason: `(concept-instances C)` $\Rightarrow$ `NIL`
- `(retrieve (?x) (?x C))`

  $\Rightarrow$ `(((?x temp123))) better?`

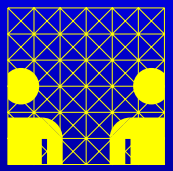


Limitations of nRQL

- Consider the ABox $\mathcal{A} =_{def} \{(\exists R.C)(i)\}$
- Logic tells us $\mathcal{A} \models \exists x.C(x)$
- But there is no $j \in \mathsf{individuals}(\mathcal{A})$ with $j^{\mathcal{I}} \in C^{\mathcal{I}}$ in all $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ with $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}) \models \mathcal{A}$
- `(retrieve (?x) (?x C))` $\Rightarrow$ `NIL`
- Reason: `(concept-instances C)` $\Rightarrow$ `NIL`
- `(retrieve (?x) (?x C))`

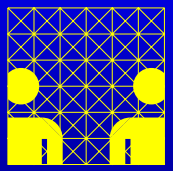
  $\Rightarrow$ `(((?x (and C (some (inv R)`
  `(one-of i))))))`

  better?



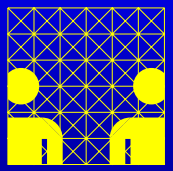
Limitations of nRQL

- Consider the ABox $\mathcal{A} =_{def} \{(\exists R.C)(i)\}$
- Logic tells us $\mathcal{A} \models \exists x.C(x)$
- But there is no $j \in \mathsf{individuals}(\mathcal{A})$ with $j^{\mathcal{I}} \in C^{\mathcal{I}}$ in all $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ with $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}) \models \mathcal{A}$
- `(retrieve (?x) (?x C))` $\Rightarrow$ `NIL`
- Reason: `(concept-instances C)` $\Rightarrow$ `NIL`
- `(retrieve (?x ?y) (?x ?y R))` $\Rightarrow$ `NIL`



Limitations of nRQL

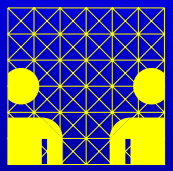
- Consider the ABox $\mathcal{A} =_{def} \{(\exists R.C)(i)\}$
- Logic tells us $\mathcal{A} \models \exists x.C(x)$
- But there is no $j \in \mathsf{individuals}(\mathcal{A})$ with $j^{\mathcal{I}} \in C^{\mathcal{I}}$ in all $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ with $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}) \models \mathcal{A}$
- `(retrieve (?x) (?x C))` $\Rightarrow$ `NIL`
- Reason: `(concept-instances C)` $\Rightarrow$ `NIL`
- `(retrieve (?x ?y) (?x ?y R))` $\Rightarrow$ `NIL`
- Reason: `(related-individuals R)` $\Rightarrow$ `NIL`



Limitations of nRQL

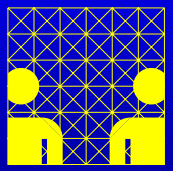
- Consider the ABox $\mathcal{A} =_{def} \{(\exists R.C)(i)\}$
- Logic tells us $\mathcal{A} \models \exists x.C(x)$
- But there is no $j \in \mathsf{individuals}(\mathcal{A})$ with $j^{\mathcal{I}} \in C^{\mathcal{I}}$ in all $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ with $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}) \models \mathcal{A}$
- `(retrieve (?x) (?x C))` $\Rightarrow$ `NIL`
- Reason: `(concept-instances C)` $\Rightarrow$ `NIL`
- `(retrieve (?x ?y) (?x ?y R))` $\Rightarrow$ `NIL`
- Reason: `(related-individuals R)` $\Rightarrow$ `NIL`
- `(retrieve (?x) (?x (some R C)))`

  $\Rightarrow$ `(((?x i)))`



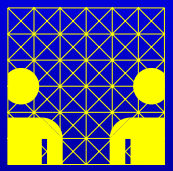
Summing up, nRQL . . .

- . . . is a pragmatic ABox query language



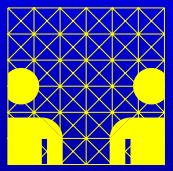
Summing up, nRQL . . .

- . . . is a pragmatic ABox query language
- . . . has been requested by users



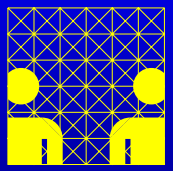
Summing up, nRQL . . .

- . . . is a pragmatic ABox query language
- . . . has been requested by users
- . . . frees them from programming loops



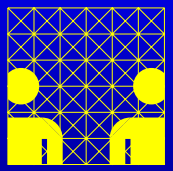
Summing up, nRQL . . .

- . . . is a pragmatic ABox query language
- . . . has been requested by users
- . . . frees them from programming loops
- . . . has been inspired by LOOM



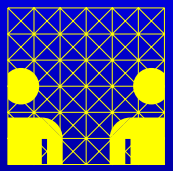
Summing up, nRQL . . .

- . . . is a pragmatic ABox query language
- . . . has been requested by users
- . . . frees them from programming loops
- . . . has been inspired by LOOM
- . . . is easy to understand (one kind of variables)



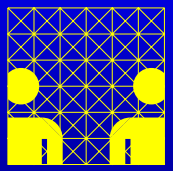
Summing up, nRQL . . .

- . . . is a pragmatic ABox query language
- . . . has been requested by users
- . . . frees them from programming loops
- . . . has been inspired by LOOM
- . . . is easy to understand (one kind of variables)
- . . . has a formal semantics



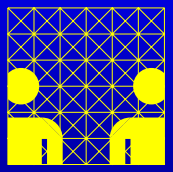
Summing up, nRQL . . .

- . . . is a pragmatic ABox query language
- . . . has been requested by users
- . . . frees them from programming loops
- . . . has been inspired by LOOM
- . . . is easy to understand (one kind of variables)
- . . . has a formal semantics
- . . . which it inherits from the semantics of standard DL ABox retrieval functions



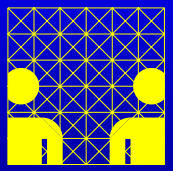
Summing up, nRQL . . .

- . . . is a pragmatic ABox query language
- . . . has been requested by users
- . . . frees them from programming loops
- . . . has been inspired by LOOM
- . . . is easy to understand (one kind of variables)
- . . . has a formal semantics
- . . . which it inherits from the semantics of standard DL ABox retrieval functions
- . . . combines in a uniform declarative way most of Racer's ABox querying functions



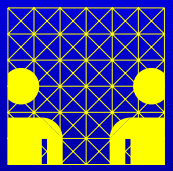
Summing up, nRQL . . .

- . . . is a pragmatic ABox query language
- . . . has been requested by users
- . . . frees them from programming loops
- . . . has been inspired by LOOM
- . . . is easy to understand (one kind of variables)
- . . . has a formal semantics
- . . . which it inherits from the semantics of standard DL ABox retrieval functions
- . . . combines in a uniform declarative way most of Racer's ABox querying functions
- . . . is efficiently implemented



Summing up, nRQL . . .

- . . . is a pragmatic ABox query language
- . . . has been requested by users
- . . . frees them from programming loops
- . . . has been inspired by LOOM
- . . . is easy to understand (one kind of variables)
- . . . has a formal semantics
- . . . which it inherits from the semantics of standard DL ABox retrieval functions
- . . . combines in a uniform declarative way most of Racer's ABox querying functions
- . . . is efficiently implemented
- . . . comes with a manual



Thank you!