

Vorlesung Semantic Web



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Vorlesung im Wintersemester 2012/2013

Dr. Heiko Paulheim

Fachgebiet Knowledge Engineering

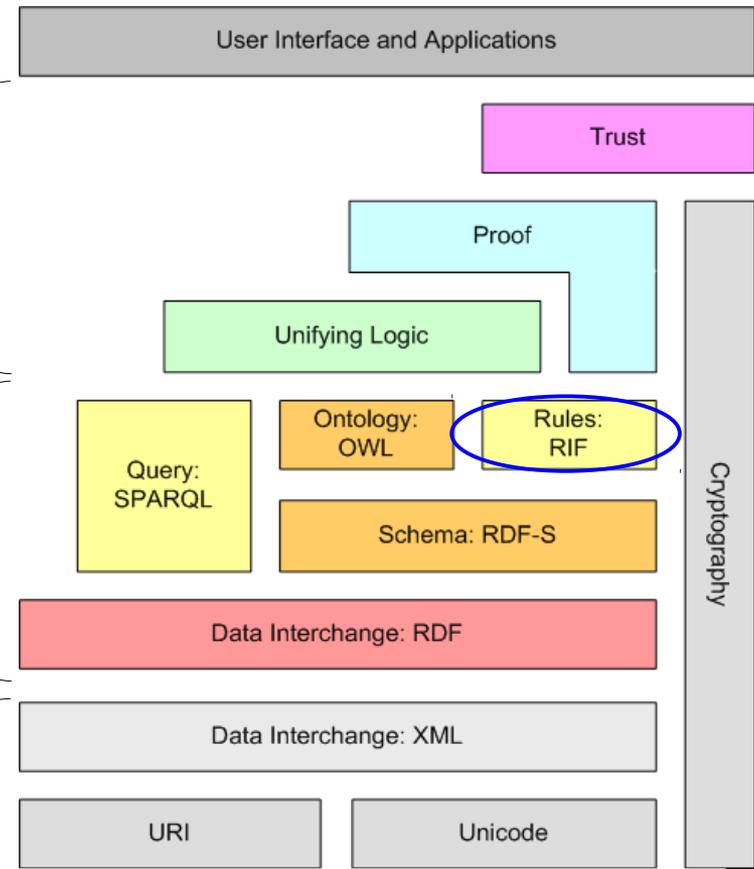
Semantic Web – Aufbau



here be dragons...

Semantic-Web-
Technologie
(Fokus der Vorlesung)

Technische
Grundlagen



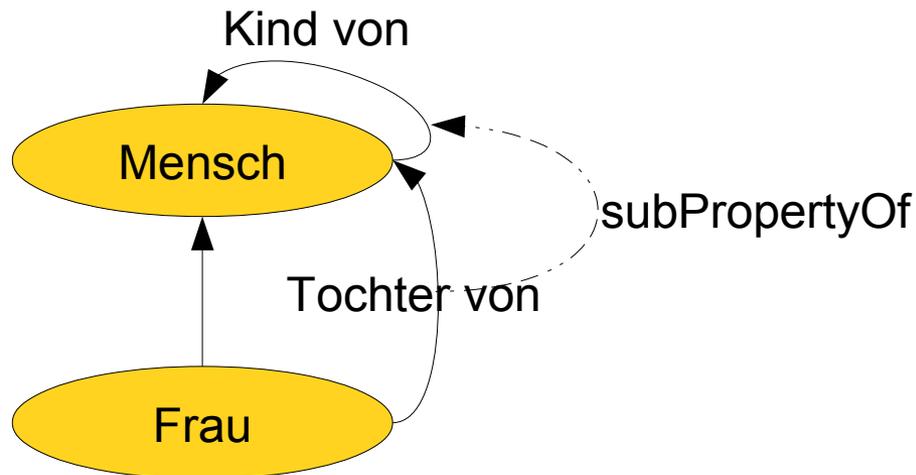
Berners-Lee (2009): *Semantic Web and Linked Data*
<http://www.w3.org/2009/Talks/0120-campus-party-tbl/>

Rückblick: Ontologien

- RDF codiert Daten
- Ontologien liefern die Semantik der Daten
 - RDF Schema (leichtgewichtig)
 - OWL (schwergewichtiger)
- ermöglichen Reasoning
 - dazu muss ein relevanter Ausschnitt der Welt modelliert sein
 - und zwar möglichst exakt

Beschränkungen von OWL

- Manche Dinge lassen sich mit OWL nicht ohne weiteres ausdrücken
- Beispiel:
 - Wenn A eine Frau und das Kind von B ist, dann ist A die Tochter von B.



Beschränkungen von OWL



- Versuchen wir das mal mit OWL:

```
:Frau rdfs:subClassOf :Mensch .  
:kindVon a owl:ObjectProperty ;  
         rdfs:domain :Mensch ;  
         rdfs:range :Mensch .  
:tochterVon a owl:ObjectProperty ;  
            rdfs:subPropertyOf :kindVon ;  
            rdfs:domain :Frau .
```

Beschränkungen von OWL

- Was schließt der Reasoner daraus?

- Zum Beispiel:

`:Julia :tochterVon :Peter .`

`→ :Julia a :Frau .`

- Was wir aber gern hätten:

`:Julia :kindVon :Peter .`

`:Julia a :Frau .`

`→ :Julia :tochterVon :Peter .`

Beschränkungen von OWL

- Was wir gern hätten:
 - $\text{tochterVon}(X,Y) \leftarrow \text{kindVon}(X,Y) \wedge \text{Frau}(X)$.
- Regeln wären da flexibel genug
- tatsächlich gibt es Regeln im Semantic Web
 - Semantic Web Rule Language (SWRL)
 - Rule Interchange Format (RIF)
 - einige weitere
- Reasoner unterstützten teilweise Regeln

SWRL

- Semantic Web Rule Language
 - eine Regel-Sprache für das Semantic Web
 - eng verzahnt mit OWL
- W3C Member Submission (2004)
 - kein Standard im engeren Sinne
 - aber sehr weit verbreitet
- Tool-Unterstützung
 - viele Reasoner
 - Protégé



Bausteine von SWRL

- Einstellige Prädikate

- als Klassen in OWL definiert

`:Peter a :Mensch . ↔ Mensch(Peter)`

- Zweistellige Prädikate

- als Object-/DataProperties in OWL definiert

`:Peter :hatMutter :Jutta . ↔ hatMutter(Peter,Jutta)`

`:Peter :hatAlter 24^^xsd:integer . ↔ hatAlter(Peter,24)`

Aufbau einer SWRL-Regel

- Grundform
 - Körper (Voraussetzung) → Kopf (Konsequenz)
 - Beides sind Konjunktionen
 - Variablen mit ?
- Beispiel
 - $\text{tochterVon}(?X,?Y) \leftarrow \text{kindVon}(?X,?Y) \wedge \text{Frau}(?X)$

Aufbau einer SWRL-Regel

- Was es in SWRL nicht gibt
 - Disjunktion (Logisches Oder)
 - Negation
 - ungebundene Variablen im Kopf
- Aber es gibt teilweise Auswege

Disjunktion

- SWRL kennt keine Disjunktion (logisches Oder)
 - zumindest im Körper der Regel lässt sich das aber simulieren
- Beispiel
 - Frauen, die an einer Universität studieren oder arbeiten, sind weibliche Universitätsangehörige
 - intuitive Lösung:
$$\text{Frau}(?X) \wedge \text{Universität}(?Y) \wedge (\text{arbeitetAn}(?X,?Y) \vee \text{studiertAn}(?X,?Y))$$

→ WeiblicheUniversitätsangehörige(?X)

Disjunktion

- Lösung:
 - erster Schritt: Körper der Regel in disjunktive Normalform bringen
 - d.h.: Disjunktion von Konjunktionen
 - zweiter Schritt: Regel in einzelne Regeln aufspalten

Disjunktion

- Beispiel:

$\text{Frau}(\text{?X}) \wedge \text{Universität}(\text{?Y}) \wedge (\text{arbeitetAn}(\text{?X}, \text{?Y}) \vee \text{studiertAn}(\text{?X}, \text{?Y}))$
→ WeiblicheUniversitätsangehörige(?X)

- wird zu

$(\text{Frau}(\text{?X}) \wedge \text{Universität}(\text{?Y}) \wedge \text{arbeitetAn}(\text{?X}, \text{?Y}))$
 $\vee (\text{Frau}(\text{?X}) \wedge \text{Universität}(\text{?Y}) \wedge \text{studiertAn}(\text{?X}, \text{?Y}))$
→ WeiblicheUniversitätsangehörige(?X)

- und das wird zu

$\text{Frau}(\text{?X}) \wedge \text{Universität}(\text{?Y}) \wedge \text{arbeitetAn}(\text{?X}, \text{?Y})$
→ WeiblicheUniversitätsangehörige(?X)
 $\text{Frau}(\text{?X}) \wedge \text{Universität}(\text{?Y}) \wedge \text{studiertAn}(\text{?X}, \text{?Y})$
→ WeiblicheUniversitätsangehörige(?X)

Disjunktion

- Disjunktionen im Kopf der Regel bekommen wir nicht so leicht los
- Beispiel:
 - Jeder Universitätsangehörige ist ein Student oder ein Mitarbeiter
 $\text{Universitätsangehöriger}(\text{?X}) \wedge \text{Universität}(\text{?Y})$
 $\rightarrow \text{arbeitetAn}(\text{?X}, \text{?Y}) \vee \text{studiertAn}(\text{?X}, \text{?Y})$
- Was soll ein Reasoner hieraus auch schließen?
 - Disjunktion im Kopf der Regel ist nicht unbedingt sinnvoll

Negation

- Negation ist in SWRL nicht vorgesehen
- Kann aber oft simuliert werden
 - SWRL verwendet man zusammen mit OWL
- Beispiel:
 - Lebewesen, die im Wasser leben, sind keine Menschen.
 - Intuitive Lösung:
 $\text{Lebewesen}(?x) \wedge \text{hatLebensraum}(?x, \text{Wasser})$
 $\rightarrow \neg \text{Mensch}(?x)$

- Simulation mit Hilfe von OWL
- Definition einer zusätzlichen Klasse:
`KeinMensch owl:complementOf Mensch .`
- Änderung der Regel:
`Lebewesen(?x) ∧ hatLebensraum(?x,Wasser)
→ KeinMensch(?x)`
- Damit kann der Reasoner schließen, dass
`:Nemo a :Lebewesen; hatLebensraum :Wasser .`
im Widerspruch steht zu:
`:Nemo a :Mensch .`

Negation

- Das funktioniert genauso für Negation im Körper:

$\text{Vogel}(?x) \wedge \neg \text{FliegendesTier}(?x)$
→ Laufvogel(?x)

- wird zu

`NichtFliegendesTier owl:complementOf FliegendesTier .`

- Und

$\text{Vogel}(?x) \wedge \text{NichtFliegendesTier}(?x)$
→ Laufvogel(?x)

Ungebundene Variablen im Kopf



- Alle Variablen, die im Kopf einer Regel vorkommen, müssen auch im Körper vorkommen
- So etwas geht also nicht:
 - Jeder Mensch hat einen Vater, der wieder ein Mensch ist
 $\text{Mensch}(?x) \rightarrow \text{hatVater}(?x, ?y) \wedge \text{Mensch}(?y)$.
- Der Reasoner müsste hier neue Instanzen erzeugen
- Mögliches Problem: Terminierung
 - Dafür gibt es in SWRL+OWL keine einfache Umgehung

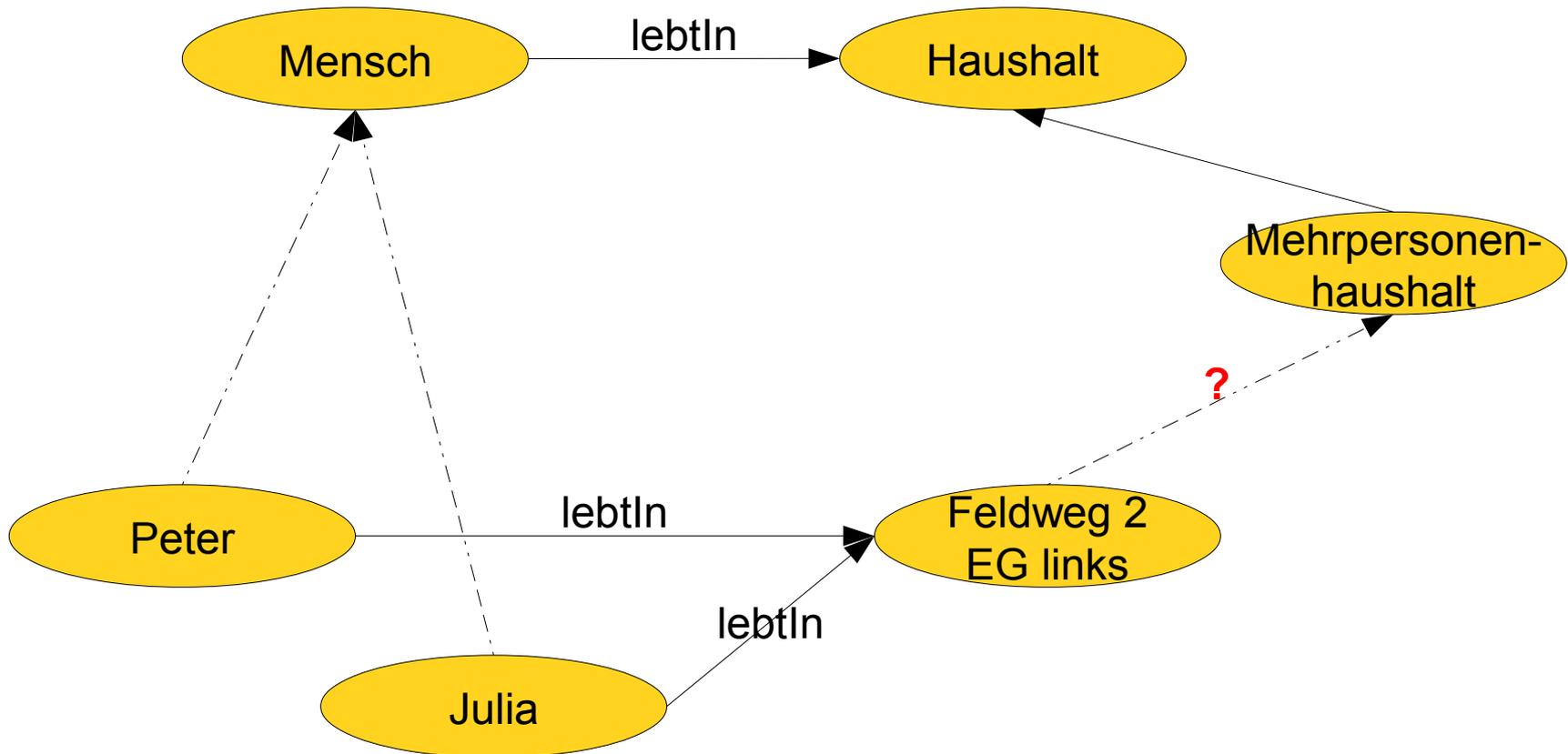
Gleichheit und Verschiedenheit

- In OWL schon vorhanden:
 - owl:sameAs, owl:differentFrom
 - können in SWRL verwendet werden
 - sowohl im Body als auch im Head
 - Kurzschreibweise: =, ≠
- Beispiel:
 - Studenten haben eine eindeutige Matrikel-Nummer:
 $\text{hasMatNr}(?x,?n) \wedge \text{hasMatNr}(?y,?n) \rightarrow ?x = ?y$

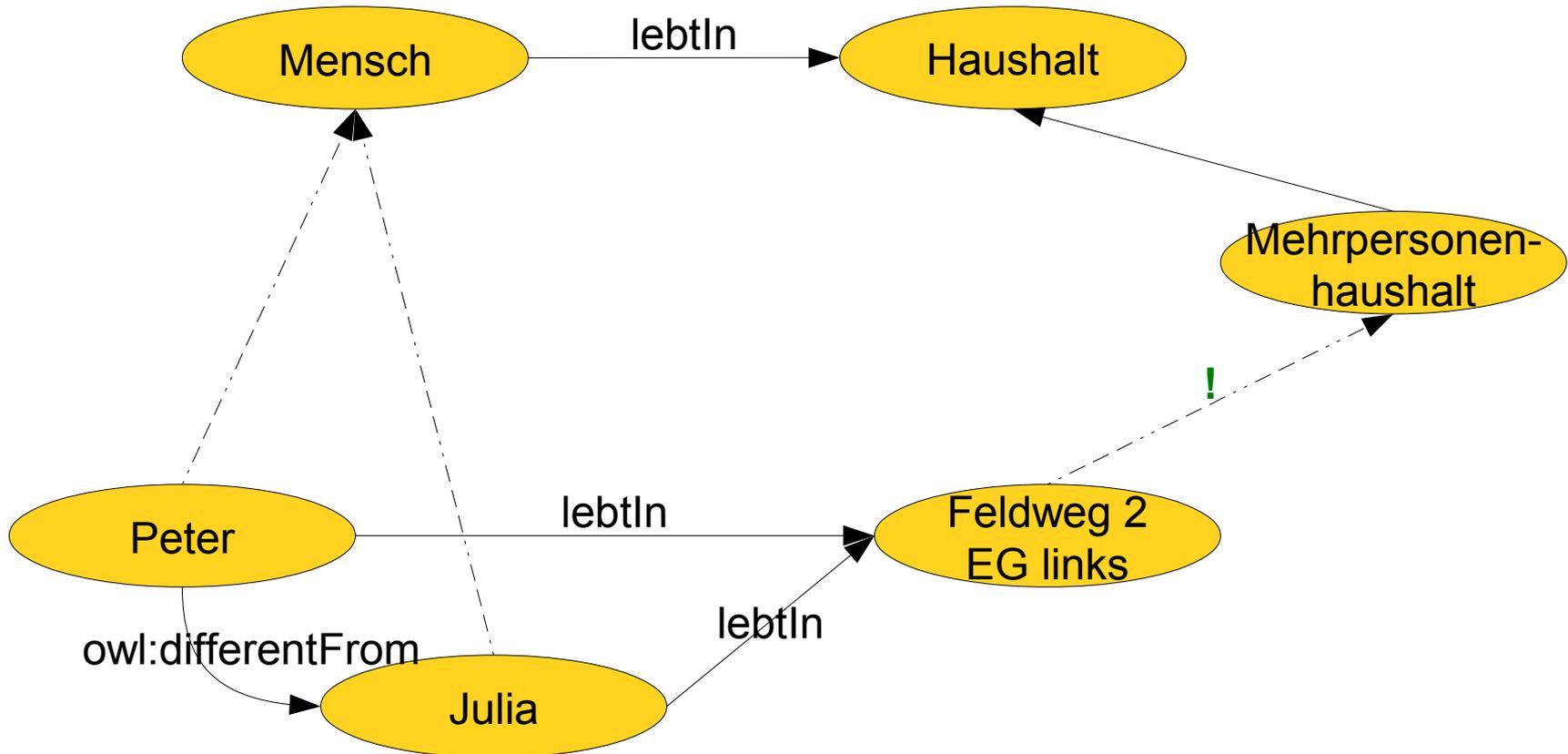
Gleichheit und Verschiedenheit

- Beispiel:
 - In einem Mehrpersonenhaushalt leben mindestens zwei (verschiedene) Personen
- Die Verschiedenheit brauchen wir explizit
 $\text{lebtIn}(?y,?x) \wedge \text{lebtIn}(?z,?x) \wedge ?y \neq ?z \rightarrow \text{Mehrpersonenhaushalt}(?x)$
- Warum?
 - Die Variablen $?y$ und $?z$ können sonst an dieselbe Entität gebunden werden

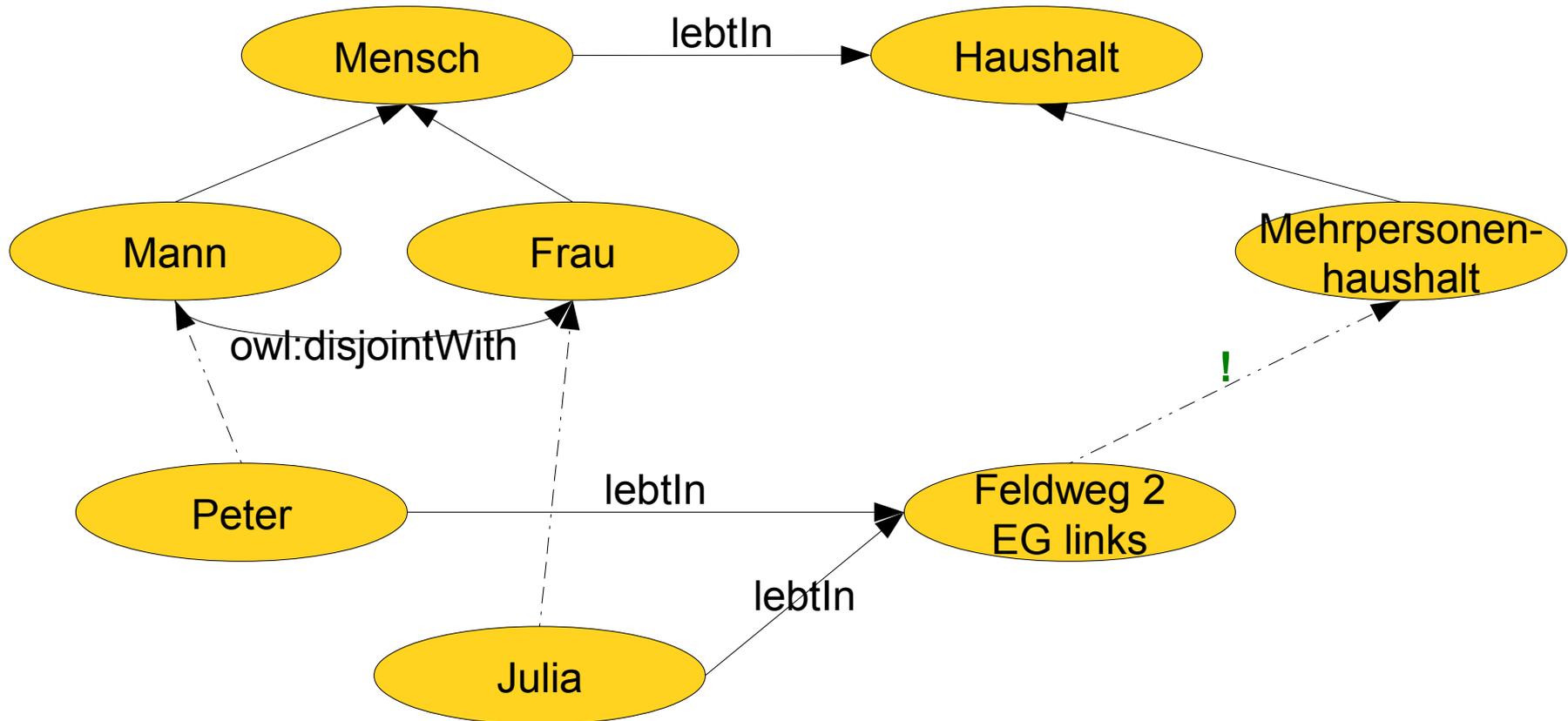
Gleichheit und Verschiedenheit



Gleichheit und Verschiedenheit



Gleichheit und Verschiedenheit



- SWRL erlaubt beliebige Prädikate als Erweiterungen
- sogenannte Built-Ins
 - nicht alle Reasoner unterstützen alle Built-Ins
 - manche Reasoner erlauben die Einbindung eigener Built-Ins
 - das ist ein sehr mächtiger Mechanismus...
- Standard-Built-Ins
 - Vergleich
 - Arithmetik (Zahlen und Kalender)
 - String-Operationen
 - ...
- Namespace meist `swrlb`

Built-Ins für Vergleiche

- Beispiel

- Definition eines Prädikates *älterAls*

- Gegeben:

```
hatAlter a owl:DatatypeProperty ;  
        rdfs:domain :Person ;  
        rdfs:range xsd:integer .
```

- Regel:

$\text{hatAlter}(?x,?ax) \wedge \text{hatAlter}(?y,?ay) \wedge \text{swrlb:greaterThan}(?ax,?ay)$
→ *älterAls*(?x,?y)

- Weitere Prädikate:

- *greaterThanOrEqual*, *lessThan*, *lessThanOrEqual*, *equal*, *notEqual*
- je nach Implementierung für Zahlen, Daten, Strings, ...

Built-Ins für Arithmetik

- Beispiel

- Definition des Prädikates `doppeltSoAltWie`

- Regel:

- $\text{hatAlter}(?x, ?ax) \wedge \text{hatAlter}(?y, ?ay) \wedge \text{swrlb:multiply}(?ax, ?ay, 2)$
→ $\text{doppeltSoAltWie}(?x, ?y)$

- Semantik des Built-Ins:

- $\text{swrlb:multiply}(?x, ?y, ?z) \leftrightarrow ?x = ?y * ?z$

- Weitere Built-Ins:

- `add`, `subtract`, `divide`, `pow`, `sin`, `cos`, `round`, ...

Built-Ins für Arithmetik

- Beispiel

- Definition des Prädikates `doppeltSoAltWie`

- Regel:

- $\text{hatAlter}(?x, ?ax) \wedge \text{hatAlter}(?y, ?ay) \wedge \text{swrlb:multiply}(?ax, ?ay, 2)$
→ $\text{doppeltSoAltWie}(?x, ?y)$

- Semantik des Built-Ins:

- $\text{swrlb:multiply}(?x, ?y, ?z) \leftrightarrow ?x = ?y * ?z$

- Weitere Built-Ins:

- `add`, `subtract`, `divide`, `pow`, `sin`, `cos`, `round`, ...

Built-Ins für Arithmetik



- Das funktioniert auch anders herum
- Regel:
 - doppeltSoAltWie(?x,?y) \wedge hatAlter(?y,?ay)
 - \wedge swrlb:multiply(?ax,?ay,2)
 - \rightarrow hatAlter(?x,?ax)
- Freie Variablen können durch Built-Ins gebunden werden
 - an welchen Stellen, hängt wiederum vom Reasoner ab

Signatur von Built-Ins



- Die Signatur bestimmt, an welchen Stellen *ungebundene* Variablen auftreten dürfen
 - gebunden: `_`, ungebunden: `?`
 - z.B. `multiply(?,_,-)`, `multiply(_,-,?)`, `multiply(_,-,?)`
 - Auch möglich: `multiply(_,-,-)`
- Was halten Sie davon:
 - `multiply(_,-,?)`
 - Erinnerung: `multiply(?x, ?y ?z) ↔ ?x = ?y * ?z`

Built-Ins für Arithmetik

- Arithmetik geht auch mit Daten

- Gegeben:

```
hatGeburtsdatum a owl:DatatypeProperty .
```

```
hatSterbedatum a owl:DatatypeProperty .
```

```
hatSterbealter a owl:DatatypeProperty .
```

- Regel:

```
hatGeburtsdatum(?x,?g) ∧ hatSterbedatum(?x,?s)  
∧ subtractDates(?a,?s,?g)  
→ hatSterbealter(?x,?a)
```

- Merke:

- das funktioniert in der Theorie
- ein Reasoner muss das aber auch unterstützen!

Built-Ins für String-Operationen



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Beispiel

- Eine Klasse für alle Menschen, deren Namen mit "S" beginnt

- Gegeben:

```
:hasName a owl:DatatypeProperty .
```

```
:PeopleWithS rdfs:subClassOf :Person .
```

- Regel:

```
hasName(?x,?n)  $\wedge$  swrlb:startsWith(?n,"S")
```

```
→ PeopleWithS(?x)
```

Semantik von SWRL-Regeln

- Für SWRL gelten dieselben Prinzipien wie für RDF und OWL
- Open World Assumption
 - es gibt keine Negation
 - man kann keine Dinge zählen
- Non-unique Name Assumption
 - weil zwei Dinge verschieden heißen, müssen sie nicht verschieden sein
 - owl:differentFrom
 - ermöglicht explizites Setzen von Verschiedenheit
 - und Test auf Verschiedenheit

- Wiederholung: Monotones vs. nicht-monotones Schlussfolgern
 - Monoton: alles, was einmal als wahr erkannt wurde, bleibt wahr
 - Nicht-monoton: gewonnene Erkenntnisse können widerrufen werden
- SWRL ist monoton
 - d.h.: alle Folgerungen addieren sich auf
 - das kann auch zu Widersprüchen führen
- Monotones Reasoning ist einfach

- Nehmen wir noch mal diese Regel:

$\text{doppeltSoAltWie}(?x,?y) \wedge \text{hatAlter}(?y,?ay)$
 $\wedge \text{swrlb:multiply}(?ax,?ay,2)$
 $\rightarrow \text{hatAlter}(?x,?ax)$

- Gegeben

```
:Person rdfs:subClassOf [ a owl:Restriction ;  
                           owl:onProperty :hatAlter ;  
                           owl:cardinality 1^^xsd:integer ].  
  
:Peter :doppeltSoAltWie :Stefan, :Markus .  
:Stefan :hatAlter 24^^xsd:integer .  
:Markus :hatAlter 25^^xsd:integer .
```

- Daraus folgt

```
:Peter :hatAlter 48^^xsd:integer .  
:Peter :hatAlter 50^^xsd:integer .
```



- Terminierung des Reasoning ist wichtig
- Bisher
 - es werden keine neuen Instanzen, Klassen, Properties und Literale erzeugt
 - damit ist die Menge der ableitbaren Aussagen beschränkt
 - C*I Klassenaussagen
 - I*O*I ObjectProperty-Aussagen
 - I*D*L DatatypeProperty-Aussagen
- Dadurch terminiert der Reasoner irgendwann

- Betrachten wir einmal folgendes Beispiel:

```
:Person rdfs:subClassOf [  
  a owl:Restriction ;  
  owl:onProperty :hasFather ;  
  owl:cardinality 1^^xsd:integer ] .  
:hasFather rdfs:range :Person .  
:Grandchild rdfs:subClassOf :Person .
```

$\text{hasFather}(?x,?y) \wedge \text{hasFather}(?y,?z) \rightarrow \text{Grandchild}(?x)$

- Gegeben

:Peter a :Person .

- Was folgt daraus?

- Mögliche Lösung
 - Wir wissen ja, dass jede Person einen Vater hat, der selbst Person ist
 - auch wenn wir diesen nicht kennen
 - es gilt also
 - `:Peter :hasFather _:p0 . _:p0 :hasFather _:p1 . :p1 ...`
 - daraus folgt
 - `:Peter a :Grandchild .`
- Was ist der Preis dieser Lösung?
 - Wir erlauben die Erzeugung neuer Instanzen
 - Damit wird die Terminierung aufs Spiel gesetzt

Sichere Regeln

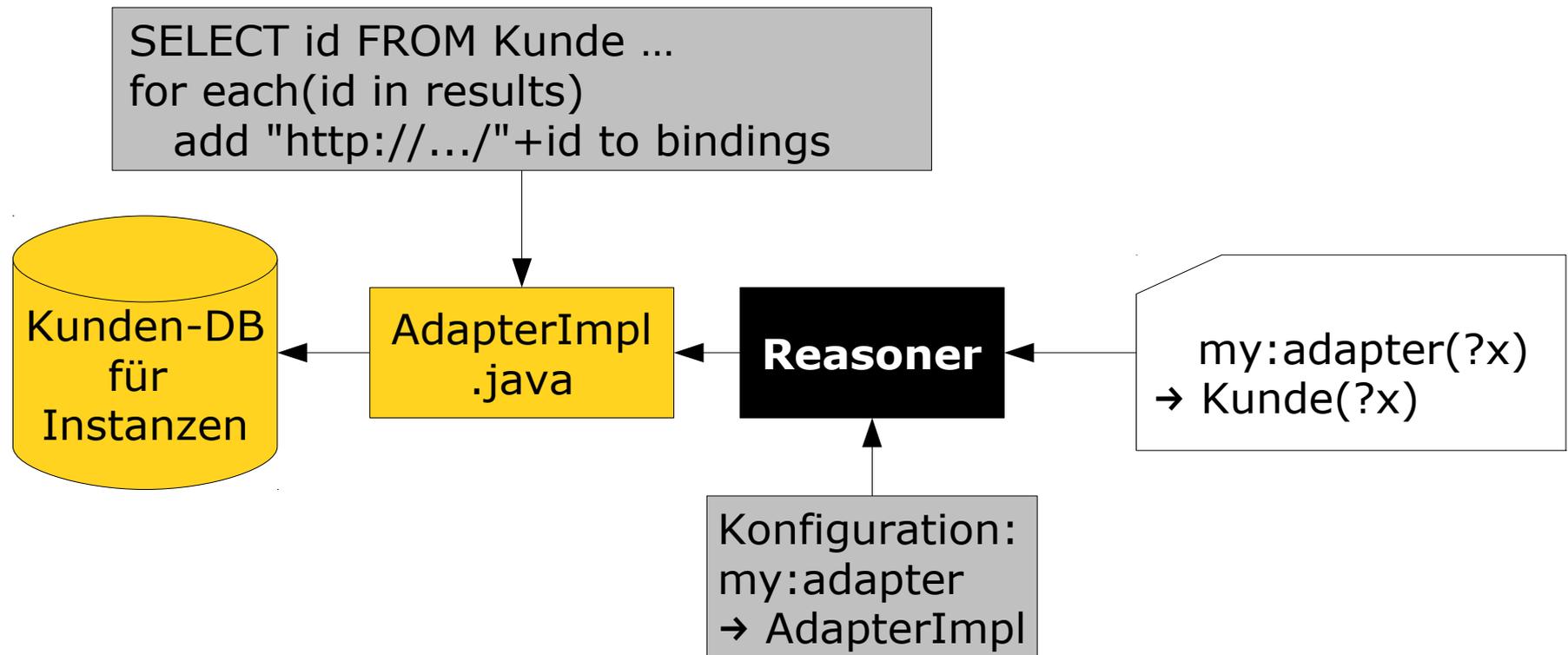
- Lösung DL-Safe Rules:
 - Variablen in Regeln werden nur an *existierende* Instanzen gebunden
 - Es werden keine Instanzen erzeugt
- Damit folgt aus unserer Wissensbasis *nicht*
 - `:Peter a :Grandchild .`
- Abwägung zwischen
 - Terminierung
 - sinnvollen Schlussfolgerungen

Built-Ins und Terminierung

- Built-Ins können auch Probleme für die Terminierung bedeuten
 - $\text{hasValue}(?x,?n) \wedge \text{add}(?n1,?n,1) \rightarrow \text{hasValue}(?x,?n1)$
- Achtung: hier werden *neue* Literale erzeugt!

Eigene Built-Ins

- Einige Reasoner ermöglichen eigene Built-Ins
- Darüber lassen sich Reasoner mit anderen System verdrahten



Beispiel: Reasoning über Daten aus laufenden Systemen

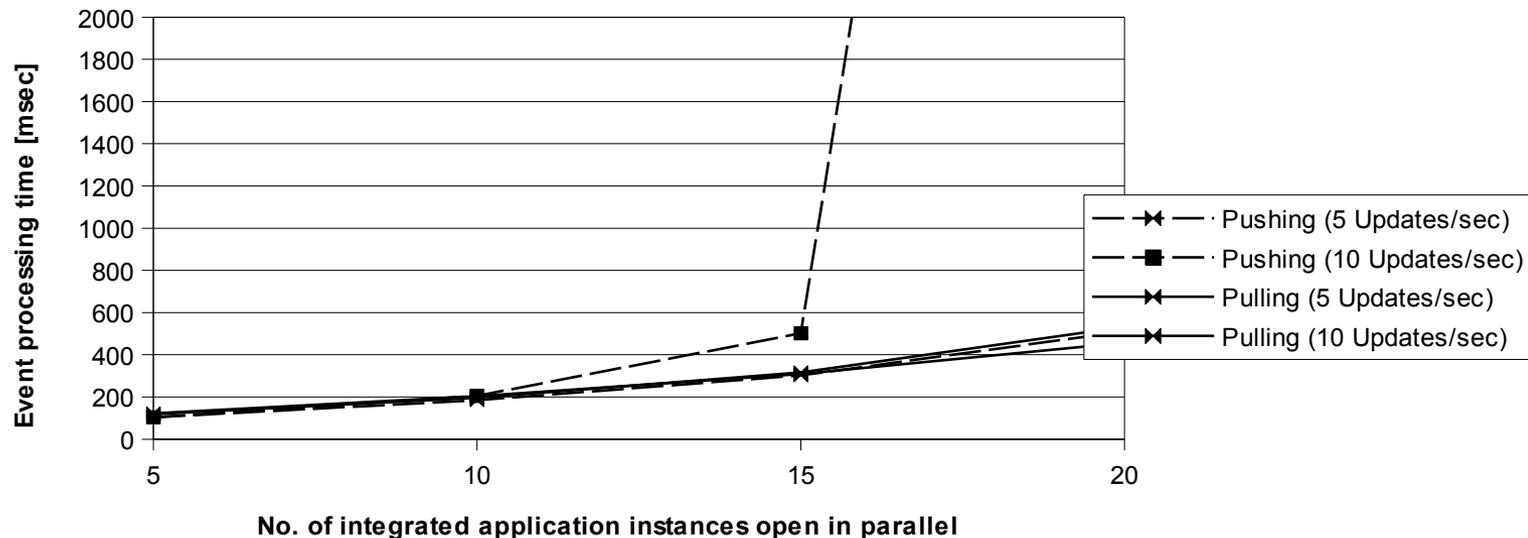


- Ziel:
 - Events applikationsübergreifend mit Reasoner verarbeiten
 - Ergebnis hängt ab von Status der Applikationen
- Zwei Möglichkeiten:
 - Pushing: A-Box ständig aktualisieren
 - Pulling: A-Box-Daten dynamisch mit Built-In holen:
$$\text{Applikation}(?a) \wedge \text{Status}(?s) \wedge \text{getStatusBuiltIn}(?a,?s)$$

→ $\text{hatStatus}(?a,?s)$

Beispiel: Reasoning über Daten aus laufenden Systemen

- Deutliche Unterschiede in der Skalierbarkeit
 - nur Variante mit Pulling funktioniert skalierbar



Weitere Anwendungsfälle für eigene Built-Ins



- Aktuelle Live-Daten
 - Wetter
 - Wechselkurs
 - Börsenkurs
 - Produktverfügbarkeit
 - ...
- Komplexere Berechnungen
 - Fahrzeit von A nach B (z.B. mit Google Maps API)
 - Simulationen und Prognosen
 - ...

OWL und SWRL

- SWRL ist für das Semantic Web konzipiert
 - wie der Name schon sagt...
- lässt sich mit OWL kombinieren
- viele Überlappungen
- SWRL lässt sich komplett mit RDF darstellen

- Das meiste, was man mit OWL sagen kann, geht auch in SWRL

```
:Frau rdfs:subClassOf :Mensch .
```

```
Mensch(X) ← Frau(X)
```

```
:tochterVon rdfs:domain :Frau .
```

```
Frau(X) ← tochterVon(X,Y) .
```

```
:tochterVon rdfs:subPropertyOf :kindVon .
```

```
kindVon(X,Y) ← tochterVon(X,Y) .
```

```
:kindVon owl:inversePropertyOf :elternteilVon .
```

```
kindVon(X,Y) ← elternteilVon(Y,X) .
```

```
elternteilVon(X,Y) ← kindVon(Y,X) .
```

- OWL-Konstrukte, die sich mit SWRL nicht ausdrücken lassen
 - z.B. minimale Kardinalitäten, someValuesFrom

- Beispiel

```
:Auto rdfs:subClassOf [  
  a owl:Restriction ;  
  owl:onProperty :hatMotor ;  
  owl:minCardinality 1^^xsd:integer ] .
```

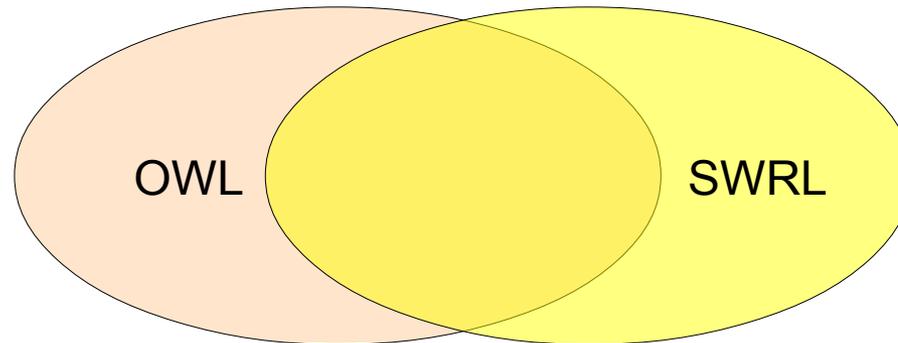
- Angedachte Regel

Auto(?x) → hatMotor(?x, ?y) .

- Achtung, ungebundene Variable im Kopf!

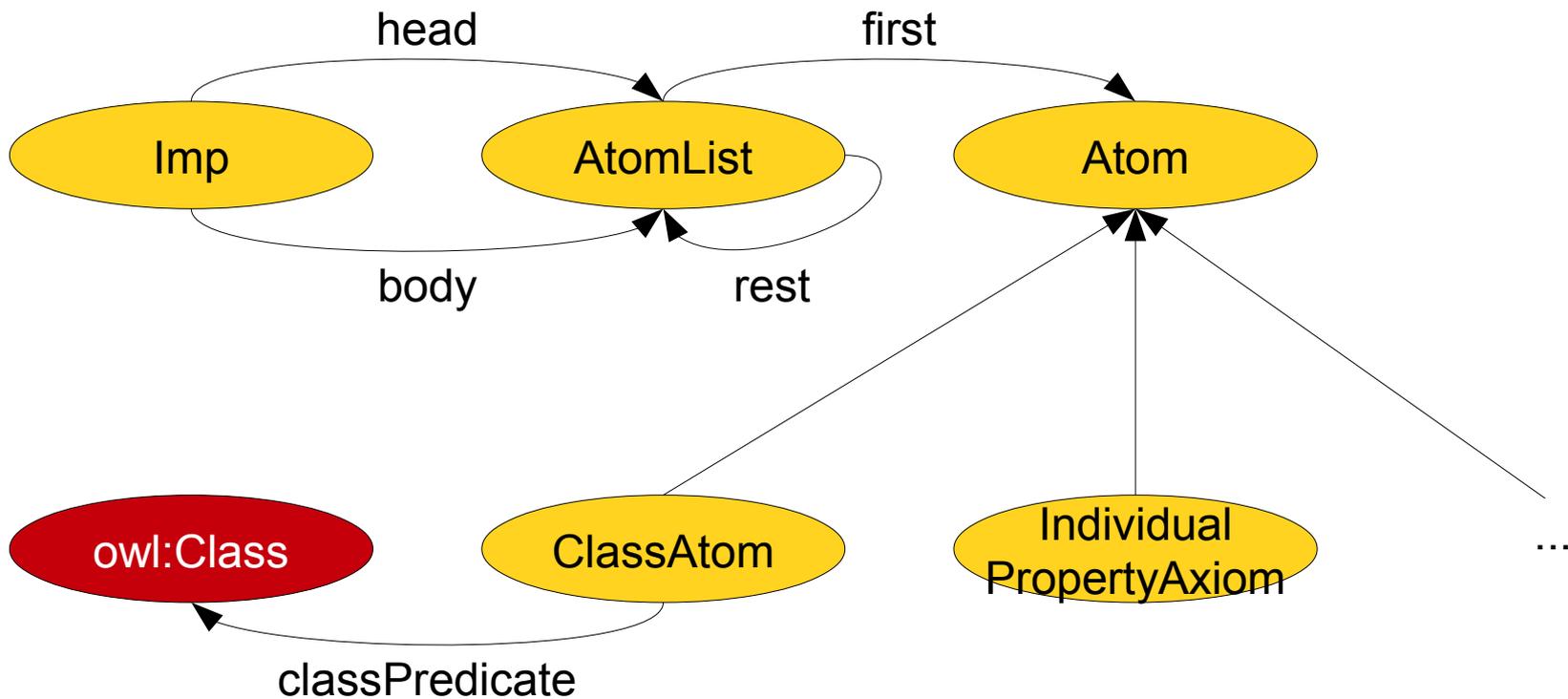
OWL und SWRL

- OWL und SWRL haben große Überlappungen
- Best Practice:
 - OWL nehmen, so weit es geht
- Grund: Optimierungen der Reasoner



SWRL in RDF

- SWRL-Ontologie (Auszug)



SWRL in RDF

- Mit der SWRL-Ontologie verlassen wir OWL DL!

```
:ca a :ClassAxiom .  
:ca swrl:classPredicate onto:Mensch .  
onto:Mensch a owl:Class .
```

- Vorverarbeitung für den Reasoner nötig
 - Regeln von der restlichen Ontologie trennen

SWRL in RDF



```
<ruleml:imp>
  <ruleml:_rlabel ruleml:href="#example1"/>
  <ruleml:_body>
    <swrlx:individualPropertyAtom swrlx:property="hasParent">
      <ruleml:var>x1</ruleml:var>
      <ruleml:var>x2</ruleml:var>
    </swrlx:individualPropertyAtom>
    <swrlx:individualPropertyAtom swrlx:property="hasBrother">
      <ruleml:var>x2</ruleml:var>
      <ruleml:var>x3</ruleml:var>
    </swrlx:individualPropertyAtom>
  </ruleml:_body>
  <ruleml:_head>
    <swrlx:individualPropertyAtom swrlx:property="hasUncle">
      <ruleml:var>x1</ruleml:var>
      <ruleml:var>x3</ruleml:var>
    </swrlx:individualPropertyAtom>
  </ruleml:_head>
</ruleml:imp>
```

Beispiel von <http://www.w3.org/Submission/SWRL/>



Anwendung von SWRL: Ontology Mappings

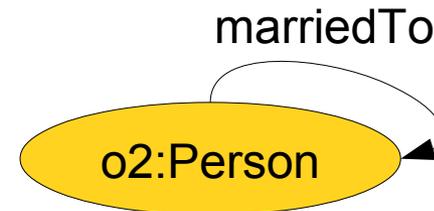
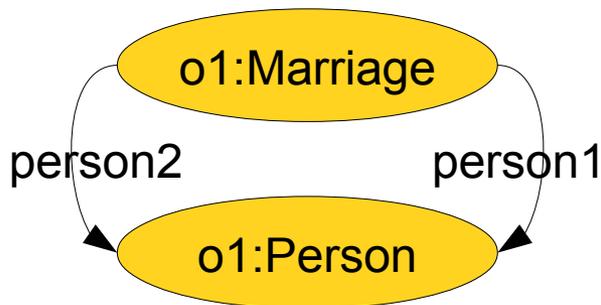
- Rückblick
 - Ontology Mappings bilden Ontologien aufeinander ab
 - bisher: OWL-Sprachmittel (z.B. `equivalentClass`)
- SWRL bietet mehr Möglichkeiten
 - z.B. Umrechnungen
 - `onto1:hasSizeInCm(?x,?c) ∧ swrlb:multiply(?i,?c,2.54)`
 - `onto2:hasSizeInInch(?x,?i)`

Anwendung von SWRL: Ontology Mappings

- Beispiel: Strukturunterschiede überwinden

$o1:Person(?p) \rightarrow o2:Person(?)$

$o1:Marriage(?m) \wedge o1:person1(?m,?p1) \wedge o1:person2(?m,?p2)$
 $\rightarrow o2:marriedTo(?p1,?p2)$



Rückblick SWRL



- Regelsprache für das Semantic Web
- wird mit OWL kombiniert
- Eigenschaften
 - meistens entscheidbar
 - monoton
 - eigene Built-Ins möglich
- Anwendungen
 - komplexeres Reasoning
 - Ausdrucksstarkes Ontology Mapping

Nicht-Monotone Regeln

- Manchmal sind monotone Regeln nicht zielführend
 - Beispiel: *wenn ein Student die Prüfung "Semantic Web" besteht, erhöht sich seine Credit-Point-Anzahl um 6*
- Erster Anlauf mit monotonen Regeln:
 - Student(?x) \wedge hatBestanden(?x,:SemanticWeb)
 - \wedge hatCredits(?x,?c) \wedge add(?nc,?c,6)
 - \rightarrow hatCredits(?x,?nc) .
- Ist das eine gute Idee?

Nicht-monotone Regeln

- Schauen wir uns ein Beispiel an:

`:Peter a :Student .`

`:Peter :hasCredits 26^^xsd:integer .`

`:Peter :hatBestanden :SemanticWeb .`

- Jetzt kommt die Regel zum Einsatz :

`:Peter :hasCredits 32^^xsd:integer .`

- Das gilt jetzt zusätzlich zu

`:Peter :hasCredits 26^^xsd:integer .`

- ...und der Reasoner ist noch lange nicht fertig

Nicht-monotone Regeln

- Was passiert:

```
:Peter :hasCredits 26^^xsd:integer .  
:Peter :hasCredits 32^^xsd:integer .  
:Peter :hasCredits 38^^xsd:integer .  
:Peter :hasCredits 44^^xsd:integer .  
:Peter :hasCredits 50^^xsd:integer .  
...
```

- Wir brauchen also Mechanismen für die Terminierung

Jena Rules

- Jena haben wir schon kennen gelernt
 - ein Programmierframework für Semantic Web
 - mit Reasoner
 - hat auch eine eigene Regel-Engine
 - ...mit eigener Sprache

- Grundidee:
 - Regeln arbeiten auf RDF-Daten
 - Kopf und Körper einer Regel sind Mengen von Tripeln
 - Tripel können Variablen enthalten
- Einfaches Beispiel:
[rule1: (?x :fatherOf ?y) (?y :fatherOf ?z) → (?x :grandfatherOf ?z)]

Built-Ins in Jena Rules

- Viele der aus SWRL bekannten Built-Ins gibt es hier auch
 - lessThan, sum, product, ...
- Interessant für Datumsoperationen
 - now(?d)
- Tests auf RDF-Knoten, Stringoperationen
 - isLiteral, isBlank, ...
 - regex
- Negation
 - noValue(?s ?p ?o)
 - heißt: kein solches Tripel existiert

Built-Ins für nicht-monotone Regeln

- Ein Built-In ist hier besonders interessant:
 - `remove(n)`
 - Löscht ein Tripel
 - `n` steht für die Nummer des Tripels in der Regel (Beginn bei 0)
- Beispiel:

```
[swcred: (?x a :Student) (?x :hatBestanden :SemanticWeb)
  (?x hatCredits ?c) sum(?c, 6, ?nc)
-> remove(2) (?x hatCredits ?nc) ]
```

Built-Ins für nicht-monotone Regeln

- Was passiert jetzt?

- Gegeben:

```
:Peter :hatCredits 26^^xsd:integer .
```

- Nach der Regel:

```
:Peter :hatCredits 32^^xsd:integer .
```

- Aber nach der Regel ist vor der Regel...

```
:Peter :hatCredits 38^^xsd:integer .
```

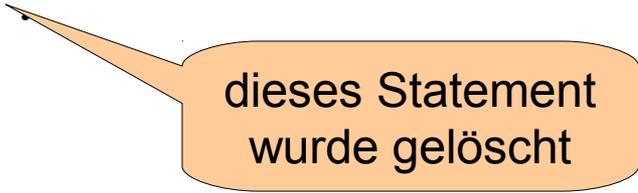
```
:Peter :hatCredits 44^^xsd:integer .
```

```
:Peter :hatCredits 50^^xsd:integer .
```

```
:Peter :hatCredits 56^^xsd:integer .
```

```
:Peter :hatCredits 62^^xsd:integer .
```

...



dieses Statement
wurde gelöscht

Abbruchkriterien in nicht-monotonen Regeln

- Da müssen wir uns vorsehen
- Neuer Versuch:

```
[swcred: (?x a :Student) (?x :hatBestanden :SemanticWeb)
  (?x hatCredits ?c) sum(?c,6,?nc)
  noValue(?x :swcredFiredFor :SemanticWeb)
->
  remove(2) (?x hatCredits ?nc)
  (?x :swcredFiredFor :SemanticWeb)]
```

- Damit wird die Regel nur 1x gefeuert
- Mit einer zusätzlichen Anweisung können wir das interne Hilfsstatement "verstecken":

```
hide (:swcredFiredFor)
```

Nicht-monotone Regeln: Neue Terminierungsprobleme

- Nicht-monotone Regeln können neue Terminierungsprobleme aufwerfen

```
[ex1:      (?x hatWert 1)
           noValue(?x hatWert 0)
->         remove(0) (?x hatWert 0) ]
[ex2:      (?x hatWert 0)
           noValue(?x hatWert 1)
->         remove(0) (?x hatWert 1) ]
```

- Das passiert bei monotonen Regeln nicht!

Jena Rules: Instanzen erzeugen



- Auch dafür gibt es Built-Ins
 - `makeInstance(?s ?p ?t ?v)`
 - Erzeuge eine neue Instanz vom Typ `?t`, binde sie an `?v`
 - Erzeuge das Tripel `?s ?p ?v`
- Beispiel:
 - Jedes Auto hat einen Motor

```
[motor: (?x a :Auto) noValue(?x :hatMotor ?y)
->      makeInstance(?x :hatMotor :Motor ?m) ]
```

Jena Rules: Instanzen erzeugen



- Auch das kann schiefgehen
 - Jeder Mensch hat einen Vater

```
[vater: (?x a :Mensch) noValue(?x :hatVater ?y)  
->      makeInstance(?x :hatVater :Mensch ?v) ]
```

- Vor der Regel:

```
:Peter a :Mensch .
```

- Nach der Regel:

```
_:v0 a :Mensch .  
:Peter :hatVater _:v0 .
```

- Weiter geht's:

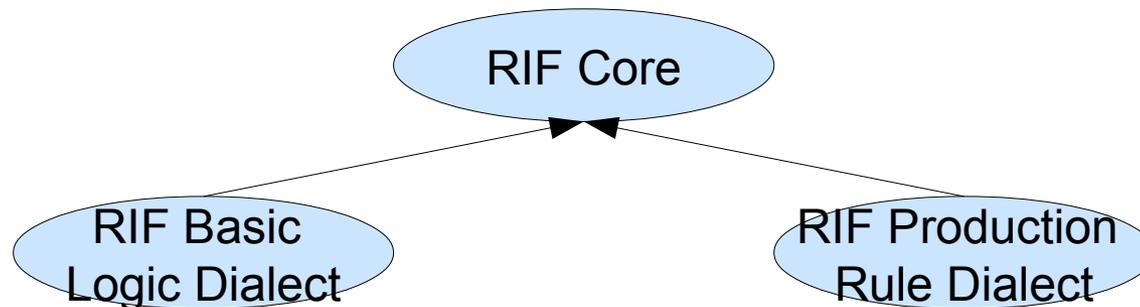
```
_:v0 :hatVater _:v1 .  
...
```

Zusammenfassung Jena Rules

- Ein mächtiges Werkzeug
 - "normale" Ableitungsregeln
 - Negation
 - Ändern, Hinzufügen, Löschen von Instanzen
- aber Terminierung nicht garantiert!

W3C-Standard: RIF

- Es gibt verschiedene Regelsprachen
 - für reine Ableitungsregeln (z.B. SWRL)
 - für Produktionsregeln (z.B. Jena Rules)
 - ...
- W3C-Standard RIF: gemeinsames Austauschformat
- Standardisiert 2010



Zusammenfassung

- Regeln können mehr als Ontologien
 - Trade-Off: Entscheidbarkeit
- Es gibt mehrere Arten von Regeln
 - Ableitungsregeln
 - Produktionsregeln
 - ...
- SWRL: Regelsprache für das Semantic Web
 - Monotone Regeln
- Jena Rules: regelbasiertes Programmieren
 - Nicht-monotone Regeln möglich
- RIF: standardisiertes Austauschformat

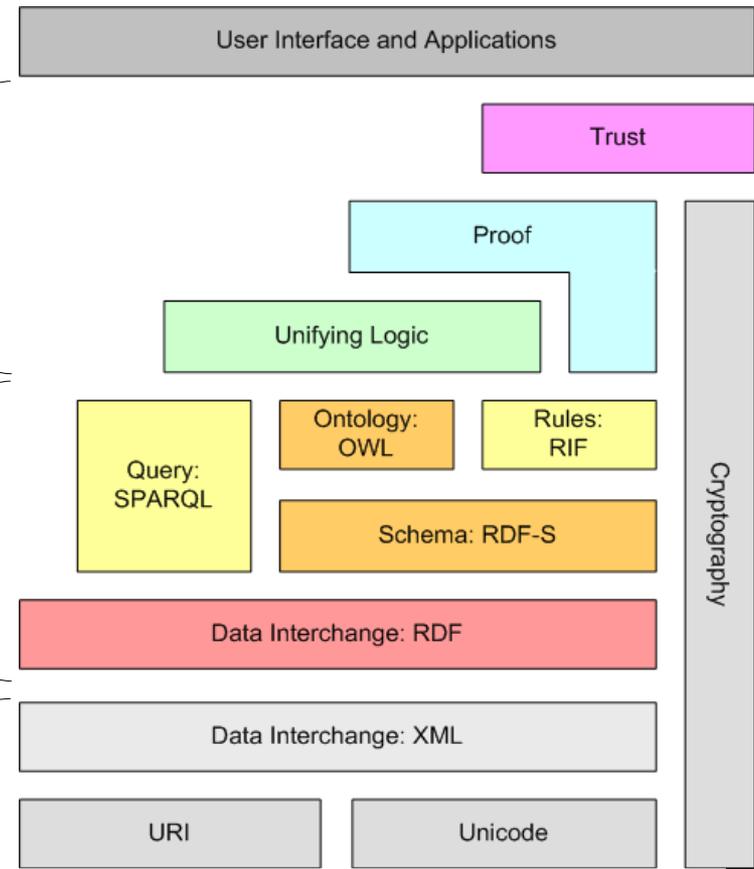
Semantic Web – Aufbau



here be dragons...

Semantic-Web-
Technologie
(Fokus der Vorlesung)

Technische
Grundlagen



Berners-Lee (2009): *Semantic Web and Linked Data*
<http://www.w3.org/2009/Talks/0120-campus-party-tbl/>

Vorlesung Semantic Web



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Vorlesung im Wintersemester 2012/2013

Dr. Heiko Paulheim

Fachgebiet Knowledge Engineering