Inference rules in DL and SWRL

G. Falquet
C. Métral

Expressivity of DL

Vocabulary:

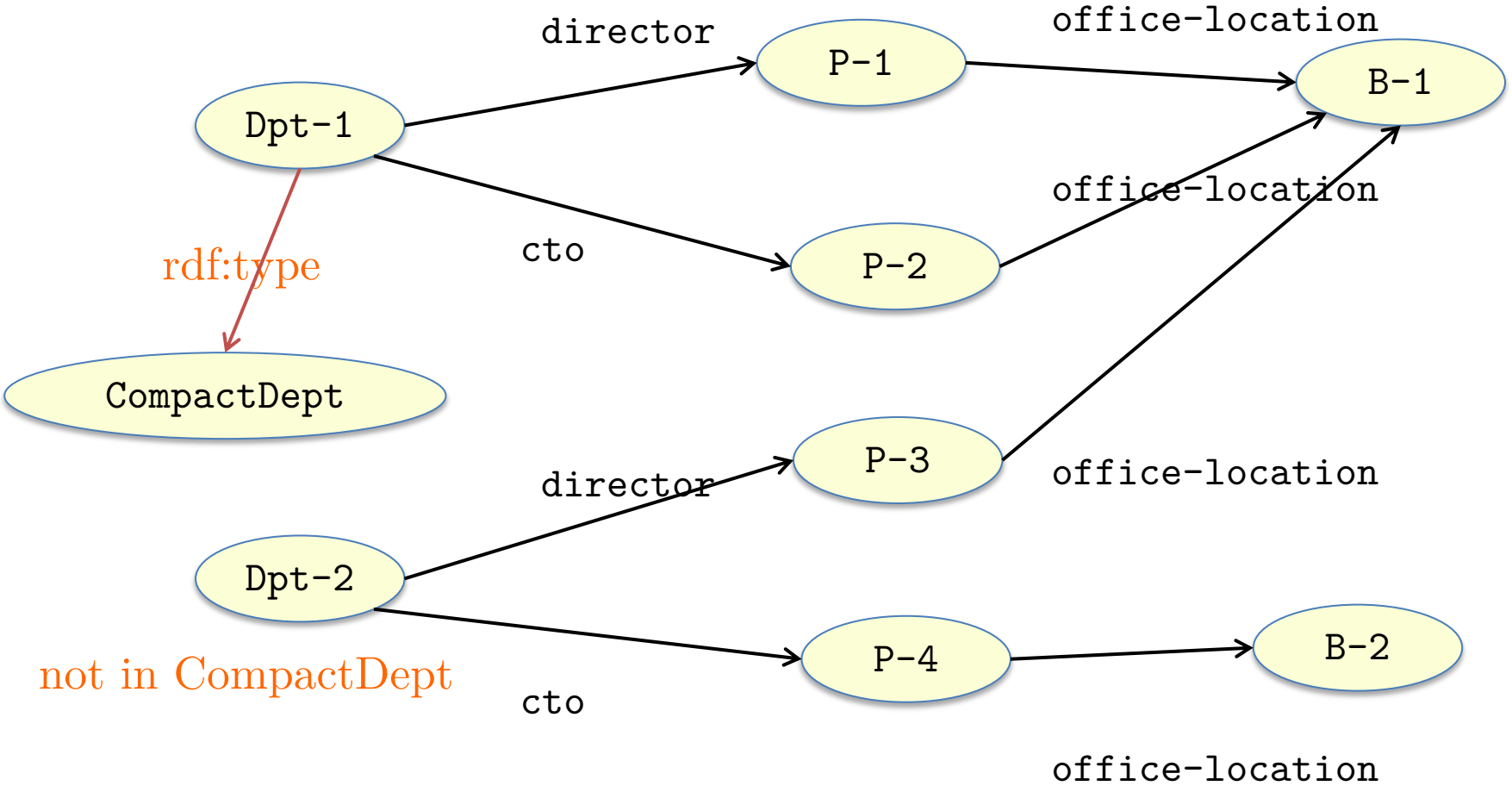
classes: `Department, Employee, Building`

properties: `director, cto, office-location`

How to define a class `CompactDept` to represent

departments that have their director and chief technology officer offices located in the same building

`CompactDept` ≡ ???


director
office-location
P-1
B-1
Dpt-1
office-location
rdf:type
cto
P-2
CompactDept
P-3
director
office-location
Dpt-2
P-4
B-2
not in CompactDept
cto
office-location

In DL (OWL 2)

Impossible to define `CompactDept` in OWL-2

Many other examples cannot be defined in OWL-2

Theoretical reason: most DLs enjoy the Tree Model Property.

if a Tbox has a model

then it has a model that doesn't contain cycles

A fact is a consequence of a Tbox if it is true in every model of the Tbox

⇒ no "cyclic fact" is a consequence of a TBox.

⇒ Need for a another language to express these facts

Inference rules

Rules to produce

New type assertions

x is a member of class C

New property assertions

x is connected to y through property p

Inference rules

To produce type assertions

$$B_1 \wedge B_2 \wedge \cdots \wedge B_k \rightarrow C(x)$$

To produce new property assertions

$$B_1 \wedge B_2 \wedge \cdots \wedge B_k \rightarrow p(x, y)$$

B_i is either a class assertion $C(t)$ or a property assertion $p(u, v)$
t, u, v are either individual names or variables

$$Restaurant(x) \wedge hasMenu(x, m) \wedge contains(m, caviar) \rightarrow Expensive(x)$$

$$hasChild(x, y) \rightarrow hasParent(y, x)$$

SWRL Rules - syntax

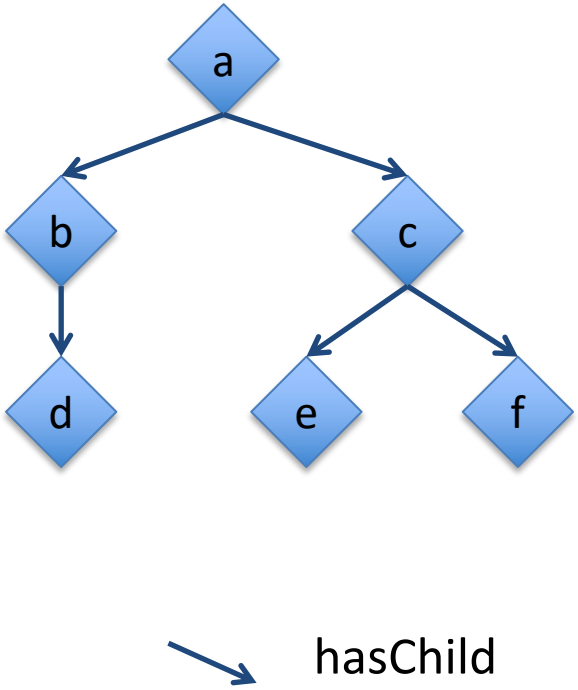
```
rule ::= antecedant -> consequent

antecedant  ::= atom, atom, ...
consequent ::= atom, atom, ...

atom ::= description '(' i-object ')'
| dataRange '(' d-object ')'
| individualvaluedPropertyID '(' i-object i-object ')'
| datavaluedPropertyID '(' i-object d-object ')'
| sameAs '(' i-object i-object ')'
| differentFrom '(' i-object i-object ')'
| builtIn '(' builtinID { d-object } ')'
```

Person(?x), Person(?y), Person(?z), hasChild(?x, ?y), hasChild(?y, ?z) ->
hasGrandChild(?x, ?z)

Interpretation

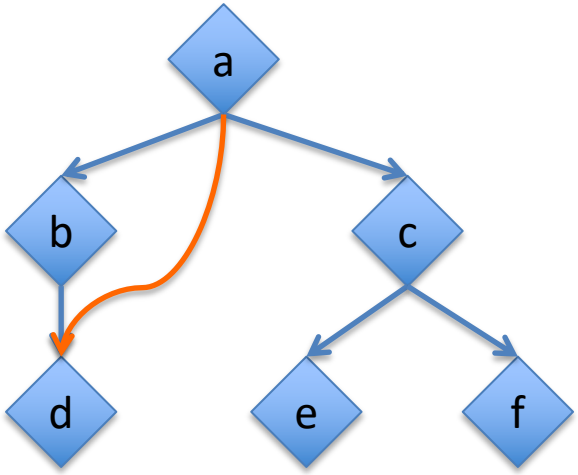
- Find all the variable bindings that satisfy the antecedent
- For each such binding the consequent must be satisfied

```
hasChild(?x, ?y), hasChild(?y, ?z)
    -> hasGrandChild(?x, ?z)
```



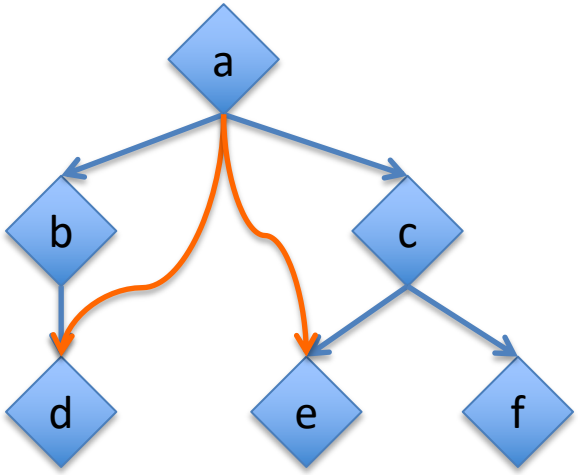
Interpretation

```
hasChild(?x, ?y), hasChild(?y, ?z)
    -> hasGrandChild(?x, ?z)

x=a, y=b, z=d -> hasGrandChild(a,d)
```



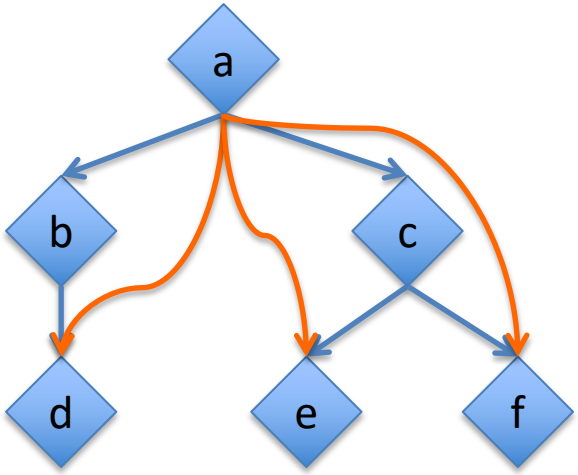
Interpretation

```
hasChild(?x, ?y), hasChild(?y, ?z)
    -> hasGrandChild(?x, ?z)

x=a, y=b, z=d -> hasGrandChild(a,d)
x=a, y=c, z=e -> hasGrandChild(a,e)
```



Interpretation

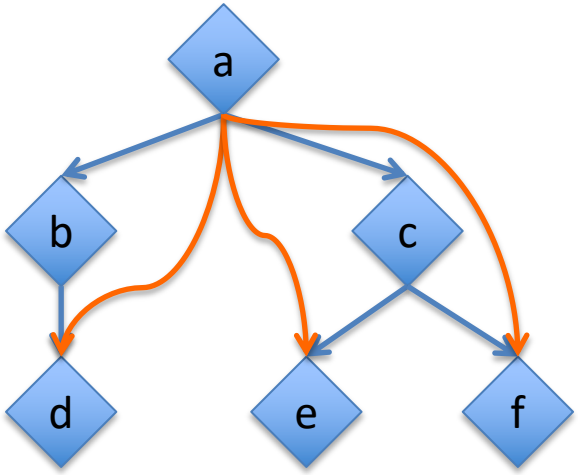
```
hasChild(?x, ?y), hasChild(?y, ?z)
    -> hasGrandChild(?x, ?z)

x=a, y=b, z=d -> hasGrandChild(a,d)
x=a, y=c, z=e -> hasGrandChild(a,e)
x=a, y=c, z=f -> hasGrandChild(a,f)
```



Interpretation

```
hasChild(?x, ?y), hasChild(?y, ?z)
    -> hasGrandChild(?x, ?z)
```

```
x=a, y=b, z=d -> hasGrandChild(a,d)
x=a, y=c, z=e -> hasGrandChild(a,e)
x=a, y=c, z=f -> hasGrandChild(a,f)
```

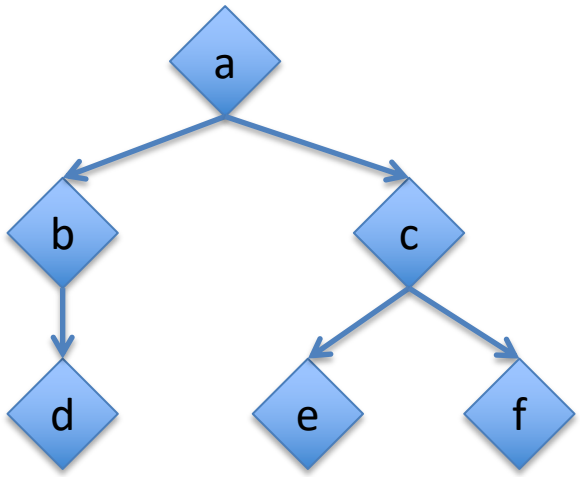


an interpretation that satisfies the rule

DifferentFrom

Variables with different names may represent the same individual !

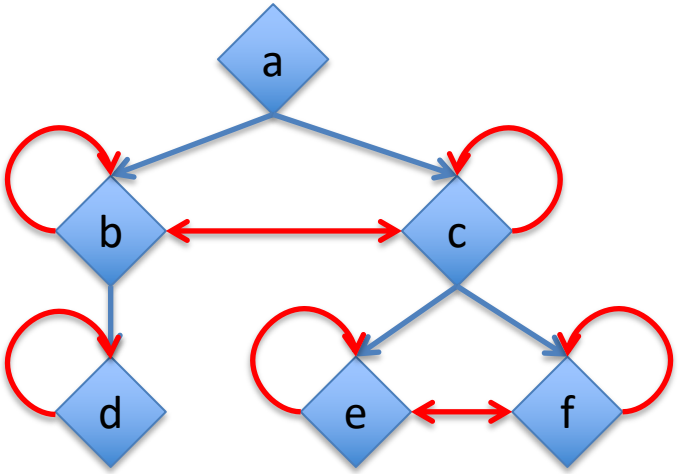
```
hasChild(?x, ?y), hasChild(?x, ?z)
-> hasSibling(?y, ?z)
```



DifferentFrom

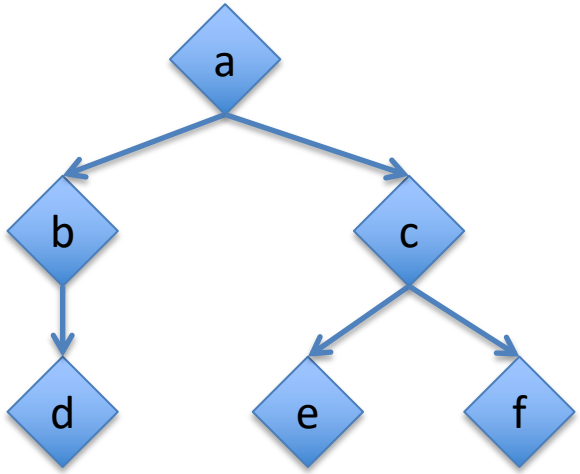
Variables with different names may represent the same individual !

```
hasChild(?x, ?y), hasChild(?x, ?z)
-> hasSibling(?y, ?z)
```



DifferentFrom

```
hasChild(?x, ?y), hasChild(?x, ?z),  DifferentFrom (?y, ?z)
-> hasSibling(?y, ?z)
```

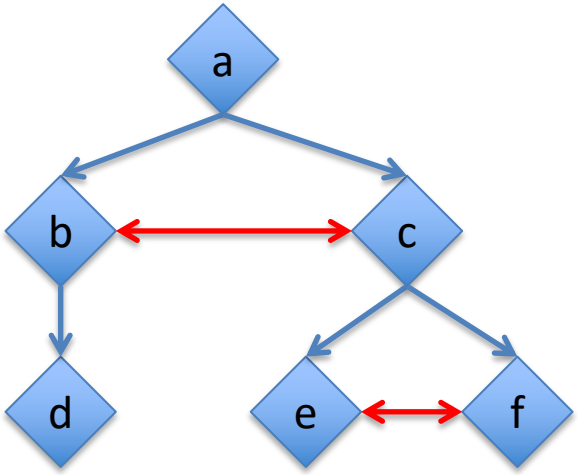


DifferentFrom

```
hasChild(?x, ?y), hasChild(?x, ?z),  DifferentFrom (?y, ?z)
-> hasSibling(?y, ?z)
```

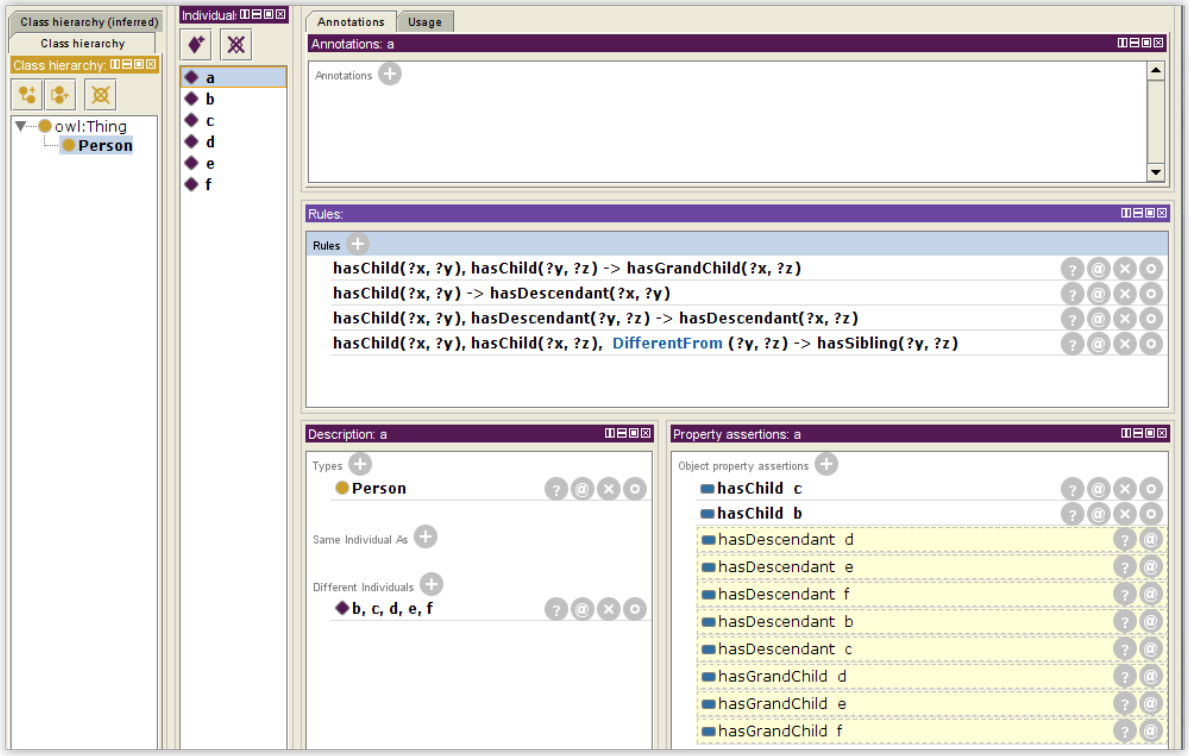
works only if

```
DifferentIndividual(b,c)
DifferentIndividual(e,f)
```



Example

```
hasChild(?x, ?y) -> hasDescendant(?x, ?y)
hasChild(?x, ?y), hasDescendant(?y, ?z) -> hasDescendant(?x, ?z)
```



DL-safe rules

Query answering for DL-axioms + rules is undecidable

It is decidable if rules are DL-safe

A rule r is called DL-safe if each variable in r occurs in a non-DL-atom in the rule body.

Practically: the variables in rules can only be bound to known individuals

Axioms:

TBox: Parent ≡ hasChild some Person

ABox: Parent(a), Parent(b), Parent(c), Person(d), hasChild(a,d)

Rule:

hasChild(?x, ?y) -> PersonWithChild(?x)

consequence:

PersonWithChild(a)

without the DL-safe restriction:

PersonWithChild(a), PersonWithChild(b), PersonWithChild(c)

Builtin predicates

To deal with numbers, strings, etc.

```
Rectangle(?x), hasWidthInMetres(?x, ?w), greaterThan(?w, 10)
-> WideRectangle(?x)
```

```
Rectangle(?x), hasHeightInMetres(?x, ?h), hasWidthInMetres(?x,
?w), greaterThan(?a, 100), multiply(?a, ?w, ?h)
-> LargeRectangle(?x)
```

```
swrlb:equal
swrlb:notEqual
swrlb:lessThan
swrlb:lessThanOrEqual
swrlb:greaterThan
swrlb:greaterThanOrEqual
```

swrlb:add
swrlb:subtract
swrlb:multiply
swrlb:divide
swrlb:integerDivide
swrlb:mod
swrlb:pow
swrlb:unaryPlus
swrlb:unaryMinus
swrlb:abs
swrlb:ceiling
swrlb:floor
swrlb:round
swrlb:roundHalfToEven
swrlb:sin
swrlb:cos
swrlb:tan

swrlb:stringEqualIgnoreCase
swrlb:stringConcat
swrlb:substring
swrlb:stringLength
swrlb:normalizeSpace
swrlb:upperCase
swrlb:lowerCase
swrlb:translate
swrlb:contains
swrlb:containsIgnoreCase
swrlb:startsWith
swrlb:endsWith
swrlb:substringBefore
swrlb:substringAfter
swrlb:matches
swrlb:replace
swrlb:tokenize

When you don't need SWRL: DL rules

Some SWRL rules can be encoded in OWL expressions

Example

Man(?x) $\wedge$ hasBrother(?x,?y) $\wedge$ hasChild(?y,?z) $\rightarrow$ Uncle(?x)

becomes

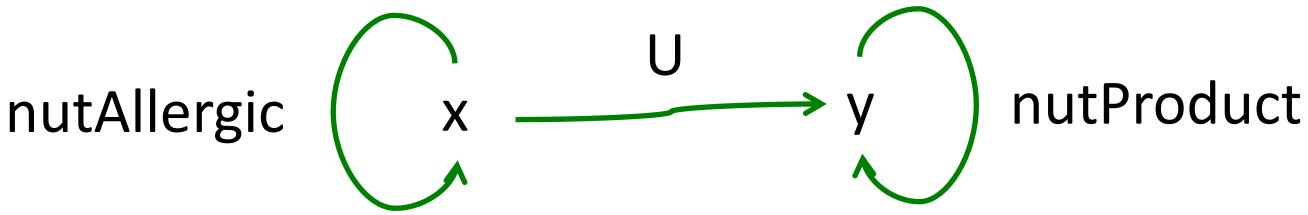
Man $\sqcap$ $\exists$hasBrother.$\exists$hasChild.$\top$ $\sqsubseteq$ Uncle

it's sometimes tricky ...

NutAllergic(x) ∧ NutProduct(y) → dislikes(x,y)

NutAllergic ≡ ∃nutAllergic.Self
NutProduct ≡ ∃nutProduct.Self
nutAllergic o U o nutProduct ⊑ dislikes

U = universal property (x U y is always true)

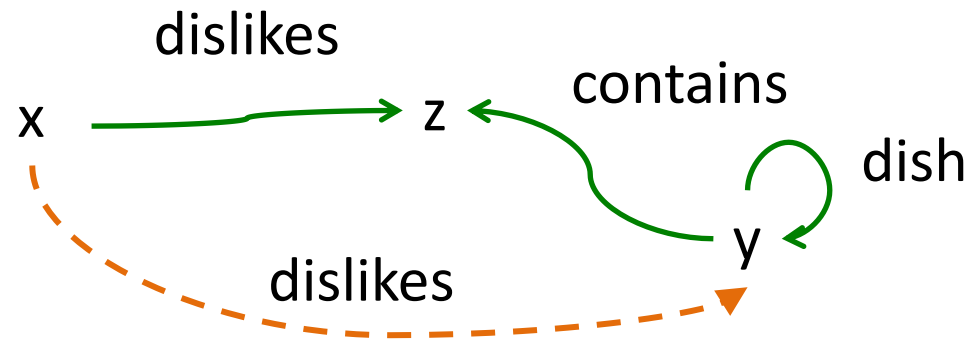


... more

dislikes(x,z) ∧ Dish(y) ∧ contains(y,z) → dislikes(x,y)

becomes

- **Dish ≡ ∃dish.Self**
- **dislikes o contains⁻ o dish ⊑ dislikes**



Rules vs. SPARQL queries

- Rules are “executed” globally
  - all rules must be satisfied simultaneously
- Rules may have interactions
  - the outcome of a rule may trigger another one
- SPARQL queries are executed independently

Simulating rules with queries

define a 'construct' query for each rule

repeat

- execute each query
- add the results to the RDF graph

until nothing new is created

parent(?x, ?y)∧ ancestor(?y, ?z) → ancestor(?x, ?z)

construct {?x ancestor ?z.}
where {?x parent ?y. ?y ancestor ?z.}

Simulating rules with queries

parent(?x, ?y)$\wedge$ ancestor(?y, ?z) $\rightarrow$ ancestor(?x, ?z)

repeat

```
    construct {?x ancestor ?z.}
    where {?x parent ?y. ?y ancestor ?z.}
```

until nothing new

Can be extremely inefficient