Ontology patterns in practice: Reasonable Ontology Templates (OTTR)

Martin G. Skjæveland, Leif Harald Karlsen, Daniel P. Lupp

Improving the efficiency and quality of ontology engineering

Problem:

- Current ontology engineering methods are too low-level
- Makes ontology construction repetitive and error-prone
- Difficult to engage end-users
- Difficult to generate sustainable large ontologies
- Difficult to efficiently maintain ontologies

Improving the efficiency and quality of ontology engineering

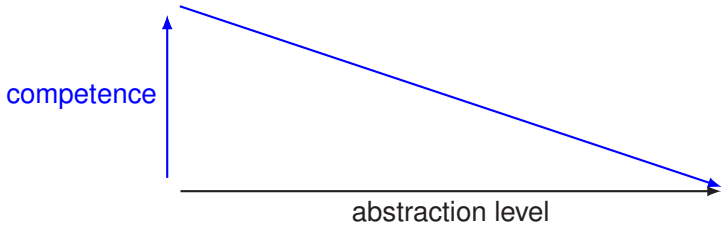
Problem:

- Current ontology engineering methods are too low-level
- Makes ontology construction repetitive and error-prone
- Difficult to engage end-users
- Difficult to generate sustainable large ontologies
- Difficult to efficiently maintain ontologies

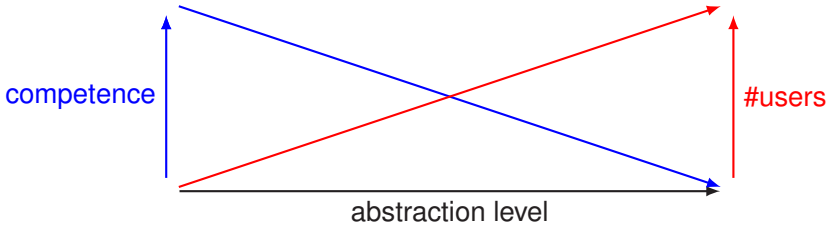
We need:

- Abstractions!
- Capture and instantiate reoccurring modelling patterns
- Formats adapted to domain experts, data managers and ontology experts

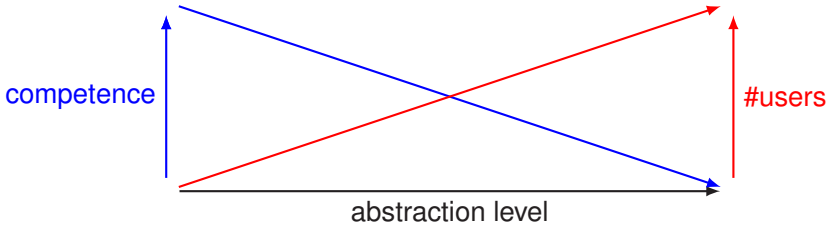
Abstractions



Abstractions

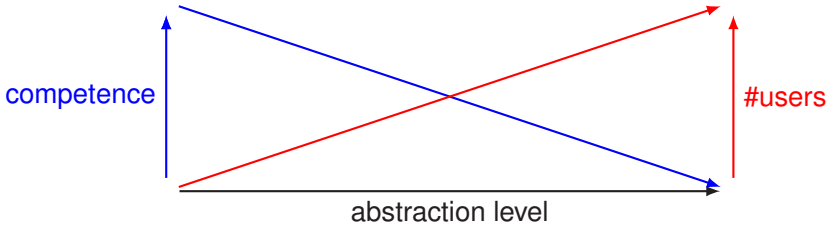


Abstractions



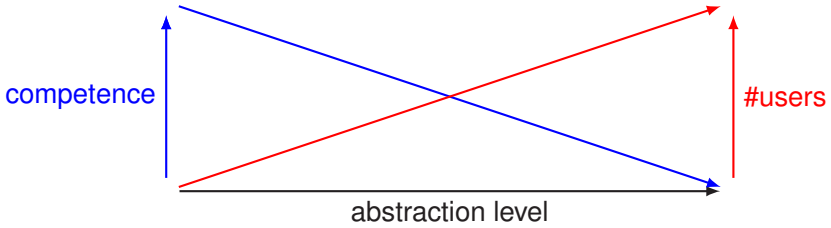
- Programming: assembly vs. high-level languages

Abstractions



- Programming: assembly vs. high-level languages
- Java: garbage collector — I don't care!

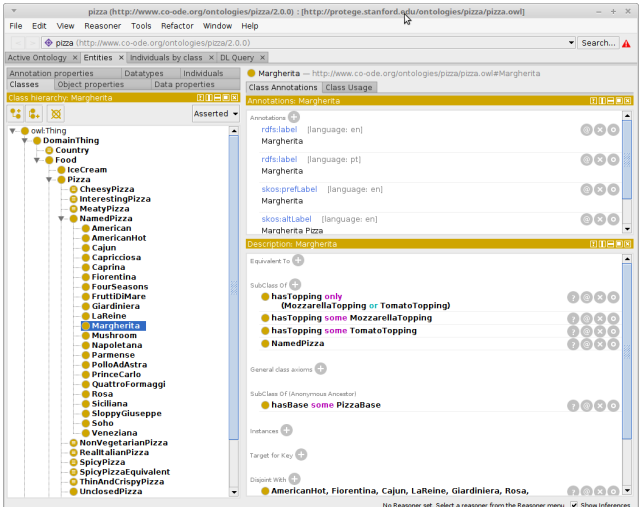
Abstractions



- Programming: assembly vs. high-level languages
- Java: garbage collector — I don't care!
- Ontology engineering:
  - still low-level OWL "coding"
  - impossible to not care about logic

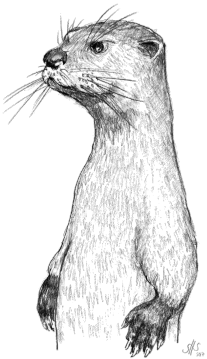
Pizza Ontology

- Pizza ontology tutorial
- 22 NamedPizza-s
- Difficult to find all instances—understanding the ontology
- Error-prone to update pattern



Reasonable Ontology Templates (OTTR)

- Can be viewed as a macro language for knowledge bases
- Abstractions over RDF and OWL
  - hiding complexity and
  - providing friendly interfaces
- Build and interact with ontologies via templates
  - DRY: Do not Repeat Yourself
  - Uniform modelling
  - Completeness of input
- Leverage existing W3C stack and tools



OTTR

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .

p:Hawaii rdf:type owl:Class .
p:Hawaii rdfs:subClassOf p:Pizza .
p:Hawaii rdfs:label "Hawaii"@en .

p:Grandiosa rdf:type owl:Class .
p:Grandiosa rdfs:subClassOf p:Pizza .
p:Grandiosa rdfs:label "Grandiosa"@no .
```

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .

p:Hawaii rdf:type owl:Class .
p:Hawaii rdfs:subClassOf p:Pizza .
p:Hawaii rdfs:label "Hawaii"@en .

p:Grandiosa rdf:type owl:Class .
p:Grandiosa rdfs:subClassOf p:Pizza .
p:Grandiosa rdfs:label "Grandiosa"@no .
```

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .

p:Hawaii rdf:type owl:Class .
p:Hawaii rdfs:subClassOf p:Pizza .
p:Hawaii rdfs:label "Hawaii"@en .

p:Grandiosa rdf:type owl:Class .
p:Grandiosa rdfs:subClassOf p:Pizza .
p:Grandiosa rdfs:label "Grandiosa"@no .
```

```
?name rdf:type owl:Class   .
?name rdfs:subClassOf p:Pizza   .
?name rdfs:label ?label   .
```

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .

p:Hawaii rdf:type owl:Class .
p:Hawaii rdfs:subClassOf p:Pizza .
p:Hawaii rdfs:label "Hawaii"@en .

p:Grandiosa rdf:type owl:Class .
p:Grandiosa rdfs:subClassOf p:Pizza .
p:Grandiosa rdfs:label "Grandiosa"@no .
```

PIZZA[?name, ?label] :: {

```
        ?name rdf:type owl:Class   .
        ?name rdfs:subClassOf p:Pizza   .
        ?name rdfs:label ?label   .
```

} .

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .

p:Hawaii rdf:type owl:Class .
p:Hawaii rdfs:subClassOf p:Pizza .
p:Hawaii rdfs:label "Hawaii"@en .

p:Grandiosa rdf:type owl:Class .
p:Grandiosa rdfs:subClassOf p:Pizza .
p:Grandiosa rdfs:label "Grandiosa"@no .
```

$$
\begin{aligned}
&\textsc{Pizza}[?\text{name}, ?\text{label}] :: \{ \\
&\quad \textsc{Triple}(?\text{name}, \texttt{rdf:type}, \texttt{owl:Class}), \\
&\quad \textsc{Triple}(?\text{name}, \texttt{rdfs:subClassOf}, \texttt{p:Pizza}), \\
&\quad \textsc{Triple}(?\text{name}, \texttt{rdfs:label}, ?\text{label}) \\
&\} .
\end{aligned}
$$

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .

p:Hawaii rdf:type owl:Class .
p:Hawaii rdfs:subClassOf p:Pizza .
p:Hawaii rdfs:label "Hawaii"@en .

p:Grandiosa rdf:type owl:Class .
p:Grandiosa rdfs:subClassOf p:Pizza .
p:Grandiosa rdfs:label "Grandiosa"@no .
```

$$
\begin{aligned}
&\textsc{Pizza}[?\text{name}, ?\text{label}] :: \{ \\
&\quad \textsc{Triple}(?\text{name}, \texttt{rdf:type}, \texttt{owl:Class}), \\
&\quad \textsc{Triple}(?\text{name}, \texttt{rdfs:subClassOf}, \texttt{p:Pizza}), \\
&\quad \textsc{Triple}(?\text{name}, \texttt{rdfs:label}, ?\text{label}) \\
&\} . \\
&\textsc{Pizza}(\texttt{p:Margherita}, \texttt{"Margherita"@it}) . \\
&\textsc{Pizza}(\texttt{p:Hawaii}, \texttt{"Hawaii"@en}) . \\
&\textsc{Pizza}(\texttt{p:Grandiosa}, \texttt{"Grandiosa"@no}) .
\end{aligned}
$$

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .

p:Hawaii rdf:type owl:Class .
p:Hawaii rdfs:subClassOf p:Pizza .
p:Hawaii rdfs:label "Hawaii"@en .

p:Grandiosa rdf:type owl:Class .
p:Grandiosa rdfs:subClassOf p:Pizza .
p:Grandiosa rdfs:label "Grandiosa"@no .
```

```
SubClassOf[?sub, ?super] :: {
   Triple(?sub, rdfs:subClassOf, ?super)
} .
Pizza[?name, ?label] :: {
   Triple(?name, rdf:type, owl:Class),
   Triple(?name, rdfs:subClassOf, p:Pizza),
   Triple(?name, rdfs:label, ?label)
} .
Pizza(p:Margherita, "Margherita"@it) .
Pizza(p:Hawaii, "Hawaii"@en) .
Pizza(p:Grandiosa, "Grandiosa"@no) .
```

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .

p:Hawaii rdf:type owl:Class .
p:Hawaii rdfs:subClassOf p:Pizza .
p:Hawaii rdfs:label "Hawaii"@en .

p:Grandiosa rdf:type owl:Class .
p:Grandiosa rdfs:subClassOf p:Pizza .
p:Grandiosa rdfs:label "Grandiosa"@no .
```

```
SubClassOf[?sub, ?super] :: {
   Triple(?sub, rdfs:subClassOf, ?super)
} .
Pizza[?name, ?label] :: {
   Triple(?name, rdf:type, owl:Class),
   SubClassOf(?name, p:Pizza),
   Triple(?name, rdfs:label, ?label)
} .
Pizza(p:Margherita, "Margherita"@it) .
Pizza(p:Hawaii, "Hawaii"@en) .
Pizza(p:Grandiosa, "Grandiosa"@no) .
```

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .
```

- Abstractions

```
SubClassOf[?sub, ?super] :: {
    Triple(?sub, rdfs:subClassOf, ?super)
} .
Pizza[?name, ?label] :: {
    Triple(?name, rdf:type, owl:Class),
    SubClassOf(?name, p:Pizza),
    Triple(?name, rdfs:label, ?label)
} .
Pizza(p:Margherita, "Margherita"@it) .
Pizza(p:Hawaii, "Hawaii"@en) .
Pizza(p:Grandiosa, "Grandiosa"@no) .
```

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .
```

- Abstractions
- Don't Repeat Yourself (DRY)

SubClassOf[?sub, ?super] :: {
  Triple(?sub, rdfs:subClassOf, ?super)
} .
Pizza[?name, ?label] :: {
  Triple(?name, rdf:type, owl:Class),
  SubClassOf(?name, p:Pizza),
  Triple(?name, rdfs:label, ?label)
} .
Pizza(p:Margherita, "Margherita"@it) .
Pizza(p:Hawaii, "Hawaii"@en) .
Pizza(p:Grandiosa, "Grandiosa"@no) .

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .
```

- Abstractions
- Don't Repeat Yourself (DRY)
- Compositional patterns
- Encapsulate complexity

```
SubClassOf[?sub, ?super] :: {
   Triple(?sub, rdfs:subClassOf, ?super)
} .
Pizza[?name, ?label] :: {
   Triple(?name, rdf:type, owl:Class),
   SubClassOf(?name, p:Pizza),
   Triple(?name, rdfs:label, ?label)
} .
Pizza(p:Margherita, "Margherita"@it) .
Pizza(p:Hawaii, "Hawaii"@en) .
Pizza(p:Grandiosa, "Grandiosa"@no) .
```

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .
```

- Abstractions
- Don't Repeat Yourself (DRY)
- Compositional patterns
- Encapsulate complexity
- Uniform modelling

$$
\begin{aligned}
&\textsc{SubClassOf}[?\text{sub}, ?\text{super}] :: \{ \\
&\quad \textsc{Triple}(?\text{sub}, \texttt{rdfs:subClassOf}, ?\text{super}) \\
&\} . \\
&\textsc{Pizza}[?\text{name}, ?\text{label}] :: \{ \\
&\quad \textsc{Triple}(?\text{name}, \texttt{rdf:type}, \texttt{owl:Class}), \\
&\quad \textsc{SubClassOf}(?\text{name}, \texttt{p:Pizza}), \\
&\quad \textsc{Triple}(?\text{name}, \texttt{rdfs:label}, ?\text{label}) \\
&\} . \\
&\textsc{Pizza}(\texttt{p:Margherita}, \texttt{"Margherita"@it}) . \\
&\textsc{Pizza}(\texttt{p:Hawaii}, \texttt{"Hawaii"@en}) . \\
&\textsc{Pizza}(\texttt{p:Grandiosa}, \texttt{"Grandiosa"@no}) .
\end{aligned}
$$

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .
```

- Abstractions
- Don't Repeat Yourself (DRY)
- Compositional patterns
- Encapsulate complexity
- Uniform modelling
- Separate design and content

```
SubClassOf[?sub, ?super] :: {
    Triple(?sub, rdfs:subClassOf, ?super)
} .
Pizza[?name, ?label] :: {
    Triple(?name, rdf:type, owl:Class),
    SubClassOf(?name, p:Pizza),
    Triple(?name, rdfs:label, ?label)
} .
Pizza(p:Margherita, "Margherita"@it) .
Pizza(p:Hawaii, "Hawaii"@en) .
Pizza(p:Grandiosa, "Grandiosa"@no) .
```

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .
```

- Abstractions
- Don't Repeat Yourself (DRY)
- Compositional patterns
- Encapsulate complexity
- Uniform modelling
- Separate design and content
- Ensure completeness of input
- Simple input format

```
SubClassOf[?sub, ?super] :: {
   Triple(?sub, rdfs:subClassOf, ?super)
} .
Pizza[?name, ?label] :: {
   Triple(?name, rdf:type, owl:Class),
   SubClassOf(?name, p:Pizza),
   Triple(?name, rdfs:label, ?label)
} .
Pizza(p:Margherita, "Margherita"@it) .
Pizza(p:Hawaii, "Hawaii"@en) .
Pizza(p:Grandiosa, "Grandiosa"@no) .
```

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .
```

- Abstractions
- Don't Repeat Yourself (DRY)
- Compositional patterns
- Encapsulate complexity
- Uniform modelling
- Separate design and content
- Ensure completeness of input
- Simple input format

- Substitution
- Macro expansion

```
SubClassOf[?sub, ?super] :: {
   Triple(?sub, rdfs:subClassOf, ?super)
} .
Pizza[?name, ?label] :: {
   Triple(?name, rdf:type, owl:Class),
   SubClassOf(?name, p:Pizza),
   Triple(?name, rdfs:label, ?label)
} .
Pizza(p:Margherita, "Margherita"@it) .
Pizza(p:Hawaii, "Hawaii"@en) .
Pizza(p:Grandiosa, "Grandiosa"@no) .
```

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .
```

- Abstractions
- Don't Repeat Yourself (DRY)
- Compositional patterns
- Encapsulate complexity
- Uniform modelling
- Separate design and content
- Ensure completeness of input
- Simple input format

- Substitution
- Macro expansion

```
SubClassOf[?sub, ?super] :: {
   Triple(?sub, rdfs:subClassOf, ?super)
} .
Pizza[?name, ?label] :: {
   Triple(?name, rdf:type, owl:Class),
   SubClassOf(?name, p:Pizza),
   Triple(?name, rdfs:label, ?label)
} .
Pizza(p:Margherita, "Margherita"@it) .
Pizza(p:Hawaii, "Hawaii"@en) .
Pizza(p:Grandiosa, "Grandiosa"@no) .
```

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .
```

- Abstractions
- Don't Repeat Yourself (DRY)
- Compositional patterns
- Encapsulate complexity
- Uniform modelling
- Separate design and content
- Ensure completeness of input
- Simple input format

- Substitution
- Macro expansion

```
SubClassOf[?sub, ?super] :: {
   Triple(?sub, rdfs:subClassOf, ?super)
} .
Pizza[p:Margherita, "Margherita"@it] :: {
   Triple(p:Margherita, rdf:type, owl:Class),
   SubClassOf(p:Margherita, p:Pizza),
   Triple(p:Margherita, rdfs:label, "Margherita"@it)
} .
Pizza(p:Margherita, "Margherita"@it) .
Pizza(p:Hawaii, "Hawaii"@en) .
Pizza(p:Grandiosa, "Grandiosa"@no) .
```

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .
```

- Abstractions
- Don't Repeat Yourself (DRY)
- Compositional patterns
- Encapsulate complexity
- Uniform modelling
- Separate design and content
- Ensure completeness of input
- Simple input format

- Substitution
- Macro expansion

```
SubClassOf[?sub, ?super] :: {
   Triple(?sub, rdfs:subClassOf, ?super)
} .
Pizza[p:Margherita, "Margherita"@it] :: {
   Triple(p:Margherita, rdf:type, owl:Class),
   SubClassOf(p:Margherita, p:Pizza),
   Triple(p:Margherita, rdfs:label, "Margherita"@it)
} .
Pizza(p:Margherita, "Margherita"@it) .
Pizza(p:Hawaii, "Hawaii"@en) .
Pizza(p:Grandiosa, "Grandiosa"@no) .
```

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .
```

- Abstractions
- Don't Repeat Yourself (DRY)
- Compositional patterns
- Encapsulate complexity
- Uniform modelling
- Separate design and content
- Ensure completeness of input
- Simple input format

- Substitution
- Macro expansion

$$
\begin{aligned}
&\textsc{SubClassOf}[\texttt{p:Margherita}, \texttt{p:Pizza}] :: \{ \\
&\quad \textsc{Triple}(\texttt{p:Margherita}, \texttt{rdfs:subClassOf}, \texttt{p:Pizza}) \\
&\} . \\
&\textsc{Pizza}[\texttt{p:Margherita}, \texttt{"Margherita"@it}] :: \{ \\
&\quad \textsc{Triple}(\texttt{p:Margherita}, \texttt{rdf:type}, \texttt{owl:Class}), \\
&\quad \textsc{SubClassOf}(\texttt{p:Margherita}, \texttt{p:Pizza}), \\
&\quad \textsc{Triple}(\texttt{p:Margherita}, \texttt{rdfs:label}, \texttt{"Margherita"@it}) \\
&\} . \\
&\textsc{Pizza}(\texttt{p:Margherita}, \texttt{"Margherita"@it}) . \\
&\textsc{Pizza}(\texttt{p:Hawaii}, \texttt{"Hawaii"@en}) . \\
&\textsc{Pizza}(\texttt{p:Grandiosa}, \texttt{"Grandiosa"@no}) .
\end{aligned}
$$

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .
```

- Abstractions
- Don't Repeat Yourself (DRY)
- Compositional patterns
- Encapsulate complexity
- Uniform modelling
- Separate design and content
- Ensure completeness of input
- Simple input format

- Substitution
- Macro expansion

$$
\begin{aligned}
&\textsc{SubClassOf}[\texttt{p:Margherita}, \texttt{p:Pizza}] :: \{ \\
&\quad \textsc{Triple}(\texttt{p:Margherita}, \texttt{rdfs:subClassOf}, \texttt{p:Pizza}) \\
&\} . \\
&\textsc{Pizza}[\texttt{p:Margherita}, \texttt{"Margherita"@it}] :: \{ \\
&\quad \textsc{Triple}(\texttt{p:Margherita}, \texttt{rdf:type}, \texttt{owl:Class}), \\
&\quad \textsc{Triple}(\texttt{p:Margherita}, \texttt{rdfs:subClassOf}, \texttt{p:Pizza}), \\
&\quad \textsc{Triple}(\texttt{p:Margherita}, \texttt{rdfs:label}, \texttt{"Margherita"@it}) \\
&\} . \\
&\textsc{Pizza}(\texttt{p:Margherita}, \texttt{"Margherita"@it}) . \\
&\textsc{Pizza}(\texttt{p:Hawaii}, \texttt{"Hawaii"@en}) . \\
&\textsc{Pizza}(\texttt{p:Grandiosa}, \texttt{"Grandiosa"@no}) .
\end{aligned}
$$

Reasonable Ontology Templates (OTTR): Example

```
p:Margherita rdf:type owl:Class .
p:Margherita rdfs:subClassOf p:Pizza .
p:Margherita rdfs:label "Margherita"@it .
```

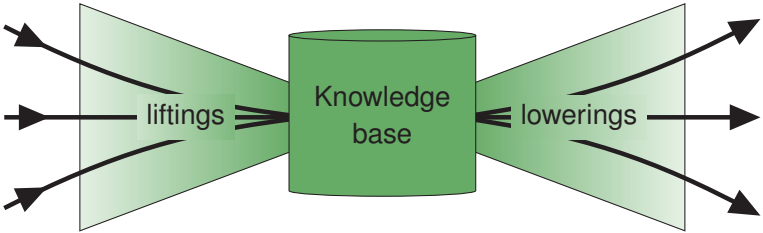
- Abstractions
- Don't Repeat Yourself (DRY)
- Compositional patterns
- Encapsulate complexity
- Uniform modelling
- Separate design and content
- Ensure completeness of input
- Simple input format

- Substitution
- Macro expansion

$$\textsc{Triple}(\texttt{p:Margherita}, \texttt{rdf:type}, \texttt{owl:Class}),$$
$$\textsc{Triple}(\texttt{p:Margherita}, \texttt{rdfs:subClassOf}, \texttt{p:Pizza}),$$
$$\textsc{Triple}(\texttt{p:Margherita}, \texttt{rdfs:label}, \texttt{"Margherita"@it})$$

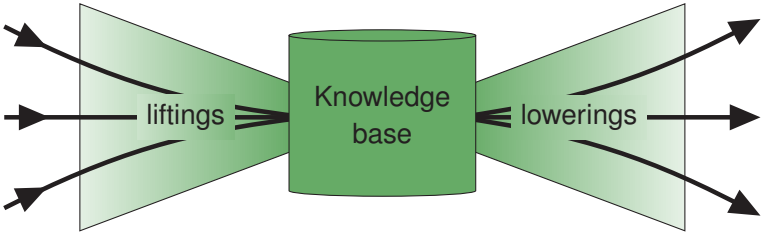
$$\textsc{Pizza}(\texttt{p:Margherita}, \texttt{"Margherita"@it}) .$$
$$\textsc{Pizza}(\texttt{p:Hawaii}, \texttt{"Hawaii"@en}) .$$
$$\textsc{Pizza}(\texttt{p:Grandiosa}, \texttt{"Grandiosa"@no}) .$$

Vision: Template-driven methodology



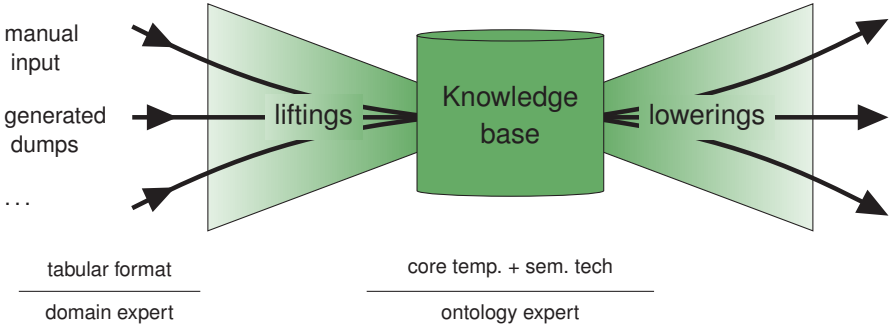
- “All” knowledge base interaction via templates
- “API for RDF/OWL”
- Maintain “all” interaction with KB by maintaining templates

Vision: Template-driven methodology



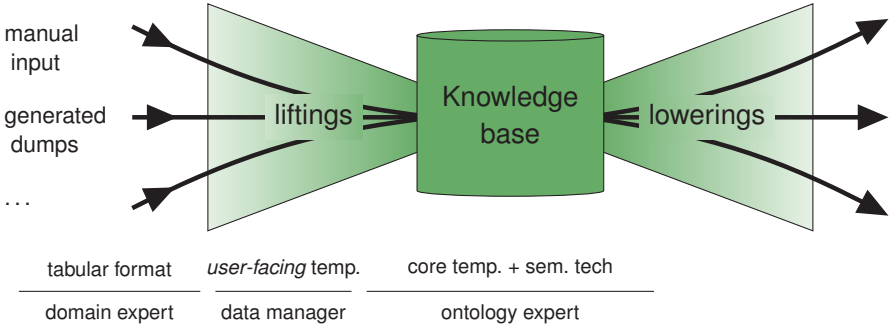
- “All” knowledge base interaction via templates
- “API for RDF/OWL”
- Maintain “all” interaction with KB by maintaining templates

Vision: Template-driven methodology



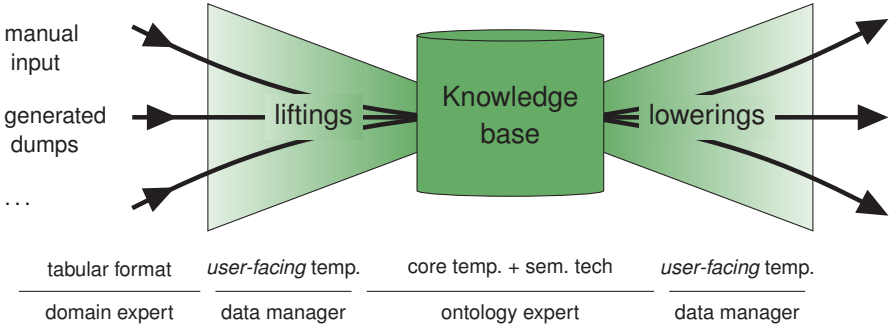
- “All” knowledge base interaction via templates
- “API for RDF/OWL”
- Maintain “all” interaction with KB by maintaining templates

Vision: Template-driven methodology



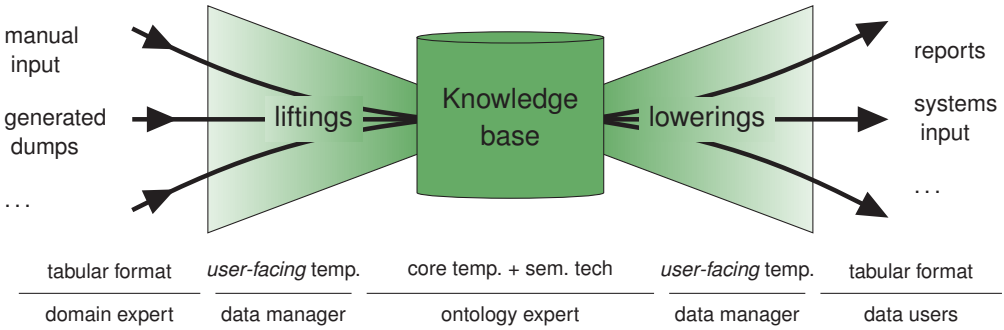
- “All” knowledge base interaction via templates
- “API for RDF/OWL”
- Maintain “all” interaction with KB by maintaining templates

Vision: Template-driven methodology



- "All" knowledge base interaction via templates
- "API for RDF/OWL"
- Maintain "all" interaction with KB by maintaining templates

Vision: Template-driven methodology



- “All” knowledge base interaction via templates
- “API for RDF/OWL”
- Maintain “all” interaction with KB by maintaining templates

stOTTR: Easy to read and write

```
NamedPizza[ owl:Class  ?Name, ? owl:Individual  ?Country, List<owl:Class>  ?Toppings] :: {
        SubClassOf(?Name, :NamedPizza),
        SubObjectHasValue(?Name, :hasCountryOfOrigin, ?Country),
        SubObjectAllValuesFrom(?Name, :hasTopping, _:b1),
          ObjectUnionOf(_:b1, ?Toppings),
        cross |  SubObjectSomeValuesFrom(?Name, :hasTopping, ++?Toppings)
} .


NamedPizza(Margherita, Italy, (Tomato, Mozzarella)) .
NamedPizza(Grandiosa, ottr:none, (Tomato, Jarlsberg, Ham, SweetPepper)) .
```

wOTTR: OWL vocabulary for semantic web use

```
pizza:NamedPizza a ottr:Template ;

    ottr:parameters (
        [ ottr:type owl:Class; ottr:variable _:pizza ]
        [ ottr:type owl:NamedIndividual; ottr:variable _:country; ottr:modifier ottr:optional ]
        [ ottr:type ( rdf:List owl:Class ); ottr:variable _:toppings; ] ) ;

    ottr:pattern
        [ ottr:of ax:SubClassOf ; ottr:values ( _:pizza p:NamedPizza ) ] ,
        [ ottr:of ax:SubObjectHasValue ; ottr:values ( _:pizza p:hasCountryOfOrigin _:country ) ] ,
        [ ottr:of ax:SubObjectAllValuesFrom ; ottr:values ( _:pizza p:hasTopping _:alltoppings ) ] ,
        [ ottr:of rstr:ObjectUnionOf ; ottr:values ( _:alltoppings _:toppings ) ] ,
        [ ottr:of ax:SubObjectSomeValuesFrom ;  ottr:modifier ottr:cross ;
          ottr:arguments ( [ ottr:value _:pizza ]
                           [ ottr:value p:hasTopping ]
                           [ ottr:value _:toppings; ottr:modifier ottr:listExpand ] ) ] .
```

tabOTTR: tabular format for bulk template instances

9			
10	#OTTR	template	http://draft.ottr.xyz/pizza/NamedPizza
11	Pizza	Country	Toppings
12	1	2	3
13	iri	iri	iri+
14	p:Veneziana	p:Italy	p:OnionTopping | p:MozzarellaTopping | p:CaperTopping | p:OliveTopping | p:
15	p:AmericanH	p:America	p:JalapenoPepperTopping | p:MozzarellaTopping | p:HotGreenPepperTopping
16	p:Margherita		p:MozzarellaTopping | p:TomatoTopping
17	p:FourSeasons		p:TomatoTopping | p:PeperoniSausageTopping | p:CaperTopping | p:Mushroo
18	p:Fiorentina		p:TomatoTopping | p:GarlicTopping | p:ParmesanTopping | p:MozzarellaTopp
19	p:PrinceCarlo		p:LeekTopping | p:RosemaryTopping | p:MozzarellaTopping | p:TomatoToppin
20	p:LaReine		p:MushroomTopping | p:HamTopping | p:MozzarellaTopping | p:OliveTopping
21	p:American	p:America	p:TomatoTopping | p:MozzarellaTopping | p:PeperoniSausageTopping
22	p:Caprina		p:MozzarellaTopping | p:TomatoTopping | p:GoatsCheeseTopping | p:Sundrie
23	p:PolloAdAstra		p:TomatoTopping | p:RedOnionTopping | p:MozzarellaTopping | p:SweetPepp

OTTR templates

$$\texttt{Margherita} \sqsubseteq \texttt{NamedPizza}$$
$$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasCountryOfOrigin}.\{\texttt{Italy}\}$$
$$\texttt{Margherita} \sqsubseteq \forall\, \texttt{hasTopping}.(\texttt{Tomato} \sqcup \texttt{Mozzarella})$$
$$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Tomato}$$
$$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Mozzarella}$$

OTTR templates

$\texttt{Margherita} \sqsubseteq \texttt{NamedPizza}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasCountryOfOrigin}.\{\texttt{Italy}\}$
$\texttt{Margherita} \sqsubseteq \forall\, \texttt{hasTopping}.(\texttt{Tomato} \sqcup \texttt{Mozzarella})$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Tomato}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Mozzarella}$

```
NamedPizza[ owl:Class ?Name, ? owl:Individual ?Country, List<owl:Class> ?Toppings]
   :: {
        SubClassOf(?Name, :NamedPizza),
        SubObjectHasValue(?Name, :hasCountryOfOrigin, ?Country),
        SubObjectAllValuesFrom(?Name, :hasTopping, _:b1),
          ObjectUnionOf(_:b1, ?Toppings),
        cross | SubObjectSomeValuesFrom(?Name, :hasTopping, ++ ?Toppings)
} .
```

$?\mathit{Name} \sqsubseteq \texttt{:NamedPizza}$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasCO}.\{?\mathit{Country}\}$

$?\mathit{Name} \sqsubseteq \forall\, \texttt{hasT}.(\sqcup ?\mathit{Toppings})$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasT}.?\mathit{Toppings}\ldots$

OTTR templates

$\texttt{Margherita} \sqsubseteq \texttt{NamedPizza}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasCountryOfOrigin}.\{\texttt{Italy}\}$
$\texttt{Margherita} \sqsubseteq \forall\, \texttt{hasTopping}.(\texttt{Tomato} \sqcup \texttt{Mozzarella})$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Tomato}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Mozzarella}$

template name

```
NamedPizza[ owl:Class ?Name, ? owl:Individual ?Country, List<owl:Class> ?Toppings]
  :: {
        SubClassOf(?Name, :NamedPizza),
        SubObjectHasValue(?Name, :hasCountryOfOrigin, ?Country),
        SubObjectAllValuesFrom(?Name, :hasTopping, _:b1),
          ObjectUnionOf(_:b1, ?Toppings),
        cross | SubObjectSomeValuesFrom(?Name, :hasTopping, ++ ?Toppings)
} .
```

$?\mathit{Name} \sqsubseteq \texttt{:NamedPizza}$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasCO}.\{?\mathit{Country}\}$

$?\mathit{Name} \sqsubseteq \forall\, \texttt{hasT}.(\sqcup ?\mathit{Toppings})$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasT}.?\mathit{Toppings}\ldots$

OTTR templates

$\texttt{Margherita} \sqsubseteq \texttt{NamedPizza}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasCountryOfOrigin}.\{\texttt{Italy}\}$
$\texttt{Margherita} \sqsubseteq \forall\, \texttt{hasTopping}.(\texttt{Tomato} \sqcup \texttt{Mozzarella})$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Tomato}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Mozzarella}$

template name

NamedPizza[owl:Class ?Name, ? owl:Individual ?Country, List<owl:Class> ?Toppings]
:: {
SubClassOf(?Name, :NamedPizza),
SubObjectHasValue(?Name, :hasCountryOfOrigin, ?Country),
SubObjectAllValuesFrom(?Name, :hasTopping, _:b1),
ObjectUnionOf(_:b1, ?Toppings),
cross | SubObjectSomeValuesFrom(?Name, :hasTopping, ++ ?Toppings)
} .

$?\mathit{Name} \sqsubseteq \texttt{:NamedPizza}$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasCO}.\{?\mathit{Country}\}$

$?\mathit{Name} \sqsubseteq \forall\, \texttt{hasT}.(\sqcup ?\mathit{Toppings})$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasT}.?\mathit{Toppings}\ldots$

OTTR templates

$\texttt{Margherita} \sqsubseteq \texttt{NamedPizza}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasCountryOfOrigin}.\{\texttt{Italy}\}$
$\texttt{Margherita} \sqsubseteq \forall\, \texttt{hasTopping}.(\texttt{Tomato} \sqcup \texttt{Mozzarella})$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Tomato}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Mozzarella}$

template name · type · parameter name

```
NamedPizza[ owl:Class ?Name, ? owl:Individual ?Country, List<owl:Class> ?Toppings]
  :: {
        SubClassOf(?Name, :NamedPizza),
        SubObjectHasValue(?Name, :hasCountryOfOrigin, ?Country),
        SubObjectAllValuesFrom(?Name, :hasTopping, _:b1),
          ObjectUnionOf(_:b1, ?Toppings),
        cross | SubObjectSomeValuesFrom(?Name, :hasTopping, ++ ?Toppings)
} .
```

$?\mathit{Name} \sqsubseteq \texttt{:NamedPizza}$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasCO}.\{?\mathit{Country}\}$

$?\mathit{Name} \sqsubseteq \forall\, \texttt{hasT}.(\sqcup ?\mathit{Toppings})$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasT}.?\mathit{Toppings}\ldots$

OTTR templates

$\texttt{Margherita} \sqsubseteq \texttt{NamedPizza}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasCountryOfOrigin}.\{\texttt{Italy}\}$
$\texttt{Margherita} \sqsubseteq \forall\, \texttt{hasTopping}.(\texttt{Tomato} \sqcup \texttt{Mozzarella})$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Tomato}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Mozzarella}$

template name · type · parameter name · optional input

```
NamedPizza[ owl:Class ?Name, ? owl:Individual ?Country, List<owl:Class> ?Toppings]
  :: {
       SubClassOf(?Name, :NamedPizza),
       SubObjectHasValue(?Name, :hasCountryOfOrigin, ?Country),
       SubObjectAllValuesFrom(?Name, :hasTopping, _:b1),
         ObjectUnionOf(_:b1, ?Toppings),
       cross | SubObjectSomeValuesFrom(?Name, :hasTopping, ++ ?Toppings)
} .
```

$?\mathit{Name} \sqsubseteq \texttt{:NamedPizza}$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasCO}.\{?\mathit{Country}\}$

$?\mathit{Name} \sqsubseteq \forall\, \texttt{hasT}.(\sqcup ?\mathit{Toppings})$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasT}.?\mathit{Toppings}\ldots$

OTTR templates

$\texttt{Margherita} \sqsubseteq \texttt{NamedPizza}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasCountryOfOrigin}.\{\texttt{Italy}\}$
$\texttt{Margherita} \sqsubseteq \forall\, \texttt{hasTopping}.(\texttt{Tomato} \sqcup \texttt{Mozzarella})$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Tomato}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Mozzarella}$

template name · type · parameter name · optional input · list input

```
NamedPizza[ owl:Class ?Name, ? owl:Individual ?Country, List<owl:Class> ?Toppings]
  :: {
       SubClassOf(?Name, :NamedPizza),
       SubObjectHasValue(?Name, :hasCountryOfOrigin, ?Country),
       SubObjectAllValuesFrom(?Name, :hasTopping, _:b1),
         ObjectUnionOf(_:b1, ?Toppings),
       cross | SubObjectSomeValuesFrom(?Name, :hasTopping, ++ ?Toppings)
} .
```

$?\mathit{Name} \sqsubseteq \texttt{:NamedPizza}$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasCO}.\{?\mathit{Country}\}$

$?\mathit{Name} \sqsubseteq \forall\, \texttt{hasT}.(\sqcup ?\mathit{Toppings})$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasT}.?\mathit{Toppings}\ldots$

OTTR templates

$\texttt{Margherita} \sqsubseteq \texttt{NamedPizza}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasCountryOfOrigin}.\{\texttt{Italy}\}$
$\texttt{Margherita} \sqsubseteq \forall\, \texttt{hasTopping}.(\texttt{Tomato} \sqcup \texttt{Mozzarella})$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Tomato}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Mozzarella}$

template name · type · parameter name · optional input · list input

```
NamedPizza[ owl:Class ?Name, ? owl:Individual ?Country, List<owl:Class> ?Toppings]
  :: {
       SubClassOf(?Name, :NamedPizza),
       SubObjectHasValue(?Name, :hasCountryOfOrigin, ?Country),
       SubObjectAllValuesFrom(?Name, :hasTopping, _:b1),
         ObjectUnionOf(_:b1, ?Toppings),
       cross | SubObjectSomeValuesFrom(?Name, :hasTopping, ++ ?Toppings)
  } .
```

list expansion

$?\mathit{Name} \sqsubseteq \texttt{:NamedPizza}$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasCO}.\{?\mathit{Country}\}$

$?\mathit{Name} \sqsubseteq \forall\, \texttt{hasT}.(\sqcup ?\mathit{Toppings})$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasT}.?\mathit{Toppings}\ldots$

OTTR templates

$\texttt{Margherita} \sqsubseteq \texttt{NamedPizza}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasCountryOfOrigin}.\{\texttt{Italy}\}$
$\texttt{Margherita} \sqsubseteq \forall\, \texttt{hasTopping}.(\texttt{Tomato} \sqcup \texttt{Mozzarella})$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Tomato}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Mozzarella}$

template name · type · parameter name · optional input · list input

```
NamedPizza[ owl:Class ?Name, ? owl:Individual ?Country, List<owl:Class> ?Toppings]
  :: {
        SubClassOf(?Name, :NamedPizza),
        SubObjectHasValue(?Name, :hasCountryOfOrigin, ?Country),
        SubObjectAllValuesFrom(?Name, :hasTopping, _:b1),
        ObjectUnionOf(_:b1, ?Toppings),
        cross | SubObjectSomeValuesFrom(?Name, :hasTopping, ++ ?Toppings)
  } .
```

list expansion · marked for expansion

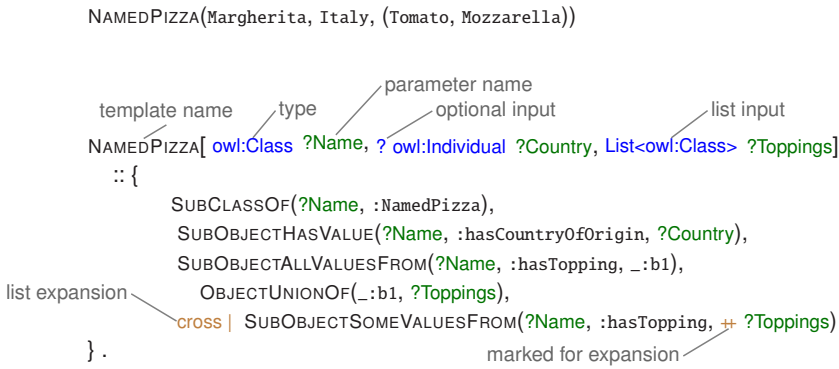
$?\mathit{Name} \sqsubseteq \texttt{:NamedPizza}$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasCO}.\{?\mathit{Country}\}$

$?\mathit{Name} \sqsubseteq \forall\, \texttt{hasT}.(\sqcup ?\mathit{Toppings})$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasT}.?\mathit{Toppings}\ldots$

OTTR templates

$\texttt{Margherita} \sqsubseteq \texttt{NamedPizza}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasCountryOfOrigin}.\{\texttt{Italy}\}$
$\texttt{Margherita} \sqsubseteq \forall\, \texttt{hasTopping}.(\texttt{Tomato} \sqcup \texttt{Mozzarella})$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Tomato}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Mozzarella}$

NAMEDPIZZA(Margherita, Italy, (Tomato, Mozzarella))



$?\mathit{Name} \sqsubseteq \texttt{:NamedPizza}$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasCO}.\{?\mathit{Country}\}$
$?\mathit{Name} \sqsubseteq \forall\, \texttt{hasT}.(\sqcup ?\mathit{Toppings})$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasT}.?\mathit{Toppings}\ldots$

OTTR templates

$\texttt{Margherita} \sqsubseteq \texttt{NamedPizza}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasCountryOfOrigin}.\{\texttt{Italy}\}$
$\texttt{Margherita} \sqsubseteq \forall\, \texttt{hasTopping}.(\texttt{Tomato} \sqcup \texttt{Mozzarella})$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Tomato}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Mozzarella}$

NamedPizza(Margherita, Italy, (Tomato, Mozzarella))
NamedPizza(Grandiosa, ottr:none, (Tomato, Jarlsberg, Ham, SweetPepper))

template name · type · parameter name · optional input · list input

```
NamedPizza[ owl:Class ?Name, ? owl:Individual ?Country, List<owl:Class> ?Toppings]
  :: {
        SubClassOf(?Name, :NamedPizza),
        SubObjectHasValue(?Name, :hasCountryOfOrigin, ?Country),
        SubObjectAllValuesFrom(?Name, :hasTopping, _:b1),
          ObjectUnionOf(_:b1, ?Toppings),
cross |   SubObjectSomeValuesFrom(?Name, :hasTopping, ++ ?Toppings)
  } .
```

list expansion · marked for expansion

$?\mathit{Name} \sqsubseteq \texttt{:NamedPizza}$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasCO}.\{?\mathit{Country}\}$

$?\mathit{Name} \sqsubseteq \forall\, \texttt{hasT}.(\sqcup ?\mathit{Toppings})$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasT}.?\mathit{Toppings}\ldots$

OTTR templates

$\texttt{Margherita} \sqsubseteq \texttt{NamedPizza}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasCountryOfOrigin}.\{\texttt{Italy}\}$
$\texttt{Margherita} \sqsubseteq \forall\, \texttt{hasTopping}.(\texttt{Tomato} \sqcup \texttt{Mozzarella})$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Tomato}$
$\texttt{Margherita} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Mozzarella}$

$\texttt{Grandiosa} \sqsubseteq \texttt{NamedPizza}$
$\texttt{Grandiosa} \sqsubseteq \forall\, \texttt{hasTopping}.(\texttt{Tomato} \sqcup \texttt{Jarlsberg} \sqcup \texttt{Ham} \sqcup \texttt{SweetPepper})$
$\texttt{Grandiosa} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Tomato}$
$\texttt{Grandiosa} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Jarlsberg}$
$\texttt{Grandiosa} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{Ham}$
$\texttt{Grandiosa} \sqsubseteq \exists\, \texttt{hasTopping}.\texttt{SweetPepper}$

```
NamedPizza(Margherita, Italy, (Tomato, Mozzarella))
NamedPizza(Grandiosa, ottr:none, (Tomato, Jarlsberg, Ham, SweetPepper))

NamedPizza[ owl:Class ?Name, ? owl:Individual ?Country, List<owl:Class> ?Toppings]
  :: {
       SubClassOf(?Name, :NamedPizza),
       SubObjectHasValue(?Name, :hasCountryOfOrigin, ?Country),
       SubObjectAllValuesFrom(?Name, :hasTopping, _:b1),
         ObjectUnionOf(_:b1, ?Toppings),
cross |  SubObjectSomeValuesFrom(?Name, :hasTopping, ++ ?Toppings)
  } .
```

template name · type · parameter name · optional input · list input

list expansion

marked for expansion

$?\mathit{Name} \sqsubseteq \texttt{:NamedPizza}$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasCO}.\{?\mathit{Country}\}$

$?\mathit{Name} \sqsubseteq \forall\, \texttt{hasT}.(\sqcup ?\mathit{Toppings})$
$?\mathit{Name} \sqsubseteq \exists\, \texttt{hasT}.?\mathit{Toppings} \ldots$

Template signatures

- A *template signature* gives the *interface* of a template, and contains just its name and parameter declarations, e.g.

  NAMEDPIZZA[owl:Class ?Name, ? owl:NamedIndividual ?Country, List<owl:Class> ?Toppings] .

Template signatures

- A *template signature* gives the *interface* of a template, and contains just its name and parameter declarations, e.g.

  NAMEDPIZZA[owl:Class ?Name, ? owl:NamedIndividual ?Country, List<owl:Class> ?Toppings] .

- Giving just a signature, instead of the full definition, is convenient when:
  - you have not yet defined the template
  - to ensure correct usage of a template, but the definition is unavailable
  - for examples or documentation

Base templates

- A *base template* is a template without definition

Base templates

- A *base template* is a template without definition
- Currently, the only base template is

```
TRIPLE[ottt:IRI ?subject, ! ottt:IRI ?predicate, rdfs:Resource ?object] :: BASE .
```

Base templates

- A *base template* is a template without definition
- Currently, the only base template is

$$\textsc{Triple}\big[\texttt{ottt:IRI ?subject}, \texttt{! ottt:IRI ?predicate}, \texttt{rdfs:Resource ?object}\big] \;::\; \texttt{BASE} \;.$$

- A TRIPLE-instance represents an RDF triple, e.g.:

$$\textsc{Triple}\big(\texttt{ex:martin}, \texttt{ex:knows}, \texttt{ex:peter}\big) \quad \rightsquigarrow \quad \texttt{ex:martin ex:knows ex:peter} \;.$$

Base templates

- A *base template* is a template without definition
- Currently, the only base template is

$$\textsc{Triple}[\texttt{ottt:IRI ?subject}, \texttt{! ottt:IRI ?predicate}, \texttt{rdfs:Resource ?object}] \ :: \ \texttt{BASE} \ .$$

- A TRIPLE-instance represents an RDF triple, e.g.:

$$\textsc{Triple}(\texttt{ex:martin}, \texttt{ex:knows}, \texttt{ex:peter}) \quad \rightsquigarrow \quad \texttt{ex:martin ex:knows ex:peter} \ .$$

- Translation done by implementation

Base templates

- A *base template* is a template without definition
- Currently, the only base template is

$$\textsc{Triple}\big[\texttt{ottt:IRI ?subject}, \texttt{! ottt:IRI ?predicate}, \texttt{rdfs:Resource ?object}\big] \;::\; \texttt{BASE} \;.$$

- A TRIPLE-instance represents an RDF triple, e.g.:

$$\textsc{Triple}\big(\texttt{ex:martin}, \texttt{ex:knows}, \texttt{ex:peter}\big) \quad \rightsquigarrow \quad \texttt{ex:martin ex:knows ex:peter} \;.$$

- Translation done by implementation
- Other base templates possible: E.g. QUAD

Base templates

- A *base template* is a template without definition
- Currently, the only base template is

$$\textsc{Triple}\big[\texttt{ottt:IRI ?subject}, \texttt{! ottt:IRI ?predicate}, \texttt{rdfs:Resource ?object}\big] \;::\; \texttt{BASE} .$$

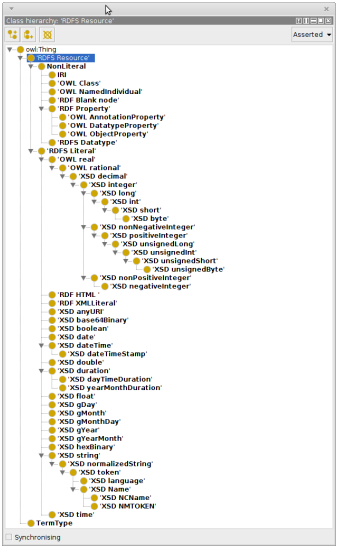
- A TRIPLE-instance represents an RDF triple, e.g.:

$$\textsc{Triple}\big(\texttt{ex:martin}, \texttt{ex:knows}, \texttt{ex:peter}\big) \quad \rightsquigarrow \quad \texttt{ex:martin ex:knows ex:peter .}$$

- Translation done by implementation
- Other base templates possible: E.g. QUAD
- Note: “Templates all the way down”

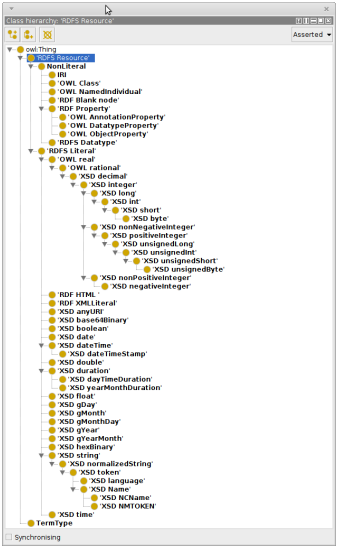
Types

- Specify permissible arguments and consistent use of resources



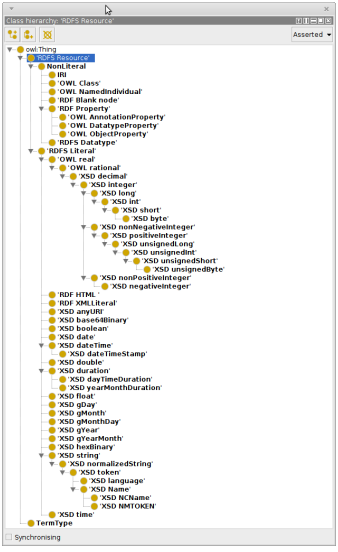
Types

- Specify permissible arguments and consistent use of resources
- Base-types limited to classes and datatypes defined in XSD, RDF, RDFS, and OWL
  - `xsd:int`, `rdf:Property`, `rdfs:Resource`, ...
  - `rdfs:Resource` is default and most general



Types

- Specify permissible arguments and consistent use of resources
- Base-types limited to classes and datatypes defined in XSD, RDF, RDFS, and OWL
  - `xsd:int`, `rdf:Property`, `rdfs:Resource`, ...
  - `rdfs:Resource` is default and most general
- Also supports lists and non-empty lists (e.g. NEList⟨List⟨xsd:string⟩⟩)



Types

- Similar to types in programming languages (e.g. Java)

[1] https://dev.spec.ottr.xyz/rOTTR/develop/types.owl.ttl
[2] https://dev.spec.ottr.xyz/rOTTR/develop/puntypes.owl.ttl

Types

- Similar to types in programming languages (e.g. Java)
- But adapted to ontologies

[1] https://dev.spec.ottr.xyz/rOTTR/develop/types.owl.ttl
[2] https://dev.spec.ottr.xyz/rOTTR/develop/puntypes.owl.ttl

Types

- Similar to types in programming languages (e.g. Java)
- But adapted to ontologies
- Can e.g. allow[1] or disallow[2] punning, depending on chosen hierarchy

[1] https://dev.spec.ottr.xyz/rOTTR/develop/types.owl.ttl
[2] https://dev.spec.ottr.xyz/rOTTR/develop/puntypes.owl.ttl

- Type checking instances

  SubClassOf[owl:Class ?sub, owl:Class ?super] .

  SubClassOf(p:Margherita, p:Pizza) .

  SubClassOf("Margherita", p:Pizza) .

- Type checking instances

SubClassOf[owl:Class ?sub, owl:Class ?super] .

SubClassOf(p:Margherita, p:Pizza) . OK

SubClassOf("Margherita", p:Pizza) . not OK

- Type checking instances

```
SubClassOf[owl:Class ?sub, owl:Class ?super] .

SubClassOf(p:Margherita, p:Pizza) .      OK
SubClassOf("Margherita", p:Pizza) .      not OK
```

- Type checking template definitions

```
Pizza[owl:Class ?name, xsd:string ?label] :: {
    Triple(?name, rdf:type, owl:Class),
    SubClassOf(?label, p:Pizza),
    Triple(?name, rdfs:label, ?label)
} .
```

- Type checking instances

  SubClassOf[owl:Class ?sub, owl:Class ?super] .

  SubClassOf(p:Margherita, p:Pizza) . OK

  SubClassOf("Margherita", p:Pizza) . not OK

- Type checking template definitions

  Pizza[owl:Class ?name, xsd:string ?label] :: {
  Triple(?name, rdf:type, owl:Class),
  SubClassOf(?label, p:Pizza), not OK
  Triple(?name, rdfs:label, ?label)
  } .

Types – Consistent use

Examples

INSTANCEOF[owl:Individual ?elem, owl:Class ?class] .

INSTANCEOF(_:blank, p:Pizza)

INSTANCEOF(p:mypizza, _:blank)

Types – Consistent use

Examples

INSTANCEOF[owl:Individual ?elem, owl:Class ?class] .

INSTANCEOF(_:blank, p:Pizza)

INSTANCEOF(p:mypizza, _:blank) OK? (depending on type hierarchy)

Types – Consistent use

Examples

INSTANCEOF[owl:Individual ?elem, owl:Class ?class] .

INSTANCEOF(_:blank, p:Pizza)

INSTANCEOF(p:mypizza, _:blank) OK? (depending on type hierarchy)

LABEL[ottr:IRI ?elem, xsd:string ?label] .

LABEL(p:mypizza, "A pizza")

Types – Consistent use

Examples

INSTANCEOF[owl:Individual ?elem, owl:Class ?class] .

INSTANCEOF(_:blank, p:Pizza)

INSTANCEOF(p:mypizza, _:blank) OK? (depending on type hierarchy)

LABEL[ottr:IRI ?elem, xsd:string ?label] .

LABEL(p:mypizza, "A pizza") OK

Types – Consistent use

Examples

InstanceOf[owl:Individual ?elem, owl:Class ?class] .

InstanceOf(_:blank, p:Pizza)

InstanceOf(p:mypizza, _:blank) OK? (depending on type hierarchy)

Label[ottr:IRI ?elem, xsd:string ?label] .

Label(p:mypizza, "A pizza") OK

InstanceOf(_:blank2, p:Pizza)

Label(p:mypizza, _:blank2)

Types – Consistent use

Examples

INSTANCEOF[owl:Individual ?elem, owl:Class ?class] .

INSTANCEOF(_:blank, p:Pizza)

INSTANCEOF(p:mypizza, _:blank) OK? (depending on type hierarchy)

LABEL[ottr:IRI ?elem, xsd:string ?label] .

LABEL(p:mypizza, "A pizza") OK

INSTANCEOF(_:blank2, p:Pizza)

LABEL(p:mypizza, _:blank2) not OK

Types – Motivation

- The type system ensures that:
  - Templates are used correctly

Types – Motivation

- The type system ensures that:
  - Templates are used correctly
  - Resources are used consistently

Types – Motivation

- The type system ensures that:
  - Templates are used correctly
  - Resources are used consistently
  - Any set of valid instances expands into a valid RDF-graph (e.g. no literal in subject position)

Types – Motivation

- The type system ensures that:
  - Templates are used correctly
  - Resources are used consistently
  - Any set of valid instances expands into a valid RDF-graph (e.g. no literal in subject position)
  - With our templates for OWL-axioms, any set of valid instances expands to a proper OWL-ontology

Types – Motivation

- The type system ensures that:
  - Templates are used correctly
  - Resources are used consistently
  - Any set of valid instances expands into a valid RDF-graph (e.g. no literal in subject position)
  - With our templates for OWL-axioms, any set of valid instances expands to a proper OWL-ontology
- These invariants also hold for more abstract templates (e.g. domain specific and user-facing templates)

Types – Motivation

- The type system ensures that:
  - Templates are used correctly
  - Resources are used consistently
  - Any set of valid instances expands into a valid RDF-graph (e.g. no literal in subject position)
  - With our templates for OWL-axioms, any set of valid instances expands to a proper OWL-ontology
- These invariants also hold for more abstract templates (e.g. domain specific and user-facing templates)
- Helps in highlighting the template's intention

Non-blank

- Non-blank (written !) – requires a node that is not blank as argument

Non-blank

- Non-blank (written !) – requires a node that is not blank as argument
- Parameters used as arguments to non-blanks must also be non-blank.

Non-blank

- Non-blank (written !) – requires a node that is not blank as argument
- Parameters used as arguments to non-blanks must also be non-blank.

TRIPLE[ottt:IRI ?subject, ! ottt:IRI ?predicate, rdfs:Resource ?object] :: BASE .

TRIPLE(_:blank, :hasLength, 5)

TRIPLE(p:mypizza, _:blank2, 5)

Non-blank

- Non-blank (written !) – requires a node that is not blank as argument
- Parameters used as arguments to non-blanks must also be non-blank.

TRIPLE[ottt:IRI ?subject, ! ottt:IRI ?predicate, rdfs:Resource ?object] :: BASE .

TRIPLE(_:blank, :hasLength, 5) OK

TRIPLE(p:mypizza, _:blank2, 5) not OK

Non-blank

- Non-blank (written !) – requires a node that is not blank as argument
- Parameters used as arguments to non-blanks must also be non-blank.

```
TRIPLE[ottt:IRI ?subject, ! ottt:IRI ?predicate, rdfs:Resource ?object] :: BASE .

TRIPLE(_:blank, :hasLength, 5)        OK
TRIPLE(p:mypizza, _:blank2, 5)        not OK


RELATEDTOALL[ottr:IRI ?subs,  rdf:Property ?relation, rdf:List<ottr:IRI> ?objs] :: {
   cross | TRIPLE(?sub, ?relation, ++?objs)
} .
```

Non-blank

- Non-blank (written !) – requires a node that is not blank as argument
- Parameters used as arguments to non-blanks must also be non-blank.

```
Triple[ottt:IRI ?subject, ! ottt:IRI ?predicate, rdfs:Resource ?object] :: BASE .

Triple(_:blank, :hasLength, 5)        OK
Triple(p:mypizza, _:blank2, 5)        not OK

RelatedToAll[ottr:IRI ?subs,  rdf:Property ?relation, rdf:List<ottr:IRI> ?objs] :: {   not OK
   cross | Triple(?sub, ?relation, ++?objs)
} .
```

Non-blank

- Non-blank (written !) – requires a node that is not blank as argument
- Parameters used as arguments to non-blanks must also be non-blank.

```
TRIPLE[ottt:IRI ?subject, ! ottt:IRI ?predicate, rdfs:Resource ?object] :: BASE .

TRIPLE(_:blank, :hasLength, 5)        OK
TRIPLE(p:mypizza, _:blank2, 5)        not OK


RELATEDTOALL[ottr:IRI ?subs, ! rdf:Property ?relation, rdf:List<ottr:IRI> ?objs] :: {    OK
   cross | TRIPLE(?sub, ?relation, ++?objs)
} .
```

Non-blank

- Non-blank (written !) – requires a node that is not blank as argument
- Parameters used as arguments to non-blanks must also be non-blank.

```
TRIPLE[ottt:IRI ?subject, ! ottt:IRI ?predicate, rdfs:Resource ?object] :: BASE .

TRIPLE(_:blank, :hasLength, 5)        OK
TRIPLE(p:mypizza, _:blank2, 5)        not OK

RELATEDTOALL[ottr:IRI ?subs, ! rdf:Property ?relation, rdf:List<ottr:IRI> ?objs] :: {    OK
   cross | TRIPLE(?sub, ?relation, ++?objs)
} .
```

- With the non-blank flag, a user can be certain that no RDF-predicate is blank

Non-blank

- Non-blank (written !) – requires a node that is not blank as argument
- Parameters used as arguments to non-blanks must also be non-blank.

```
TRIPLE[ottt:IRI ?subject, ! ottt:IRI ?predicate, rdfs:Resource ?object] :: BASE .

TRIPLE(_:blank, :hasLength, 5)        OK
TRIPLE(p:mypizza, _:blank2, 5)        not OK

RELATEDTOALL[ottr:IRI ?subs, ! rdf:Property ?relation, rdf:List<ottr:IRI> ?objs] :: {   OK
   cross | TRIPLE(?sub, ?relation, ++?objs)
} .
```

- With the non-blank flag, a user can be certain that no RDF-predicate is blank
- Can also force IRIs of entities

Expansion modes

- Instances with lists as arguments can be used together with an *expansion mode*

Expansion modes

- Instances with lists as arguments can be used together with an *expansion mode*
- If a mode is present, each argument to an instance is treated as a list (non-list element are treated as lists of one element).

Expansion modes

- Instances with lists as arguments can be used together with an *expansion mode*
- If a mode is present, each argument to an instance is treated as a list (non-list element are treated as lists of one element).
- We have three modes:
  - Cross (written `cross`), the template is applied to the *cross product* of the argument lists
  - ZipMax (`zipMax`) the template is applied to the *zip max* of the argument lists
  - ZipMin (`zipMin`) the template is applied to the *zip min* of the argument lists

Expansion modes

Example

```
SubClassOf[ owl:Class ?sub, List<owl:Class> ?super] :: {
    cross | Triple(?sub, rdfs:subClassOf, ++?super)
} .
```

```
SubClassOf[ owl:Class ?sub, List<owl:Class> ?super] :: {
    cross | Triple(?sub, rdfs:subClassOf, ++?super)
} .

SubClassOf(A, (X, Y, Z)) .
```

Expansion modes

Example

SubClassOf[owl:Class ?sub, List<owl:Class> ?super] :: {

cross | Triple(?sub, rdfs:subClassOf, ++?super)

} .

SubClassOf(A, (X, Y, Z)) .

Triple(A, rdfs:subClassOf, X)

Triple(A, rdfs:subClassOf, Y)

Triple(A, rdfs:subClassOf, Z)

```
SubClassOf[List< owl:Class> ?sub, List<owl:Class> ?super] :: {
    cross | Triple(++?sub, rdfs:subClassOf, ++?super)
} .

SubClassOf((A, B, C), (X, Y, Z)) .
```

Expansion modes

Example

SubClassOf[List< owl:Class> ?sub, List<owl:Class> ?super] :: {

cross | Triple(++?sub, rdfs:subClassOf, ++?super)

} .

SubClassOf((A, B, C), (X, Y, Z)) .

Triple(A, rdfs:subClassOf, X) Triple(B, rdfs:subClassOf, X) Triple(C, rdfs:subClassOf, X)
Triple(A, rdfs:subClassOf, Y) Triple(B, rdfs:subClassOf, Y) Triple(C, rdfs:subClassOf, Y)
Triple(A, rdfs:subClassOf, Z) Triple(B, rdfs:subClassOf, Z) Triple(C, rdfs:subClassOf, Z)

Expansion modes

Example

```
SubClassOf[List< owl:Class> ?sub, List<owl:Class> ?super] :: {
    zipMin | Triple(++?sub, rdfs:subClassOf, ++?super)
} .

SubClassOf((A, B, C), (X, Y, Z)) .
```

```
SubClassOf[List< owl:Class> ?sub, List<owl:Class> ?super] :: {
    zipMin | Triple(++?sub, rdfs:subClassOf, ++?super)
} .

SubClassOf((A, B, C), (X, Y, Z)) .

Triple(A, rdfs:subClassOf, X)
                    Triple(B, rdfs:subClassOf, Y)
                                        Triple(C, rdfs:subClassOf, Z)
```

Expansion modes

Example

```
SubClassOf[List< owl:Class> ?sub, List<owl:Class> ?super] :: {
    zipMin | Triple(++?sub, rdfs:subClassOf, ++?super)
} .

SubClassOf((A, B), (X, Y, Z)) .
```

```
SubClassOf[List< owl:Class> ?sub, List<owl:Class> ?super] :: {
    zipMin | Triple(++?sub, rdfs:subClassOf, ++?super)
} .

SubClassOf((A, B), (X, Y, Z)) .

Triple(A, rdfs:subClassOf, X)
                              Triple(B, rdfs:subClassOf, Y)
```

Expansion modes

Example

```
SubClassOf[List< owl:Class> ?sub, List<owl:Class> ?super] :: {
    zipMax | Triple(++?sub, rdfs:subClassOf, ++?super)
} .

SubClassOf((A, B), (X, Y, Z)) .
```

Expansion modes

Example

```
SubClassOf[List< owl:Class> ?sub, List<owl:Class> ?super] :: {
    zipMax | Triple(++?sub, rdfs:subClassOf, ++?super)
} .

SubClassOf((A, B), (X, Y, Z)) .

Triple(A, rdfs:subClassOf, X)
                    Triple(B, rdfs:subClassOf, Y)
                                        Triple(none, rdfs:subClassOf, Z)
```

Expansion modes – Motivation

- Removes repetition, e.g. instead of

$$\textsc{SubClassOf}(\mathtt{A}, \mathtt{B}), \textsc{SubClassOf}(\mathtt{A}, \mathtt{C}), \textsc{SubClassOf}(\mathtt{A}, \mathtt{D})$$

Expansion modes – Motivation

- Removes repetition, e.g. instead of

  $$\textsc{SubClassOf}(\mathtt{A}, \mathtt{B}), \textsc{SubClassOf}(\mathtt{A}, \mathtt{C}), \textsc{SubClassOf}(\mathtt{A}, \mathtt{D})$$

  we can simply write

  $$\text{cross} \mid \textsc{SubClassOf}(\mathtt{A}, +\!\!+(\mathtt{B}, \mathtt{C}, \mathtt{D}))$$

- DRY Principle (“Do not repeat yourself”) – Easier to maintain

Expansion modes – Motivation

- Removes repetition, e.g. instead of

  $$\textsc{SubClassOf}(\mathtt{A}, \mathtt{B}), \textsc{SubClassOf}(\mathtt{A}, \mathtt{C}), \textsc{SubClassOf}(\mathtt{A}, \mathtt{D})$$

  we can simply write

  $$\text{cross} \mid \textsc{SubClassOf}(\mathtt{A}, +\!\!+(\mathtt{B}, \mathtt{C}, \mathtt{D}))$$

- DRY Principle (“Do not repeat yourself”) – Easier to maintain
- Allows a template to take arbitrary number of arguments

Expansion modes – Motivation

- Removes repetition, e.g. instead of

  $$\textsc{SubClassOf}(\mathtt{A}, \mathtt{B}), \textsc{SubClassOf}(\mathtt{A}, \mathtt{C}), \textsc{SubClassOf}(\mathtt{A}, \mathtt{D})$$

  we can simply write

  $$\mathrm{cross} \mid \textsc{SubClassOf}(\mathtt{A}, +\!\!+(\mathtt{B}, \mathtt{C}, \mathtt{D}))$$

- DRY Principle (“Do not repeat yourself”) – Easier to maintain
- Allows a template to take arbitrary number of arguments
- Can use both the elements of a list and the list itself as arguments

Modifiers – Optional

- Optional (?) – permits expansion even if argument is a missing value
- Missing values are denoted by `ottr:none` (and empty cells in tabOTTR)
- Instances with missing values passed to parameter without ? are discarded

Optional
Example

$\textsc{Pizza}[\text{class ?name, ? xsd:string ?label}]$.

$\textsc{Pizza}(\texttt{p:Margherita}, \texttt{"Margherita"@it})$.

$\textsc{Pizza}(\texttt{p:WheatEver}, \texttt{none})$.

Pizza[class ?name, ? xsd:string ?label] .

Pizza(p:Margherita, "Margherita"@it) . OK

Pizza(p:WheatEver, none) . OK

PIZZA[class ?name, ? xsd:string ?label] .

PIZZA(p:Margherita, "Margherita"@it) . OK

PIZZA(p:WheatEver, none) . OK

PIZZA(p:WheatEver, none) .

$\textsc{Pizza}[\text{class ?name, ? xsd:string ?label}]$.

$\textsc{Pizza}(\texttt{p:Margherita}, \texttt{"Margherita"@it})$. OK

$\textsc{Pizza}(\texttt{p:WheatEver}, \texttt{none})$. OK

$\textsc{Pizza}(\texttt{p:WheatEver}, \texttt{none})$.

$\downarrow$

$\textsc{Triple}(\texttt{p:WheatEver}, \texttt{rdf:type}, \texttt{owl:Class})$

$\textsc{Triple}(\texttt{p:WheatEver}, \texttt{rdfs:subClassOf}, \texttt{p:Pizza})$

$\textsc{Triple}(\texttt{p:WheatEver}, \texttt{rdfs:label}, \texttt{none})$

Optional
Example

$\textsc{Pizza}[\text{class ?name}, \text{? xsd:string ?label}]$.

$\textsc{Pizza}(\texttt{p:Margherita}, \texttt{"Margherita"@it})$. OK

$\textsc{Pizza}(\texttt{p:WheatEver}, \texttt{none})$. OK

$\textsc{Pizza}(\texttt{p:WheatEver}, \texttt{none})$.

$\downarrow$

$\textsc{Triple}(\texttt{p:WheatEver}, \texttt{rdf:type}, \texttt{owl:Class})$

$\textsc{Triple}(\texttt{p:WheatEver}, \texttt{rdfs:subClassOf}, \texttt{p:Pizza})$

$\textsc{Triple}(\texttt{p:WheatEver}, \texttt{rdfs:label}, \texttt{none})$

Pizza[class ?name, ? xsd:string ?label] .

Pizza(p:Margherita, "Margherita"@it) . OK

Pizza(p:WheatEver, none) . OK

Pizza(none, "No pizza"@en) .

Pizza[class ?name, ? xsd:string ?label] .

Pizza(p:Margherita, "Margherita"@it) . OK

Pizza(p:WheatEver, none) . OK

Pizza(none, "No pizza"@en) . OK

Pizza[class ?name, ? xsd:string ?label] .

Pizza(p:Margherita, "Margherita"@it) . OK

Pizza(p:WheatEver, none) . OK

Pizza(none, "No pizza"@en) . OK

Default Values

- We also allow default values for parameters

Default Values

- We also allow default values for parameters
- Used if the argument to the parameter is `none`

Default Values

- We also allow default values for parameters
- Used if the argument to the parameter is `none`
- For example:

SubObjectSomeValuesFrom[owl:Class ?class, owl:ObjectProperty ?prop, owl:Class ?range = owl:Thing] :: {
    SubClassOf(?class, _:b),
    ObjectSomeValuesFrom(_:b, ?prop, ?range)
} .

(i.e. ?class $\sqsubseteq \exists$?prop . ?range)

Default Values

- We also allow default values for parameters
- Used if the argument to the parameter is `none`
- For example:

SubObjectSomeValuesFrom[owl:Class ?class, owl:ObjectProperty ?prop, owl:Class ?range = `owl:Thing`] :: {
  SubClassOf(?class, _:b),
  ObjectSomeValuesFrom(_:b, ?prop, ?range)
} . (i.e. ?class $\sqsubseteq \exists$?prop . ?range)

SubObjectSomeValuesFrom(`:Cat`, `:hasMother`, `:Cat`)
$\rightsquigarrow$ {SubClassOf(`:Cat`, _:c), ObjectSomeValuesFrom(_:c, `:hasMother`, `:Cat`)} (i.e. $Cat \sqsubseteq \exists hasMother.Cat$)

SubObjectSomeValuesFrom(`:Container`, `:contains`, `none`)
$\rightsquigarrow$ {SubClassOf(`:Container`, _:d), ObjectSomeValuesFrom(_:d, `:contains`, `owl:Thing`)} (i.e. $Container \sqsubseteq \exists contains.\top$)

OTTR Overview

- In addition to the core pattern mechanism, OTTR supports:
  - Signatures and base templates
  - Types and non-blank flag, and checks on these
  - Expansion modes and list support
  - Default and optional values
- The features are based on well-known principles in programming language design
- Provides
  - a simple, yet powerful language for expressing ontology patterns
  - that ensures correctness, easy maintenance, and uniform modelling
  - across all levels of abstraction

wOTTR – Quick overview

HasComponent[owl:Class ?whole, owl:Class ?component] :: {
  SubObjectSomeValuesFrom(?whole, ex:hasComponent, ?component)
} .

wOTTR – Quick overview

```
HasComponent[
   owl:Class ?whole,
   owl:Class ?component
] :: {
   SubObjectSomeValuesFrom
   (?whole, ex:hasComponent, ?component)
} .
```

wOTTR – Quick overview

HASCOMPONENT[
  owl:Class ?whole,
  owl:Class ?component
] :: {
  SUBOBJECTSOMEVALUESFROM
  (?whole, ex:hasComponent, ?component)
} .

```
:HasComponent a ottr:Template ;
  ottr:parameters (
    [ ottr:type owl:Class; ottr:variable :whole ]
    [ ottr:type owl:Class; ottr:variable :component ]
  ) ;
  ottr:pattern
    [ ottr:of ax:SubObjectSomeValuesFrom ;
      ottr:values (:whole ex:hasComponent :component ) ] .
```

wOTTR – Quick overview

```
HasComponent[
  owl:Class ?whole,
  owl:Class ?component
] :: {
  SubObjectSomeValuesFrom
  (?whole, ex:hasComponent, ?component)
} .
```

```
HasComponents[
  owl:Class ?whole,
  NEList<owl:Class> ?components
] :: {
  cross | HasComponent(
        ?whole,
        ++?components)
} .
```

```
:HasComponent a ottr:Template ;
  ottr:parameters (
    [ ottr:type owl:Class; ottr:variable :whole ]
    [ ottr:type owl:Class; ottr:variable :component ]
  ) ;
  ottr:pattern
    [ ottr:of ax:SubObjectSomeValuesFrom ;
      ottr:values (:whole ex:hasComponent :component ) ] .
```

wOTTR – Quick overview

```
HasComponent[
  owl:Class ?whole,
  owl:Class ?component
] :: {
  SubObjectSomeValuesFrom
  (?whole, ex:hasComponent, ?component)
} .
```

```
:HasComponent a ottr:Template ;
  ottr:parameters (
    [ ottr:type owl:Class; ottr:variable :whole ]
    [ ottr:type owl:Class; ottr:variable :component ]
  ) ;
  ottr:pattern
    [ ottr:of ax:SubObjectSomeValuesFrom ;
      ottr:values (:whole ex:hasComponent :component ) ] .
```

```
HasComponents[
  owl:Class ?whole,
  NEList<owl:Class> ?components
] :: {
  cross | HasComponent(
        ?whole,
        ++?components)
} .
```

```
:HasComponents a ottr:Template ;
  ottr:parameters (
    [ ottr:type owl:Class; ottr:variable :whole ]
    [ ottr:type (ottt:NEList owl:Class); ottr:variable :components ]
  ) ;
  ottr:pattern
    [ ottr:modifier ottr:cross; ottr:of :HasComponent ;
      ottr:arguments (
        [ ottr:value :whole ]
        [ ottr:modifier ottr:listExpand; ottr:value :components ]) ] .
```

wOTTR Vocabulary: Templates

In wOTTR[3], templates

- are resources typed with `ottr:Template`
- is `ottr:parameters`-related to an RDF list of resources representing its parameters
- is `ottr:pattern`-related to a set of resources representing its body instances

Each parameter

- is a resource (often a blank node)
- has a type, related to it via `ottr:type`
- has a variable name, related to it via `ottr:variable`

[3] See `https://spec.ottr.xyz/wOTTR/0.4/` for the full spec.

wOTTR Vocabulary: Instances

Each instance

- is a resource (often a blank node)
- is `ottr:of`-related to the template it is an instance of
- has a list of arguments, related to it via
  - `ottr:values` if argument values are given directly
  - `ottr:arguments` if some of the arguments have modifiers (e.g. `ottr:listExpand`)

- Templates can be built with:
  - Any text editor (wOTTR, soon: stOTTR)
  - Soon: GUI

- Templates can be built with:
  - Any text editor (wOTTR, soon: stOTTR)
  - Soon: GUI
- Template instances can be made using:
  - Any text editor (wOTTR, soon: stOTTR)
  - Excel/LibreOffice etc. (tabOTTR)
  - Work in progress: General mapping language from RDBMSs, CSV, JSON, XML, etc.
  - From ontology with templates as queries (qOTTR – experimental)

- *Lutra*[4] is our open source CLI-tool for working with templates

[4] https://gitlab.com/ottr/lutra/lutra/releases

- *Lutra*[4] is our open source CLI-tool for working with templates
- Java 8, Streams, Jena

[4] https://gitlab.com/ottr/lutra/lutra/releases

- *Lutra*[4] is our open source CLI-tool for working with templates
- Java 8, Streams, Jena
- Features:
  - Expand template library and instances

[4] https://gitlab.com/ottr/lutra/lutra/releases

- *Lutra*[4] is our open source CLI-tool for working with templates
- Java 8, Streams, Jena
- Features:
  - Expand template library and instances
  - Type-checking, cycle detection, and other sanity checks

[4] `https://gitlab.com/ottr/lutra/lutra/releases`

- *Lutra*[4] is our open source CLI-tool for working with templates
- Java 8, Streams, Jena
- Features:
  - Expand template library and instances
  - Type-checking, cycle detection, and other sanity checks
  - Input: tabOTTR, wOTTR, legacy wOTTR (others soon to come)

[4] https://gitlab.com/ottr/lutra/lutra/releases

- *Lutra*[4] is our open source CLI-tool for working with templates
- Java 8, Streams, Jena
- Features:
  - Expand template library and instances
  - Type-checking, cycle detection, and other sanity checks
  - Input: tabOTTR, wOTTR, legacy wOTTR (others soon to come)
  - Output: wOTTR (others soon to come)

[4] https://gitlab.com/ottr/lutra/lutra/releases

- *Lutra*[4] is our open source CLI-tool for working with templates
- Java 8, Streams, Jena
- Features:
  - Expand template library and instances
  - Type-checking, cycle detection, and other sanity checks
  - Input: tabOTTR, wOTTR, legacy wOTTR (others soon to come)
  - Output: wOTTR (others soon to come)
  - Also an API for working programmatically with templates
- To run Lutra: `$ java -jar lutra.jar`
- For help: `$ java -jar lutra.jar --help`

[4] `https://gitlab.com/ottr/lutra/lutra/releases`

Example scenario: Device factory

Demo/Exercises

DEVICE[owl:Class ?itemClass, ottt:NEList<owl:Class> ?components, ? xsd:string ?label] :: {

Demo/Exercises

DEVICE[owl:Class ?itemClass, ottt:NEList<owl:Class> ?components, ? xsd:string ?label] :: {

SUBCLASSOF(?itemClass, `class:Device`),

Demo/Exercises

ConsistsOf[owl:Class ?whole, NEList<owl:Class> ?components] .

"States that ?whole consists of exactly the components in ?components."

Device[owl:Class ?itemClass, ottt:NEList<owl:Class> ?components, ? xsd:string ?label] :: {

SubClassOf(?itemClass, class:Device),

Demo/Exercises

ConsistsOf[owl:Class ?whole, NEList<owl:Class> ?components] .

"States that ?whole consists of exactly the components in ?components."

Device[owl:Class ?itemClass, ottt:NEList<owl:Class> ?components, ? xsd:string ?label] :: {
  SubClassOf(?itemClass, class:Device),
  ConsistsOf(?itemClass, ?components),

ConsistsOf[owl:Class ?whole, NEList<owl:Class> ?components] .

"States that ?whole consists of exactly the components in ?components."

```
Device[owl:Class ?itemClass, ottt:NEList<owl:Class> ?components, ? xsd:string ?label] :: {
  SubClassOf(?itemClass, class:Device),
  ConsistsOf(?itemClass, ?components),
  Triple(?itemClass, rdfs:label, ?label)
} .
```

Modelling

1. Open up `instances.xlxs` containing some instances, and use Lutra to expand these instances into an Ontology via

   ```
   $ java -jar lutra.jar -m expand -I tabottr -l ./templates -o expanded.ttl instances.xlsx
   ```

2. Open the expanded file in Protégé and examine the result
3. Add instances stating the following:
   - that a `ThinkPad_L420` has components of types `Intel_i5`, `GPU_Nvidia`, `RAM_2GB`, `HDD_1TB`, and has label `"ThinkPad L420 Laptop"`,

Device instances

DEVICEINSTANCE[owl:NamedIndividual ?item, owl:Class ?itemClass, ottt:NEList<owl:NamedIndividual> ?components] :: {

Device instances

```
DeviceInstance[owl:NamedIndividual ?item, owl:Class ?itemClass, ottt:NEList<owl:NamedIndividual> ?components] :: {
  Triple(?item, rdf:type, ?itemClass),
```

Device instances

```
DeviceInstance[owl:NamedIndividual ?item, owl:Class ?itemClass, ottt:NEList<owl:NamedIndividual> ?components] :: {
  Triple(?item, rdf:type, ?itemClass),
  cross | Triple(?item, ex:hasComponent, ++?components)
} .
```

Individuals

3. Add instances stating the following:
   - that the concrete ThinkPad L420 `pc1` has concrete components `cpu1`, `ram1`, `graphics1`, and `hdd1`.
4. Expand the instances and reopen it in Protégé

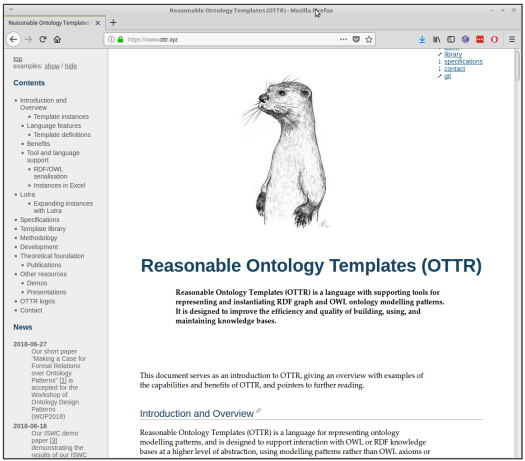
Templates

```
DEVICE[owl:Class ?itemClass, ottt:NEList<owl:Class> ?components, ? xsd:string ?label] :: {
  SUBCLASSOF(?itemClass, class:Device),
  CONSISTSOF(?itemClass, ?components),
  TRIPLE(?itemClass, ?label)
}

DEVICEINSTANCE[owl:NamedIndividual ?item, owl:Class ?itemClass, ottt:NEList<owl:NamedIndividual> ?components] :: {
  TRIPLE(?item, rdf:type, ?itemClass),
  cross | TRIPLE(?item, ex:hasComponent, ++?components)
}
```

Resources

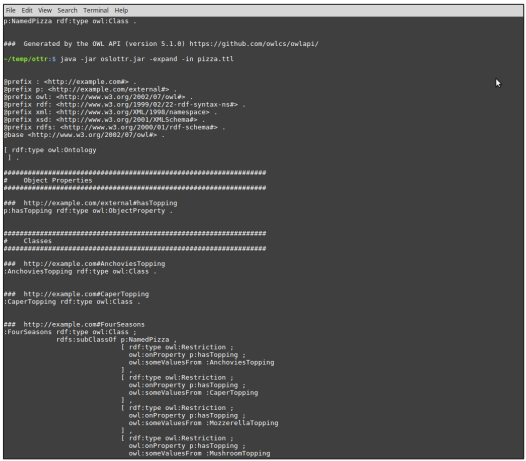
Available at `ottr.xyz`



Resources

Available at `ottr.xyz`

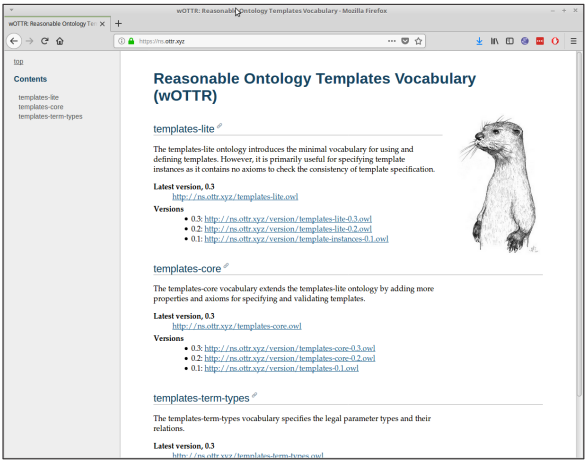
- *Lutra* CLI Java tool



Resources

Available at `ottr.xyz`

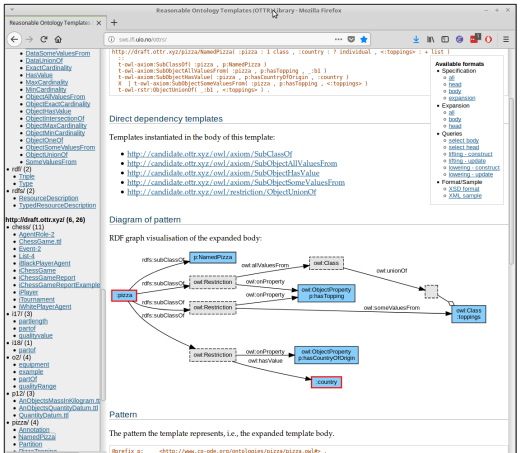
- *Lutra* CLI Java tool
- Specs: wOTTR, stOTTR, tabOTTR, mOTTR, rOTTR



Resources

Available at `ottr.xyz`

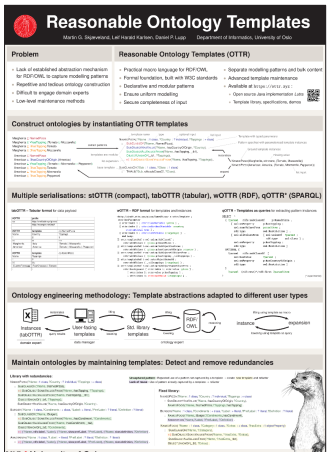
- *Lutra* CLI Java tool
- Specs: wOTTR, stOTTR, tabOTTR, mOTTR, rOTTR
- Library of OTTR templates



Resources

Available at `ottr.xyz`

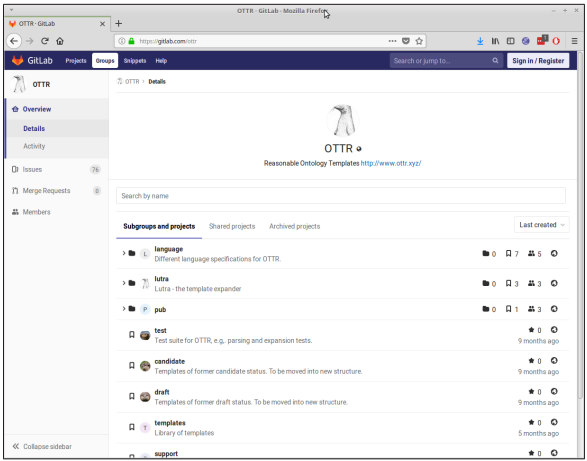
- *Lutra* CLI Java tool
- Specs: wOTTR, stOTTR, tabOTTR, mOTTR, rOTTR
- Library of OTTR templates
- Research papers, posters and demos



Resources

Available at `ottr.xyz`

- *Lutra* CLI Java tool
- Specs: wOTTR, stOTTR, tabOTTR, mOTTR, rOTTR
- Library of OTTR templates
- Research papers, posters and demos
- Git: `gitlab.com/ottr/`



Oblig on OTTR

- Published after lecture (before 17:00)
- Same procedure as the small obligs (one week, one attempt)
- Can use MrOblig to check your answers
- Can/Should use Lutra, but only need to use

```
$ java -jar lutra.jar -m expand -l <library> -o <expanded> <instances>
```

where

- `<library>` is the folder containing the templates
- `<expanded>` is the file to write the expanded instances to
- `<instances>` is the file containing the instances to expand