OWL-RL Documentation

Release 5.2.2

CSIRO Land and Water

Oct 10, 2021

Getting Started

CHAPTER 1

OWL-RL

A simple implementation of the OWL2 RL Profile, as well as a basic RDFS inference, on top of RDFLib. Based mechanical forward chaining. The distribution contains:

OWL-RL: the Python library. You should copy the directory somewhere into your `PYTHONPATH`. Alternatively, you can also run the `python setup.py install` script in the directory.

- `scripts/RDFConvertService`: can be used as a CGI script to invoke the library. It may have to be adapted to the local server setup.
- `scripts/owlrl`: script that can be run locally on to transform a file into RDF (on the standard output). Run the script with `-h` to get the available flags.

The package requires Python version 3.5 or higher; it depends on RDFLib; version 4.2.2 or higher is required. If you need the python 2.7.x compatible version, see the @/py2 branch in this repository.

For the details on RDFS, see the RDF Semantics Specification; for OWL 2 RL, see the OWL 2 Profile specification.

View the **OWL-RL documentation** online: http://owl-rl.readthedocs.io/

To view the changelog for this software library, see CHANGELOG.rst.

1.1 Installation

Coming soon.

1.2 Usage

Coming soon.

Note: Refer to `owlrl` for package entry details, etc.

1.3 Indices and tables

- genindex
- modindex
- search

1.4 owlrl

This module is a brute force implementation of the 'finite' version of RDFS semantics and of OWL 2 RL on the top of RDFLib (with some caveats, see below). Some extensions to these are also implemented.

Brute force means that, in all cases, simple forward chaining rules are used to extend (recursively) the incoming graph with all triples that the rule sets permit (ie, the "deductive closure" of the graph is computed). There is an extra options whether the axiomatic triples are added to the graph (prior to the forward chaining step). These, typically set the domain and range for properties or define some core classes. In the case of RDFS, the implementation uses a 'finite' version of the axiomatic triples only (as proposed, for example, by Herman ter Horst). This means that it adds only those `rdf:_i` type predicates that do appear in the original graph, thereby keeping this step finite. For OWL 2 RL, OWL 2 does not define axiomatic triples formally; but they can be deduced from the OWL 2 RDF Based Semantics document and are listed in Appendix 6 (though informally).

Note: This implementation adds only those triples that refer to OWL terms that are meaningful for the OWL 2 RL case.

1.4.1 Package Entry Points

The main entry point to the package is via the `DeductiveClosure` class. This class should be initialized to control the parameters of the deductive closure; the forward chaining is done via the L{expand<DeductiveClosure.expand>} method. The simplest way to use the package from an RDFLib application is as follows:

```
graph = Graph()                                    # creation of an RDFLib graph
...
...                                                # normal RDFLib application, eg,␣
↪parsing RDF data
...
DeductiveClosure(OWLRL_Semantics).expand(graph)  # calculate an OWL 2 RL deductive␣
↪closure of graph
                                                   # without axiomatic triples
```

The first argument of the `DeductiveClosure` initialization can be replaced by other classes, providing different types of deductive closure; other arguments are also possible. For example:

```
DeductiveClosure(OWLRL_Extension, rdfs_closure = True, axiomatic_triples = True,␣
↪datatype_axioms = True).expand(graph)
```

This will calculate the deductive closure including RDFS and some extensions to OWL 2 RL, and with all possible axiomatic triples added to the graph (this is about the maximum the package can do. . .)

The same instance of `DeductiveClosure` can be used for several graph expansions. In other words, the expand function does not change any state.

For convenience, a second entry point to the package is provided in the form of a function called *convert_graph()*, that expects a directory with various options, including a file name. The function parses the file, creates the expanded graph, and serializes the result into RDF/XML or Turtle. This function is particularly useful as an entry point for a CGI call (where the HTML form parameters are in a directory) and is easy to use with a command line interface. The package distribution contains an example for both.

There are major closure type (ie, semantic closure possibilities); these can be controlled through the appropriate parameters of the `DeductiveClosure` class:

- using the *RDFS_Semantics* class, implementing the RDFS semantics.
- using the `OWLRL.OWLRL_Semantics` class, implementing the OWL 2 RL.
- using *CombinedClosure.RDFS_OWLRL_Semantics* class, implementing a combined semantics of RDFS semantics and OWL 2 RL.

In all three cases there are other dimensions that can control the exact closure being generated:

- for convenience, the so called axiomatic triples (see, eg, the axiomatic triples in RDFS are, by default, I{not} added to the graph closure to reduce the number of generated triples. These can be controlled through a separate initialization argument.
- similarly, the axiomatic triples for D-entailment are separated.

1.4.2 Some Technical/implementation aspects

The core processing is done in the in the *Closure.Core* class, which is subclassed by the *RDFSClosure.RDFS_Semantics* and the `OWLRL.OWLRL_Semantics` classes (these two are then, on their turn, subclassed by the *CombinedClosure.RDFS_OWLRL_Semantics* class). The core implements the core functionality of cycling through the rules, whereas the rules themselves are defined and implemented in the subclasses. There are also methods that are executed only once either at the beginning or at the end of the full processing cycle. Adding axiomatic triples is handled separately, which allows a finer user control over these features.

Literals must be handled separately. Indeed, the functionality relies on 'extended' RDF graphs, that allows literals to be in a subject position, too. Because RDFLib does not allow that, processing begins by exchanging all literals in the graph for bnodes (identical literals get the same associated bnode). Processing occurs on these bnodes; at the end of the process all these bnodes are replaced by their corresponding literals if possible (if the bnode occurs in a subject position, that triple is removed from the resulting graph). Details of this processing is handled in the separate `Literals.LiteralProxies` class.

The OWL specification includes references to datatypes that are not in the core RDFS specification, consequently not directly implemented by RDFLib. These are added in a separate module of the package.

Problems with Literals with datatypes

The current distribution of RDFLib is fairly poor in handling datatypes, particularly in checking whether a lexical form of a literal is "proper" as for its declared datatype. A typical example is:

```
"-1234"^^xsd:nonNegativeInteger
```

which should not be accepted as valid literal. Because the requirements of OWL 2 RL are much stricter in this respect, an alternative set of datatype handling (essentially, conversions) had to be implemented (see the *XsdDatatypes* module).

The DeductiveClosure class has an additional instance variable whether the default RDFLib conversion routines should be exchanged against the new ones. If this flag is set to True and instance creation (this is the default), then the conversion routines are set back to the originals once the expansion is complete, thereby avoiding to influence older application that may not work properly with the new set of conversion routines.

If the user wants to use these alternative lexical conversions everywhere in the application, then the *DeductiveClosure.use_improved_datatypes_conversions()* method can be invoked. That method changes the conversion routines and, from that point on, all usage of DeductiveClosure instances will use the improved conversion methods without resetting them. Ie, the code structure can be something like:

```
DeductiveClosure().use_improved_datatypes_conversions()
... RDFLib application
DeductiveClosure().expand(graph)
...
```

The default situation can be set back using the *DeductiveClosure.use_rdflib_datatypes_conversions()* call.

It is, however, not *required* to use these methods at all. I.e., the user can use:

```
DeductiveClosure(improved_datatypes=False).expand(graph)
```

which will result in a proper graph expansion except for the datatype specific comparisons which will be incomplete.

Requires:

- RDFLib, 4.0.0 and higher.
- rdflib_jsonld

License: This software is available for use under the W3C Software License

Organization: World Wide Web Consortium

Author: Ivan Herman

class owlrl.**DeductiveClosure**(*closure_class*, *improved_datatypes=True*, *rdfs_closure=False*, *axiomatic_triples=False*, *datatype_axioms=False*)

Bases: object

Entry point to generate the deductive closure of a graph. The exact choice deductive closure is controlled by a class reference. The important initialization parameter is the closure_class, a Class object referring to a subclass of *Closure.Core*. Although this package includes a number of such subclasses OWLRL_Semantics, *RDFS_Semantics*, *RDFS_OWLRL_Semantics*, and *OWLRL_Extension*, the user can use his/her own if additional rules are implemented.

Note that owl:imports statements are *not* interpreted in this class, that has to be done beforehand on the graph that is to be expanded.

Parameters

- **closure_class** (subclass of *Closure.Core*) – A closure class reference.
- **improved_datatypes** (*bool*) – Whether the improved set of lexical-to-Python conversions should be used for datatype handling. See the introduction for more details. Default: True.
- **rdfs_closure** (*bool*) – Whether the RDFS closure should also be executed. Default: False.

- **axiomatic_triples** (*bool*) – Whether relevant axiomatic triples are added before chaining, except for datatype axiomatic triples. Default: False.
- **datatype_axioms** (*bool*) – Whether further datatype axiomatic triples are added to the output. Default: false.

Variables ***improved_datatype_generic*** – Whether the improved set of lexical-to-Python conversions should be used for datatype handling *in general*, I.e., not only for a particular instance and not only for inference purposes. Default: False.

expand(*graph*)
Expand the graph using forward chaining, and with the relevant closure type.

Parameters **graph** (`rdflib.Graph`) – The RDF graph.

improved_datatype_generic = False

static use_improved_datatypes_conversions()
Switch the system to use the improved datatype conversion routines.

static use_rdflib_datatypes_conversions()
Switch the system to use the generic (RDFLib) datatype conversion routines

owlrl.**convert_graph**(*options*, *closureClass=None*)
Entry point for external scripts (CGI or command line) to parse an RDF file(s), possibly execute OWL and/or RDFS closures, and serialize back the result in some format.

Note that this entry point can be used requiring no entailment at all; because both the input and the output format for the package can be RDF/XML or Turtle, such usage would simply mean a format conversion.

If OWL 2 RL processing is required, that also means that the `owl:imports` statements are interpreted. I.e., ontologies can be spread over several files. Note, however, that the output of the process would then include all imported ontologies, too.

Parameters

- **options** (*object*) – Object with specific attributes.
- **options.sources** (*list*) – List of uris or file names for the source data; for each one if the name ends with 'ttl', it is considered to be turtle, RDF/XML otherwise (this can be overwritten by the options.iformat, though)
- **options.text** (*str*) – Direct Turtle encoding of a graph as a text string (useful, eg, for a CGI call using a text field).
- **options.owlClosure** (*bool*) – Can be yes or no.
- **options.rdfsClosure** (*bool*) – Can be yes or no.
- **options.owlExtras** (*bool*) – Can be yes or no; whether the extra rules beyond OWL 2 RL are used or not.
- **options.axioms** (*bool*) – Whether relevant axiomatic triples are added before chaining (can be a boolean, or the strings "yes" or "no").
- **options.daxioms** (*bool*) – Further datatype axiomatic triples are added to the output (can be a boolean, or the strings "yes" or "no").
- **options.format** (*str*) – Output format, can be "turtle" or "rdfxml".
- **options.iformat** (*str*) – Input format, can be "turtle", "rdfa", "json", "rdfxml", or "auto". "auto" means that the suffix of the file is considered: '.ttl'. '.html', 'json' or '.jsonld' respectively with 'xml' as a fallback.

- **options.trimming** (*bool*) – Whether the extension to OWLRL should also include trimming.
- **closureClass** (TODO(edmond.chuc@csiro.au): What class is this supposed to be?) – Explicit class reference. If set, this overrides the various different other options to be used as an extension.

owlrl.**interpret_owl_imports**(*iformat*, *graph*)

Interpret the owl import statements. Essentially, recursively merge with all the objects in the owl import statement, and remove the corresponding triples from the graph.

This method can be used by an application prior to expansion. It is *not* done by the the DeductiveClosure class.

Parameters **iformat** – Input format; can be one of AUTO, TURTLE, or RDFXML. AUTO means that

the suffix of the file name or URI will decide: '.ttl' means Turtle, RDF/XML otherwise. :type iformat: str

Parameters **graph** (RDFLib.Graph) – The RDFLib Graph instance to parse into.

owlrl.**return_closure_class**(*owl_closure*, *rdfs_closure*, *owl_extras*, *trimming=False*)

Return the right semantic extension class based on three possible choices (this method is here to help potential users, the result can be fed into a DeductiveClosure instance at initialization).

Parameters

- **owl_closure** (*bool*) – Whether OWL 2 RL deductive closure should be calculated.
- **rdfs_closure** (*bool*) – Whether RDFS deductive closure should be calculated. In case owl_closure==True, this parameter should also be used in the initialization of DeductiveClosure.
- **owl_extras** (*bool*) – Whether the extra possibilities (rational datatype, etc) should be added to an OWL 2 RL deductive closure. This parameter has no effect in case owl_closure==False.
- **trimming** (*bool*) – Whether extra trimming is done on the OWL RL + Extension output.

Returns Deductive class reference or None.

Return type DeductiveClosure or None

1.5 AxiomaticTriples

Axiomatic triples to be (possibly) added to the final graph.

Requires: RDFLib, 4.0.0 and higher.

License: This software is available for use under the W3C Software License.

Organization: World Wide Web Consortium

Author: Ivan Herman

See also:

View the source code AxiomaticTriples.

1.6 Closure

The generic superclasses for various rule based semantics and the possible extensions.

Requires: RDFLib, 4.0.0 and higher.

License: This software is available for use under the W3C Software License.

Organization: World Wide Web Consortium

Author: Ivan Herman

class owlrl.Closure.**Core**(*graph*, *axioms*, *daxioms*, *rdfs=False*)
Bases: object

Core of the semantics management, dealing with the RDFS and other Semantic triples. The only reason to have it in a separate class is for an easier maintainability.

This is a common superclass only. In the present module, it is subclassed by a *RDFSClosure.RDFS_Semantics* class and a OWLRL.OWLRL_Semantics classes. There are some methods that are implemented in the subclasses only, ie, this class cannot be used by itself!

Parameters

- **graph** (rdflib.Graph) – The RDF graph to be extended.
- **axioms** (*bool*) – Whether axioms should be added or not.
- **daxioms** (*bool*) – Whether datatype axioms should be added or not.
- **rdfs** (*bool*) – Whether RDFS inference is also done (used in subclassed only).

Variables

- **IMaxNum** – Maximal index of rdf:_i occurrence in the graph.
- **graph** – The real graph.
- **axioms** – Whether axioms should be added or not.
- **daxioms** – Whether datatype axioms should be added or not.
- **added_triples** – Triples added to the graph, conceptually, during one processing cycle.
- **error_messages** – Error messages (typically inconsistency messages in OWL RL) found during processing. These are added to the final graph at the very end as separate BNodes with error messages.
- **rdfs** – Whether RDFS inference is also done (used in subclassed only).

add_axioms()
Add axioms.

This is only a placeholder and raises an exception by default; subclasses *must* fill this with real content

add_d_axioms()
Add d axioms.

This is only a placeholder and raises an exception by default; subclasses I{must} fill this with real content

add_error(*message*)
Add an error message

Parameters **message** (*str*) – Error message.

closure()
Generate the closure the graph. This is the real 'core'.

The processing rules store new triples via the separate method `Core.store_triple()` which stores them in the `added_triples` array. If that array is empty at the end of a cycle, it means that the whole process can be stopped.

If required, the relevant axiomatic triples are added to the graph before processing in cycles. Similarly the exchange of literals against bnodes is also done in this step (and restored after all cycles are over).

empty_stored_triples()
Empty the internal store for triples.

flush_stored_triples()
Send the stored triples to the graph, and empty the container.

one_time_rules()
This is only a placeholder; subclasses should fill this with real content. By default, it is just an empty call. This set of rules is invoked only once and not in a cycle.

post_process()
Do some post-processing step. This method when all processing is done, but before handling possible errors (ie, the method can add its own error messages). By default, this method is empty, subclasses can add content to it by overriding it.

pre_process()
Do some pre-processing step. This method before anything else in the closure. By default, this method is empty, subclasses can add content to it by overriding it.

rules(*t*, *cycle_num*)
The core processing cycles through every tuple in the graph and dispatches it to the various methods implementing a specific group of rules. By default, this method raises an exception; indeed, subclasses *must* add content to by overriding it.

Parameters

- **t** (*tuple*) – One triple on which to apply the rules.
- **cycle_num** (*int*) – Which cycle are we in, starting with 1. This value is forwarded to all local rules; it is also used locally to collect the bnodes in the graph.

store_triple(*t*)
In contrast to its name, this does not yet add anything to the graph itself, it just stores the tuple in an internal set (`Core.added_triples`). (It is important for this to be a set: some of the rules in the various closures may generate the same tuples several times.) Before adding the tuple to the set, the method checks whether the tuple is in the final graph already (if yes, it is not added to the set).

The set itself is emptied at the start of every processing cycle; the triples are then effectively added to the graph at the end of such a cycle. If the set is actually empty at that point, this means that the cycle has not added any new triple, and the full processing can stop.

Parameters **t** (*tuple (s,p,o)*) – The triple to be added to the graph, unless it is already there

1.7 CombinedClosure

The combined closure: performing *both* the OWL 2 RL and RDFS closures.

The two are very close but there are some rules in RDFS that are not in OWL 2 RL (eg, the axiomatic triples concerning the container membership properties). Using this closure class the OWL 2 RL implementation becomes a full extension of RDFS.

Requires: RDFLib, 4.0.0 and higher.

License: This software is available for use under the W3C Software License.

Organization: World Wide Web Consortium

Author: Ivan Herman

class owlrl.CombinedClosure.**RDFS_OWLRL_Semantics**(*graph*, *axioms*, *daxioms*, *rdfs=True*)
Bases: *owlrl.RDFSClosure.RDFS_Semantics*, owlrl.OWLRL.OWLRL_Semantics

Common subclass of the RDFS and OWL 2 RL semantic classes. All methods simply call back to the functions in the superclasses. This may lead to some unnecessary duplication of terms and rules, but it it not so bad. Also, the additional identification defined for OWL Full, ie, Resource being the same as Thing and OWL and RDFS classes being identical are added to the triple store.

Note that this class is also a possible user extension point: subclasses can be created that extend the standard functionality by extending this class. This class *always*} performs RDFS inferences. Subclasses have to set the `self.rdfs` flag explicitly to the requested value if that is to be controlled.

Parameters

- **graph** (rdflib.Graph) – The RDF graph to be extended.
- **axioms** (*bool*) – Whether (non-datatype) axiomatic triples should be added or not.
- **daxioms** (*bool*) – Whether datatype axiomatic triples should be added or not.
- **rdfs** (*bool*) – Placeholder flag (used in subclassed only, it is always defaulted to True in this class)

Variables

- ***full_binding_triples*** – Additional axiom type triples that are added to the combined semantics; these 'bind' the RDFS and the OWL worlds together.
- **rdfs** – (bool) Whether RDFS inference is to be performed or not. In this class instance the value is *always* `True`, subclasses may explicitly change it at initialization time.

add_axioms()
Add axioms

add_d_axioms()
This is not really complete, because it just uses the comparison possibilities that RDFLib provides.

add_error(*message*)
Add an error message

Parameters **message** (*str*) – Error message.

static add_new_datatype(*uri*, *conversion_function*, *datatype_list*, *subsumption_dict=None*, *subsumption_key=None*, *subsumption_list=None*)
If an extension wants to add new datatypes, this method should be invoked at initialization time.

Parameters

- **uri** – URI for the new datatypes, like owl_ns["Rational"].
- **conversion_function** – A function converting the lexical representation of the datatype to a Python value, possibly raising an exception in case of unsuitable lexical form.

- **datatype_list** (*list*) – List of datatypes already in use that has to be checked.
- **subsumption_dict** (*dict*) – Dictionary of subsumption hierarchies (indexed by the datatype URI-s).
- **subsumption_key** (*str*) – Key in the dictionary, if None, the uri parameter is used.
- **subsumption_list** (*list*) – List of subsumptions associated to a subsumption key (ie, all datatypes that are superclasses of the new datatype).

closure()
Generate the closure the graph. This is the real 'core'.

The processing rules store new triples via the separate method `Core.store_triple()` which stores them in the `added_triples` array. If that array is empty at the end of a cycle, it means that the whole process can be stopped.

If required, the relevant axiomatic triples are added to the graph before processing in cycles. Similarly the exchange of literals against bnodes is also done in this step (and restored after all cycles are over).

empty_stored_triples()
Empty the internal store for triples.

flush_stored_triples()
Send the stored triples to the graph, and empty the container.

full_binding_triples = [(rdflib.term.URIRef('http://www.w3.org/2002/07/owl#Thing'), rd

one_time_rules()
Adds some extra axioms and calls for the d_axiom part of the OWL Semantics.

post_process()
Do some post-processing step. This method when all processing is done, but before handling possible errors (I.e., the method can add its own error messages). By default, this method is empty, subclasses can add content to it by overriding it.

pre_process()
Do some pre-processing step. This method before anything else in the closure. By default, this method is empty, subclasses can add content to it by overriding it.

restriction_typing_check(*v*, *t*)
Helping method to check whether a type statement is in line with a possible restriction. This method is invoked by rule "cls-avf" before setting a type on an allValuesFrom restriction.

The method is a placeholder at this level. It is typically implemented by subclasses for extra checks, e.g., for datatype facet checks.

Parameters

- **v** – The resource that is to be 'typed'.
- **t** – The targeted type (ie, Class).

Returns Boolean.

Return type bool

rules(*t*, *cycle_num*)

Parameters

- **t** (*tuple*) – A triple (in the form of a tuple).
- **cycle_num** (*int*) – Which cycle are we in, starting with 1. This value is forwarded to all local rules; it is also used locally to collect the bnodes in the graph.

store_triple(*t*)

In contrast to its name, this does not yet add anything to the graph itself, it just stores the tuple in an internal set (`Core.added_triples`). (It is important for this to be a set: some of the rules in the various closures may generate the same tuples several times.) Before adding the tuple to the set, the method checks whether the tuple is in the final graph already (if yes, it is not added to the set).

The set itself is emptied at the start of every processing cycle; the triples are then effectively added to the graph at the end of such a cycle. If the set is actually empty at that point, this means that the cycle has not added any new triple, and the full processing can stop.

Parameters **t** (*tuple (s,p,o)*) – The triple to be added to the graph, unless it is already there

1.8 DatatypeHandling

Most of the XSD datatypes are handled directly by RDFLib. However, in some cases, that is not good enough. There are two major reasons for this:

1. Some datatypes are missing from RDFLib and required by OWL 2 RL and/or RDFS.
2. In other cases, though the datatype is present, RDFLib is fairly lax in checking the lexical value of those datatypes. Typical case is boolean.

Some of these deficiencies are handled by this module. All the functions convert the lexical value into a python datatype (or return the original string if this is not possible) which will be used, e.g., for comparisons (equalities). If the lexical value constraints are not met, exceptions are raised.

Requires: RDFLib, 4.0.0 and higher.

License: This software is available for use under the W3C Software License.

Organization: World Wide Web Consortium

Author: Ivan Herman

owlrl.DatatypeHandling.**use_Alt_lexical_conversions**()

Registering the datatypes item for RDFLib, ie, bind the dictionary values. The 'bind' method of RDFLib adds extra datatypes to the registered ones in RDFLib, though the table used here (I.e., `AltXSDToPYTHON`) actually overrides all of the default conversion routines. The method also add a Decimal entry to the `PythonToXSD` list of RDFLib.

owlrl.DatatypeHandling.**use_RDFLib_lexical_conversions**()

Restore the original (ie, RDFLib) set of lexical conversion routines.

1.8.1 AltXSDToPYTHON Table

Note: The code below is not extracted automatically from the source code.

If there are any errors, please make a pull request or an issue: https://github.com/RDFLib/OWL-RL

```
AltXSDToPYTHON = {
    XSD.language: lambda v: _strToVal_Regexp(v, _re_language),
    XSD.NMTOKEN: lambda v: _strToVal_Regexp(v, _re_NMTOKEN, re.U),
    XSD.Name: lambda v: _strToVal_Regexp(v, _re_NMTOKEN, re.U, _re_Name_ex),
    XSD.NCName: lambda v: _strToVal_Regexp(v, _re_NCName, re.U, _re_NCName_ex),
```

(continues on next page)

(continued from previous page)

```
    XSD.token: _strToToken,
    RDF.plainLiteral: _strToPlainLiteral,
    XSD.boolean: _strToBool,
    XSD.decimal: _strToDecimal,
    XSD.anyURI: _strToAnyURI,
    XSD.base64Binary: _strToBase64Binary,
    XSD.double: _strToDouble,
    XSD.float: _strToFloat,
    XSD.byte: lambda v: _strToBoundNumeral(v, _limits_byte, int),
    XSD.int: lambda v: _strToBoundNumeral(v, _limits_int, int),
    XSD.long: lambda v: _strToBoundNumeral(v, _limits_long, int),
    XSD.positiveInteger: lambda v: _strToBoundNumeral(v, _limits_positiveInteger,␣
↪int),
    XSD.nonPositiveInteger: lambda v: _strToBoundNumeral(v, _limits_
↪nonPositiveInteger, int),
    XSD.negativeInteger: lambda v: _strToBoundNumeral(v, _limits_negativeInteger,␣
↪int),
    XSD.nonNegativeInteger: lambda v: _strToBoundNumeral(v, _limits_
↪nonNegativeInteger, int),
    XSD.short: lambda v: _strToBoundNumeral(v, _limits_short, int),
    XSD.unsignedByte: lambda v: _strToBoundNumeral(v, _limits_unsignedByte, int),
    XSD.unsignedShort: lambda v: _strToBoundNumeral(v, _limits_unsignedShort, int),
    XSD.unsignedInt: lambda v: _strToBoundNumeral(v, _limits_unsignedInt, int),
    XSD.unsignedLong: lambda v: _strToBoundNumeral(v, _limits_unsignedLong, int),
    XSD.hexBinary: _strToHexBinary,
    XSD.dateTime: lambda v: _strToDateTimeAndStamp(v, False),
    XSD.dateTimeStamp: lambda v: _strToDateTimeAndStamp(v, True),
    RDF.XMLLiteral: _strToXMLLiteral,
    XSD.integer: int,
    XSD.string: lambda v: v,
    RDF.HTML: lambda v: v,
    XSD.normalizedString: lambda v: _strToVal_Regexp(v, _re_token),

    # These are RDFS specific...
    XSD.time: _strToTime,
    XSD.date: _strToDate,
    XSD.gYearMonth: _strTogYearMonth,
    XSD.gYear: _strTogYear,
    XSD.gMonthDay: _strTogMonthDay,
    XSD.gDay: _strTogDay,
    XSD.gMonth: _strTogMonth,
}
```

See also:

View the source code DatatypeHandling.

1.9 owlrl

This module is a brute force implementation of the 'finite' version of RDFS semantics and of OWL 2 RL on the top of RDFLib (with some caveats, see below). Some extensions to these are also implemented.

Brute force means that, in all cases, simple forward chaining rules are used to extend (recursively) the incoming graph with all triples that the rule sets permit (ie, the "deductive closure" of the graph is computed). There is an extra options whether the axiomatic triples are added to the graph (prior to the forward chaining step). These, typically set the

domain and range for properties or define some core classes. In the case of RDFS, the implementation uses a 'finite' version of the axiomatic triples only (as proposed, for example, by Herman ter Horst). This means that it adds only those `rdf:_i` type predicates that do appear in the original graph, thereby keeping this step finite. For OWL 2 RL, OWL 2 does not define axiomatic triples formally; but they can be deduced from the OWL 2 RDF Based Semantics document and are listed in Appendix 6 (though informally).

Note: This implementation adds only those triples that refer to OWL terms that are meaningful for the OWL 2 RL case.

1.9.1 Package Entry Points

The main entry point to the package is via the `DeductiveClosure` class. This class should be initialized to control the parameters of the deductive closure; the forward chaining is done via the L{expand<DeductiveClosure.expand>} method. The simplest way to use the package from an RDFLib application is as follows:

```
graph = Graph()                                    # creation of an RDFLib graph
...
...                                                # normal RDFLib application, eg,␣
↪parsing RDF data
...
DeductiveClosure(OWLRL_Semantics).expand(graph)  # calculate an OWL 2 RL deductive␣
↪closure of graph
                                                   # without axiomatic triples
```

The first argument of the `DeductiveClosure` initialization can be replaced by other classes, providing different types of deductive closure; other arguments are also possible. For example:

```
DeductiveClosure(OWLRL_Extension, rdfs_closure = True, axiomatic_triples = True,␣
↪datatype_axioms = True).expand(graph)
```

This will calculate the deductive closure including RDFS and some extensions to OWL 2 RL, and with all possible axiomatic triples added to the graph (this is about the maximum the package can do...)

The same instance of `DeductiveClosure` can be used for several graph expansions. In other words, the expand function does not change any state.

For convenience, a second entry point to the package is provided in the form of a function called *`convert_graph()`*, that expects a directory with various options, including a file name. The function parses the file, creates the expanded graph, and serializes the result into RDF/XML or Turtle. This function is particularly useful as an entry point for a CGI call (where the HTML form parameters are in a directory) and is easy to use with a command line interface. The package distribution contains an example for both.

There are major closure type (ie, semantic closure possibilities); these can be controlled through the appropriate parameters of the `DeductiveClosure` class:

- using the *`RDFS_Semantics`* class, implementing the RDFS semantics.
- using the `OWLRL.OWLRL_Semantics` class, implementing the OWL 2 RL.
- using *`CombinedClosure.RDFS_OWLRL_Semantics`* class, implementing a combined semantics of RDFS semantics and OWL 2 RL.

In all three cases there are other dimensions that can control the exact closure being generated:

- for convenience, the so called axiomatic triples (see, eg, the axiomatic triples in RDFS are, by default, I{not} added to the graph closure to reduce the number of generated triples. These can be controlled through a separate initialization argument.

- similarly, the axiomatic triples for D-entailment are separated.

1.9.2 Some Technical/implementation aspects

The core processing is done in the in the *Closure.Core* class, which is subclassed by the *RDFSClosure.RDFS_Semantics* and the `OWLRL.OWLRL_Semantics` classes (these two are then, on their turn, subclassed by the *CombinedClosure.RDFS_OWLRL_Semantics* class). The core implements the core functionality of cycling through the rules, whereas the rules themselves are defined and implemented in the subclasses. There are also methods that are executed only once either at the beginning or at the end of the full processing cycle. Adding axiomatic triples is handled separately, which allows a finer user control over these features.

Literals must be handled separately. Indeed, the functionality relies on 'extended' RDF graphs, that allows literals to be in a subject position, too. Because RDFLib does not allow that, processing begins by exchanging all literals in the graph for bnodes (identical literals get the same associated bnode). Processing occurs on these bnodes; at the end of the process all these bnodes are replaced by their corresponding literals if possible (if the bnode occurs in a subject position, that triple is removed from the resulting graph). Details of this processing is handled in the separate `Literals.LiteralProxies` class.

The OWL specification includes references to datatypes that are not in the core RDFS specification, consequently not directly implemented by RDFLib. These are added in a separate module of the package.

Problems with Literals with datatypes

The current distribution of RDFLib is fairly poor in handling datatypes, particularly in checking whether a lexical form of a literal is "proper" as for its declared datatype. A typical example is:

```
"-1234"^^xsd:nonNegativeInteger
```

which should not be accepted as valid literal. Because the requirements of OWL 2 RL are much stricter in this respect, an alternative set of datatype handling (essentially, conversions) had to be implemented (see the *XsdDatatypes* module).

The `DeductiveClosure` class has an additional instance variable whether the default RDFLib conversion routines should be exchanged against the new ones. If this flag is set to True and instance creation (this is the default), then the conversion routines are set back to the originals once the expansion is complete, thereby avoiding to influence older application that may not work properly with the new set of conversion routines.

If the user wants to use these alternative lexical conversions everywhere in the application, then the *DeductiveClosure.use_improved_datatypes_conversions()* method can be invoked. That method changes the conversion routines and, from that point on, all usage of `DeductiveClosure` instances will use the improved conversion methods without resetting them. Ie, the code structure can be something like:

```
DeductiveClosure().use_improved_datatypes_conversions()
... RDFLib application
DeductiveClosure().expand(graph)
...
```

The default situation can be set back using the *DeductiveClosure.use_rdflib_datatypes_conversions()* call.

It is, however, not *required* to use these methods at all. I.e., the user can use:

```
DeductiveClosure(improved_datatypes=False).expand(graph)
```

which will result in a proper graph expansion except for the datatype specific comparisons which will be incomplete.

Requires:

- RDFLib, 4.0.0 and higher.
- rdflib_jsonld

License: This software is available for use under the W3C Software License

Organization: World Wide Web Consortium

Author: Ivan Herman

class owlrl.**DeductiveClosure**(*closure_class*, *improved_datatypes=True*, *rdfs_closure=False*, *axiomatic_triples=False*, *datatype_axioms=False*)

Bases: object

Entry point to generate the deductive closure of a graph. The exact choice deductive closure is controlled by a class reference. The important initialization parameter is the closure_class, a Class object referring to a subclass of *Closure.Core*. Although this package includes a number of such subclasses OWLRL_Semantics, *RDFS_Semantics*, *RDFS_OWLRL_Semantics*, and *OWLRL_Extension*, the user can use his/her own if additional rules are implemented.

Note that owl:imports statements are *not* interpreted in this class, that has to be done beforehand on the graph that is to be expanded.

Parameters

- **closure_class** (subclass of *Closure.Core*) – A closure class reference.
- **improved_datatypes** (*bool*) – Whether the improved set of lexical-to-Python conversions should be used for datatype handling. See the introduction for more details. Default: True.
- **rdfs_closure** (*bool*) – Whether the RDFS closure should also be executed. Default: False.
- **axiomatic_triples** (*bool*) – Whether relevant axiomatic triples are added before chaining, except for datatype axiomatic triples. Default: False.
- **datatype_axioms** (*bool*) – Whether further datatype axiomatic triples are added to the output. Default: false.

Variables ***improved_datatype_generic*** – Whether the improved set of lexical-to-Python conversions should be used for datatype handling *in general*, I.e., not only for a particular instance and not only for inference purposes. Default: False.

expand(*graph*)

Expand the graph using forward chaining, and with the relevant closure type.

Parameters **graph** (rdflib.Graph) – The RDF graph.

improved_datatype_generic = False

static use_improved_datatypes_conversions()

Switch the system to use the improved datatype conversion routines.

static use_rdflib_datatypes_conversions()

Switch the system to use the generic (RDFLib) datatype conversion routines

owlrl.**convert_graph**(*options*, *closureClass=None*)

Entry point for external scripts (CGI or command line) to parse an RDF file(s), possibly execute OWL and/or RDFS closures, and serialize back the result in some format.

Note that this entry point can be used requiring no entailment at all; because both the input and the output format for the package can be RDF/XML or Turtle, such usage would simply mean a format conversion.

If OWL 2 RL processing is required, that also means that the `owl:imports` statements are interpreted. I.e., ontologies can be spread over several files. Note, however, that the output of the process would then include all imported ontologies, too.

Parameters

- **options** (*object*) – Object with specific attributes.
- **options.sources** (*list*) – List of uris or file names for the source data; for each one if the name ends with 'ttl', it is considered to be turtle, RDF/XML otherwise (this can be overwritten by the options.iformat, though)
- **options.text** (*str*) – Direct Turtle encoding of a graph as a text string (useful, eg, for a CGI call using a text field).
- **options.owlClosure** (*bool*) – Can be yes or no.
- **options.rdfsClosure** (*bool*) – Can be yes or no.
- **options.owlExtras** (*bool*) – Can be yes or no; whether the extra rules beyond OWL 2 RL are used or not.
- **options.axioms** (*bool*) – Whether relevant axiomatic triples are added before chaining (can be a boolean, or the strings "yes" or "no").
- **options.daxioms** (*bool*) – Further datatype axiomatic triples are added to the output (can be a boolean, or the strings "yes" or "no").
- **options.format** (*str*) – Output format, can be "turtle" or "rdfxml".
- **options.iformat** (*str*) – Input format, can be "turtle", "rdfa", "json", "rdfxml", or "auto". "auto" means that the suffix of the file is considered: '.ttl'. '.html', 'json' or '.jsonld' respectively with 'xml' as a fallback.
- **options.trimming** (*bool*) – Whether the extension to OWLRL should also include trimming.
- **closureClass** (TODO(edmond.chuc@csiro.au): What class is this supposed to be?) – Explicit class reference. If set, this overrides the various different other options to be used as an extension.

`owlrl.`**`interpret_owl_imports`**(*iformat*, *graph*)

Interpret the owl import statements. Essentially, recursively merge with all the objects in the owl import statement, and remove the corresponding triples from the graph.

This method can be used by an application prior to expansion. It is *not* done by the the `DeductiveClosure` class.

Parameters **iformat** – Input format; can be one of `AUTO`, `TURTLE`, or `RDFXML`. `AUTO` means that

the suffix of the file name or URI will decide: '.ttl' means Turtle, RDF/XML otherwise. :type iformat: str

Parameters **graph** (`RDFLib.Graph`) – The RDFLib Graph instance to parse into.

`owlrl.`**`return_closure_class`**(*owl_closure*, *rdfs_closure*, *owl_extras*, *trimming=False*)

Return the right semantic extension class based on three possible choices (this method is here to help potential users, the result can be fed into a `DeductiveClosure` instance at initialization).

Parameters

- **owl_closure** (*bool*) – Whether OWL 2 RL deductive closure should be calculated.

- **rdfs_closure** (*bool*) – Whether RDFS deductive closure should be calculated. In case `owl_closure==True`, this parameter should also be used in the initialization of `DeductiveClosure`.
- **owl_extras** (*bool*) – Whether the extra possibilities (rational datatype, etc) should be added to an OWL 2 RL deductive closure. This parameter has no effect in case `owl_closure==False`.
- **trimming** (*bool*) – Whether extra trimming is done on the OWL RL + Extension output.

Returns Deductive class reference or None.

Return type `DeductiveClosure` or None

1.10 OWLRLExtras

Extension to OWL 2 RL, ie, some additional rules added to the system from OWL 2 Full. It is implemented through the *OWLRL_Extension* class, whose reference has to be passed to the relevant semantic class (i.e., either the OWL 2 RL or the combined closure class) as an 'extension'.

The added rules and features are:

- self restriction
- owl:rational datatype
- datatype restrictions via facets

In more details, the rules that are added:

1. self restriction 1: `?z owl:hasSelf ?x. ?x owl:onProperty ?p. ?y rdf:type ?z. => ?y ?p ?y.`
2. self restriction 2: `?z owl:hasSelf ?x. ?x owl:onProperty ?p. ?y ?p ?y. => ?y rdf:type ?z.`

Requires: RDFLib, 4.0.0 and higher.

License: This software is available for use under the W3C Software License.

Organization: World Wide Web Consortium

Author: Ivan Herman

class `owlrl.OWLRLExtras.`**OWLRL_Extension**(*graph*, *axioms*, *daxioms*, *rdfs=False*)
Bases: *owlrl.CombinedClosure.RDFS_OWLRL_Semantics*

Additional rules to OWL 2 RL. The initialization method also adds the `owl:rational` datatype to the set of allowed datatypes with the `_strToRational()` function as a conversion between the literal form and a Rational. The `xsd:decimal` datatype is also set to be a subclass of `owl:rational`. Furthermore, the restricted datatypes are extracted from the graph using a separate method in a different module (*RestrictedDatatype.extract_faceted_datatypes()*), and all those datatypes are also added to the set of allowed datatypes. In the case of the restricted datatypes and extra subsumption relationship is set up between the restricted and the base datatypes.

Variables

- ***extra_axioms*** – Additional axioms that have to be added to the deductive closure (in case the axiomatic triples are required).
- **restricted_datatypes** – list of the datatype restriction from *RestrictedDatatype*.

add_axioms()
Add the `OWLRL_Extension.extra_axioms`, related to the self restrictions.

add_d_axioms()
This is not really complete, because it just uses the comparison possibilities that RDFLib provides.

add_error(*message*)
Add an error message

Parameters **message** (*str*) – Error message.

static add_new_datatype(*uri*, *conversion_function*, *datatype_list*, *subsumption_dict=None*, *subsumption_key=None*, *subsumption_list=None*)
If an extension wants to add new datatypes, this method should be invoked at initialization time.

Parameters

- **uri** – URI for the new datatypes, like owl_ns["Rational"].
- **conversion_function** – A function converting the lexical representation of the datatype to a Python value, possibly raising an exception in case of unsuitable lexical form.
- **datatype_list** (*list*) – List of datatypes already in use that has to be checked.
- **subsumption_dict** (*dict*) – Dictionary of subsumption hierarchies (indexed by the datatype URI-s).
- **subsumption_key** (*str*) – Key in the dictionary, if None, the uri parameter is used.
- **subsumption_list** (*list*) – List of subsumptions associated to a subsumption key (ie, all datatypes that are superclasses of the new datatype).

closure()
Generate the closure the graph. This is the real 'core'.

The processing rules store new triples via the separate method `Core.store_triple()` which stores them in the `added_triples` array. If that array is empty at the end of a cycle, it means that the whole process can be stopped.

If required, the relevant axiomatic triples are added to the graph before processing in cycles. Similarly the exchange of literals against bnodes is also done in this step (and restored after all cycles are over).

empty_stored_triples()
Empty the internal store for triples.

extra_axioms = [(rdflib.term.URIRef('http://www.w3.org/2002/07/owl#hasSelf'), rdflib.t

flush_stored_triples()
Send the stored triples to the graph, and empty the container.

full_binding_triples = [(rdflib.term.URIRef('http://www.w3.org/2002/07/owl#Thing'), rd

one_time_rules()
This method is invoked only once at the beginning, and prior of, the forward chaining process.

At present, only the L{subsumption} of restricted datatypes<_subsume_restricted_datatypes>} is performed.

post_process()
Do some post-processing step. This method when all processing is done, but before handling possible errors (I.e., the method can add its own error messages). By default, this method is empty, subclasses can add content to it by overriding it.

pre_process()
Do some pre-processing step. This method before anything else in the closure. By default, this method is empty, subclasses can add content to it by overriding it.

restriction_typing_check(*v*, *t*)
Helping method to check whether a type statement is in line with a possible restriction. This method is invoked by rule "cls-avf" before setting a type on an allValuesFrom restriction.

The method is a placeholder at this level. It is typically implemented by subclasses for extra checks, e.g., for datatype facet checks.

Parameters

- **v** – the resource that is to be 'typed'.
- **t** – the targeted type (i.e., Class).

Returns Boolean.

Return type bool

rules(*t*, *cycle_num*)
Go through the additional rules implemented by this module.

Parameters

- **t** (*tuple*) – A triple (in the form of a tuple).
- **cycle_num** (*int*) – Which cycle are we in, starting with 1. This value is forwarded to all local rules; it is also used locally to collect the bnodes in the graph.

store_triple(*t*)
In contrast to its name, this does not yet add anything to the graph itself, it just stores the tuple in an internal set (`Core.added_triples`). (It is important for this to be a set: some of the rules in the various closures may generate the same tuples several times.) Before adding the tuple to the set, the method checks whether the tuple is in the final graph already (if yes, it is not added to the set).

The set itself is emptied at the start of every processing cycle; the triples are then effectively added to the graph at the end of such a cycle. If the set is actually empty at that point, this means that the cycle has not added any new triple, and the full processing can stop.

Parameters **t** (*tuple (s,p,o)*) – The triple to be added to the graph, unless it is already there

class owlrl.OWLRLExtras.**OWLRL_Extension_Trimming**(*graph*, *axioms*, *daxioms*, *rdfs=False*)
Bases: *owlrl.OWLRLExtras.OWLRL_Extension*

This Class adds only one feature to *OWLRL_Extension*: to initialize with a trimming flag set to `True` by default.

This is pretty experimental and probably contentious: this class *removes* a number of triples from the Graph at the very end of the processing steps. These triples are either the by-products of the deductive closure calculation or are axiom like triples that are added following the rules of OWL 2 RL. While these triples *are necessary* for the correct inference of really 'useful' triples, they may not be of interest for the application for the end result. The triples that are removed are of the form (following a SPARQL-like notation):

- `?x owl:sameAs ?x`, `?x rdfs:subClassOf ?x`, `?x rdfs:subPropertyOf ?x`, `?x owl:equivalentClass ?x` type triples.
- `?x rdfs:subClassOf rdf:Resource`, `?x rdfs:subClassOf owl:Thing`, `?x rdf:type rdf:Resource`, `owl:Nothing rdfs:subClassOf ?x` type triples.

- For a datatype that does *not* appear explicitly in a type assignments (ie, in a `?x rdf:type dt`) the corresponding `dt rdf:type owl:Datatype` and `dt rdf:type owl:DataRange` triples, as well as the disjointness statements with other datatypes.
- annotation property axioms.
- a number of axiomatic triples on `owl:Thing`, `owl:Nothing` and `rdf:Resource` (eg, `owl:Nothing rdf:type owl:Class`, `owl:Thing owl:equivalentClass rdf:Resource`, etc).

Trimming is the only feature of this class, done in the *post_process()* step. If users extend *OWLRL_Extension*, this class can be safely mixed in via multiple inheritance.

Parameters

- **graph** (`rdflib.Graph`) – The RDF graph to be extended.
- **axioms** (*bool*) – Whether (non-datatype) axiomatic triples should be added or not.
- **daxioms** (*bool*) – Whether datatype axiomatic triples should be added or not.
- **rdfs** (*bool*) – Whether RDFS extension is done.

add_axioms()
Add the `OWLRL_Extension.extra_axioms`, related to the self restrictions.

add_d_axioms()
This is not really complete, because it just uses the comparison possibilities that RDFLib provides.

add_error(*message*)
Add an error message

Parameters **message** (*str*) – Error message.

static add_new_datatype(*uri*, *conversion_function*, *datatype_list*, *subsumption_dict=None*, *subsumption_key=None*, *subsumption_list=None*)
If an extension wants to add new datatypes, this method should be invoked at initialization time.

Parameters

- **uri** – URI for the new datatypes, like owl_ns["Rational"].
- **conversion_function** – A function converting the lexical representation of the datatype to a Python value, possibly raising an exception in case of unsuitable lexical form.
- **datatype_list** (*list*) – List of datatypes already in use that has to be checked.
- **subsumption_dict** (*dict*) – Dictionary of subsumption hierarchies (indexed by the datatype URI-s).
- **subsumption_key** (*str*) – Key in the dictionary, if None, the uri parameter is used.
- **subsumption_list** (*list*) – List of subsumptions associated to a subsumption key (ie, all datatypes that are superclasses of the new datatype).

closure()
Generate the closure the graph. This is the real 'core'.

The processing rules store new triples via the separate method `Core.store_triple()` which stores them in the `added_triples` array. If that array is empty at the end of a cycle, it means that the whole process can be stopped.

If required, the relevant axiomatic triples are added to the graph before processing in cycles. Similarly the exchange of literals against bnodes is also done in this step (and restored after all cycles are over).

empty_stored_triples()
Empty the internal store for triples.

extra_axioms = [(rdflib.term.URIRef('http://www.w3.org/2002/07/owl#hasSelf'), rdflib.t

flush_stored_triples()
Send the stored triples to the graph, and empty the container.

full_binding_triples = [(rdflib.term.URIRef('http://www.w3.org/2002/07/owl#Thing'), rd

one_time_rules()
This method is invoked only once at the beginning, and prior of, the forward chaining process.

At present, only the L{subsumption} of restricted datatypes<_subsume_restricted_datatypes>} is performed.

post_process()
Do some post-processing step performing the trimming of the graph. See the *OWLRL_Extension_Trimming* class for further details.

pre_process()
Do some pre-processing step. This method before anything else in the closure. By default, this method is empty, subclasses can add content to it by overriding it.

restriction_typing_check(*v*, *t*)
Helping method to check whether a type statement is in line with a possible restriction. This method is invoked by rule "cls-avf" before setting a type on an allValuesFrom restriction.

The method is a placeholder at this level. It is typically implemented by subclasses for extra checks, e.g., for datatype facet checks.

Parameters

- **v** – the resource that is to be 'typed'.
- **t** – the targeted type (i.e., Class).

Returns Boolean.

Return type bool

rules(*t*, *cycle_num*)
Go through the additional rules implemented by this module.

Parameters

- **t** (*tuple*) – A triple (in the form of a tuple).
- **cycle_num** (*int*) – Which cycle are we in, starting with 1. This value is forwarded to all local rules; it is also used locally to collect the bnodes in the graph.

store_triple(*t*)
In contrast to its name, this does not yet add anything to the graph itself, it just stores the tuple in an internal set (`Core.added_triples`). (It is important for this to be a set: some of the rules in the various closures may generate the same tuples several times.) Before adding the tuple to the set, the method checks whether the tuple is in the final graph already (if yes, it is not added to the set).

The set itself is emptied at the start of every processing cycle; the triples are then effectively added to the graph at the end of such a cycle. If the set is actually empty at that point, this means that the cycle has not added any new triple, and the full processing can stop.

Parameters **t** (*tuple (s,p,o)*) – The triple to be added to the graph, unless it is already there

1.11 RDFSClosure

This module is brute force implementation of the RDFS semantics on the top of RDFLib (with some caveats, see in the introductory text).

Requires: RDFLib, 4.0.0 and higher.

License: This software is available for use under the W3C Software License.

Organization: World Wide Web Consortium

Author: Ivan Herman

class owlrl.RDFSClosure.**RDFS_Semantics**(*graph*, *axioms*, *daxioms*, *rdfs*)
Bases: *owlrl.Closure.Core*

RDFS Semantics class, ie, implementation of the RDFS closure graph.

Note: Note that the module does *not* implement the so called Datatype entailment rules, simply because the underlying RDFLib does not implement the datatypes (ie, RDFLib will not make the literal "1.00" and "1.00000" identical, although even with all the ambiguities on datatypes, this I{should} be made equal...).

Also, the so-called extensional entailment rules (Section 7.3.1 in the RDF Semantics document) have not been implemented either.

The comments and references to the various rule follow the names as used in the RDF Semantics document.

Parameters

- **graph** (rdflib.Graph) – The RDF graph to be extended.
- **axioms** (*bool*) – Whether (non-datatype) axiomatic triples should be added or not.
- **daxioms** (*bool*) – Whether datatype axiomatic triples should be added or not.
- **rdfs** (*bool*) – Whether RDFS inference is also done (used in subclassed only).

add_axioms()
Add axioms

add_d_axioms()
This is not really complete, because it just uses the comparison possibilities that RDFLib provides.

add_error(*message*)
Add an error message

Parameters **message** (*str*) – Error message.

closure()
Generate the closure the graph. This is the real 'core'.

The processing rules store new triples via the separate method Core.store_triple() which stores them in the added_triples array. If that array is empty at the end of a cycle, it means that the whole process can be stopped.

If required, the relevant axiomatic triples are added to the graph before processing in cycles. Similarly the exchange of literals against bnodes is also done in this step (and restored after all cycles are over).

empty_stored_triples()
Empty the internal store for triples.

flush_stored_triples()
Send the stored triples to the graph, and empty the container.

one_time_rules()
Some of the rules in the rule set are axiomatic in nature, meaning that they really have to be added only once, there is no reason to add these in a cycle. These are performed by this method that is invoked only once at the beginning of the process.

In this case this is related to a 'hidden' same as rules on literals with identical values (though different lexical values).

post_process()
Do some post-processing step. This method when all processing is done, but before handling possible errors (ie, the method can add its own error messages). By default, this method is empty, subclasses can add content to it by overriding it.

pre_process()
Do some pre-processing step. This method before anything else in the closure. By default, this method is empty, subclasses can add content to it by overriding it.

rules(*t*, *cycle_num*)
Go through the RDFS entailment rules rdf1, rdfs4-rdfs12, by extending the graph.

Parameters

- **t** (*tuple*) – A triple (in the form of a tuple).
- **cycle_num** (*int*) – Which cycle are we in, starting with 1. Can be used for some (though minor) optimization.

store_triple(*t*)
In contrast to its name, this does not yet add anything to the graph itself, it just stores the tuple in an internal set (`Core.added_triples`). (It is important for this to be a set: some of the rules in the various closures may generate the same tuples several times.) Before adding the tuple to the set, the method checks whether the tuple is in the final graph already (if yes, it is not added to the set).

The set itself is emptied at the start of every processing cycle; the triples are then effectively added to the graph at the end of such a cycle. If the set is actually empty at that point, this means that the cycle has not added any new triple, and the full processing can stop.

Parameters **t** (*tuple (s,p,o)*) – The triple to be added to the graph, unless it is already there

1.12 RestrictedDatatype

Module to datatype restrictions, i.e., data ranges.

The module implements the following aspects of datatype restrictions:

- a new datatype is created run-time and added to the allowed and accepted datatypes; literals are checked whether they abide to the restrictions
- the new datatype is defined to be a 'subClass' of the restricted datatype
- literals of the restricted datatype and that abide to the restrictions defined by the facets are also assigned to be of the new type

The last item is important to handle the following structures:

```
ex:RE a owl:Restriction ;
   owl:onProperty ex:p ;
   owl:someValuesFrom [
```

(continues on next page)

(continued from previous page)

```
       a rdfs:Datatype ;
       owl:onDatatype xsd:string ;
       owl:withRestrictions (
           [ xsd:minLength "3"^^xsd:integer ]
           [ xsd:maxLength "6"^^xsd:integer ]
       )
   ]
.
ex:q ex:p "abcd"^^xsd:string.
```

In the case above the system can then infer that `ex:q` is also of type `ex:RE`.

Datatype restrictions are used by the *OWLRLExtras.OWLRL_Extension* extension class.

The implementation is **not** 100% complete. Some things that an ideal implementation should do are not done yet like:

- checking whether a facet is of a datatype that is allowed for that facet
- handling of non-literals in the facets (ie, if the resource is defined to be of type literal, but whose value is defined via a separate `owl:sameAs` somewhere else)

Requires: RDFLib, 4.0.0 and higher.

License: This software is available for use under the W3C Software License.

Organization: World Wide Web Consortium

Author: Ivan Herman

class owlrl.RestrictedDatatype.**RestrictedDatatype**(*type_uri*, *base_type*, *facets*)

Bases: *owlrl.RestrictedDatatype.RestrictedDatatypeCore*

Implementation of a datatype with facets, ie, datatype with restrictions.

Parameters

- **type_uri** – URI of the datatype being defined.
- **base_type** – URI of the base datatype, ie, the one being restricted.
- **facets** – List of `(facetURI, value)` pairs.

:ivar datatype : The URI for this datatype.

Variables

- **base_type** – URI of the datatype that is restricted.
- **converter** – Method to convert a literal of the base type to a Python value (`DatatypeHandling.AltXSDToPYTHON`).
- **minExclusive** – Value for the :code‘xsd:minExclusive‘ facet, initialized to `None` and set to the right value if a facet is around.
- **minInclusive** – Value for the `xsd:minInclusive` facet, initialized to `None` and set to the right value if a facet is around.
- **maxExclusive** – Value for the `xsd:maxExclusive` facet, initialized to `None` and set to the right value if a facet is around.
- **maxInclusive** – Value for the `xsd:maxInclusive` facet, initialized to `None` and set to the right value if a facet is around.
- **minLength** – Value for the `xsd:minLength` facet, initialized to `None` and set to the right value if a facet is around.

- **maxLength** – Value for the `xsd:maxLength` facet, initialized to `None` and set to the right value if a facet is around.
- **length** – Value for the `xsd:length` facet, initialized to `None` and set to the right value if a facet is around.
- **pattern** – Array of patterns for the `xsd:pattern` facet, initialized to `[]` and set to the right value if a facet is around.
- **langRange** – Array of language ranges for the `rdf:langRange` facet, initialized to `[]` and set to the right value if a facet is around.
- **check_methods** – List of class methods that are relevant for the given `base_type`.
- **toPython** – Function to convert a Literal of the specified type to a Python value. Is defined by `lambda v: _lit_to_value(self, v)`, see *_lit_to_value()*.

checkValue(*value*)

Check whether the (Python) value abides to the constraints defined by the current facets.

Parameters **value** – The value to be checked.

Return type bool

class owlrl.RestrictedDatatype.**RestrictedDatatypeCore**(*type_uri*, *base_type*)

Bases: `object`

An 'abstract' superclass for datatype restrictions. The instance variables listed here are used in general, without the specificities of the concrete restricted datatype.

This module defines the *RestrictedDatatype* class that corresponds to the datatypes and their restrictions defined in the OWL 2 standard. Other modules may subclass this class to define new datatypes with restrictions.

Variables

- **type_uri** – The URI for this datatype.
- **base_type** – URI of the datatype that is restricted.
- **toPython** – Function to convert a Literal of the specified type to a Python value.

checkValue(*value*)

Check whether the (Python) value abides to the constraints defined by the current facets.

Parameters **value** – The value to be checked.

Return type bool

owlrl.RestrictedDatatype.**extract_faceted_datatypes**(*core*, *graph*)

Extractions of restricted (i.e., faceted) datatypes from the graph.

Parameters

- **core** (*Closure.Core*) – The core closure instance that is being handled.
- **graph** (`RDFLib.Graph`) – RDFLib graph.

Returns List of *RestrictedDatatype* instances.

Return type list

owlrl.RestrictedDatatype.**_lit_to_value**(*dt*, *v*)

This method is used to convert a string to a value with facet checking. RDF Literals are converted to Python values using this method; if there is a problem, an exception is raised (and caught higher up to generate an error message).

The method is the equivalent of all the methods in the *DatatypeHandling* module, and is registered to the system run time, as new restricted datatypes are discovered.

(Technically, the registration is done via a `lambda v: _lit_to_value(self,v)` setting from within a *RestrictedDatatype* instance).

Parameters

- **dt** (*RestrictedDatatype*) – Faceted datatype.
- **v** – Literal to be converted and checked.

Raises **ValueError** – Invalid literal value.

1.13 XsdDatatypes

Lists of XSD datatypes and their mutual relationships

Requires: RDFLib, 4.0.0 and higher.

License: This software is available for use under the W3C Software License.

Organization: World Wide Web Consortium

Author: Ivan Herman

See also:

View the source code XsdDatatypes

Python Module Index

o

Index

Symbols

A

C

D

E

F

I

O

P

R

S

U