Project Number:	**215219**
Project Acronym:	**SOA4All**
Project Title:	**Service Oriented Architectures for All**
Instrument:	**Integrated Project**
Thematic Priority:	**Information and Communication Technologies**

D3.2.5 Second Prototype Repository Reasoner for WSML-Core v2.0

Activity N:	Activity 2	
Work Package:	WP3	
Due Date:		28/02/2010
Submission Date:		28/02/2010
Start Date of Project:		01/03/2008
Duration of Project:		36 Months
Organisation Responsible for Deliverable:		UIBK
Revision:		1.1
Author(s):	Daniel Winkler, UIBK Barry Bishop, UIBK	
Reviewer(s):	Barry Norton	

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission)	
RE	Restricted to a group specified by the consortium (including the Commission)	
CO	Confidential, only for members of the consortium (including the Commission)	

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	25/01/2010	First Draft	Daniel Winkler
0.2	15/02/2010	Addition of minor items and proof reading	Barry Bishop
1.0	24/02/2010	Corrections after peer review.	Barry Bishop
1.1	28/02/2010	Format corrections	Julia Wells (ATOS)

Table of Contents

List of Listings

Glossary of Acronyms

Acronym	Definition
D	Deliverable
DL	Description Logic
EC	European Commission
WP	Work Package
WSML	Web Service Modelling Language

Executive summary

In order to automate tasks such as discovery and composition, Semantic Web Services must be described in a well-defined formal language. The Web Services Modelling Language (WSML) is based on the conceptual model of the Web Service Modelling Ontology [2] (WSMO) and as such can be used for modelling all aspects of Web services and associated ontologies. WSMO-Lite and MicroWSMO service descriptions include annotation mechanisms for linking services, operations and message types with entities from ontologies described using WSML.

WSML is actually a family of several language variants, each of which is based upon a different logical formalism. The family of languages are unified under one syntactic umbrella, with a concrete syntax for modelling ontologies, web services, etc, according to the WSMO meta-model, which forms the basis of WSMO-Lite and MicroWSMO.

This deliverable, along with others, describes the second prototype repository reasoner for WSML-Core v2.0, in particular improvements as well as bug fixes to the extensions for *instance equivalence* [1]. The software development process for the reasoning components has been more or less continuous over the last twelve months and many issues and bugs have been discovered and dealt with. However, the main contribution described in this deliverable concerns two algorithms that extend classical semi-naive evaluation for recursive Datalog programs. These extensions are required when processing logical programs that contain rules that infer the equivalence of two objects.

The WSML-Core v2.0 reasoning component of the WSML2Reasoner framework translates ontologies that are described using WSML-Core to Datalog with specific extensions (one of which is rue-head equality). The reasoner used for processing the Datalog representation is IRIS [9], an open-source, Java implementation that is developed as part of the WSML reasoning framework.

1. Introduction

The Web Service Modelling Language WSML is a formal language for the specification of ontologies and different aspects of Web services, based on the conceptual model of WSMO [2]. Several different WSML language variants exist, which are based upon different logical formalisms. The main formalisms exploited for this purpose are Description Logics [11], Logic Programming [5], and the intersection of these two families of logics, namely 'Description Logic Programs' [10], which is the basic of WSML-Core. Furthermore, WSML has been influenced by F-Logic [12] and frame-based representation systems.

This deliverable discusses the implementation of the second prototype repository reasoner for WSML-core v2.0. The WSML-Core v2.0 language aims to provide a minimal but useful expressivity and is inspired by minimal representation from project LarKC[1] and DLP [10]. It belongs to a set of related deliverables, which discuss the second prototype implementations of several WSML 2.0 variants, namely:

- **D3.2.5 Second Prototype Repository Reasoner for WSML-Core v2.0**
- D3.2.6 Second Prototype Rule Reasoner for WSML Rule v2.0
- D3.2.7 Second Prototype for Description Logic Reasoner for WSML DL v2.0

1.1 Purpose and Scope

This document is a progress report on the software implementation for the second prototype repository reasoner for WSML-Core v2.0. Its main audience are developers who wish to integrate the WSML reasoning framework in to their components and others who want to understand some of the issues regarding processing information represented using this formalism.

The semantics of WSML allow the modelling of knowledge using these formal languages with well-defined semantics. In the case of SOA4All, the reasoner is expected to be used for semantic discovery and Web service composition.

Reasoning for WSML-Core v2.0 is performed by transforming the WSML representation to an extended Datalog, which is then processed by a separate reasoning engine. This engine and the transformation process must provide the necessary WSML-Core v2.0 semantics. The default reasoner used by the WSML2Reasoner framework is IRIS, which has been developed internally at UIBK to support several of the WSML variants.

The objective of this deliverable is to provide information about the features of the second prototype repository reasoner for WSML-Core v2.0. In particular, it explains the changes required for the *instance equivalence* feature, i.e. *equality in rule heads*, which was implemented as part of the implementation work of the first prototype WSML-Core v2.0 version in the Datalog reasoner *IRIS*[2] and the *WSML2Reasoner*[3] reasoning framework. This first version exhibited incorrect behaviour in certain circumstances and some unexpected modifications were required to the reasoning algorithms.

[1] LarKC – European Project http://www.larkc.eu/

[2] IRIS Reasoner http://www.iris-reasoner.org

[3] WSML2Reasoner http://www.wsml2reasoner.org/

1.2 Structure of the document

The structure of this deliverable is as follows: Section 2 discusses the actual implementation and its changes according to the Specification. Section 3 describes how to install and use the prototype reasoner for WSML-Core v2.0. The main implementation of the prototype as well as the algorithms extending conventional semi-naive Datalog evaluation are described in Section 4. Section 5 concludes with a short summary of the deliverable and references can be found in Section 6.

2. Reflection on the Specification

The WSML-Core v2.0 language was designed to provide a level of expressivity that intersects both rule-based languages and descriptions logics, i.e. broadly similar to DLP [10]. The result is a language that has minimal, but useful expressivity whose computational requirements scale well in relation to the size of the knowledge base.

WSML-Quark [4] is a lightweight and intuitive language variant that enables the hierarchical organization of concepts. It forms the most basic (and inexpressive) layer of the WSML language variants hierarchy and is suited for simple classification systems.

For compatibility reasons, WSML-Core v2.0 includes instance equality, which allows the inference that two distinct identifiers refer to the same real world object, e.g. that 'FredJones' and 'Mr.F.Jones' are one and the same thing. To accomplish this, the IRIS [9] implementation and the WSML2Reasoner framework have been modified to add new transformation and reasoning behaviour.

The updated implementation described in this report provides the API and behaviour to accurately support the WSML-Core v2.0 language as specified in deliverable *D3.1.2 Defining the features of the WSML-Core v2.0 language*. This implementation will also support reasoning with WSML-Quark ontologies.

3. Installation and Configuration

3.1 Installation

In order to install and configure the reasoning framework, a Java Virtual Machine (version 1.5 or later) is required. The WSML2Reasoner binary distribution can be obtained from sourceforge[4] (or via the WSML2Reasoner homepage[5]).

The latest version of WSML2Reasoner (0.7.0) includes all the latest bundled software including IRIS (version 0.6.0), WSMO4J (version 2.0.1 – a separate branch of the main trunk) and Elly (version 0.1.0).

The WSML2Reasoner source code can be downloaded from the sourceforge subversion repository[6], which provides the features described in this deliverable.

The *WSML2Reasoner* software is licensed under the GNU Lesser GPL (LGPL). However, there are three release variants in accordance with the license agreements for the bundled reasoning engine libraries:

1. *LGPL:* This release includes all the LGPL libraries used by WSML2Reasoner, including the IRIS and PELLET reasoning engines.
2. *GPL:* In addition to the LGPL libraries and packages, this release includes the MINS reasoning engine, which is licensed under the GNU GPL.
3. *Proprietary:* This release version does not include any further libraries or reasoning engines. However, it does include wrapper classes that allow the WSML2Reasoner framework to use the KAON2 reasoning engine.

The package of the WSML2Reasoner framework consists of the following components:

1. wsml2reasoner-src-x.x.x.zip: The source code of the reasoning framework.
2. wsml2reasoner-javadoc-x.x.x.zip: The JavaDoc of the reasoning framework API.
3. wsml2reasoner-x.x.x.jar: The main executable.
4. lib folder: The required and optional libraries.

To use the WSML2Reasoner one has to use the binaries located in the WSML2Reasoner Java archive file or compile the source and add the libraries found in the lib-folder to the classpath.

3.2 Configuration

The WSML2Reasoner framework provides two ways to obtain a reasoner instance. The first method to create a WSML-Core reasoner is by passing a WSML-Core ontology when calling the `createWSMLReasoner` method of the `DefaultWSMLReasonerFactory`. This will determine the according WSML variant of the specified ontology and create a predefined WSML reasoner. However, one can also create a WSML-Core reasoner without specifying an ontology by calling `createCoreReasoner.` Optionally, the underlying Datalog reasoner type can also be specified using the `Map<String, Object> params` parameter.

[4] https://sourceforge.net/projects/wsml2reasoner/files/

[5] http://tools.sti-innsbruck.at/wsml2reasoner/download

[6] svn co https://wsml2reasoner.svn.sourceforge.net/svnroot/wsml2reasoner wsml2reasoner

3.3 Reasoning Example

Listing 1 gives an example Java program that executes a query against a WSML-Core ontology. The ontology is given in Listing 2 (filename '`instance-equality.wsml`'). For the sake of simplicity exceptions are also not handled in the example.

As indicated by the filename, the example ontology shows the use of instance equality in a rule head, as described in detail in Section 4.1.

```java
public class Example {
        public static void main(String[] args) throws Exception {
                // Create a parser and parse the example ontlogy file. For simplicity we do
                // not take care of exceptions at the moment.
                Parser parser = Factory.createParser(null);
                TopEntity[] identifiables = parser
                                .parse(loadFile("instance-equality.wsml"));

                // We can be sure here, that we only parse a single ontology.
                Ontology ontology = (Ontology) identifiables[0];

                // Create a query, that should bind x to both instances A and B.
                String query = "p(?x)";

                // Instantiate the desired reasoner using the default reasoner factory.
                LPReasoner reasoner = DefaultWSMLReasonerFactory.getFactory()
                                .createCoreReasoner(null);

                // Register the ontology.
                reasoner.registerOntology(ontology);

                // Create the logical expression factory.
                LogicalExpressionFactory factory = Factory
                                .createLogicalExpressionFactory(null);

                // Transform the query in string form to a logical expression object.
                LogicalExpression expression = factory.createLogicalExpression(query,
                                ontology);

                // Execute the query and assign the result to ‘bindings’.
                Set<Map<Variable, Term>> bindings = reasoner.executeQuery(expression);
        }
}
```

Listing 1: Reasoning Example

The example WSML-Core v2.0 ontology defined in Listing 2 (`instance-equality.wsml`) can be used to show the instance equality feature. The Ontology defines two concepts with two instances; one of them has an attribute.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-core"
namespace { _"http://simple#"
}
ontology simple

concept C1
concept C2

instance a memberOf C1
       name hasValue aName

instance b memberOf C2

axiom equalInHead definedBy
       a = b :- true.
       p(?x) :- ?x memberOf C2 and ?x[name hasValue aName].
```

Listing 2: WSML-Core v2.0 example file (`instance-equality.wsml`)

Listing 3 gives the results for the query '`?- p(?x)`', which returns both instances since they are set to be equal in the axiom '`equalInHead`'.

```
2 results to the query:
(1) - {?x=http://simple#b}
(2) - {?x=http://simple#a}
```

Listing 3: Results to query specified in Listing 1

4. Software Description

WSML-Quark [4] is a lightweight and intuitive language variant that enables the hierarchical organization of concepts. It forms the most basic layer of the WSML language variants hierarchy. WSML-Core is semantically layered upon WSML Quark, and has been updated (version 2.0) to align with results of ongoing standardization efforts (e.g. OWL 2 RL) as well as research results such as the L2 language, which has similar language features.

The WSML2Reasoner framework is a collection of reasoning components for reasoning with all the WSML language variants. It includes many normalisation and axiomatisation algorithms, along with components for translating between WSML and other established formalisms, e.g. Datalog, Description Logics, OWL, etc.

The Datalog reasoner IRIS is included in this framework and is developed in conjunction with the other components. Therefore, IRIS has also been modified in order to support the new features introduced with the new versions of the WSML language variants[3]. In fact IRIS now supports all the WSML variants (Quark, Core, Flight, Rule and even DL) as will be reported in later deliverables.

The following sections describe some of the more important development activities.

4.1 Equality in rule heads

WSML-Core v2.0 introduces instance equivalence, also known as *equality in rule heads*. In WSML this allows the declaration that different instance identifiers (IRIs) refer to the same object. In Datalog *equality in rule heads* allows the declaration of equivalence between constant terms, such as strings or integers. *Equality in rule heads* has been integrated into the Datalog reasoner *IRIS*. Two approaches have been implemented to realize this feature, a rewriting technique and integrated support for equivalence in rule heads:

- *Rewriting:* For a given Datalog program containing rules with equality in the head, this technique creates new rules to provide support for *equivalence in rule heads*. Firstly, all occurrences of equality in the head of a rule are replaced by a special predicate (in the following examples denoted by *equivalent*). Then, new rules are created to ensure the correct evaluation of rule head equality. Note that rule (1) and (2) are unsafe rules, since the property "*each variable in the rule head appears in a non-negated, ordinary relation*" is violated.

  ```
  (1) equivalent(?X,?X) :- .
  (2) equivalent(?X,?Y) :- ?X = ?Y
  (3) equivalent(?X,?Y) :- equivalent(?Y,?X).
  (4) equivalent(?X,?Y) :- equivalent(?X,?Z), equivalent(?Z,?Y).
  ```

  These rules basically provide the semantics of 'equivalence', i.e. all objects are equivalent to themselves (1), objects that are equal are equivalent (2), equivalence is symmetric (3) and transitive (4).
  The rewriting algorithm then creates new rules for each predicate occurring in the program, this includes both intentional and extensional predicates. The number of new rules depends on the arity of the predicates. For each predicate p this technique creates *n* new rules, where *n* is the arity of p. Assume a predicate `hasName(?X,?Y,?Z)` with arity 3. For this predicate the following three rules are created:

  ```
  (1) hasName(?U,?Y,?Z) :- hasName(?X,?Y,?Z), equivalent(?X,?U).
  ```

```
(2) hasName(?X,?U,?Z) :- hasName(?X,?Y,?Z), equivalent(?Y,?U).
(3) hasName(?X,?Y,?U) :- hasName(?X,?Y,?Z), equivalent(?Z,?U).
```

This may create a large number of additional rules. Furthermore, it is required, that unsafe rules are created. However, an advantage of this approach is, that the resulting program can be evaluated using any Datalog reasoner that supports unsafe rules, regardless of whether the reasoner explicitly supports equivalence in rule heads or not. This brings support for equality in rule heads with reasoners which do not support it.

- *Integration:* Due to the disadvantages of the approach described above (the requirement for the Datalog reasoner to support unsafe rules and rule/complexity explosion), an alternative method for supporting equivalence in rule heads has been integrated into *IRIS* by modifying the way rules are evaluated.
  During evaluation of a Datalog program the reasoner keeps track of all the terms between which equivalence has been inferred. These equivalencies are maintained in a special data structure that keeps track of every equivalence class (set of objects that are equivalent to each other). During rule evaluation, which basically consists of executing natural joins over the predicates found in rule bodies, instances considered to be equal to 1) objects that are physically the same (in memory), 2) objects that are semantically equal according to the definitions of [7] and objects that are equal to objects that are in the same equivalence class. When evaluating rules, this equality is taken into account by the reasoner in order to compute the correct minimal model of the Datalog program. For instance, when using a view `p(?X,'a')` over a relation, the evaluation returns all tuples `(x,y)` where x is some term and y is any term equivalent to `'a'` (note that `'a'` is equivalent to itself).

4.2 Bug in Evaluation

The integrated evaluation strategy had a bug related to direct evaluation of Datalog rules. As example consider the following rule base

```
a = b :- true.
test() :- a = b.
```

for which IRIS did not infer `test()`. The reason was that the updated evaluation strategy had not been considered when evaluating equality of two non-variable terms. Due to optimizations, the dedicated equality built-ins are used to determine equality of concrete terms. This has however the result that two string terms, say `'a'` and `'b'` are not equal when using the built-in equality that checks for string equality.

The bug fix is straight-forward, the strategy for evaluating equality was changed to consider the internally maintained equivalence classes data structure if the built-in equality check fails to infer equality of two terms.

4.3 Stratification

Stratified negation semantics are applied to a rule-set to maintain monotonic behaviour in the presence of default negation. The principle requires that the rules can be grouped in to strata, where for any rule, the positive literals of the rule body have a dependency only on rules in the same or lower strata and the negative literals have a dependency only on rules in lower strata.

However, the ability for a rule to infer the equivalence of to objects means that it has the ability to affect the outcome of any other rule and hence any rule with equality in the head must exist in the lowest stratum, i.e. either it must have no direct or indirect dependency on a negative literal or else the negative literal cannot exist in the head of any other rule. If this is not the case, then the stratified negation semantics cannot be applied.

The stratification algorithms in IRIS have been updated accordingly.

4.4 Semi-naive Evaluation

A more serious problem arises when rule head equality interferes with the evaluation technique "*semi-naive evaluation*", which is the default forward-chaining evaluation technique. Semi-naïve evaluation reduces the number tuples considered during joins by attempting to avoid generating the same inferences in the same way. As such it is an improvement on "*naive evaluation*" which simply evaluates each rule over the entire contents of the relations associated with the rule body predicates. Appendix 7.1 and 7.2 list formal algorithms for naive and semi-naive evaluation, respectively.

The advantage, and thus problem of semi-naive evaluation is that tuples that were already tested for joining, are no more considered in following iterations. Considering ground statements `p(a,b)` and `q(c,d)` and a rule

```
    r(x,z) :- p(x,y), q(y,z).
```

semi-naive evaluation iterates the contents of the relation for p, but does not find anything that joins (and so does not fire the rule). If at some later point in the evaluation `b = c` is inferred, then the `∆Pi`'s do not contain these already considered tuples, thus `p(a,b)` and `q(c,d)` are not joined to infer `r(a,d)`. This problem does not occur if IRIS uses naive evaluation, since this strategy considers all tuples in every iteration.

One workaround is thus to use the naive evaluation strategy as fallback evaluation for the case that a new tuple in the equality relation is computed, i.e., `∆EQUAL` is not empty. This has the effect that the evaluation benefits from the advantages of semi-naive evaluation, but for the case of a computed equivalence the algorithm swaps to a naive evaluation to re-compute using already considered tuples. An even simpler approach would be to always use naive evaluation for the lowest stratum of rules whenever any rule has equality in the head. (As mentioned above, all rules with equality in the head must exist in the lowest stratum.)

A more sophisticated approach is to make use of the `∆EQUAL` relation directly. The idea is to compute relations by means of the semi-naïve evaluation and use `∆EQUAL` for a post-processing step to handle possibly generated equalities. Thus, when joining the predicates `p(x,y)` and `q(y,z)`, the equality relation is used in between the join of p and q. Naively, this results in a rewriting of the join to `p(x,eq1), EQUAL(eq1, eq2), q(eq2,z).` The `EQUAL` relation clearly needs to be defined as reflexive, symmetric and transitive in order to capture all necessary joins.

The rewriting of the rule is correct, but inefficient from an implementation point of view, thus Appendix 7.3 defines a formal approach on how to compute the minimal model of a Datalog program in a more efficient manner. The improvement ignores the implicitly defined equality given by equality to itself (such that the relation is not reflexive) and use the equality relation solely to capture equalities that are implied by the semantics of the Datalog Program.

4.5 W3C XML Schema Datatypes

Even though [1] does not mention built-in datatypes, the WSML specification [6] does and thus the underlying reasoner needs updates in order to be compliant to the current working draft (version 1.1 part 2) of the W3C XML Schema Definition Language (XSD) [7].

For reasons of backward compatibility the changes in the definition required the following updates to the original implementation based on XML Schema Definition Language version 1 part 2:

- Implementation of built-in datatypes `yearMonthDuration` and `dayTimeDuration`
- Various updates to Date/time Datatypes
- The `rdf:text` literal was renamed to `rdf:PlainLiteral` [8]

These changes are in addition to the new datatypes specified in [13].

The remaining issue is related to the notions of "equality" and "identity" for datatypes, the definition of float and double data values serves as example for the distinction: "The (numeric) equality of values is now distinguished from the identity of the values themselves; this allows float and double to treat positive and negative zero as distinct values, but nevertheless to treat them as equal for purposes of bounds checking. This allows a better alignment with the expectations of users working with IEEE floating-point binary numbers" [7].

5. Conclusions

The introduction of support for rule-head equality in IRIS brought some unexpected computational issues. This deliverable has discussed several bugs that have arisen while implementing this extension. The main focus, however, is the evaluation strategy that is used for evaluating Datalog programs that make use of rule-head equality. Semi-naïve evaluation, which results in notable evaluation speed-up for most Datalog programs, does not behave correctly and extensions to this evaluation strategy have been devised. The easier and more intuitive solution is to use naïve evaluation whenever rule-head equality occurs. The more complex approach, which needs to be evaluated for performance, utilizes the idea of semi-naïve evaluation and enhances it by post-processing to retain correctness.

Further improvements for support for XML Schema datatypes have been implemented.

6. References

[1] Unel, G., Keller, U., Fischer, F. and Bishop, B., SOA4All deliverable “D3.1.2 Defining the Features of the WSML-Core v2.0 Language”, 2009.

[2] Roman, D., Lausen, H. and Keller, U., “Web Service Modeling Ontology (WSMO)” WSMO Working Draft, 2004.

[3] Pressnig, M., SOA4All deliverable “D3.2.2 First Prototype Reasoner for WSML-Core v2.0”, 2009.

[4] Unel, G., Keller, U., Fischer, F. and Bishop, B., SOA4All deliverable “D3.1.1 Defining the Features of the WSML-Quark Language”, 2009.

[5] Ullman, J. D., "Principles of Database and Knowledge-Base Systems”, vol. I. Chapter 3 (Logic as a Data Model), 1988.

[6] The WSML Working Group. "D16.1v1.0 WSML Language Reference”, 2008.

[7] Peterson, D., Gao, S., Malhotra, A., Sperberg-McQueen, C. M., Thompson, H. S. “W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes”, W3C Working Draft, 3 December 2009.

[8] Bao, J., Hawke, S., Motik, B., Patel-Schneider, P. F., Polleres, A., "rdf:PlainLiteral: A Datatype for RDF Plain Literals”, W3C Recommendation, 27 October 2009.

[9] The IRIS Datalog reasoner website: http://www.iris-reasoner.org/

[10] Grosof, B. N., Horrocks, I., Volz, R., and Decker, S. 2003. Description logic programs: combining logic programs with description logic. In Proceedings of the 12th international Conference on World Wide Web (Budapest, Hungary, May 20 - 24, 2003). WWW '03. ACM, New York, NY, 48-57. DOI= http://doi.acm.org/10.1145/775152.775160

[11] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., Patel-Schneider, P.F. (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, 2003.

[12] Kifer, M., Lausen, G., and Wu, J. 1995. Logical foundations of object-oriented and frame-based languages. J. ACM 42, 4 (Jul. 1995).

[13] Toma, I., Bishop, B., Fischer, F, SOA4All deliverable “D3.1.4 Defining the features of the WSML-Rule v2.0 language”, 2009.

7. Appendix

7.1 Naive Datalog Evaluation

Algorithm 1 [5]: Evaluation of Datalog Equations.

INPUT: A collection of datalog rules with EDB predicates `r1, ..., rk` and IDB predicates `p1, ..., pm`. Also, a list of relations `R1, ..., Rk` to serve as values of the EDB predicates.

OUTPUT: The last fixed point solution to the datalog equations obtained from these rules.

METHOD: Begin by setting up the equations for the rules. These equations have variables `P1, ..., Pm` corresponding to the IDB predicates, and the equation for `Pi` is `Pi = EVAL(pi, R1, ..., Rk, P1, ..., Pm)`. We then initialize each `Pi`'s. When no more tuples can be added to any IDB relation, we have our desired output. The details are given in the following program:

```
for i := 1 to m do
  Pi := Ø;
repeat
  for i := 1 to m do
    Qi := Pi; /* save old values of Pi's */
  for i := 1 to m do
    Pi := EVAL(pi, R1, ..., Rk, Q1, ..., Qm);
until Pi = Qi for all i, 1 <= i <= m;
output Pi's
```

The expression `EVAL-RULE(r, R1, ..., Rn)` computes for a rule `r` from the relations `R1, ..., Rn` a relation `R(X1, ..., Xn)` with all and only the tuple `(a1, ..., am)` such that, when we substitute `aj` for `Xj`, `1 <= j <= m`, all the sub-goals `S1, ..., Sn` are made true.

`EVAL` is defined as the union of `EVAL-RULE(...)` for each of the rules `r` for a predicate `pi`, projected onto the variables of the head.

7.2 Semi-Naive Datalog Evaluation

Algorithm 2 [5]: Semi-Naive Evaluation of Datalog Equations.

INPUT: A collection of rectified datalog rules with EDB predicates `r1, ..., rk` and IDB predicates `p1, ..., pm`. Also, a list of relations `R1, ..., Rk` to serve as values of the EDB predicates.

OUTPUT: The last fixed point solution to the datalog equations obtained from these rules.

METHOD: We use `EVAL` once to get the computation of relations started, and then use `EVAL-INCR` repeatedly on incremental IDB relations. The computation is shown in the following program, where for each IDB predicate `pi`, there is an associated relation `Pi` that holds all the tuples, and there is an incremental relation `∆Pi` that holds only the tuples added on the previous round.

```
for i := 1 to m do begin
  △Pi := EVAL(pi, R1, ..., Rk, Ø, ..., Ø);
  Pi := △Pi;
end;
repeat
  for i := 1 to m do
    △Qi := △Pi; /* save old △P's */
  for i := 1 to m do begin
    △Pi := EVAL-INCR(pi, R1, ..., Rk, P1, ..., Pm, △Q1, ..., △Qm);
    △Pi := △Pi - Pi; /* remove "new" tuples that appeared before */
  end;
  for i := 1 to m do

    Pi := Pi ∪ △Pi; /* save old △P's */
until △Pi = Ø for all i;
output Pi's
```

The **incremental relation** `EVAL-RULE-INCR` for rule `r` is the union of the n relations

`EVAL-RULE(r, R1, ..., Ri-1, △Ri, Ri+1, ..., Rn)`

for `1 <= i <= n`. Like with `EVAL`, the expression `EVAL-INCR` is defined as the union over all rules `r` for a predicate `pi`.

7.3 Improved Semi-Naive Datalog Evaluation

Algorithm 3: Improved Semi-Naive Evaluation of Datalog Equations.

INPUT: A collection of rectified datalog rules that may assert rule-head equality with EDB predicates `r1, ..., rk` and IDB predicates `p1, ..., pm`. Also, a list of relations `R1, ..., Rk` to serve as values of the EDB predicates.

OUTPUT: The last fixed point solution to the datalog equations obtained from these rules.

METHOD: We use EVAL once to get the computation of relations started, and then use `EVAL-INCR` followed by the defined function `EVAL-EQUAL` repeatedly on incremental IDB relations. The computation is shown in the following program, where for each IDB predicate `pi`, there is an associated relation `Pi` that holds all the tuples, and there is an incremental relation `△Pi` that holds only the tuples added on the previous round. Additionally, the relation `△EQUAL` holds all asserted instance equalities in a symmetric and transitive manner. The relation `EQUAL` could be initialized as a reflexive set over all instances; this is however avoided as self-equality is implicitly captured by the `EVAL` and `EVAL-INCR` operations.

```
for i := 1 to m do begin
  ΔPi := EVAL(pi, R1,..., Rk, Ø,..., Ø);
  Pi := ΔPi;
end;
EQUAL := Ø; /* initialize without reflexive statements */
repeat
  for i := 1 to m do
    ΔQi := ΔPi; /* save old ΔP's */
  EQUAL := EQUAL + ΔEQUAL;
  ΔEQUAL := EVAL-INCR(peq, R1,..., Rk, P1,..., Pm, ΔQ1,..., ΔQm,
            EQUAL);
  ΔEQUAL := truncate(ΔEQUAL); /* remove reflexive statements */
  /* EQUAL and peq are excluded in the following from Pi and pi */
  for i := 1 to m do begin
    ΔPi := EVAL-INCR(pi, R1,..., Rk, P1,..., Pm, ΔQ1,..., ΔQm);
    ΔPi := ΔPi + EVAL-EQUAL(pi, R1,..., Rk, P1,..., Pm,
           ΔQ1,..., ΔQm, EQUAL);
    ΔPi := ΔPi - Pi; /* remove "new" tuples that appeared before */
  end;
  for i := 1 to m do
    Pi := Pi + ΔPi; /* save old ΔP's */
until ΔPi = Ø for all i and ΔEQUAL = Ø;
output Pi's and EQUAL
```

The algorithm makes use of the already defined EVAL-INCR to compute new tuples in the EQUAL relation. This is done by using the equality predicate peq and the equality relation EQUAL. Note that the resulting relation ΔEQUAL is truncated in the subsequent step, i.e. all reflexive statements, e.g. EQUAL(a,a), are removed from the relation as they are handled implicitly by EVAL-INCR expression. The iteration over all predicates and resulting relations excludes the EQUAL relation and peq predicate, they are handled explicitly in the preceding step.

The algorithm applies a post-processing of every relation by the expression EVAL-EQUAL, which is based on an extended version of EVAL-RULE-INCR that does not join relations directly, rather via an intermediate EQUAL relation. This has the effect, that joins contingent upon instance equality, which were not considered in the semi-naive evaluation, are handled by the algorithm. Additionally, this approach is an improvement to naive evaluation, since the join of two relations is reduced from a full join to a join of those facts that have equal instances in the Datalog program.

The algorithm terminates for the case that all ΔPi and additionally ΔEQUAL are empty. ΔEQUAL is thus only needed to decide the termination condition of the algorithm.