Abductive Logic Programming for Datalog$^{\pm}$ ontologies

Marco Gavanelli[1], Evelina Lamma[1], Fabrizio Riguzzi[2], Elena Bellodi[1], Riccardo Zese[1], and Giuseppe Cota[1]

[1] Dipartimento di Ingegneria – University of Ferrara
[2] Dipartimento di Matematica e Informatica – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy [name.surname]@unife.it

Abstract. Ontologies are a fundamental component of the Semantic Web since they provide a formal and machine manipulable model of a domain. Description Logics (DLs) are often the languages of choice for modeling ontologies. Great effort has been spent in identifying decidable or even tractable fragments of DLs. Conversely, for knowledge representation and reasoning, integration with rules and rule-based reasoning is crucial in the so-called Semantic Web stack vision. Datalog$^{\pm}$ is an extension of Datalog which can be used for representing lightweight ontologies, and is able to express the DL-Lite family of ontology languages, with tractable query answering under certain language restrictions. In this work, we show that Abductive Logic Programming (ALP) is also a suitable framework for representing Datalog$^{\pm}$ ontologies, supporting query answering through an abductive proof procedure, and smoothly achieving the integration of ontologies and rule-based reasoning. In particular, we consider an Abductive Logic Programming framework named $\mathcal{S}$CIFF and derived from the IFF abductive framework, able to deal with existentially (and universally) quantified variables in rule heads, and Constraint Logic Programming constraints. Forward and backward reasoning is naturally supported in the ALP framework. The $\mathcal{S}$CIFF language smoothly supports the integration of rules, expressed in a Logic Programming language, with Datalog$^{\pm}$ ontologies, mapped into $\mathcal{S}$CIFF (forward) integrity constraints. The main advantage is that this integration is achieved within a single language, grounded on abduction in computational logic.

1 Introduction

The main idea of the Semantic Web is making information available in a form that is understandable and automatically manageable by machines [21]. Ontologies are engineering artefacts consisting of a vocabulary describing some domain, and an explicit specification of the intended meaning of the vocabulary (i.e., how concepts should be classified), possibly together with constraints capturing additional knowledge about the domain. Ontologies therefore provide a formal and machine manipulable model of a domain, and this justifies their use in the Semantic Web.

In order to realize this vision, the W3C has supported the development of a family of knowledge representation formalisms of increasing complexity for defining ontologies, called Web Ontology Language (OWL). Ontologies are a fundamental component of the Semantic Web, and Description Logics (DLs) are often the languages of choice for modeling them.

Several DL reasoners, such us Pellet [32], RacerPro [19] and HermiT [31], are used to extract implicit information from the modeled ontologies, and most of them implement the tableau algorithm in a procedural language. Nonetheless, some tableau expansion rules are non-deterministic, thus requiring to implement a search strategy in an or-branching search space. A different approach is to provide a Prolog-based implementation for the tableau expansion rules [34].

Extensive work focused on developing tractable DLs, identifying the *DL-Lite* family [14], for which answering conjunctive queries is in AC_0 in data complexity.

In a related research direction, [11] proposed Datalog$^{\pm}$, an extension of Datalog with existential rules for defining ontologies. Datalog$^{\pm}$ can be used for representing lightweight ontologies, and encompasses the DL-Lite family [10]. By suitably restricting the language syntax and adopting appropriate syntactic conditions, also Datalog$^{\pm}$ achieves tractability [9].

In this work we consider the Datalog$^{\pm}$ language and show how ontologies expressed in this language can be also modeled in an Abductive Logic Programming (ALP) framework, where query answering is supported by the underlying ALP proof procedure. ALP has been proved a powerful tool for knowledge representation and reasoning [24], taking advantage from ALP operational support as (static or dynamic) verification tool. ALP languages are usually equipped with a declarative (model-theoretic) semantics, and an operational semantics given in terms of a proof-procedure. Several abductive proof procedures have been defined (both backward, forward, and a mix of the two such), with many different applications (diagnosis, monitoring, verification, etc.). Among them, the IFF abductive proof-procedure [17] was proposed to deal with forward rules, and with non-ground abducibles. This proof procedure has been later extended [4], and the resulting proof procedure, named $\mathcal{S}$CIFF, can deal with both existentially and universally quantified variables in rule heads, and Constraint Logic Programming (CLP) constraints [23]. The resulting system was used for modeling and implementing several knowledge representation frameworks, such as deontic logic [6], normative systems, interaction protocols for multi-agent systems [7], Web services choreographies [2], etc.

Here we concentrate on Datalog$^{\pm}$ ontologies, and show how an ALP language enriched with quantified variables (existential to our purposes) can be a useful knowledge representation and reasoning framework for them. We do not focus here on complexity results of the overall system, which is, however, not tractable.

Forward and backward reasoning is naturally supported by the ALP proof procedure, and the considered $\mathcal{S}$CIFF language smoothly supports the integration of rules, expressed in a Logic Programming language, with ontologies expressed in Datalog$^{\pm}$. In fact, $\mathcal{S}$CIFF allows us to map Datalog$^{\pm}$ ontologies into the forward integrity constraints on which it is based.

In the following, Section 2 briefly introduces Datalog$^{\pm}$. Section 3 introduces Abductive Logic Programming and the $\mathcal{S}$CIFF language, with a mention to its abductive proof procedure. Section 4 shows how the considered Datalog$^{\pm}$ language can be mapped into $\mathcal{S}$CIFF, and the kind of queries that the abductive proof procedure can handle. Section 5 illustrates related work. Section 6 concludes the paper, and outlines future work.

2 Datalog$^{\pm}$

Datalog$^{\pm}$ extends Datalog by allowing existential quantifiers, the equality predicate and the truth constant *false* in rule heads. Datalog$^{\pm}$ can be used for representing lightweight ontologies and is able to express the DL-Lite family of ontology languages [10]. By suitably restricting the language syntax, Datalog$^{\pm}$ achieves tractability [9].

In order to describe Datalog$^{\pm}$, let us assume (i) an infinite set of data constants Δ, (ii) an infinite set of labeled nulls Δ_N (used as "fresh" Skolem terms), and (iii) an infinite set of variables Δ_V. Different constants represent different values (unique name assumption), while different nulls may represent the same value. We assume a lexicographic order on $\Delta \cup \Delta_N$, with every symbol in Δ_N following all symbols in Δ. We denote by $\mathbf{X}$ vectors of variables $X_1, \ldots, X_k$ with $k \geq 0$. A relational schema $\mathcal{R}$ is a finite set of relation names (or predicates). A term t is a constant, null or variable. An atomic formula (or atom) has the form $p(t_1, \ldots, t_n)$, where p is an n-ary predicate, and $t_1, \ldots, t_n$ are terms. A database D for $\mathcal{R}$ is a possibly infinite set of atoms with predicates from $\mathcal{R}$ and arguments from $\Delta \cup \Delta_N$. A conjunctive query (CQ) over $\mathcal{R}$ has the form $q(\mathbf{X}) = \exists \mathbf{Y} \Phi(\mathbf{X}, \mathbf{Y})$, where $\Phi(\mathbf{X}, \mathbf{Y})$ is a conjunction of atoms having as arguments variables $\mathbf{X}$ and $\mathbf{Y}$ and constants (but no nulls). A Boolean CQ (BCQ) over $\mathcal{R}$ is a CQ having head predicate q of arity 0 (i.e., no variables in $\mathbf{X}$).

We often write a BCQ as the set of all its atoms, having constants and variables as arguments, and omitting the quantifiers. Answers to CQs and BCQs are defined via homomorphisms, which are mappings $\mu : \Delta \cup \Delta_N \cup \Delta_V \rightarrow \Delta \cup \Delta_N \cup \Delta_V$ such that (i) $c \in \Delta$ implies $\mu(c) = c$, (ii) $c \in \Delta_N$ implies $\mu(c) \in \Delta \cup \Delta_N$, and (iii) μ is naturally extended to term vectors, atoms, sets of atoms, and conjunctions of atoms. The set of all answers to a CQ $q(\mathbf{X}) = \exists \mathbf{Y} \Phi(\mathbf{X}, \mathbf{Y})$ over a database D, denoted $q(D)$, is the set of all tuples $\mathbf{t}$ over Δ for which there exists a homomorphism $\mu : \mathbf{X} \cup \mathbf{Y} \rightarrow \Delta \cup \Delta_N$ such that $\mu(\Phi(\mathbf{X}, \mathbf{Y})) \subseteq D$ and $\mu(\mathbf{X}) = \mathbf{t}$. The answer to a BCQ $q = \exists \mathbf{Y} \Phi(\mathbf{Y})$ over a database D, denoted $q(D)$, is Yes, denoted $D \models q$, iff there exists a homomorphism $\mu : \mathbf{Y} \rightarrow \Delta \cup \Delta_N$ such that $\mu(\Phi(\mathbf{Y})) \subseteq D$, i.e., if $q(D) \neq \emptyset$.

Given a relational schema $\mathcal{R}$, a *tuple-generating dependency* (or TGD) F is a first-order formula of the form $\forall \mathbf{X} \forall \mathbf{Y} \Phi(\mathbf{X}, \mathbf{Y}) \rightarrow \exists \mathbf{Z} \Psi(\mathbf{X}, \mathbf{Z})$, where $\Phi(\mathbf{X}, \mathbf{Y})$ and $\Psi(\mathbf{X}, \mathbf{Z})$ are conjunctions of atoms over $\mathcal{R}$, called the *body* and the *head* of F, respectively. Such F is satisfied in a database D for $\mathcal{R}$ iff, whenever there exists a homomorphism h such that $h(\Phi(\mathbf{X}, \mathbf{Y})) \subseteq D$, there exists an extension h' of h such that $h'(\Psi(\mathbf{X}, \mathbf{Z})) \subseteq D$. We usually omit the universal quantifiers in

TGDs. A TGD is *guarded* iff it contains an atom in its body that involves all variables appearing in the body.

Query answering under TGDs is defined as follows. For a set of TGDs T on $\mathcal{R}$, and a database D for $\mathcal{R}$, the set of models of D given T, denoted $mods(D,T)$, is the set of all (possibly infinite) databases B such that $D \subseteq B$ and every $F \in T$ is satisfied in B. The set of answers to a CQ q on D given T, denoted $ans(q,D,T)$, is the set of all tuples $\mathbf{t}$ such that $\mathbf{t} \in q(B)$ for all $B \in mods(D,T)$. The answer to a BCQ q over D given T is Yes, denoted $D \cup T \models q$, iff $B \models q$ for all $B \in mods(D,T)$.

A Datalog$^{\pm}$ theory may contain also *negative constraints* (or NC), which are first-order formulas of the form $\forall \mathbf{X} \Phi(\mathbf{X}) \rightarrow \bot$, where $\Phi(\mathbf{X})$ is a conjunction of atoms (not necessarily guarded). The universal quantifiers are usually left implicit.

Equality-generating dependencies (or EGDs) are the third component of a Datalog$^{\pm}$ theory. An EGD F is a first-order formula of the form $\forall \mathbf{X} \Phi(\mathbf{X}) \rightarrow X_i = X_j$, where $\Phi(\mathbf{X})$, called the *body* of F, is a conjunction of atoms, and X_i and X_j are variables from $\mathbf{X}$. We call $X_i = X_j$ the *head* of F. Such F is satisfied in a database D for $\mathcal{R}$ iff, whenever there exists a homomorphism h such that $h(\Phi(\mathbf{X})) \subseteq D$, it holds that $h(X_i) = h(X_j)$. We usually omit the universal quantifiers in EGDs.

The *chase* is a bottom-up procedure for deriving atoms entailed by a database and a Datalog$^{\pm}$ theory. The chase works on a database through the so-called TGD and EGD chase rules.

The TGD chase rule is defined as follows. Given a relational database D for a schema $\mathcal{R}$, and a TGD F on $\mathcal{R}$ of the form $\forall \mathbf{X} \forall \mathbf{Y} \Phi(\mathbf{X}, \mathbf{Y}) \rightarrow \exists \mathbf{Z} \Psi(\mathbf{X}, \mathbf{Z})$, F is *applicable to* D if there is a homomorphism h that maps the atoms of $\Phi(\mathbf{X}, \mathbf{Y})$ to atoms of D. Let F be applicable and h_1 be a homomorphism that extends h as follows: for each $X_i \in \mathbf{X}$, $h_1(X_i) = h(X_i)$; for each $Z_j \in \mathbf{Z}$, $h_1(Z_j) = z_j$, where z_j is a "fresh" null, i.e., $z_j \in \Delta_N, z_j \notin D$, and z_j lexicographically follows all other labeled nulls already introduced. The result of the application of the TGD chase rule for F is the addition to D of all the atomic formulas in $h_1(\Psi(\mathbf{X}, \mathbf{Z}))$ that are not already in D.

The EGD chase rule is defined as follows. An EGD F on $\mathcal{R}$ of the form $\Phi(\mathbf{X}) \rightarrow X_i = X_j$ is *applicable to* a database D for $\mathcal{R}$ iff there exists a homomorphism $h : \Phi(\mathbf{X}) \rightarrow D$ such that $h(X_i)$ and $h(X_j)$ are different and not both constants. If $h(X_i)$ and $h(X_j)$ are different constants in Δ, then there is a *hard violation* of F. Otherwise, the result of the application of F to D is the database $h(D)$ obtained from D by replacing every occurrence of $h(X_i)$ with $h(X_j)$ if $h(X_i)$ precedes $h(X_j)$ in the lexicographic order, and every occurrence of $h(X_j)$ with $h(X_i)$ if $h(X_j)$ precedes $h(X_i)$ in the lexicographic order.

The chase algorithm consists of an exhaustive application of the TGD and EGD chase rules that may lead to an infinite result. The chase rules are applied iteratively: in each iteration (1) a single TGD is applied once and then (2) the EGDs are applied until a fix point is reached. EGDs are assumed to be separable [12]. Intuitively, separability holds whenever: (i) if there is a hard violation of an

EGD in the chase, then there is also one on the database w.r.t. the set of EGDs alone (i.e., without considering the TGDs); and (ii) if there is no hard violation, then the answers to a BCQ w.r.t. the entire set of dependencies equals those w.r.t. the TGDs alone (i.e., without the EGDs).

The two problems of CQ and BCQ evaluation under TGDs and EGDs are LOGSPACE-equivalent [11]. Moreover, query answering under TGDs is equivalent to query answering under TGDs with only single atoms in their heads [9]. Henceforth, we focus only on the BCQ evaluation problem and we assume that every TGD has a single atom in its head. A BCQ q on a database D, a set T_T of TGDs and a set T_E of EGDs can be answered by performing the chase and checking whether the query is entailed by the extended database that is obtained. In this case we write $D \cup T_T \cup T_E \models q$.

Example 1. Let us consider the following ontology for a real estate information extraction system, a slight modification of the one presented in Gottlob et al. [18]:

$F_1 = ann(X, label), ann(X, price), visible(X) \rightarrow priceElem(X)$
If X is annotated as a label, as a price and is visible, then it is a price element.

$F_2 = ann(X, label), ann(X, priceRange), visible(X) \rightarrow priceElem(X)$
If X is annotated as a label, as a price range, and is visible, then it is a price element.

$F_3 = priceElem(E), group(E, X) \rightarrow forSale(X)$
If E is a price element and is grouped with X, then X is for sale.

$F_4 = forSale(X) \rightarrow \exists P\ price(X, P)$
If X is for sale, then there exists a price for X.

$F_5 = hasCode(X, C), codeLoc(C, L) \rightarrow loc(X, L)$
If X has postal code C, and C's location is L, then X's location is L.

$F_6 = hasCode(X, C) \rightarrow \exists L\ codeLoc(C, L), loc(X, L)$
If X has postal code C, then there exists L such that C has location L and so does X.

$F_7 = loc(X, L1), loc(X, L2) \rightarrow L1 = L2$
If X has the locations $L1$ and $L2$, then $L1$ and $L2$ are the same.

$F_8 = loc(X, L) \rightarrow advertised(X)$
If X has a location L then X is advertised.

Suppose we are given the database

$$codeLoc(ox1, central), codeLoc(ox1, south), codeLoc(ox2, summertown)$$
$$hasCode(prop1, ox2), ann(e1, price), ann(e1, label), visible(e1),$$
$$group(e1, prop1)$$

The atomic BCQs $priceElem(e1)$, $forSale(prop1)$ and $advertised(prop1)$ evaluate to true, while the CQ $loc(prop1, L)$ has answers $q(L) = \{summertown\}$. In fact, even if $loc(prop1, z_1)$ with $z_1 \in \Delta_N$ is entailed by formula F_5, formula F_7 imposes that $summertown = z_1$. If F_7 were absent then $q(L) = \{summertown, z_1\}$.

Answering BCQs q over databases and ontologies containing NCs can be performed by first checking whether the BCQ $\Phi(\mathbf{X})$ evaluates to false for each NC of the form $\forall \mathbf{X} \Phi(\mathbf{X}) \rightarrow \bot$. If one of these checks fails, then the answer to the original BCQ q is positive, otherwise the negative constraints can be simply ignored when answering the original BCQ q.

A guarded Datalog$^{\pm}$ ontology is a quadruple (D, T_T, T_C, T_E) consisting of a database D, a finite set of guarded TGDs T_T, a finite set of negative constraints T_C and a finite set of EGDs T_E that are separable from T_T. The data complexity (i.e., the complexity where both the query and the theory are fixed) of evaluating BCQs relative to a guarded Datalog$^{\pm}$ theory is polynomial [9].

In the case in which the EGDs are key dependencies and the TGDs are inclusion dependencies, Calì et al. [13] proposed a backward chaining algorithm for answering BCQ. A *key dependency* κ is a set of EGDs of the form

$$\{r(\mathbf{X}, Y_1, \ldots, Y_m), r(\mathbf{X}, Y_1', ..., Y_m') \rightarrow Y_i = Y_i'\}_{1 \leq i \leq m}$$

A TGD of the form $r_1(\mathbf{X}, \mathbf{Y}) \rightarrow \exists \mathbf{Z} r_2(\mathbf{X}, \mathbf{Z})$, where r_1 and r_2 are predicate names and no variable appears more than once in the body nor in the head, is called an *inclusion dependency*. The key dependencies must not interact with the inclusion dependencies, similarly to the semantic separability condition mentioned above for TGDs and EGDs. In this case once it is known that no hard violation occurs, queries can be answered by considering the inclusion dependencies only, ignoring the key dependencies. A necessary and sufficient syntactic condition for non interaction is based on the construction of CD-graphs [13].

3 ALP and its proof procedure

Abductive Logic Programming (ALP, for short) is a family of programming languages that integrate abductive reasoning into logic programming. An ALP program is a logic program, consisting of a set of clauses, that can contain in the body some distinguished predicates, belonging to a set $\mathcal{A}$ and called *abducibles*. The aim is finding a set of abducibles **EXP**, built from symbols in $\mathcal{A}$ that, together with the knowledge base, is an explanation for a given known effect (also called *goal* $\mathcal{G}$):

$$KB \cup \mathbf{EXP} \models \mathcal{G}. \tag{1}$$

Also, **EXP** should satisfy a set of logic formulae, called *Integrity Constraints* IC:

$$KB \cup \mathbf{EXP} \models IC. \tag{2}$$

E.g., a knowledge base might contain a set of rules stating that a person is a nature lover

$$\begin{aligned} &natureLover(X) \leftarrow \mathbf{hasAnimal}(X, Y), \mathbf{pet}(Y). \\ &natureLover(X) \leftarrow \mathbf{biologist}(X). \end{aligned}$$

From this knowledge base one can infer, e.g., that each person who owns a pet is a nature lover. However, in some cases we might have the information that

kevin is a nature lover, and wish to infer more information about him. In such a case we might label predicates **hasAnimal**, **pet** and **biologist** as abducible (in the following, abducible predicates are written in **bold**) and apply an abductive proof procedure to the knowledge base. Two explanations are possible: either there exists an animal that is owned by *kevin* and that is a pet:

$$(\exists Y) \qquad \mathbf{hasAnimal}(kevin, Y), \mathbf{pet}(Y)$$

or *kevin* is a biologist:

$$\mathbf{biologist}(kevin)$$

We see that the computed answer includes abduced atoms, which can contain variables.

Integrity constraints can help reducing the number of computed explanations, ruling out those that are not possible. For example, the following integrity constraint states that to become a biologist one needs to be at least 25 years old:

$$\mathbf{biologist}(X), \mathbf{age}(X, A) \rightarrow A \geq 25$$

We might know that *kevin* is a child, and have a definition of the predicate *child*:

$$child(X) \leftarrow \mathbf{age}(X, A), A < 10.$$

In this example we see the usefulness of constraints as in Constraint Logic Programming [23]: the symbols $<, \geq, ...$ are handled as constraints, i.e., they are not predicates defined in a knowledge base, but they associate a numeric domain to the involved variables and restrict it according to constraint propagation. Now, the goal $natureLover(kevin), child(kevin)$ returns only one possible explanation:

$$(\exists Y)(\exists A) \qquad \mathbf{hasAnimal}(kevin, Y), \mathbf{pet}(Y), \mathbf{age}(kevin, A) \qquad A < 10$$

since the option that *kevin* is a biologist is ruled out. Note that we do not need to know the exact age of *kevin* to rule out the biologist hypothesis.

$\mathcal{S}$CIFF [4] is a language in the ALP class, originally designed to model and verify interactions in open societies of agents [7], and it is an extension of the IFF proof-procedure [17]. As in the IFF language, it considers forward integrity constraints of the form

$$body \rightarrow head$$

where the *body* is a conjunction of literals and the head is a disjunction of conjunctions of literals. While in the IFF the literals can be built only on defined or abducible predicates, in $\mathcal{S}$CIFF they can also be CLP constraints, occurring events (only in the body), or positive and negative expectations.

Definition 1. *A* $\mathcal{S}$CIFF Program *is a pair* $\langle KB, \mathcal{IC} \rangle$ *where KB is a set of clauses and* $\mathcal{IC}$ *is a set of forward rules called* Integrity Constraints *(ICs, for short in the following).*

$\mathcal{S}$CIFF considers a (possibly dynamically growing) set of facts (named event set) **HAP**, that contains ground atoms $\mathbf{H}(Event[,Time])$. This set can grow dynamically, during the computation, thus implementing a dynamic acquisition of events. Some distinguished abducibles are called *expectations*. A *positive expectation*, written $\mathbf{E}(Event[,Time])$ means that a corresponding event $\mathbf{H}(Event[,Time])$ is expected to happen, while $\mathbf{EN}(Event[,Time])$ is a *negative expectation*, and requires events $\mathbf{H}(Event[,Time])$ not to happen. To simplify the notation, we will omit the $Time$ argument from events and expectations when not needed, as it is for our purposes.

While events are ground atoms, expectations can contain variables. In positive expectations all variables are existentially quantified (expressing the idea that a single event is enough to support them), while negative expectations are universally quantified, so that any event matching with a negative expectation leads to inconsistency with the current hypothesis. CLP [23] constraints can be imposed on variables. The computed answer includes in general three elements: a substitution for the variables in the goal (as usual in Prolog), the constraint store (as in CLP), and the set **EXP** of abduced literals.

The declarative semantics of $\mathcal{S}$CIFF includes the classic conditions of abductive logic programming

$$KB \cup \mathbf{HAP} \cup \mathbf{EXP} \models \mathcal{G}$$
$$KB \cup \mathbf{HAP} \cup \mathbf{EXP} \models \mathcal{IC}$$

plus specific conditions to support the confirmation of expectations.

Positive expectations are confirmed if

$$KB \cup \mathbf{HAP} \cup \mathbf{EXP} \models \mathbf{E}(X) \rightarrow \mathbf{H}(X),$$

while negative expectations are confirmed (or better they are not violated) if

$$KB \cup \mathbf{HAP} \cup \mathbf{EXP} \models \mathbf{EN}(X) \wedge \mathbf{H}(X) \rightarrow \mathit{false}.$$

The declarative semantics of $\mathcal{S}$CIFF also requires that the same event cannot be expected both to happen and not to happen

$$KB \cup \mathbf{HAP} \cup \mathbf{EXP} \models \mathbf{E}(X) \wedge \mathbf{EN}(X) \rightarrow \mathit{false} \tag{3}$$

The $\mathcal{S}$CIFF proof-procedure is a rewriting system that defines a proof tree, whose nodes represent states of the computation. A set of transitions rewrite a node into one or more children nodes. $\mathcal{S}$CIFF inherits the transitions of the IFF proof-procedure [17], and extends it in various directions. We recall the basics of $\mathcal{S}$CIFF; a complete description is in [4], with proofs of soundness, completeness, and termination. An efficient implementation of $\mathcal{S}$CIFF is described in [5].

Each node of the proof is a tuple $T \equiv \langle R, CS, PSIC, \mathbf{EXP} \rangle$, where R is the resolvent, CS is the CLP constraint store, PSIC is a set of implications (called *Partially Solved Integrity Constraints*) derived from propagation of integrity constraints, and **EXP** is the current set of abduced literals. The main transitions, inherited from the IFF are:

Unfolding replaces a (non abducible) atom with its definitions;
Propagation if an abduced atom $\mathbf{a}(X)$ occurs in the condition of an IC (e.g., $\mathbf{a}(Y) \rightarrow p$), the atom is removed from the condition (generating $X = Y \rightarrow p$);
Case Analysis given an implication containing an equality in the condition (e.g., $X = Y \rightarrow p$), generates two children in logical or (in the example, either $X = Y$ and p, or $X \neq Y$);
Equality rewriting rewrites equalities as in the Clark's equality theory;
Logical simplifications other simplifications like $(\text{true} \rightarrow A) \Leftrightarrow A$, etc.

$\mathcal{S}$CIFF includes also the transitions of CLP [23] for constraint solving. Finally, in this paper we consider the *generative version* of $\mathcal{S}$CIFF, previously called also g-$\mathcal{S}$CIFF [3], in which also the **H** events in the set **HAP** are considered as abducibles, and can be assumed like the other abducible predicates, beside being provided as input in the event set **HAP**.

4 Mapping Datalog$^{\pm}$ into ALP programs

In this section, we show that a Datalog$^{\pm}$ program can be represented as a set of $\mathcal{S}$CIFF integrity constraints and an event set. $\mathcal{S}$CIFF abductive declarative semantics provides the model-theoretic counterpart to Datalog$^{\pm}$ semantics. Operationally, query answering is achieved bottom-up via the *chase* in Datalog$^{\pm}$, while in the ALP framework it is supported by the $\mathcal{S}$CIFF proof procedure. $\mathcal{S}$CIFF is able to integrate a knowledge base KB, expressed in terms of Logic Programming clauses, possibly with abducibles in their body, and to deal with integrity constraints.

To our purposes, we consider only $\mathcal{S}$CIFF programs with an empty KB, ICs with only conjunctions of positive expectations and CLP constraints (or *false*) in their heads. We show that this subset of the language suffices to represent Datalog$^{\pm}$ ontologies.

We map the finite set of relation names of a Datalog$^{\pm}$ relational schema $\mathcal{R}$ into the set of predicates of the corresponding $\mathcal{S}$CIFF program.

Definition 2. *The τ mapping is recursively defined as follows, where A is an atom,* **M** *can be either* **H** *or* **E**, *and F_1, F_2, ... are formulae:*

$$\begin{aligned}
\tau(Body \rightarrow Head) &= \tau_{\mathbf{H}}(Body) \rightarrow \tau_{\mathbf{E}}(Head) \\
\tau_{\mathbf{H}}(A) &= \mathbf{H}(A) \\
\tau_{\mathbf{E}}(A) &= \mathbf{E}(A) \\
\tau_{\mathbf{M}}(F_1 \wedge F_2) &= \tau_{\mathbf{M}}(F_1) \wedge \tau_{\mathbf{M}}(F_2) \\
\tau_{\mathbf{M}}(\mathit{false}) &= \mathit{false} \\
\tau_{\mathbf{M}}(Y_i = Y_j) &= Y_i = Y_j \\
\tau_{\mathbf{E}}(\exists \mathbf{X}\ A) &= A
\end{aligned}$$

A Datalog$^{\pm}$ database D for $\mathcal{R}$ corresponds to the (possibly infinite) $\mathcal{S}$CIFF event set **HAP**, since there is a one-to-one correspondence between each tuple in D and each (ground) fact in **HAP**. This mapping is denoted as $\mathbf{HAP} = \tau_{\mathbf{H}}(D)$.

Notice that since the $\mathcal{S}$CIFF event set can dynamically grow, new constants can be introduced as a new event occurs (these new constants correspond to those in the set Δ_N of Datalog$^{\pm}$).

A Datalog TGD F of the kind *body* $\rightarrow$ *head* is mapped into the $\mathcal{S}$CIFF integrity constraint $IC = \tau(F)$, where the *body* is mapped into conjunctions of $\mathcal{S}$CIFF atoms, and *head* into conjunctions of $\mathcal{S}$CIFF abducible atoms. Existential quantifications of variables occurring in the *head* of the TGD are maintained in the head of the $\mathcal{S}$CIFF IC, but they are left implicit in the $\mathcal{S}$CIFF syntax, while the rest of the variables are universally quantified with scope the entire IC.

Given a set of TGDs T, let us denote the mapping of T into the corresponding set $\mathcal{IC}$ of $\mathcal{S}$CIFF integrity constraints, as $\mathcal{IC} = \tau(T)$.

Recall that for a set of TGDs T on $\mathcal{R}$, and a database D for $\mathcal{R}$, the set of models of D given T, denoted $mods(D, T)$, is the set of all (possibly infinite) databases B such that $D \subseteq B$ and every $F \in T$ is satisfied in B. For any such database B, we can prove that there exists an abductive explanation $\mathbf{EXP} = \tau_{\mathbf{E}}(B)$, $\mathbf{HAP'} = \tau_{\mathbf{H}}(B)$ such that:

$$\mathbf{HAP'} \cup \mathbf{EXP} \models \mathcal{IC}$$

where $\mathbf{HAP'} \supseteq \mathbf{HAP} = \tau_{\mathbf{H}}(D)$, and $\mathcal{IC} = \tau(T)$.

Finally, Datalog$^{\pm}$ negative constraints NC are mapped into $\mathcal{S}$CIFF ICs with head *false*, and equality-generating dependencies EGDs into $\mathcal{S}$CIFF ICs, each one with an equality CLP constraint in its head.

Therefore, informally speaking, the set of models of D given T, $mods(D, T)$, corresponds to the set of all the abductive explanations **EXP** satisfying the set of $\mathcal{S}$CIFF integrity constraints $\mathcal{IC} = \tau(\mathcal{T})$.

A Datalog$^{\pm}$ CQ $q(\mathbf{X}) = \exists \mathbf{Y} \Phi(\mathbf{X}, \mathbf{Y})$ over $\mathcal{R}$ is mapped into a $\mathcal{S}$CIFF goal $G = \tau_{\mathbf{E}}(\Phi(\mathbf{X}, \mathbf{Y}))$, where $\tau_{\mathbf{E}}(\Phi(\mathbf{X}, \mathbf{Y}))$ is a conjunction of $\mathcal{S}$CIFF atoms. Notice that in the $\mathcal{S}$CIFF framework we have therefore a goal with existential variables only, and among them, we are interested in computed answer substitutions for the original (tuple of) variables $\mathbf{X}$ (and therefore $\mathbf{Y}$ variables can be made anonymous).

A Datalog$^{\pm}$ BCQ $q = \Phi(\mathbf{Y})$ is mapped similarly: $G = \tau_{\mathbf{E}}(\Phi(\mathbf{Y}))$.

Recall that in Datalog$^{\pm}$ the set of answers to a CQ q on D given T, denoted $ans(q, D, T)$, is the set of all tuples $\mathbf{t}$ such that $\mathbf{t} \in q(B)$ for all $B \in mods(D, T)$. With abuse of notation, we will write $q(\mathbf{t})$ to mean answer $\mathbf{t}$ for q on D given T.

We can hence state the following theorems for (model-theoretic) completeness of query answering.

Theorem 1 (Completeness of query answering). *For each answer $q(\mathbf{t})$ of a CQ $q(\mathbf{X}) = \exists \mathbf{Y} \Phi(\mathbf{X}, \mathbf{Y})$ on D given T, in the corresponding $\mathcal{S}$CIFF program $\langle \emptyset, \tau(\mathcal{IC}) \rangle$ there exists an answer substitution θ and an abductive explanation $\mathbf{EXP} \cup \mathbf{HAP'}$ for goal $G = \tau_{\mathbf{E}}(\Phi(\mathbf{X}, \_))$ such that:*

$$\langle \emptyset, \tau(\mathcal{IC}) \rangle \models^{g}_{\mathbf{HAP}} G\theta$$

where $\mathbf{HAP} = \tau_{\mathbf{H}}(D)$, $\mathcal{IC} = \tau(T)$, and $G\theta = \tau_{\mathbf{E}}(\Phi(\mathbf{t}, \_))$.

Corollary 1 (Completeness of boolean query answering). *If the answer to a BCQ* $q = \exists \mathbf{Y} \Phi(\mathbf{Y})$ *over* D *given* T *is Yes, denoted* $D \cup T \models q$*, then in the corresponding* $\mathcal{S}$*CIFF program there exists an abductive explanation* $\mathbf{EXP} \cup \mathbf{HAP}'$ *such that:*

$$\langle \emptyset, \tau(\mathcal{IC}) \rangle \models^{g}_{\mathbf{HAP}} G\theta$$

where $\mathbf{HAP} = \tau_{\mathbf{H}}(D)$*,* $\mathcal{IC} = \tau(T)$*, and* $G = \tau_{\mathbf{E}}(\Phi(\_))$*.*

The $\mathcal{S}$CIFF proof procedure has been proved sound w.r.t. $\mathcal{S}$CIFF declarative semantics in [4], therefore for each abductive explanation $\mathbf{EXP}$ for a given goal G in a $\mathcal{S}$CIFF program, there exists a $\mathcal{S}$CIFF-based computation producing a set of abducibles (positive expectations to our purposes) $\delta \subseteq \mathbf{EXP}$, and a computed answer substitution for goal G possibly more general than θ.

Example 2 (Real estate information extraction system in ALP). Let us conclude this section by re-considering the Datalog$^{\pm}$ ontology for the real estate information extraction system of Example 1. TGDs F_1-F_8 are one-to-one mapped into the following $\mathcal{S}$CIFF ICs:

$IC_1 : \mathbf{H}(ann(X, label)), \mathbf{H}(ann(X, price)), \mathbf{H}(visible(X)) \rightarrow \mathbf{E}(priceElem(X))$
$IC_2 : \mathbf{H}(ann(X, label)), \mathbf{H}(ann(X, priceRange)), \mathbf{H}(visible(X))$
$\quad \rightarrow \mathbf{E}(priceElem(X))$
$IC_3 : \mathbf{H}(priceElem(E)), \mathbf{H}(group(E, X)) \rightarrow \mathbf{E}(forSale(X))$
$IC_4 : \mathbf{H}(forSale(X)) \rightarrow (\exists P)\ \mathbf{E}(price(X, P))$
$IC_5 : \mathbf{H}(hasCode(X, C)), \mathbf{H}(codeLoc(C, L)) \rightarrow \mathbf{E}(loc(X, L))$
$IC_6 : \mathbf{H}(hasCode(X, C)) \rightarrow (\exists L)\ \mathbf{E}(codeLoc(C, L)), \mathbf{E}(loc(X, L))$
$IC_7 : \mathbf{H}(loc(X, L1)), \mathbf{H}(loc(X, L2)) \rightarrow L1 = L2$
$IC_8 : \mathbf{H}(loc(X, L)) \rightarrow \mathbf{E}(advertised(X))$

The database is then simply mapped into the following event set $\mathbf{HAP}$:

$$\{\mathbf{H}(codeLoc(ox1, central)), \mathbf{H}(codeLoc(ox1, south)),$$
$$\mathbf{H}(codeLoc(ox2, summertown)), \mathbf{H}(hasCode(prop1, ox2)), \mathbf{H}(ann(e1, price)),$$
$$\mathbf{H}(ann(e1, label)), \mathbf{H}(visible(e1)), \mathbf{H}(group(e1, prop1))\}$$

The $\mathcal{S}$CIFF proof procedure applies ICs in a forward manner, and it infers the following set of abducibles from the program above:

$$\mathbf{EXP} = \{\mathbf{E}(priceElem(e1)), \mathbf{E}(forSale(prop1)), \exists P\ \mathbf{E}(price(prop1, P)),$$
$$\mathbf{E}(loc(prop1, summertown)), \mathbf{E}(advertised(prop1))\}$$

plus the corresponding $\mathbf{H}$ atoms, that are not reported for the sake of brevity.

Each of the (ground) atomic queries of Example 1 is entailed in the $\mathcal{S}$CIFF program above, since there exist sets $\mathbf{EXP}$ and $\mathbf{HAP}'$ such that:

$$\mathbf{HAP}' \cup \mathbf{EXP} \models \mathbf{E}(priceElem(e1)), \mathbf{E}(forSale(prop1)), \mathbf{E}(advertised(prop1))$$

The query $\exists L\ \mathbf{E}(loc(prop1, L))$ is entailed as well, considering the unification $L = summertown$ since:

$$\mathbf{HAP}' \cup \mathbf{EXP} \models \mathbf{E}(loc(prop1, summertown)).$$

It is worth noting that the $\mathcal{S}$CIFF framework is much more expressive than the restricted version used in this paper; in fact, in the mapping we used an empty KB, but in general the Knowledge Base can be a logic program, that can includes expectations, abducible literals, as well as CLP constraints. Beside the forward propagation of Integrity Constraints, $\mathcal{S}$CIFF supports also backward reasoning.

5 Related Work

Various approaches has been followed to reason upon ontologies.

Usually, DL reasoners implement a tableau algorithm using a procedural language. Since some tableau expansion rules are non-deterministic, the developers have to implement a search strategy from scratch.

Pellet [32] is a free open-source Java-based reasoner for SROIQ with simple datatypes (i.e., for OWL 1.1). It implements a tableau-based decision procedure for general TBoxes (subsumption, satisfiability, classification) and ABoxes (retrieval, conjunctive query answering). It supports the OWL-API, the DIG interface, and the Jena interface and comes with numerous other features.

Pellet can compute the set of all the explanations for given queries by exploiting the tableau algorithm. An explanation is roughly a subset of the knowledge base (KB) that is sufficient for entailing the query. It applies Reiter's *hitting set algorithm* [29] to find all the explanations. This is a black box method: Pellet repeatedly removes an axiom from the KB and then computes again a new explanation exploiting the tableau algorithm on the new KB, recording all the different explanations so found.

Differently from Pellet, reasoners written in Prolog can exploit Prolog's backtracking facilities for performing the search. This has been observed in various works. In [8, 28] the authors proposed a tableau reasoner in Prolog for First Order Logic (FOL) based on free-variable semantic tableaux. However, the reasoner is not tailored to DLs.

Hustadt, Motik and Sattler [22] presented the KAON2 algorithm that exploits basic superposition, a refutational theorem proving method for FOL with equality, and a new inference rule, called decomposition, to reduce a $\mathcal{SHIQ}$ KB into a disjunctive Datalog program, while DLog [25, 33] is an ABox reasoning algorithm for the $\mathcal{SHIQ}$ language that allows to store the content of the ABox externally in a database and to answer instance check and instance retrieval queries by transforming the KB into a Prolog program.

Meissner presented the implementation of a reasoner for the DL $\mathcal{ALCN}$ written in Prolog [26] which was then extended and reimplemented in the Oz language [27]. Starting from [26], Herchenröder [20] implemented heuristic search techniques in order to reduce the inference time for the DL $\mathcal{ALC}$. Faizi [15] added to [20] the possibility of returning information about the steps executed during the inference process for queries but still handled only $\mathcal{ALC}$.

A different approach is the one by Ricca et al. [30] that presented OntoDLV, a system for reasoning on a logic-based ontology representation language called

OntoDLP. This is an extension of (disjunctive) ASP and can interoperate with OWL. OntoDLV rewrites the OWL KB into the OntoDLP language, can retrieve information directly from external OWL Ontologies and answers queries by using ASP.

TRILL [34, 35] adopts a Prolog-based implementation for the tableau expansion rules for $\mathcal{ALC}$ description logics. Differently from previous reasoners, TRILL is also able to return explanations for the given queries. Moreover, TRILL differs in particular from DLog for the possibility of answering general queries instead of instance check and instance retrieval only.

As reported in Section 2, reasoning upon Datalog$^{\pm}$ ontologies is achieved, instead, via the *chase* bottom-up procedure, which is exploited for deriving atoms entailed by a database and a Datalog$^{\pm}$ theory.

In this work, instead, we apply an abductive logic programming proof-procedure to reason upon ontologic data. It is worth to notice that in a previous work [1] the $\mathcal{S}$CIFF proof-procedure was interfaced with Pellet to perform ontological reasoning; in the current work, instead, $\mathcal{S}$CIFF is directly used to perform the reasoning by mapping atoms in the ontology to $\mathcal{S}$CIFF concepts (like events and expectations).

6 Conclusions and Future Work

In this paper, we addressed representation and reasoning for Datalog$^{\pm}$ ontologies in an Abductive Logic Programming framework, with existential (and universal) variables, and Constraint Logic Programming constraints in rule heads. The underlying proof procedure, named $\mathcal{S}$CIFF, is inspired by the IFF proof procedure, and had been implemented in Constraint Handling Rules [16]. The $\mathcal{S}$CIFF system has already been used for modeling and implementing several knowledge representation frameworks, also providing an effective reasoning system.

Here we have considered Datalog$^{\pm}$ ontologies, and shown how the $\mathcal{S}$CIFF language can be a useful knowledge representation and reasoning framework for them. In fact, the underlying abductive proof procedure can be directly exploited as an ontological reasoner for query answering and consistency check. To the best of our knowledge, this is the first application of ALP to model and reason upon ontologies.

Moreover, the considered $\mathcal{S}$CIFF language smoothly supports the integration of rules, expressed in a Logic Programming language, with ontologies expressed in Datalog$^{\pm}$, since a logic program can be added as (non-empty) KB to the set of ICs, therefore considering deductive rules besides the forward ICs themselves. Moreover, through dynamic acquisition of events in its **HAP** set, $\mathcal{S}$CIFF might also supports inline incrementality of the extensional part of the knowledge base (namely, the $\mathcal{A}$Box).

Many issues have not been addressed in this paper, and they will be subject of future work. First of all, we have not focused here on complexity results. Future work will be devoted to identify syntactic conditions guaranteeing tractable ontologies in $\mathcal{S}$CIFF, in the style of what has been done for Datalog$^{\pm}$.

A second issue for future work concerns experimentation and comparison with other approaches, even not Logic Programming (LP, for short) based, on real-size ontologies.

Finally, $\mathcal{S}$CIFF language is richer than the subset here used to represent Datalog$^{\pm}$ ontologies. It can support, in fact, negative expectations in rule heads, with universally quantified variables too, which basically represent the fact that something ought not to happen, and the proof procedure can identify violations to them.

Therefore, the richness of the language, and the potential of its abductive proof procedure pave the way to add further features to Datalog$^{\pm}$ ontologies.

References

1. Alberti, M., Cattafi, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M., Torroni, P.: A computational logic application framework for service discovery and contracting. International Journal of Web Services Research 8(3), 1–25 (2011)
2. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M.: An abductive framework for a-priori verification of web services. In: Maher, M. (ed.) Proceedings of the Eighth Symposium on Principles and Practice of Declarative Programming. pp. 39–50. ACM Press, New York, USA (Jul 2006)
3. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Security protocols verification in Abductive Logic Programming: a case study. In: Dikenelli, O., Gleizes, M.P., Ricci, A. (eds.) Proceedings of ESAW'05, Lecture Notes in Artificial Intelligence, vol. 3963, pp. 106–124. Springer-Verlag (2006)
4. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: the SCIFF framework. ACM Transactions on Computational Logic 9(4) (2008)
5. Alberti, M., Gavanelli, M., Lamma, E.: The CHR-based implementation of the SCIFF abductive system. Fundamenta Informaticae 124(4), 365–381 (2013)
6. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Sartor, G., Torroni, P.: Mapping deontic operators to abductive expectations. Computational and Mathematical Organization Theory 12(2–3), 205 – 225 (Oct 2006)
7. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Specification and verification of agent interactions using social integrity constraints. Electronic Notes in Theoretical Computer Science 85(2) (2003)
8. Beckert, B., Posegga, J.: leanTAP: Lean tableau-based deduction. J. Autom. Reasoning 15(3), 339–358 (1995)
9. Calì, A., Gottlob, G., Kifer, M.: Taming the infinite chase: Query answering under expressive relational constraints. In: International Conference on Principles of Knowledge Representation and Reasoning. pp. 70–80. AAAI Press (2008)
10. Calì, A., Gottlob, G., Lukasiewicz, T.: A general datalog-based framework for tractable query answering over ontologies. In: Symposium on Principles of Database Systems. pp. 77–86. ACM (2009)
11. Calì, A., Gottlob, G., Lukasiewicz, T.: Tractable query answering over ontologies with Datalog$^{\pm}$. In: International Workshop on Description Logics. CEUR Workshop Proceedings, vol. 477. CEUR-WS.org (2009)
12. Calì, A., Gottlob, G., Lukasiewicz, T., Marnette, B., Pieris, A.: Datalog$^{\pm}$: A family of logical knowledge representation and query languages for new applications. In:

IEEE Symposium on Logic in Computer Science. pp. 228–242. IEEE Computer Society (2010)
13. Calì, A., Gottlob, G., Pieris, A.: Tractable query answering over conceptual schemata. In: International Conference on Conceptual Modeling. LNCS, vol. 5829, pp. 175–190. Springer (2009)
14. Calvanese, D., Giacomo, G.D., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The *dl-lite* family. J. Autom. Reasoning 39(3), 385–429 (2007)
15. Faizi, I.: A Description Logic Prover in Prolog. Bachelor's thesis, Informatics Mathematical Modelling, Technical University of Denmark (2011)
16. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming 37(1-3), 95–138 (Oct 1998)
17. Fung, T.H., Kowalski, R.A.: The IFF proof procedure for abductive logic programming. Journal of Logic Programming 33(2), 151–165 (Nov 1997)
18. Gottlob, G., Lukasiewicz, T., Simari, G.I.: Conjunctive query answering in probabilistic Datalog+/- ontologies. In: International Conference on Web Reasoning and Rule Systems. LNCS, vol. 6902, pp. 77–92. Springer (2011)
19. Haarslev, V., Hidde, K., Möller, R., Wessel, M.: The racerpro knowledge representation and reasoning system. Semantic Web 3(3), 267–277 (2012)
20. Herchenröder, T.: Lightweight Semantic Web Oriented Reasoning in Prolog: Tableaux Inference for Description Logics. Master's thesis, School of Informatics, University of Edinburgh (2006)
21. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. CRCPress (2009)
22. Hustadt, U., Motik, B., Sattler, U.: Deciding expressive description logics in the framework of resolution. Inf. Comput. 206(5), 579–601 (2008)
23. Jaffar, J., Maher, M.: Constraint logic programming: a survey. Journal of Logic Programming 19-20, 503–582 (1994)
24. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive Logic Programming. Journal of Logic and Computation 2(6), 719–770 (1993)
25. Lukácsy, G., Szeredi, P.: Efficient description logic reasoning in Prolog: The DLog system. TPLP 9(3), 343–414 (2009)
26. Meissner, A.: An automated deduction system for description logic with ALCN language. Studia z Automatyki i Informatyki 28-29, 91–110 (2004)
27. Meissner, A.: A simple distributed reasoning system for the connection calculus. Vietnam Journal of Computer Science 1(4), 231–239 (2014), http://dx.doi.org/10.1007/s40595-014-0023-8
28. Posegga, J., Schmitt, P.: Implementing semantic tableaux. In: DAgostino, M., Gabbay, D., Hähnle, R., Posegga, J. (eds.) Handbook of Tableau Methods, pp. 581–629. Springer Netherlands (1999), http://dx.doi.org/10.1007/978-94-017-1754-0_10
29. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. 32(1), 57–95 (1987)
30. Ricca, F., Gallucci, L., Schindlauer, R., Dell'Armi, T., Grasso, G., Leone, N.: OntoDLV: An ASP-based system for enterprise ontologies. J. Log. Comput. 19(4), 643–670 (2009)
31. Shearer, R., Motik, B., Horrocks, I.: Hermit: A highly-efficient owl reasoner. In: OWLED (2008)
32. Sirin, E., Parsia, B., Cuenca-Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Journal of Web Semantics 5(2), 51–53 (2007)

33. Straccia, U., Lopes, N., Lukacsy, G., Polleres, A.: A general framework for representing and reasoning with annotated semantic web data. In: Fox, M., Poole, D. (eds.) Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010. AAAI Press (2010), http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1590
34. Zese, R., Bellodi, E., Lamma, E., Riguzzi, F.: A description logics tableau reasoner in Prolog. In: Cantone, D., Asmundo, M.N. (eds.) CILC. CEUR Workshop Proceedings, vol. 1068, pp. 33–47. CEUR-WS.org (2013)
35. Zese, R., Bellodi, E., Lamma, E., Riguzzi, F., Aguiari, F.: Semantics and inference for probabilistic description logics. In: Bobillo, F., Carvalho, R.N., da Costa, P.C.G., d'Amato, C., Fanizzi, N., Laskey, K.B., Laskey, K.J., Lukasiewicz, T., Nickles, M., Pool, M. (eds.) Uncertainty Reasoning for the Semantic Web III - ISWC International Workshops, URSW 2011-2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8816, pp. 79–99. Springer (2014)