Typed Hybrid Description Logic Programs with Order-Sorted Semantic Web Type Systems based on OWL and RDFS

Adrian Paschke

Internet-based Information Systems, Dept. of Informatics, TU Munich, Germany
adrian.paschke@gmx.de
Technical Report December 2005

Abstract. In the recent years rule-based programming in terms of declarative logic programming has formed the basis for many Artificial Intelligence (AI) applications and is well integrated in the mainstream information technology capturing higher-level decision logics. Typically, the standard rule systems and rule-based logic programming languages such as Prolog derivatives are based on the untyped theory of predicate calculus with untyped logical objects (untyped terms), i.e. the logical reasoning algorithms apply pure syntactical reasoning. From a rule engineering perspective this is a serious restriction which lacks major Software Engineering principles such as data abstraction or modularization, which become more and more important when rule applications grow larger and more complex. To support such principles in logic programming and capture the rule engineer's intended meaning of a logic program, types and typed objects play an important role. Moreover, from a computational point of view, the use of types drastically reduces the search space, i.e. proofs can be kept at a more abstract level and it offers the option to restrict the application of rules and to control the level of generality in queries. In this paper I introduce a multi-sorted logic which allows using external Semantic Web type systems for term typing in rules. Using Semantic Web ontologies as type systems facilitates exchange of domain-independent rules over domain boundaries via dynamically typing and mapping of explicitly defined type terminologies. That is, rule description can take advantage of the DL representation and reasoning capabilities, where different type systems, assigning different domain-specific vocabularies to rules, can be merged or disjoined and used dynamically at runtime. Until recently, the use of Semantic Web languages such as RDFS, OWL Lite or OWL DL has been limited primarily to representing Web contents and previous works towards integration of rules and ontologies in the Semantic Web mainly focus on extending the general expressiveness of either the rule language or the ontology language. I elaborate on a specific application, namely DL-typed unification which integrates polymorphic order-sorted type inference into the semantics of hybrid description logic programs. This approach provides a technical separation with minimal interfaces (based on query entailment) between the inferences in the logic programming component and the description logic inferences, which results in more efficient computations, enhanced language expressiveness, a flexible and robustly decidable hybrid integration, even in the case where the rule language is far more expressive than Datalog. The multi-sorted logic supports subtypes, so called order-sorted type systems, where types are analyzed at run-time possibly permitting ad-hoc polymorphism of typed variables, i.e. variables might change their types during runtime. I present syntax and semantic of my hybrid DL-

typed logic programming language. I have developed a prototype system as part of the ContractLog KR which supports typed unification and hybrid knowledge bases and integrate the backward-reasoning ContractLog/Prova rule engine with the Semantic Web API Jena and the OWL-DL reasoner Pellet. Finally, I illustrate the integration of types into the Rule Markup Language RuleML with serialization of RDFS/OWL based type systems in XML syntax resp. XML/RDF syntax. In contrast to homogeneous integration approaches such as DLP or hybrid approaches based on additional DL constraints (DL-atoms/queries) my hybrid approach benefits from the prescriptive typing approach and the typed unification using an external tableaux DL-reasoner (Jena/Pellet) for DL type checking as a black box from the view of a rule engineer.

Key words: Multi-Sorted Logic, Homogenous and Heterogeneous Description Logic Programs, Polymorphic Order-Sorted Typed Unification, Rule Interchange, Semantic Web Type Systems

1 On the Need for Semantic Web Types in Declarative Logic Programming

Traditional logic programming languages such as most Prolog derivatives are typically purely based on the untyped theory of predicate calculus with untyped logical objects (untyped terms). That is, the logical reasoning algorithms apply pure syntactical reasoning and use flat untyped logical objects (terms) in their rule descriptions. From the engineering perspective of a rule-based SLA specification this is a serious restriction which lacks major Software Engineering (SE) principles such as *data abstraction* or *modularization*. Such principles become important when rule applications grow larger and more complex, are engineered and maintained by different people and are managed and interchanged in a distributed environment over domain-boundaries. To support such SE principles in declarative programming and capture the rule engineer's intended meaning of a contract, types and typed objects play an important role. Types are an appropriate way to integrate domain specific ontologies into dynamic domain-independent rules. As a result, such typed rules are much easier to interchange between domain-boundaries in distributed environments such as the Semantic Web where the domain vocabularies are expressed as webized ontologies written in expressive ontology languages such as RDF(S) or OWL. From a descriptive point of view types attach additional information, which can be used as *type constraints* for selecting specific goals, i.e. they constrain the level of generality in queries, lead to much smaller search spaces and improve the execution efficiency of query answering.

In the following I describe a multi-sorted logic with a order-sorted typed unification as operational semantics which allows using external DL type systems, i.e. Semantic Web ontologies resp. knowledge bases, for term typing in rules. In contrast to existing hybrid integration approaches which use additional DL constraints or atoms/queries in the body of rules I propose a prescriptive typing approach where types are direct properties of the logical formulas, i.e. are directly attached to rule terms. The typed order-sorted unification, which

supports dynamic type checking of terms during unification, provides a built-in technical separation between the inferences in the rule component and the type checks in the DL type system which results in:

- in more efficient computations, since the type restrictions directly apply during the unification of typed terms
- higher flexibility with minimal interfaces (based on entailment) between rules component and the type system
- robustly decidable combinations, even in case where the rule language is far more expressive than Datalog and no safeness restrictions apply
- enhanced language expressiveness, e.g. ad-hoc polymorphism with overloading is possible, type casting during order-sorted unification, data abstraction and modularization
- existing implementations can be reused
- rules can be more intuitively modelled and easily combined and interchanged

The further paper is structured as follows: I first review the history of types in logic programming languages and describe basic concepts in section 2. Then, in section 3, I elaborate on the integration of rules and ontologies. In section 4 I describe the syntax of my DL-typed rule language and in section 4 and 5 its declarative and operational semantics, respectively. In section 6 I will discussion implementation and integrating of types into rule markup languages and will discuss the ContractLog approach towards a hybrid DL-typed logic. Finally, I will conclude this paper with a short summary in section 7.

2 Types in Logic Programming

Definition 1. *(**Type System**) A type system [Car97] is responsible for assigning types to variables and expressions. It uses the type expressions for static type checking at compile time and/or dynamic type checking at runtime. Type systems typically define a type relation of the form $t : r$, denoting that the term t is of type r. The primary purpose of a type system is to prevent typing errors which violate the intended semantics of a rule or function in a LP.*

That is, a type system is used to ensure robustness of a LP. By ensuring that the structure is well-defined types enable a more disciplined and regular engineering process and facilitate modularity and partial specification of the intended use of the logical functions and their arguments in a logic program. It has been demonstrated that types play an important role to capture the programmer's intended meaning of a logic program, see e.g. [Nai92] and that they can be used to dramatically reduce the rule search space - see e.g. Schubert's Steamroller Problem which used to illustrate this advantage [Sti86].

The theory of types in LP has been studied in several works and many different approaches for the introduction of types into logic programming have been proposed reaching from many-sorted or order-sorted systems with sub-typing to ad-hoc and parametric polymorphic typing and combinations such as parametric

polymorphic order-sorted type systems. Most of the works on type systems and their properties are based on the theory of λ-calculus [Bar88] which gives some fundamentals for reasoning about types in functional languages and which has been generalized to different programming languages such as object-oriented or declarative logic programming languages. Based on this theory one of the most well-known type systems is the Hindley-Milner type system [Mil78]. Different forms of type declarations have been proposed such as declarations which use a rather general constraint language [HS88], logical formulas [Xu188,Nai92], regular sets [Mis84,Mis85,HJ92,DZ92,AE93], equational specifications [Han92] or typed terms over a order-sorted structure [MO84,Lak91,SNGM9b,HT92,Han91] (see e.g. [Pfe92] for an overview).

In general, the works can be classified into two different views on types in logic programming, namely *descriptive types* and *prescriptive* types [Red88] (a.k.a. explicit and implicit types or syntactic and semantic typing). Early work on types in logic programming mainly concentrate on descriptive type systems, e.g. [DZ2a,Mis84,Hen94,Zob87] which attach type information with programs without changing the language used and without affecting the meaning of logic programs. These descriptive approaches are seeking to approximate the structure of a program for use by an optimizing compiler at compile time. For instance, Mycroft and O'Keefe [MO84] demonstrated that the polymorphic type discipline of Milner [Mil78] can be represented in pure Prolog, where the type declarations for variables occur outside of the clauses and do not change the semantics of the pure Prolog program. Dietrich and Hagl's [Die88] extend this approach with input/output mode declarations and subsorts. In contrast, prescriptive type systems, e.g. Gdel [HL94], Typed Prolog [Lak91], λ-Prolog [Mil86,MNFS91], consider types as properties of the formulas one wants to give a meaning to, i.e. they use a *typed logic* for programming leading to languages with higher expressiveness. The purpose is to identify ill-typed programs, so that the actual semantics of a program satisfies the intended semantics of the rule engineer. Lakshman and Reddy have redefined the Mycroft-O'Keefe type discipline as Typed Prolog [Lak91] which adopts the prescriptive view and gives a semantics to the typed logic programs. Several other works follow this approach of defining semantics for prescriptive typed LPs, e.g. [HJ92,HT92,Lak91].

These semantics approaches, to which also this ContractLog work contributes, base logic programming with types on typed logics that use sorts (type definitions). Following the terminology of abstract data types [GTW78,EM85] in *many-sorted type systems* sorts are defined by their constructors, i.e. the constituent elements of a sort. The sorts are used to define the number of arguments and their types for every predicate in a logic program. In the many-sorted case sorts are not allowed to have subsort relations between them and accordingly type checking can be done statically at compile time, e.g. realized by a preprocessor without any extensions to the underlying unsorted predicate logic. In the *order-sorted approach* subsort hierarchies are supported typically by the use of a *order-sorted unification* (typed unification) in order to incorporate some form of *subtyping polymorphism* for untyped variables which as-

sume the type of a unified typed term at runtime. A first order-sorted logic was given by Oberschelp [Obe62] and an order-sorted algebra was developed by Goguen et. al. [GM87,Smo89] which forms the basis for the language Eqlog [GM86]. An extended order-sorted algebra with error-handling was proposed by Gogolla [Gog86]. Several other order-sorted approaches have been described using order-sorted unification [Wal87,Hub87]. Different forms of polymorphism such as *generic polymorphism* (see e.g. ML programming language [Mil78]), *ad-hoc polymorphism* or *parametric polymorphism* haven been introduced into logic programming. For a discussion of the differences between parametric and ad-hoc polymorphism see e.g. [Str00]. In the context of polymorphism terms (variables) are authorized to change their types dynamically at runtime, which makes static compile-time analysis insufficient in general. If the type system permits ad-hoc polymorphism, the unifiers for terms are determined by their types and the procedures (logical functions) being invoked are dependent on this information, i.e. the types affect the question of unifiability in an intrinsic way and the computation process must use some form of typed unification procedure to ensure type correctness also during runtime. The types are needed to determine the existence of unifiers and hence also the applicability of clauses. An interesting aspect of this typed semantics is that it enables overloading of clauses leading to different functional variants which are selected dynamically at runtime according to the types of the queries. As a result, the specific inferences that are performed and the correct answers to queries are directly related to the types of the terms in the program clauses and the answers to queries not only display the bindings of variables, but also their types. Typed unification has been studied for order-sorted and polymorphic type systems, see e.g. Typed Prolog [Lak91], Protos-L [BB89], λ-Prolog extensions [KNW93]. *Order-sorted unification* extends the usual term unification with additional dynamic type checking. In a nutshell, the basic idea of sorted unification of two typed variables is to find the greatest lower bound (glb) of their types based on the type hierarchy with subtype relationships, failing if it does not exist. In other words, the unification algorithm tries to find the glb of two sort restrictions yielding a variable whose sort restriction is the greatest common subsort of the two sorts of the unified terms in the given sort hierarchy. This typing approach provides higher levels of abstractions and allows *ad-hoc polymorphism* wrt *coercion*, i.e. automatic type conversion between subtypes, and *overloading*, i.e. defining multiple functions (rules with the same head but different types) taking different types, where the unification algorithm automatically does the type conversion and calls the right function (unifies a (sub-)goal with the right rule head). *Parametric polymorphic types* allow to parameterize a structured sort over some other sort, i.e. types and predicates can be written generically so that they can deal equally with any object without depending on their types, in contrast to the many-sorted case, where for each predicate variant having a different type a sort definition must be given explicitly. Typically, parametric polymorphism still maintains full static type-safety on a syntactic level, however there are approaches with a semantic notion of polymorphic types, e.g. order-sorted parametric polymorphic typed

logics which have to take the type information into account at runtime and hence require an extended unification algorithm with type inferencing, as in the order sorted case. These polymorphic type systems being very complicated artifacts both theoretically and computational wise and have been primarily designed for use in the context of higher-order logic programming. The emphasis in these higher-order type languages has been on describing conditions under which the computationally expensive type analysis can be avoided at runtime which often amounts to banishing ad hoc polymorphism and applying several restrictions to function symbols which must be type preserving. Recent works on types for LPs have concentrated on implementation techniques for efficiently checking or inferring types at runtime, in particular polymorphic types, e.g. by means of abstract interpretations [Lu98] or constraint solving techniques [Dem99].

3 Description Logic Type System

Until recently, the use of Semantic Web ontology languages such as OWL [MH04] or RDFS [BG04] has been limited primarily to define meta data vocabularies and add semantic machine-readable meta data to Web pages enabling automated processing of Web contents. Both, Description Logics [NBB+02] which form the logical basis of Semantic Web serialization languages such as OWL and (Horn) LPs (restricted to function-free Datalog LPs) are decidable fragments of first order logic, however for the most part with orthogonal expressive power. Whereas OWL is restricted to unary resp. binary axioms, but e.g. provides classical/strong negation under open world assumption (OWA) and existentially as well as universally quantified variables, Datalog LPs allow n-ary axioms and (non-monotonic) negation, but are restricted to universal quantifications and are therefore not able to reason about unknown individuals. Clearly, both approaches can benefit from a combination and several integration approaches have been proposed recently.

The works on combining rules and ontologies can be basically classified into two basic approaches: *homogeneous and heterogeneous integrations*. Starting from the early Krypthon language [Bra85] among the heterogeneous approaches, which hybridly use DL reasoning techniques and tools in combination with rule languages and rule engines are e.g. CARIN [Lev96], Life [Ait91], Al-log [DLNS91], non-monotonic dl-programs [Eit04] and r-hybrid KBs [Ric05]. Among the homogeneous approaches which combine the rule component and the DL component in one homogeneous framework sharing the combined language symbols are e.g. DLP [Gro03], KAON2 [MSS05] or SWRL [HPSB+04]. Both integration approaches have pros and cons and different integration strategies such as reductions or fixpoint iterations are applied with different restrictions to ensure decidability. These restrictions reach from only allowing the intersection of DLs and Horn rules [Gro03], to leaving full syntactic freedom for the DL component, but restricting the rules to *DL-safe rules* [MSS05], where DL variables must also occur in a non DL-atom in the rule body, or *role-safe rules* [Lev96], where at least one variable in a binary DL-query in the body of a hybrid rule must also appear in a non-DL atom in the body of the rule which

never appears in the consequent of any rule in the program or to tree-shaped rules [Hey05]. Furthermore, they can be distinguished according to their information flow which might be *uni-directional* or *bi-directional*. For instance, in homogeneous approaches bi-directional information flows between the rules and the ontology part are naturally supported and new DL constructs introduced in the rule heads can be directly used in the integrated ontology inferences, e.g. with the restriction that the variables also appear in the rule body (safeness condition). However, in these approaches the DL reasoning is typically solved completely by the rule engine and benefits of existing tableau based algorithms in DL reasoners are lost. On the other side, heterogenous approaches, benefit from the hybrid use of both reasoning concepts exploiting the advantages of both (using LP reasoning and tableaux based DL reasoning), but bi-directional information flow and fresh DL constructs in rule heads are much more difficult to implement. In summary, the question wether the Semantic Web should adopt an homogeneous or heterogenous view is still very much at the beginning and needs more investigation.

In ContractLog I have implemented support for both homogeneous and heterogeneous integration approaches, in the so called *OWL2Prova API*, which can be configured and used in rule representation. Nevertheless, in the context of term typing using ontologies or object class hierarchies as order-sorted type systems, I use a hybrid prescriptive typing approach which exploits highly optimized external DL reasoner for type checking by means of subsumption checking and instance inference permitting also equivalence reasoning or inner anonymous existentials. In contrast to homogenous approaches such as SWRL (the union of OWL-DL and Datalog [HPSB$^+$04]) or DLP (the intersection of restricted OWL and Datalog [Gro03]) the heterogeneous approach has the advantage that is does not need any non-standard homogenous reasoning services, but may reuse existing implementations exploiting e.g. highly optimized external DL reasoner for DL reasoning tasks and a rule engine for the rule reasoning tasks. In most hybrid approaches, which combine ontologies and rules, the interaction between the subsystems is solved by additional *constraints* in the body of (Datalog) rules, which restrict the values of variables and constants in the *constraint clauses* to range over the instances of the externally specified DL concepts. Although this constraint solution to some extend ensures expressiveness in the sense that it allows to reuse the binary relations (classes and object properties) defined in the external ontology, it has some serious drawbacks in the context of term typing, since this approach leaves the usual operational semantics of resolution and unification unchanged. Hence, the constraints apply only in the body of a rule according to the selection function of the LP inference algorithm which typically selects the "left-most" unified literal as next subgoal. As a result, according to the standard depth-first strategy the type constraints are used relatively late in the derivation process. In particular, they do not apply directly during the unification between (sub)goals and rule heads, which leads to large and needless search spaces (failed search trees) and might lead to unintended results since there is no way to directly verify and exclude the unification of typed free (sub)goals

and typed rule heads which do not match according to their type definitions. Moreover, fresh typed constant terms, i.e. new DL instances of a particular DL type (class), can not be introduced directly in rule heads since this would require special semantics and formalizations, e.g., with conjunctive rule heads consisting of a literal which defines the individual and another literal which defines its type (or Lloyd-Topor transformations). For instance, a rule with a variable X of type C in the rule head, would need an extra constraint, e.g. $type(X, C)$ in the body, e.g.: $p(X) \leftarrow q(X), type(X, C)$. Obviously, such a constraint rule is dependent on the order of the body atoms: if $type(X, C)$ is before $q(X)$ the variable X might not be bound to a ground individual and hence can not be verified; on the other hand, if it is after $q(X)$ it does not directly constrain the subgoal $q(X)$, which might be possibly very deep including many variable bindings which are not of type C.

In contrast to these constraint approaches or approaches which apply explicit additional DL-queries/atoms in the rules' body for querying the type system, I have implemented a hybrid typed logic in ContractLog where dynamic type checking for prescriptively typed term is directly integrated into the unification process, i.e. directly implemented within the operational semantics, rather than indirectly through the resolution-based inference mechanism solving the additional constraint/query terms in the body of rules. The interaction during typed unification between the rule component and the DL type system is based on entailment. As a result, the hybrid typed method provides a built-in technical separation between the inferences in the rule component and the type checks in the DL type system (resp. Java type system) which results in:

- in more efficient computations, since the type restrictions directly apply during the unification of typed terms
- higher flexibility with minimal interfaces (based on entailment) between rules component and the type system
- robustly decidable combinations, even in case where the rule language is far more expressive than Datalog and no safeness restrictions apply
- enhanced language expressiveness, e.g. ad-hoc polymorphism with overloading is possible, type casting during order-sorted unification, data abstraction and modularization
- existing implementations can be reused
- rules can be more intuitively modelled and easily combined and interchanged

In the following two sections I will describe the syntax and semantics of the typed logic which I have implemented in the ContractLog KR. In particular I will elaborate on hybrid description logic programs (*hybrid DLPs*), namely description logic Semantic Web type systems (DL type systems with DL-types) which are used for *term typing of LP rules based on a polymorphic, order-sorted, hybrid DL-typed unification as operational semantics for hybrid DLPs.*

3.1 Syntax of Typed ContractLog

A ContractLog LP is an extended LP (ELP) [Lif92], i.e. a LP with monotonic explicit negation and non-monotonic default negation. An ELP is a set of clauses

of the from $H \leftarrow B$, where H is a literal over L called the head of the rule, and B is a set of literals over L called the body of the rule. A literal B_i, which might be default negated "$\sim B_i$", is either an atom or the negation $\neg$ of an atom, where $\sim$ denotes default negation written as $not(...)$ and $\neg$ is denoted as explicit negation written as $neg(...)$. Roughly, default negation means, everything that can not be proven as true is assumed to be false. A rule is called a fact if it only consists of the rule head H. An atom is a n-ary formula containing terms $p(a, X, f(...))$, where p is the predicate name. A term is either a constant a, a variable X or a n-ary complex term/function $f(...)$. A goal/query G is a headless clause defining a conjunction of literals (positive or negative atoms) $L_1 \wedge .. \wedge L_i$ where each L_i is called a subgoal. A *query* is embedded in the built-in function : $-sovle(...)$ or : $-eval(...)$.

ContractLog assumes not just a single universe of discourse, but several domains, so called sorts (types). I first describe the basic extension of the language towards a multi-sorted logic, i.e. the extension of the signature and the variables of the alphabet with sorts.

Definition 2. *(**Multi-sorted Signature**) A multi-sorted signature S is defined as a tuple $\langle S_1, .., S_n, \overline{P}, \overline{F}, arity, \overline{c}, sort\rangle$ where:*

1. *$S_1, .., S_n$ is a list of type symbols called sorts (types)*
2. *$\overline{P}$ is a finite sequence of predicate symbols $\langle P_1, .., P_n\rangle$.*
3. *$\overline{F}$ is a finite sequence of function symbols $\langle F_1, .., F_m\rangle$*
4. *For each P_i respectively each F_j, $arity(P_i)$ resp. $arity(F_j)$ is a non-zero natural number denoting the arity of P_i resp. F_i.*
5. *$\overline{c} = \langle c1, .., c_o\rangle$ is a finite or infinite sequence of constant symbols.*
6. *sort is a function that associates with each predicate, function or constant its sorts*

The function sort is defined as follows:

- *if p is a predicate of arity k, then $sort(p)$ is a k-tuple of sorts $sort(p) = (X_1, ..X_k)$ where each X_i is of some sort S_j.*
- *if f is a function of arity k, then $sort(f)$ is a $k+1$-tuple of sorts defining the sorts of the domain and the range of f*
- *if c is a constant, then $sort(c)$ gives the sort of c.*

I define the following three basic types of sorts

1. *primitive sorts are given as a fixed set of primitive data types such as integer, string, etc.*
2. *function sorts are complex sorts constructed from primitive sorts (or other complex sorts) $S_1 \times ... \times S_n \rightarrow S_{n+1}$*
3. *Boolean sorts are (predicate) statement of the form $S_1 \times ... \times S_n$*

Additionally, each variable V_j in the language with a multi-sorted signature is associated with a specific sort: $sort(V_j) = S_i$. The intuitive meaning is that a predicate or function holds only if each of its terms is of the respective sort given by *sort*. A binary equality predicate = exists in the language. I write $t_1 = t2$ instead of $= (t1, t2)$.

Definition 3. *(**Multi-sorted Logic**) A multi-sorted logic associates which each term, predicate and function a particular sort:*

1. *Any constant or variable t is a term and its sort is given by $sort(t)$*
2. *Let $F(t_1, .., t_n)$ be a function then it is a term of sort S_{n+1} if $sort(F) = \langle S_1, .., S_n, S_{n+1} \rangle$, i.e. F takes argument of sort $S_1, .., S_n$ and returns arguments in sort S_{n+1}.*

I now extend the language to consider external sort/type alphabets and combined signatures as basis for combined knowledge bases and the integration of external type systems into rule bases.

Definition 4. *(**Type alphabet**) A type alphabet T is a finite set of monomorphic sort/type symbols built over the distinct set of terminological class concepts of a (external type) language Σ.*

Informally, a typed ContractLog LP consists of an extended LP with typed terms and a set of external (order-sorted) type systems in which the types (sorts) are defined based on their type alphabets. An external type system might possibly define a complete knowledge base with types/sorts (classes) and individuals associated with these types (instances of the classes). The combined signature is then the union of the two (or more) signatures, i.e. the combination of the signature of the rule component and the signatures of the external type systems / knowledge bases including their type alphabets, their functions and predicates and their individuals.

Definition 5. *(**Combined Signature**) A combined signature Σ is the union of all its constituent finite signatures: $\Sigma = \langle S_1 \cup .. \cup S_n \rangle$*

Based on this definition of a combined signature I now describe the concrete syntax of typed ContractLog. Using equalities ContractLog implements a notion of default inequality for the combined set of individuals/constants which leads to a less restrictive unique name assumption:

Definition 6. *(**Default Unique Name Assumption**) Two ground terms are assumed to be unequal, unless equality between the terms can be derived.*

The type systems considered in ContractLog are order-sorted.

Definition 7. *(**Order-sorted Type System**) A finite order-sorted type system T comes with a partial order $\leq$, i.e. T under $\leq$ has a greatest lower bound $glb(r_1, r_2)$ for any two types r_1 and r_2 having a lower bound at all. Since T is finite also a least upper bound $lub(r_1, r_2)$ exists for any two types r_1 and r_2 having a upper bound at all.*

Currently, ContractLog supports two external order-sorted type systems (with sub-type relations), one is *Java* and the other one are *Semantic Web ontologies* (defined in OWL or RDFS) respectively Description Logic KBs. In the following I describe the syntax of typed ContractLog for external DL-type systems. I

have chosen a **prescriptive typing approach**, where types are direct properties of the logical formulas, i.e. are directly attached to terms in a type relation $t : r$, denoting that a term t has a type r. This has several advantages for the semantics.

ContractLog supports webized Semantic Web ontologies defined in RDFS or OWL (OWL Lite or OWL DL) as type systems. That is, the combined signature consisting of the finite signature of the rule component and the finite signature of the ontology language(s). Note that the approach is general and can be extended to different DL languages defining DL KBs as type systems. Hence, in the following I use the term DL also for Semantic Web ontology languages and knowledge bases. For the reason of understandability, I assume only one external type system in the following.

The type alphabet T is a finite set of monomorphic type symbols built over the distinct set of terminological atomic concepts C in a Semantic Web ontology language L, i.e. defined by the atomic classes in the T-Box model. Note, that restricting types to atomic concepts is not a real restriction, because for any complex concept such as $(C \sqcap D)$ or $(C \sqcup D)$ one may introduce an atomic concept A_C, i.e. add the axiom $C \sqsubseteq A_C$ to the T-Box, and use A_C as atomic type instead of the complex type. This approach is also reasonable from a practical point of view since dynamic type checking must be computationally efficient in order to be usable in a order-sorted typed logic with possible very large rule derivation trees and typed unification, i.e. fast type checks are crucial during typed term unification. I assume that the type alphabet is fixed (but arbitrary), i.e. no new terminological concepts can be introduced in the T-Box by the rules at runtime. This ensure that I can also apply static type checking on the used DL-types in ContractLog at compile time (during parsing the LP script).

I use a prescriptive typing approach for the "has-type" relation $t : r$. The set of constants/individuals is built over the set of individual names in L, but I do not fix the constant names and allow arbitrary fresh constants (individuals) (under default UNA) to be introduced in the head of rules and facts. The precise syntax is a follows:

Definition 8. ***(DL-typed Terms)*** *A type is a terminological concept/class defined in the type system (T-Box model) possibly prefixed by a URI namespace abbreviation separated by _, i.e. namespace_Type. A typed term is denoted by the relation $t : r$ for typed variable terms and $r : t$ for typed constant terms, i.e. $ns\_a : ns\_C$ denoting that individual ns_a is of type ns_C or $X : ns\_C$, i.e. variable X is of type ns_C.*

The type ontologies are typically provided on the Web under a certain URL and types and individuals are represented as resources having an webized URI. Defined namespace can be used to avoid name conflicts and namespace abbreviations facilitate are more readable language. Note, that fresh individuals which are introduced in rules or facts apply locally within the scope of the predicate/function and rule in which they are defined, i.e. within a reasoning chain; in contrast to the individuals defined in the A-box model of the type system which apply globally as individuals of a class (type).

Example 1.

```
% Import external type systems

import("http//../dl_typing/businessVocabulary1.owl").
import("http//../dl_typing/businessVocabulary2.owl").
import("http//../dl_typing/mathVocabulary.owl").
import("http//../dl_typing/currencyVocabulary.owl").

reasoner("dl"). % configure reasoner (OWL-DL=Pellet)


% Rule-based Discount Policy

discount(X:businessVoc1_Customer, math_Percentage:10) :-
    gold(X: businessVoc1_Customer).
discount(X: businessVoc1_Customer, math_Percentage:5) :-
    silver(X: businessVoc1_Customer).
discount(X: businessVoc1_Customer, math_Percentage:2) :-
  bronze(X: businessVoc1_Customer).

% Note that this rules use a different vocabulary
% if the types "Client" and "Customer" are equal
% both typed rule sets unify
% Class equivalence between both "types"
% is defined in the second OWL-DL ontology

gold(X:businessVoc2_Client) :-
   spending(X:businessVoc2_Client, S:currency_Dollar),
   S:currency_Dollar > currency_Dollar:1000.
silver(X:businessVoc2_Client) :-
   spending(X:businessVoc2_Client, S:currency_Dollar),
   S:currency_Dollar > currency_Dollar:500.
bronze(X:businessVoc2_Client) :-
   spending(X:businessVoc2_Client, S:currency_Dollar),
   S:currency_Dollar > currency_Dollar:100.


% Facts

spending(businessVoc1_Customer:Adrian, currency_Dollar:1000).
spending(businessVoc1_Customer:Aira, currency_Dollar:200).

% Query

:-solve(discount(X:businessVoc_Client, Y:math_Percentage)).
```

The examples shows term typing with "plug-able" (imports of) Semantic Web type vocabularies. Remarkably, $businessVocabulary1$ defines a type $Customer$ which is defined to be equal to the type $Client$ in $businessVocabulary2$. Hence, both types unify and the first three rules relate to the second three rules of the discount policy as well as to the defined facts. The user can use both types interchangeable to define queries on the hybrid KB, i.e., the domain-specific vocabulary of choice can be used.

Free DL-typed variables are allowed in facts. As I will describe in the semantics section they act as instance queries on the ontology, i.e. they query all individuals of the given type and bind them to the typed variable. In addition ContractLog provides a special query predicate which can be used in the body of rules to interact with the ontology component and explicitly express queries, such as concept membership, role membership or concept inclusion on the DL knowledge base. The special query predicate *rdf* (implemented in owl.prova library) is used to query external ontologies written in RDF(S) or OWL (OWL Lite or OWL DL).

Example 2.

```
% Bind all individuals of type "Wine" to the variable "Subject"
%using the owl ontology WineProjectOWL.owl and the "rdfs" reasoner
rdf(
    "./examples/function_tests/owl/testdata/WineProjectOWL.owl",
    "rdfs",
    Subject,"rdf_type","http://www.owl-ontologies.com/unnamed.owl#Wine")

% Use the transitive reasoner and namespace abbreviations
rdf(
    "./rules/function_tests/owl/testdata/WineProjectOWL.owl",
    "transitive",
    Subject,"rdfs_subClassOf","default_Wine")
```

The first argument specifies the URL of the external ontology. The second argument specifies the external reasoner which is used to infer the ontology model and answer the query. Note, that this hybrid method using an external reasoner to answer a queries provides a technical separation between the inferences in the Description Logic part which is solved by an optimized external DL reasoner and the Logic Programming components which is solved by the rule engine. As a result the combined approach is robustly decidable, even in case where the rule language is far more expressive than Datalog. Moreover, the triple-based query language also supports queries to plain RDF data sources, e.g. Dublin Core meta data. The following predefined reasoner are supported:

- "" — "empty" — null = no reasoner
- default = OWL reasoner
- transitive = transitive reasoner
- rdfs = RDFS rule reasoner

- owl = OWL reasoner
- daml = DAML reasoner
- dl = OWL-DL reasoner
- swrl = SWRL reasoner
- rdfs_full = rdfs full reasoner
- rdfs_simple = rdfs simple reasoner
- owl_mini = owl mini reasoner
- owl_micro = owl micro reasoner

User-defined reasoners can be easily configured and used. By default the specified reasoners are used to query the external models on the fly, i.e. to dynamically answer the queries using the external reasoner. But, a pre-processing mode is also supported. Here the reasoners are used to pre-infer the ontology model, i.e. build an inferred RDF triple model where the logical DL entailments such as transitive subclasses are already resolved at compilation time. Queries then operate on the inferred model and are hence much fast to answer, however with the drawback that updates of the ontology model require a complete recompilation of the inferred model.

4 Semantics of Typed ContractLog

4.1 Declarative Semantics: Multi-Sorted Logic

The semantics of a LP P is typically defined wrt to the closed Herbrand universe $B(P)$. I now extend the domain of discourse towards a combined knowledge base defined over a combined signature where individuals and types (sorts) from one or more type systems outside of the fixed domain of the rule component are taken into account. The semantics of the combined KB based on an extended Herbrand Base is then defined wrt to the combined signature.

Definition 9. *(**Combined Knowledge Base**) The combined knowledge base of a type ContractLog LP* $KB = \langle \Pi^{LP}, \Phi^{TS} \rangle$ *consists of a finite set of (order-sorted) type systems / type knowledge bases* $\Phi^{TS} = \Phi_1^{TS} \cap .. \cap \Phi_n^{TS}$ *and a typed ContractLog KB* Π^{LP} *represented as a defeasible extended typed LP.*

The combined signature of the combined KB is the union of all constituent signatures, i.e. each interpretation of a ContractLog LP has the set of ground terms of the combined signature as its fixed universe.

Definition 10. *(**Extended Herbrand Base**) Let* $KB = \langle \Pi^{LP}, \Phi^{TS} \rangle$ *a typed combined ContractLog LP P. The extended Herbrand Base of P, denoted* $B(P)$*, is the set of all ground literals which can be formed by using the predicate symbols in the combined signature with the ground typed terms in the combined universe* $U(P)$*, which is the set of all ground typed terms which can be formed out of the constants, type and function symbols of the combined signature.*

The grounding of the combined KB is computed wrt the composite signature.

Definition 11. *(**Grounding**) Let P be a typed (combined) ContractLog LP and C its set of constant symbols in the combined signature. The grounding $ground(P)$ consists of all ground instances of all rules in P w.r.t to the combined multi-sorted signature which can be obtained as follows:*
The ground instantiation of a rule r is the collection of all formulae $r[X_1 : S_1/t_1, .., X_n : S_n/t_n]$ with $X_1, .., X_n$ denoting the variables and $S : 1, .., S_n$ the types of the variables (which must not necessarily be disjoint) which occur in r and $t_1, .., t_n$ ranging over all constants in C wrt to their types. For every explicit query/goal $Q[Y_1 : T_1, .., Y_m : T_m]$ to the type system, being either a fact with one or more free typed variables $Y_1 : T_1, .., Y_m : T_m$ or a special query atom $rdf(...)$ with variables as arguments in the triple-like query, the grounding $ground(Q)$ is an instantiation of all variables with constants (individuals) in C according to their types.

The interpretation I of a typed program P then is a subset of the extended Herbrand base $B(P)$.

Definition 12. *(**Multi-sorted Interpretation**) Let $KB = \langle \Pi^{LP}, \Phi^{TS} \rangle$ be a combined KB and C its set of constant symbols. An interpretation I for a multi-sorted combined signature Σ consists of*

1. *a universe $| I | = S_1^I \cup S_2^I \cup .. \cup S_n^I$, which is the union of the types (sorts), and*
2. *the predicates, function symbols and constansts/individuals C in the combined signature, which are interpreted in accordance with their types.*

The assignment function σ from the set of variable V into the universe $|I|$ must respect the sorts/types of the variables (in order-sorted type systems also subtypes). That is, if X is a variable of type T, then $\sigma(X) \in T^I$. In general, if ϕ is a typed atom or function in Π^{LP} and σ an assignment to the interpretation I, then $I \models \phi[\sigma]$, i.e. ϕ is true in I when each variable X of ϕ is substituted by the values $\sigma(X)$ wrt to its type. Since the assignment to constant and function symbols is fixed and the domain of discourse corresponds one-to-one with the constants C in the combined signature, it is possible to identify an interpretation I with a subset of the extended Herbrand base.

The assignment function is given as a query from the rule component to the type system, so that there is a separation between the inferences in a type system and the rule component. Moreover, explicit queries to a type system defined in the body of a rule, e.g. ontology queries (special *rdf* query or free DL-typed facts) are based on this hybrid query mechanism. The query interaction between the rules and the type system is based on entailment. I now define the notion of model for a typed ContractLog LP

Definition 13. *(**Model**) et $KB = \langle \Pi^{LP}, \Phi^{TS} \rangle$ be a combined KB of a typed ContractLog LP P. An interpretation I is a model of a untyped ground atom $a \in B(\Pi^{LP})$ or I satisfies a, denoted $I \models a$ iff $a \in I$. I is a model for a typed*

ground atom $b \in B(\Pi^{LP})$, or I satisfies b, denoted $I \models b$, iff $b \in I$ and for every typed term $t_i : s_i$ in b the type query $s_i(t_i)$ (is t_i of type s_i) is entailed in Φ^{TS}, i.e. $\Phi^{TS} \models s_i(t_i)$ (in an order sorted type system subtypes are considered, i.e. t_i is of the same or a subtype of s_i). I is an interpretation of an ground explicit query/goal Q to the type system Φ^{TS} if $\Phi^{TS} \models Q$.
I is a model of a ground rule $r : H \leftarrow B$ iff $I \models H(r)$ whenever $I \models B(r)$. I is a model a typed program P (resp. a combined knowledge base KB), denoted by $I \models P$, if $I \models r$ for all $r \in ground(P)$.

The default semantics for ContractLog is extended well-founded semantics. To support well-founded semantics for negated queries to external type systems, i.e. the negated query $\sim Q$ succeeds, if the query fails, i.e. $\Phi^{TS} \not\models Q$, I extend the definition of unfounded sets with additional query atoms to the external type systems (knowledge bases).

Definition 14. ***(Unfounded Set)*** *Let P be a typed LP (combined KB). Let I be a partial interpretation. Let $\alpha \subseteq B_P$ be a set of ground atoms. α is an unfounded set of P wrt I, if for every atom $A \in \alpha$ and every ground rule instance $A \leftarrow \beta \in ground(P)$ with a finite sequence of ground standard literals and query atoms (querying the type system) in β at least one of the following conditions holds:*

1. *at least one standard body literal $L \in \beta$ is false in I.*
2. *at least one standard positive body literal $B \in \beta$ is contained in α.*
3. *at least one query atom $q \in \beta$ is false in $I \cup \neg\alpha$.*
4. *at least one negative query atom $\sim q \in \beta$ is true in I*

Remark 1. Although, in this paper I do not consider Java type systems it should be noted, that the structures in Java type systems are usually not considered as interpretations in the strict model-theoretic definition, but are composite structures involving several different structures whose elements have a certain inner composition. However, transformations of composite structures into their flat model theoretic presentations is in the majority of cases possible. However, from a practical point of view, it is convenient to neglect the inner composition of the elements of the universe of a structure. These elements are just considered as "abstract" points devoid of any inherent meaning. Note that this does not hold for procedural functions and predicates (boolean-valued attachments / built-ins) defined on them.

This structural mapping between objects from their interpretations in the Java universe to their interpretation in the rule system ignoring finer-grained differences that might arise from the respective definitions is given by the following isomorphism.

Definition 15. *(Isomorphism) Let Σ be a signature with sorts $S_1, ..S_n$ and let M_1, M_2 be two interpretations of Σ, then $f : |M_1| \rightarrow |M_2|$ is an isomorphism of M_1 and M_2 if f is a one-to-one mapping from the universe of M_1 onto the universe of M_2 such that:*

1. *For every sort* S_i, $m \in S_i^{M_1}$ *iff* $f(m) \in S_i^{M_2}$
2. *For every constant* c, $f(c^{M_1}) = c^{M_2}$
3. *For every n-ary predicate symbol* p *wit n-tuple* $m_1, .., m_n \in |M_1|$, $\langle m_1, .., m_n \rangle \in p^{M_1}$ *iff* $\langle f(m_1), .., f(m_n) \rangle \in p^{M_2}$
4. *For every n-place function symbol* F, $f(F^{M_1}(m_1, .., m_n)) = F^{M_2}(f(m_1), .., f(m_n))$

For instance, in ContractLog an isomorphism between Boolean Java objects and their model-theoretic truth value is defined, which makes it possible to treat boolean-valued procedural attachments as body atoms in rules and establish an entailment relation as defined above between the Java type system and the rule component. Another example are String objects which are treated as standard constants in rules. Primitive datatype values from the ontology (XML) domain can be mapped similarly.

4.2 Operational Semantics: Hybrid Polymorphic Order-Sorted Unification

In the following I define the operational semantics of typed ContractLog LPs. In contrast to other hybrid approaches which apply additional constraint atoms as type guards in the rule body and leave the usual machinery of resolution and unification unchanged, the operational semantics for prescriptive types in ContractLog's typed logic is given by an order-sorted unification. Here the specific computations that are performed in this typed language are intimately related to the types attached to the atomic term symbols. The order-sorted unification yields the term of the two sorts (types) in the given sort hierarchy. This ensures that type checks apply directly during typed unification of terms at runtime enabling ad-hoc polymorphism of variables leading e.g. to different optimized rule variants and early constrained search trees. Thus, the order-sorted mechanism provides higher level of abstraction, providing more compact and complete solutions and avoiding possibly expensive backtracking.

The standard untyped unification algorithm in logic programming serves as a tool for the resolution principle. It takes a set of expressions as its input and yields the most general unifier (mgu) for this set of formulas. A substitution σ is called a unifier for the set of formulas $\{E_1, ..., E_n\}$ if $E_1\sigma = E_2\sigma = ... = E_n\sigma$. A unifier σ for the set of expressions $\{E_1, ..., E_n\}$ is called the most general unifier if, for any other unifier for the same set of formulas θ, there is yet another unifier ι such that $\theta = \sigma * \iota$. For a survey on unification theory see, e.g. [BS01,Sie89]. In the following I first define the rules for untyped unification in terms of equation-solving transformations [MM82] for elimination (E), decomposition (D), variable binding (B) and orientation (O). The judgements beneath the horizontal line is the conclusion of the rule and the judgements below the line are the premises of the rule. The computation starts with a set of equations $E = t_1 = t'_1, .., t_n = t'_n$ representing the terms to be unified, and to transform E into the solved set of equations E' using the four given rules E, D, B and O.
(E) $\frac{E \& Y \doteq Y}{E}$, where Y is a variable

(D) $\frac{E\&f(t_1,..,t_n)\doteq f(t_1^{'},..,t_n^{'})}{E\&t_1\doteq t_1^{'}\&...\&t_n\doteq t_n^{'}}$
(B) $\frac{E\&Y\doteq t}{\sigma(E)\&Y\doteq t}$, where Y is a variable, t is a constant or variable term, and Y occurs in E but not in t, and where $\sigma = \{Y/t\}$
(O) $\frac{E\&t\doteq Y}{E\&Y\doteq t}$, where Y is a variable and t is not a variable

The computation starts with a set of equations $E = \{t_1 \doteq t_1^{'}, ..., t_n \doteq t_n^{'}\}$ where $\{t_i/t_i^{'}\}$ describes the pairs of unifiable terms. Using the four rules E is transformed into a set of equations $E^{'} = \{Y_i | i \in \{1, .., n\}\}$ where Y_i are distinct variables which do not occur elsewhere in $E^{'}$, i.e. $\sigma = \{Y_1/t_1, .., Y_n/t_n\}$ is the most general unifier of the unification problem given by the original set of equations E. Unification fails, if there is an equation $f(t_1, .., t_n) \doteq g(t_1^{'}, .., t_m^{'})$ in E with $f \neq g$ or if there is an equation $Y \doteq t$ in E such that $Y \doteq t$ and Y occurs in t. I now extend this basic set of unification rules to a hybrid polymorphic order-sorted DL-typed unification. I restrict type checking to finding the lower bound of two types (r_1, r_2) under the partial order $\leq$ of the order-sorted model with an upper bound $\top$ and a lower bound $\bot \equiv$ *empty* and replace the type of a term with the more specific type concept. Therefore, I define a *lower* operation by:

$lower(r_1, r_2) = (r_2/r_1) \rightarrow r_1$, if $r_1 \leq r_2$ resp. $lower(r_1, r_2) = (r_1/r_2) \rightarrow r_2$, if $r_1 > r_2$
$lower(r_1, \top) = (\top/r_1) \rightarrow r_1$ resp. $lower(\top, r_2) = (\top/r_2) \rightarrow r_2$, where $\top =$ *untyped*
$lower(r_1, r_2) = \bot$, otherwise, where $\bot =$ empty type.

Note that, the operation *lower* requires at most two queries to the external type system to compute the lower bound of two types having a lower bound at all. To enable polymorphic typing of variables during typed unification, i.e. a variable may change its type dynamically, I introduce a set $P = \{t_1 : r_1, .., t_n : r_n\}$ of type restrictions, denoting that the term t_i (currently) has type r_i, as a prefix to the set of equations E: $P\&E$. The modified and extended type rules for order-sorted unification are as follows:

(E) $\frac{P\&E\&Y\doteq Y}{P\&E}$, where Y is a variable
(D) $\frac{P\&E\&f(t_1,..,t_n)\doteq f(t_1^{'},..,t_n^{'})}{P\&E\&t_1\doteq t_1^{'}\&...\&t_n\doteq t_n^{'}}$
(B') $\frac{P\&Y:r\&E\&Y\doteq t}{P'\&\sigma(E)\&Y\doteq t}$, where Y is a variable, t is a variable or non-variable term, and Y occurs in E but not in t, and where $\sigma = \{Y/t\}$. $P\&t : r$ reduces to P' using the auxiliary type rules ET and BT
(O) $\frac{P\&E\&t\doteq Y}{P\&E\&Y\doteq t}$, where Y is a variable and t is not a variable

The auxiliary rules for polymorphic unification of types are:
(ET) $\frac{P\&f(t_1,..,t_n):r}{P}$, if $f : r_1..r_n \rightarrow r_2$ and $r_2 \leq r$ and (ET') $\frac{P\&f(t_1,..,t_n):\top}{P}$
(BT) $\frac{P\&Y:r_1\&Z:r_2}{P\&Y:lower(r_1,r_2)}$

DL-typed unification fails
(1) if there is an equation $f(t_1,..,t_n) \doteq g(t_1^{'},..,t_m^{'})$ in E with $f \neq g$ or
(2) if there is an equation $Y \doteq t$ in E such that $Y \doteq t$ and $Y \in t$ or
(3) if there is an equation $Y \doteq t$ in E such that $Y : r_1$ and $t : r_2$, where t is a constant term and $r_2 > r_1$ or
(4) if there is an equation $Y \doteq Z$ in E such that $Y : r_1$ and $Z : r_2$ and $lower(r_1, r_2) = \perp$, where Y and Z are variable terms.
Otherwise, if $E^{'} = \{Y_i | i \in \{1,..,n\}\}$ then $\sigma = \{Y_1/t_1,..,Y_n/t_n\}$ is the mgu of the unification problem given by the original set of equations E.

In contrast to the unsorted unification, (B') now involves ad-hoc polymorphic unification of order-sorted types with a subtype resp. equivalence test $r_2 \leq r_1$ and a computation of the lower bound of two types $lower(r_1, r_2)$ in the auxiliary rules, possibly assigning the more specific type (i.e. the lower type) to a variable. The variables may change their type during unification according to the rule (BT) and the *lower* operation. (ET') is introduced to reduce unification to special cases of the binding rule (B) in the untyped case without type checking, i.e. to efficiently process untyped variables. That is the order-sorted unification coincides with the untyped unification, if the underlying LP does not contain typed terms. I require that there are no infinite function definitions such as $f(f(f(...)))$ and hence introduce the following restriction for typed unification: $\sigma = \{X/f(t1,..,tn)\}$ if $X \ni f$, i.e. the variable X is not allowed to occur in the function f with which it is unified. Furthermore, I restrict the unification algorithm to only well-typed terms, i.e. the type of the argument t_i in $f(t_1,..,t_n)$ must be a subtype of the type r_i for $f : r_1..r_n \rightarrow r$, where r is the target type of the function. I define the type of predicates, terms and lists to be untyped by default denoted by $\top$. As a result untyped complex terms or lists can be unified only with other untyped variables. Informally the polymorphic order-sorted unification rules state:

- Untyped Unification: Ordinary untyped unification without type checking
- Untyped-Typed Unification:The untyped query variable assumes the type of the typed target
- Variable-Variable Unification:
  (1) If the query variable is of the same type as the target variable or belongs to a subtype of the target variable, the query variable retains its type (according to *lower*), i.e. the target variable is replaced by the query variable.
  (2) If the query variable belongs to a super-type of the target variable, the query variable assumes the type of the target variable (according to *lower*), i.e. the query variable is replaced by the target variable.
  (3) If the query and the target variable are not assignable ($lower = \perp$) the unification fails
- Variable-Constant Term Unification:
  (1) If a variable is unified with a constant of its super-type, the unification fails.

(2) If the type of the constant is the same or a sub-type of the variable, it succeeds and the variable becomes instantiated.

– Constant-Constant Term Unification: Both constants are equal and the type of the query constant is equal to the type of the target constant.

Complex terms such as lists are untyped by default and hence are only allowed to be unified with untyped variables.

After having defined the general semantics for order-sorted typed unification, I will now discuss its implementation for DL type systems in ContractLog.

A DL resp. Semantic Web type system consists of a T-Box defining the order-sorted types and their relations and a possible empty A-Box defining global individuals of the defined types. The T-Box typically has a partial order $\leq$. ContractLog assumes $owl : Resource$ as common maximum class under the partial order of any DL type system, i.e. $\top \equiv owl : Resource \equiv java.lang.Object \equiv untyped$ and $owl : Nothing$ to be the minimum lower bound $\bot \equiv owl : Nothing \equiv empty$. Note that both type system, Java and DL, coincide in the untyped framework which can be downcasted from $java.lang.Object$ or $owl : Resource$. The DL type checks, applied as hybrid queries to the DL ontology type system(s) during the unification process, are primarily concerned with *Instantiation*, i.e. querying whether an individual is an instance of a class or deriving all individuals for a particular class, and *Subsumption*, i.e. deciding wether a class is subsumed by another class (is a subclass of). *Equivalence* inferences which check if two classes or individuals are equal (or disjoint) and hence can be unified is another important task which is provided by expressive ontology languages, e.g. OWL-DL (SHOIN(D)). The typed unification rules take into account, that the type system is defined by one or more DL ontologies and that subtype tests via subsumption are constrained by the expressiveness of the DL query language. For instance, in OWL there is no way to express the concept of a most general superclass or a most specific type. Although, it is possible to compute such statements by applying iterative subsumption queries, such an approach will impose greater computational burdens for dynamic type checking. Therefore, in ContractLog I restrict type checking to finding the lower bound of two types (r_1, r_2) under $\leq$ in the *lower* operation of the typed unification and replace the type of a term with the more specific type concept in the unification rules. The operation *lower* requires at most two subsumption queries to the external DL reasoner to compute the lower bound under the partial order $\leq$. If the type system consists of more than one DL ontology, the ontologies are merged into one combined ontology. The common super class under which all ontologies are subsumed is the concept "Resource". Hence, the partial order $\leq$ still holds for the combined ontology under the assumption that no cycles are introduced. Cross links between the component ontologies might be defined, e.g. via relating classes with $owl : equivalentClass$ or $owl : disjointWith$. Note that, this may introduce conflicts between terminological definitions which need conflict resolution strategies, e.g. defined by defeasible reasoning.

Remark 2. It is very important to note the difference between my order-sorted typing approach and hybrid approaches which apply additional DL atoms resp. DL constraints to query the DL ontology component such as e.g. dl-programs [Eit04] or Carin [Lev96]. In my prescriptive approach the type checks in terms of DL-queries to the DL component apply during the typed unification process and constrain the unification of terms. The operational semantics provides a "built-in" technical separation between the rule inferences and the DL inferences which directly applies during typed term unification and results in flexible formalisms that are robustly decidable, even in the case where the rule language is far more expressive than Datalog. This particular combination cannot be seen neither as a super or subset of homogeneous approaches such as SWRL nor as related to existing hybrid approaches which apply DL constraints, since the semantics is completely different.

5 Implementation and Integrating of Types into Rule Markup Languages

I have implemented and evaluated my hybrid DL-typed approach (by means of industrial use cases) in the ContractLog KR. ContractLog is an expressive KR framework developed in the RBSLA project (http://ibis.in.tum.de/staff/paschke/rbsla/index.htm) which focuses on sophisticated knowledge representation concepts for service level management (SLM) of IT services. ContractLog is based on Prova / Mandarax, an open-source Java-based backward-reasoning rule engine with a Prolog like scripting syntax (http://www.prova.ws/). The ContractLog KR implements several logical formalisms such as event logics, defeasible logic, deontic logics and comes with a novel sequential resolution for extended LPs with extended defeasible Well-founded Semantics. The rule engine (an extension to the Prova inference engine) coming with ContractLog/RBSLA distribution (https://rbslasourceforge.net/projects/rbsla) performs query answering on the hybrid rules. As an external service I use the Jena API (http://jena.sourceforge.net) in combination Pellet (http://www.mindswap.org/2003/pellet/), integrated as an external DL reasoner. During resolution non-grounded facts with variables are interpreted as instance queries on the external service to decide wether an individual is an instance of a class given by the type of the bound variable or to derive all individuals explicitly introduced in the DL A-Box which are instances of the type (class) of the free variable, which becomes instantiated with the found individuals. During unification dynamic type checking must be performed via subsumption queries using the external DL reasoner. To achieve decidability I restricted the rule component to LPs with finite functions and the DL component to decidable DLs reaching from *ALC* to *SHIQ(D)* resp. *SHOIN(D)*. The only restriction is on the interchange of entailments between both formalisms in the sense that there is a unidirectional flow from the DL part to the LP part. In particular this means, that only consequences involving individuals which are explicitly introduced in the A-Box are passed from the DL reasoner to the LP reasoner, but new individuals can be introduced within the LP facts or rule

(heads) which do not affect query answering with the DL reasoner. The complexity for reasoning in my hybrid DL-typed logic is derived from the complexity of the component languages and the applied semantics for DL languages and extended LPs under WFS. The worst case complexity of reasoning in my hybrid DL-typed logic is *EXPTIME* for a combination of extended LPs and OWL Lite and *NEXPTIME* for a combination with OWL-DL. Note, that Jena also support RDFS reasoning and I also allow to define type systems in RDFS. Since, the RDFS reasoner omits many complex entailments such as equality reasoning , using RDFS to define light-weight type systems will lead to more efficient type unification.

Theorem 1: *Query answering in hybrid DL-typed Datalog is EXPTIME-complete for OWL-Lite (SHIQ(D)) type systems resp. NEXPTIME-complete for OWL-DL (SHOIN(D)) type systems.*

The ContractLog KR based on Prova provides a Prolog related scripting syntax to write LPs, e.g.

```
import("./dl_typing/WineProjectOWL.owl").
reasoner("dl"). % OWL DL reasoner

serve(X:vin_Wine):- wine(X:vin_Wine).
wine(vin_White_Wine:Chardonnay). wine(X:vin_Red_Wine).
```

The example defines a rule, where the variable X is of type vin_Wine (vin is the namespace prefix). The first fact introduces a new individual (constant term) $Chardonny$ which is of type vin_White_Wine and hence unifies with $X : vin\_Wine$. The second fact defines a variable X of type vin_Red_Wine which is used to query all individuals of type vin_Red_Wine from the DL A-box using the external DL reasoner, if X is a free variable resp. to check whether X is of type vin_Red_Wine, if X is a previously bound variable. The type system is imported via the $import(< URI >)$ predicate. Different external reasoner such as "$transitive$" (simple transitive reasoner), "$rdfs$" (rdfs reasoner), "owl" (owl lite reasoner), "dl" (owl dl reasoner) etc. are supported within the ContractLog implementation (see OWL2PROVA framework in the RBSLA/ContractLog distribution available at https://rbslasourceforge.net/projects/rbsla).

In addition to this Prolog related scripting syntax I provide a XML/RDF serialization based on RuleML (http://www.ruleml.org/). RuleML (Rule Markup Language) is a standardization initiative with the goal of creating an open, producer-independent XML/RDF based web language for rules. I use the @$type$ attribute to define typed terms.

```
<Implies>
  <head> <Atom>
    <Rel>serve</Rel>
    <Var @type="vin:WhiteWine">X</Var>
```

```
  </Atom></head>
  <body><And><Atom>
     <Rel>wine</Rel>
     <Var @type="vin:WhiteWine">X</Var>
  </Atom></And></body>
</Implies>
```

The RuleML documents are transformed into the target execution language, i.e. into ContractLog/Prova scripts and then executed in the ContractLog KR using the typed unification to process hyrbid DL-typed rules. I have implemented a XSLT stylesheet which transforms typed RuleML LPs into typed ContractLog scripts.

6 Summary

The main motivation for introducing Semantic Web or Java based types into declarative logic programs comes from Software Engineering, where principles such as data abstraction, modularization and consistency checks are vital for the development and maintenance of large rule bases. Distributed system engineering and collaboration, where domain-independent rules need to be interchanged and given a domain-dependent meaning based one or more common vocabularies in their target environments is an other. The possibility to use arbitrary Semantic Web ontologies and object-oriented domain models in declarative logic programming gives a highly flexible, extensible and syntactically rich language design with a precise semantics. Semantic Web counterparts of domain-specific ontologies, e.g. OWL Time, can be easily used as type systems giving the rule terms a domain-specific meaning. This syntactic and semantic combination which allows efficient declarative programming is vital for modern Semantic Web based rule systems. From a computational point of view, the use of order-sorted types can drastically reduce the search space, hence increasing the runtime efficiency and the expressive power, e.g. enabling overloading of rule variants. The tight combination of declarative and object-oriented programming via rich Java-based procedural attachments facilitates integration of existing procedural functionalities, tools and external data sources into rule executions at runtime. In this section I have presented a hybrid approach which provides a technical separation with minimal interfaces between the rule and type components leading to a robust, flexible and expressive typed logic with support for external Java and DL type systems, Java-based procedural attachments, modes and built-ins. The implementation follows a prescriptive typing approach and incorporates type information directly into the names of symbols in the rule language. The interaction between the rules and the type system is based on entailment in a multi-sorted logic. As an operational semantics a typed unification is applied which permits dynamic type checking, ad-hoc polymorphism, i.e., variables might change their types (coercion), and overloading, i.e. overloading of rules by using different types in their rule heads, leading to variants of the same rule.

References

[AE93] K. Apt and S. Etalle. *Mathematical Foundations of Computer Science*, chapter On the Unification Free Prolog Programs, pages 1–19. 1993.

[Ait91] *Towards the meaning of LIFE.* Springer, 1991.

[Bar88] *Introduction to Lambda Calculus*, GÃ?teborg, 1988. Programming Methodology Group.

[BB89] C. Beierle and S. BÃ?ttcher. Protos-l: Towards a knowledge base programming language. Technical report, 1989.

[BG04] D. Brickley and R. V. Guha. Rdf vocabulary description language 1.0: Rdf schema, http://www.w3.org/tr/rdf-schema/, 2004.

[Bra85] *An essential hybrid reasoning system: Knowledge and symbol level accounts for krypton.*, 1985.

[BS01] F. Baader and W. Snyder. chapter Unification Theory. Handbook of Automated Reasoning, pages 445–532. 2001.

[Car97] L. Cardelli. *Type Systems.* The Computer Science and Engineering Handbook. CRC Press, 1997.

[Dem99] *Type constraint solving for parametric and ad-hoc polymorphism*, 1999.

[Die88] *A Polymorphic Type System with Subtypes for Prolog.* Springer-Verlag, 1988.

[DLNS91] F. M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. A hybrid system with datalog and concept languages. *Trends in Artificial Intelligence*, LNAI 549:88–97, 1991.

[DZ92] P. W. Dart and J. Zobel. Efficient run-time type checking of typed logic programs. *J. Log. Program.*, 14:31–69, 1992.

[DZ2a] P. W. Dart and J. Zobel. *Types in Logic Programming*, chapter A Regular Type Language for Logic Programs, pages 157–187. 1992a.

[Eit04] *Combining answer set programming with description logics for the Semantic Web*, 2004.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* Springer, 1985.

[GM86] J. A. Goguen and J. Meseguer. *Functional and Logic Programming*, chapter Eqlog: Equality, Types and Generic Modules for Logic Programming, pages 295–263. Prentice Hall, 1986.

[GM87] J. A. Goguen and J. Meseguer. Order-sorted algebra i: Partial and overloaded operators, errors and inheritance. Technical report, Menlo Park, CA, 1987.

[Gog86] M. Gogolla. *Ã?ber partiell geordnete Sortenmengen und deren Anwendung zur Fehlerbehandlung in abstrakten Datentypen.* PhD thesis, 1986.

[Gro03] *Description logic programs: Combining logic programs with description logic.* ACM, 2003.

[GTW78] J. A. Goguen, J. W. Thatcher, and E. W. Wagner. *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types.*, volume Vol. IV of *Current Trends on Programming Methodology.* Prentice-Hall Int.,, 1978.

[Han91] M. Hanus. Horn clause programs with polymorphic types: Semantics and resolution. *Theor. Comput. Sci.*, 89:63–106, 1991.

[Han92] M. Hanus. *Types in Logic Programming*, chapter Logic Programming with Type Specifications, pages 91–140. 1992.

[Hen94] *Type Analysis of Prolog Using Type Graphs*, 1994.

[Hey05] *Nonmonotonic ontological and rule-based reasoning with extended conceptual logic programs*, Heraklion, Greece, 2005. Springer.

[HJ92] N. Heintze and J. Jaffar. *Types in Logic Programming*, chapter Semantic Types for Logic Programs, pages 141–155. 1992.

[HL94] P. M. Hill and J. W. Lloyd. *The GÂ?del Programming Language.* MIT Press, 1994.

[HPSB+04] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. Swrl: A semantic web rule language combining owl and ruleml, http://www.w3.org/submission/swrl/, 2004.

[HS88] M. Hohfeld and G. Smolka. Definite relations over constraint languages. Technical report, 1988.

[HT92] P. M. Hill and R. W. Topor. *Types in Logic Programming*, chapter A Semantics for Typed Logic Programs, pages 1–62. 1992.

[Hub87] *Extended Prolog for Order-Sorted Resolution*, San Francisco, 1987.

[KNW93] K. Kwon, G. Nadathur, and D. S. Wilson. Implementing polymorphic typing in a logic programming language. Technical report, 1993.

[Lak91] *Typed Prolog: A Semantic Reconstruction of the Mycroft-O'Keefe Type System*, San Diego, California, USA, 1991.

[Lev96] *A Representation Language Combining Horn Rules and Description Logics*, 1996.

[Lif92] *Answer Sets in General Non-Monotonic Reasioning (preliminary report).* Morgan-Kaufman, 1992.

[Lu98] L. Lu. Polymorphic type analysis in logic programs by abstract intepretation. *Journal of Logic Programming*, 36:1–54, 1998.

[MH04] D. L. McGuinness and F. v. Harmelen. Owl web ontology language, http://www.w3.org/tr/owl-features/, 2004.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17:348–375, 1978.
[Mil86] *Higher-Order Logic Programming*, 1986.
[Mis84] *Towards a Theory of Types in Prolog*, 1984.
[Mis85] *Declaration-free type checking*, New Orleans, Louisiana, United States, 1985.
[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Programming Lnaguages and Systems*, 4:258–282, 1982.
[MNFS91] D. Miller, G. Nadathur, Pfenning F., and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
[MO84] A. Mycroft and R. A. O'Keefe. A polymorphic type system for prolog. *Artif. Intell.*, 23:295–307, 1984.
[MSS05] B. Motik, U. Sattler, and R. Studer. Query answering for owl-dl with rules. *Journal of Web Semantics*, 3:41–60, 2005.
[Nai92] L. Naish. *Types in Logic Programming*, chapter Types and the Intended Meaning of Logic Programs, pages 189–216. MIT Press, 1992.
[NBB+02] D. Nardi, R. J. Brachman, F. Baader, W. Nutt, F. M. Donini, U. Sattler, D. Calvanese, R. Molitor, G. De Giacomo, R. Kuesters, F. Wolter, D. L. McGuinness, P. F. Patel-Schneider, R. MÃ?ller, V. Haarslev, I. Horrocks, A. Borgida, C. Welty, A. Rector, E. Franconi, M. Lenzerini, and R. Rosati. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2002.
[Obe62] A. Oberschelp. Untersuchungen zur mehrsortigen quantorenlogik. *Math. Ann.*, 145:297–333, 1962.
[Pfe92] F. Pfenning. *Types in Logic Programming*. The MIT Press, 1992.
[Red88] *Notions of polymorphism for predicate logic programs*, 1988.
[Ric05] R. Riccardo. On the decidability and complexity of integrating ontologies and rules. *Journal of Web Semantics*, 3, 2005.
[Sie89] J. Siekmann. Unification theory. *J. of Symbolic Computation*, 7:207–274, 1989.
[Smo89] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, 1989.
[SNGM9b] G. Smolka, W. Nutt, J. A. Goguen, and J. Meseguer. *Resolution of Equations in Algebraic Structures*, chapter Order Sorted Equational Computation. Academic Press, New York, 1989b.
[Sti86] M. E. Stickel. Schubert's steamroller problem: Formulations and solutions. *Journal of Automated Reasoning*, 2:89–101, 1986.
[Str00] C. Strachey. Fundamental concepts in programming languages. *Higher Order Symbol. Comput.*, 13:11–49, 2000.
[Wal87] C. Walther. *A Many-Sorted Calculus Based on Resolution and Paramodulation*. Research Notes in Artificial Intelligence. Pitman London and Morgan Kaufman, Los Altos, CA, 1987.
[Xu188] *A Type Inference System for Prolog*, 1988.
[Zob87] *Derivation of Polymorphic Types for PROLOG Programs.*, 1987.