Types for Web Rule Languages: a preliminary study

Horatiu Cirstea, Emmanuel Coquery, Wlodzimierz Drabent, Francois Fages, Claude Kirchner, Jan Maluszynski, Benjamin Wack

I3-D2

Types for Web Rule Languages: a preliminary study.

Project number:	IST-2004-506779
Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Document type:	D (deliverable)
Nature of document	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Nancy/I3-D2/D/PU/a1
Responsible editor:	Horatiu Cirstea and Claude Kirchner
Contributing participants:	Linköping, Nancy, Paris, Warsaw
Contributing workpackages:	I1, I4
Contractual date of delivery:	31 August 2004
Actual date of delivery:	03 September 2004

Abstract

We survey and analyse the relevant existing work on typing of rules, in particular on typing of constraint logic programs and discuss applicability of these approaches to the REWERSE reasoning and query languages under development by WG I1 and by WG I4. This is related to WG I1, developing logic programming like languages for reasoning on the web and with WG I4 investigating development of declarative query languages such as XPathLog and Xcerpt.

Keyword List

constraint, rewriting, rule based language, type system

Types for Web Rule Languages: a preliminary study.

Horatiu Cirstea[1] **and Emmanuel Coquery**[2] **and Włodzimierz Drabent**[3] **and François Fages**[2] **and Claude Kirchner**[1] **and Jan Maluszynski**[4] **and Benjamin Wack**[1]

[1] LORIA: CNRS, INRIA, Universities of Nancy
615, rue du Jardin botanique,
54600 Villers-lès-Nancy Cedex, France

[2] INRIA
France

[3] Institute of Computer Science, Polish Academy of Sciences,
Warsaw, Poland

[4] Department of Computer and Information Science,
Linköping University
S-581 83 Linköping, Sweden

2 September 2004

Abstract
We survey and analyse the relevant existing work on typing of rules, in particular on typing of constraint logic programs and discuss applicability of these approaches to the REWERSE reasoning and query languages under development by WG I1 and by WG I4. This is related to WG I1, developing logic programming like languages for reasoning on the web and with WG I4 investigating development of declarative query languages such as XPathLog and Xcerpt.

Keyword List
constraint, rewriting, rule based language, type system

Contents

1 Introduction

1.1 Do we need types in the Semantic Web?

The semantics of the Semantic Web is based on two complementary concepts: first, one adds semantics to the web entity using ontologies; second, one performs transformations and deductions on the web entity.

So, the Web becomes more semantic as more computations and deductions are performed on its entities. It becomes therefore the main subject of an intense programming activity which can be classically sustained by the right elaboration and use of appropriate type systems.

The importance of types for programming languages is recognized since Algol 60. Types, especially when statically checkable, help in discovering errors in programs and make possible more efficient implementation techniques. Of particular importance is the ability of static type checking to automatically and quickly discover and locate a certain kind of program errors. Without types this requires tedious testing and debugging. The old experience is confirmed with newer programming paradigms, for instance by logic programmers comparing their experiences with programming in untyped Prolog and in typed Goedel.

As usual, a type discipline ensures certain guidelines and can prevent inconsistencies such as the erroneous composition of objects but it also induces constraints on the programming approach and on the language expressiveness. The design of good type system should therefore ensure an appropriate balance between the constraints it imposes and the obtained expressiveness. Following these ideas, elaborated type systems allowing polymorphism, dependent types, intersection types or sub-types have been developed. For example, ML is the first and one of the prominent languages to contain polymorphic type inference together with a type-safe exception handling mechanism.

On the logical side, the deBruijn-Curry-Howard isomorphism allows for the now classical correspondence between types and propositions as well as between terms and proofs. Since we are not only interested on computations on the Web but also on properties about the Web, types promise also to be of main interest for the logical side of the Semantic Web.

But of course the Web can not only be abstracted as a huge graph, it has its own specificities and the operations we want to perform on it as well as its properties are strongly influencing the design of the type systems.

In this general context, we will review works related to the rule based approach on which the REWERSE framework grounds its semantics.

1.2 What is a rule?

The rule based programming paradigm indeed covers two main different atomic concepts: production rule and rewrite rule.

1.2.1 Production rules

The production rules, as well as their declinations like business rules or constraint handling rules, are statements of the form "if condition then action" that act on a set or multiset of entities.

They are popular, in particular in AI, to describe expert systems, robot behavior, constraint simplification [Frü95], behaviour of business, ...

There are many implementations of production rules. Without being exhaustive, let us mention OPS5 [X-T88, For81], Clips [CR03], Claire [CL96] or the JRules system [Ilo02].

To the best of our knowledge, no specific type system have been defined for these languages.

1.2.2 Rewrite rules

Rewrite rules are statements of the form "replaces a matching entity by another one" that act on terms [BN98, "T02] or on graphs [Cou90].

They are popular in the theorem proving as well as in the algebraic specification communities as they are a natural way to operationalize equality. More recently they have been used from a more logical as well as computational point of view at the heart of the rewriting calculus [CK01] and logic [Mes92].

Many systems use the concept and implement the evaluation of rewrite rules, some giving also the possibility to have user defined strategies. Again, without being exhaustive we can mention ELAN [BKK+98], Maude [CDE+99], ASF+SDF [DHK96], Stratego [Vis99]. A typical dedicated rewrite language for XML is XSLT. The matching and rewriting concepts behind the above languages and systems are made available in existing programming languages with the TOM [MRV03] system (`tom.loria.fr`), currently available for Java, C and CAML.

These languages, all mainly first-order, use various type systems, some of them quite sophisticated like for Maude, where simple types (there called sorts) are completed with subtypes and membership constraints [Com91, Mes98, HKK98].

Even more elaborated type systems can be defined for rewriting, and we will survey in section 2.2 the main motivations and results obtained on the rewriting calculus.

1.3 Emerging rule languages for the Semantic Web

The Semantic Web effort of W3C resulted in a recently accepted standard for the Web Ontology Language OWL. OWL, based on an expressive Description Logic, makes it possible to give axiomatic characterizations of application domains and to reason about them, e.g. for web querying.

The W3C vision of layered Semantic Web assumes adding the rule layer on top of the ontology layer. There seems to be a general consensus (see e.g. [Sta03]) that rules with a well-defined semantics are needed in the Semantic Web applications and that they should be well integrated with the ontology level. While development of rule languages seem to be a hot research topic, the existing proposals tuned to the Semantic Web seem to be rather preliminary. The topic is also pursued in REWERSE, with the objective to support reasoning on the web and web querying.

We now briefly mention the efforts which in our opinion are most relevant for REWERSE.

1.3.1 Xcerpt.

The declarative query and transformation language Xcerpt [BS04, and references therein] for querying XML databases, defined at Munich, will be further extended by REWERSE Working Group I4 towards a semantic web query and transformation language. Xcerpt differs from most similar languages, as it is not based on XPATH selection language and the concept of path navigation in trees. Instead, queries are expressed in terms of pattern matching. There is similarity between Xcerpt and logic programming. Xcerpt programs consist of rules, similar to Horn clauses. The standard unification is however not suitable to deal with semistructured data

(where, roughly speaking, symbols do not have fixed arity). A specialized notion of simulation unification is employed instead.

1.3.2 RuleML initiative

The Rule Markup Initiative[1] (RuleML), started in August 2000, brings together researchers and practitioners from several countries with the objective to explore rule systems suitable for the Semantic Web. As stated on the RuleML home page, the goal of the Rule Markup Initiative is to develop RuleML as the canonical Web language for rules using XML markup, formal semantics, and efficient implementations.

The RuleML language is to be defined as an hierarchy of sublanguages. The basic sublanguage in this hierarchy is the language of function-free Horn clauses, known as Datalog. Its XML syntax has been formally defined and is referred to as RuleML 0.7, further extended to RuleML 0.85. The RuleML language should, among others, include various extensions of Datalog and production rules (event-condition-action rules).

Some members of REWERSE are involved in RuleML, and the planned proposal for REWERSE reasoning language will certainly be influenced by RuleML activities.

1.3.3 SWRL

Until recently the activities of RuleML did not concern the issue of integration of rules and ontologies. This gap has been addressed recently by a draft proposal [HPSB$^+$04] for a Semantic Web Rule Language (SWRL) combining OWL with the Unary/Binary Datalog RuleML sublanguages of the Rule Markup Language. The language makes it possible to extend OWL axioms with rule axioms. A rule axiom (see [HPSB$^+$04]) consists of an antecedent (body) and a consequent (head), each of which is a possibly empty set (conjunction) of atoms. Such a rule is considered an implication of body and head. The integration between rules and ontologies is achieved by using the *concepts*[2] and the *roles*[3] of the Description Logic underlying OWL, (which denote, respectively, unary and binary predicates) for building atoms of the SWRL rules. The concepts and the roles are defined by axioms expressed in OWL, which is a subset of SWRL, and used in SWRL rules. A model-theoretic semantics of the integration, sketched in the report, extends naturally the semantics of OWL. It is well-known that the resulting logic is undecidable. The report does not provide an operational semantics for SWRL; it is well-known that SLD resolution is no more complete for such an extension of Datalog.

1.3.4 Description logic programs

It is commonly agreed that Datalog will be used in many web applications. A relevant question is then how to integrate Datalog rules with ontologies. A recent proposal in that direction [GHRS03] identifies a Description Logic, called Description Horn Logic (DHL), whose formulae can be transformed into equivalent Datalog rules, and the corresponding subset of Datalog, called Description Logic Programs (DLP). The Description Horn Logic is shown to be more expressive than the RDF-S fragment of the Description Logic (see [GHRS03] for the details). Thus, on the one hand it is practically relevant, and on the other hand it has complete tractable inference procedure. This is because any DHL ontology can be transformed into the equivalent

[1] http://www.ruleml.org
[2] Sometimes called *classes*
[3] Sometimes called *properties*

Datalog program. This makes it possible to use complete and tractable inference procedures of Horn clauses for efficient answering of standard queries about classes and properties defined by a DHL ontology. The limitation of this elegant approach is that it does not apply to OWL but only to a restricted subset. The Description Logic Programs constitute also a restricted subset of Datalog, since for example the rule "If x is a brother of y and z is a son of y then x is an uncle of y" is not expressible in DLP. Thus, in this approach one achieves a complete fusion of ontology layer with rule layer by restricting the underlying logical formalisms: the Horn logic and the Description Logic.

1.3.5 Towards typing of the Semantic Web rules

The proposals for Semantic Web rule languages outlined above are essentially based on, or influenced by, pure logic programming paradigm. Most of them indicate need for further extensions, like negation, constraints, priorities, etc. A special issue is integration of functions; e.g. in SWRL it is done by using predefined built-ins of XQuery, typed with XML Schema datatypes. In all cases the operational semantics of the rules relies on some kind of rewriting. In particular execution of Xcerpt rules would require rewriting of a kind of terms. We argue that static type checking of rules could, as usual, facilitate finding errors in the rules and in some cases possibly could also speed-up the execution.

As mentioned above, most of the existing rule proposals for the Semantic Web refer to logic programming and/or to rewriting techniques. Therefore this report will survey some relevant approaches to typing of constraint logic programming (CLP) and a framework for typing of rewriting based programs.

We will also summarize some ongoing REWERSE research on application of these approaches to web reasoning languages and to web query languages.

The report is organized as follows. The next section is an overview of typing approaches for (constraint) logic programming, and for rewrite based programs. Section 3 presents a typing approach for a sublanguage of RuleML. Section 4 discusses typing web query languages.

2 Typing of rules

2.1 Typing constraint logic programs

We survey existing work on typing for logic programming (LP) and constraint logic programming (CLP). First we explain basic notions of descriptive and prescriptive typing. Then Section 2.1.2 discusses descriptive and Section 2.1.3 prescriptive typing approaches for LP and CLP.

2.1.1 Descriptive and prescriptive typing

There exist two approaches to typing logic programs and constraint logic programs – prescriptive and descriptive. Stating it briefly: descriptive types approximate the semantics of programs in an untyped language, and in the prescriptive approach the language is typed and types are part of its semantics [Pfe92, Chapter 10, "Dependent Types in Logic Programming", p. 285,286].

In **descriptive typing** we deal with untyped programming languages, whose declarative semantics is given by untyped logic. Typing means here providing an approximation of the semantics of a given program. The notion of approximation is related to some partial ordering

$\preceq$ on the semantic domain; usually $\preceq$ is induced by the set inclusion. An approximation of a program's semantics is a semantic object greater (in $\preceq$) than the semantics.

Type correctness in descriptive typing is a semantic concept: a program is correct w.r.t. given types if the types approximate its semantics. One may consider the declarative semantics, for instance that given by the least Herbrand model of a program. One may also approximate some operational semantics. In most cases it is call-success semantics, given by the set of procedure calls and the set of successes that appear in the computations for a given class of initial goals.

Such a semantic approximation is a set, or a family of sets. Thus in descriptive typing a type is a set, usually a set of terms or atoms. (And we informally call such approximation "types" or "types for the program"). Types should be decidable sets, and be defined by some rather simple formalism. Often it is necessary that the class of types is closed under the basic set operations and that there exist efficient algorithms for certain operations on types.

In the context of descriptive typing, algorithms are usually provided for type checking and type inference. *Type checking* means checking whether given types approximate the semantics of a given program. Type checking can be seen as proving correctness of the program w.r.t. a specification expressed by means of types. A difference with general proof methods is that we deal with a restricted class of specifications; thanks to this, proving is effective. *Type inference* means finding the types which approximate the semantics of a given program.

Prescriptive typing is closer to the treatment of types in functional programming. In functional programming, type systems aim to prevent some class of run-time errors corresponding to undefined operations, such as adding an integer to a list, or accessing a non-existing field in a record. In prescriptive typing of constraint logic programs, this is translated into declaring some terms to be illegal, thus providing guidelines and in fact a discipline for *composing* terms and modules. In an analogy to the previous example, the term `1+[]` is thus declared to be illegal. These declarations are made by giving fixed types (formally – signatures) to function symbols and predicates, as opposed to descriptive typing where function symbols are untyped.

The correctness of a prescriptive type system w.r.t. an execution model expresses that no illegal term will appear during the execution of a well-typed query in a well-typed program. This property is usually called *subject reduction*. The execution model itself may be typed or not, that is it may or not "take types in account". For example, the type system of Mycroft-O'Keefe is proven correct w.r.t. both an untyped execution model [MO84] and w.r.t. a typed one [LR91]

A type in prescriptive typing is an expression in some formal language of types (while in descriptive typing a type is set). Well-typedness is a syntactic notion, usually defined by a system of proof rules. Only those programs, queries (or other syntactic objects) that can be derived by the rule system are well-typed. The set of values corresponding to a given type is a kind of secondary notion and has to be defined separately. Given a prescriptive type system, one can consider the issue of its correctness w.r.t. some execution model.

Pfenning [Pfe92, Chapter 10] presents a more strict point of view on what prescriptive typing for (constraint) logic programming is. For them, in prescriptive typing the types are part of the semantics of the programming language. Its declarative semantics is given by a typed logic. Only well-typed programs have semantics. Examples of such logic programming languages are Goedel and Mercury [HL94, SHC96].

Notice that different treatment of function symbols in descriptive and prescriptive typing corresponds to their different role in logic programming and functional programming. In logic programming function symbols are constructors and thus are applicable to arbitrary terms.

In functional programming they denote functions, defined by programs. (And constraint logic programming combines the two roles of function symbols.)

In contrast to descriptive typing, in the prescriptive approach there exist programs for which no typing exists (i.e. which are not well-typed for every typing). In the descriptive approach, the meaning of any program can be approximated by types.

In prescriptive typing variables have types. If a variable appears in an answer of a program, it stands for a value from (the set of values of) its type. In descriptive typing variables are untyped. A variable in a program answer stands for an arbitrary term.

2.1.2 Descriptive type systems

We discuss here descriptive type systems for (constraint) logic programming. We first discuss the tools for expressing the approximations, then the semantics to be approximated. This makes us ready to present approaches to type checking and type inference. We discuss mainly logic programming; descriptive type approaches to constraint logic programs are generalizations of the corresponding ones for logic programs.

Regular sets of terms. To construct semantic approximations for logic programs a formalism for defining (a class of) decidable sets of terms/atoms is needed. Usually one deals with regular sets of ground terms/atoms, definable by tree automata [CDG$^+$02]. Equivalent formalisms are regular term grammars (see e.g. [DZ92]), regular unary logic programs [YS91], certain classes of set constraints [HJ90a, TDT00], and monadic second-order logic [Tho97, Nev02].

There exists an important subclass of the regular sets of terms, namely that given by deterministic top-down tree automata. Such sets are sometimes called tuple-distributive, discriminative or path-closed. In a tuple-distributive set, the set of argument tuples of a given function/predicate symbol is a Cartesian product (of tuple-distributive sets). One may argue that this is a more natural notion of a type (than that given by nondeterministic tree automata). The restriction to deterministic automata results in simpler and more efficient algorithms. For instance checking inclusion of regular sets is EXPTIME-complete, and it is polynomial for the restricted class. A technical difficulty is that the class is not closed under union, in contrast to the regular sets of terms.

Datalog (cf. Section 3) is a restricted form of logic programming. We may say that the structure of data objects is irrelevant for Datalog. (Formally, the data objects are constants, not general terms). Thus regular sets of terms are not suitable as approximations for Datalog programs. Section 3 shows how Description Logics can be used for this purpose.

In descriptive typing, type *polymorphism* means dealing with families of sets. Parametric families of regular sets of terms can be defined by parameterizing one of formalisms discussed above. For instance a parametric version of regular tree grammars is used in [DMP02].

In the context of constraint logic programming (CLP), instead of terms we deal with constrained terms. A constrained term is a pair $c \,[]\, t$ of a constraint c and a term t, where each free variable of c occurs in t. The notion of a regular set of terms has to be generalized accordingly. A generalization is proposed in [DMP02]. It employs a fixed finite collection of primitive sets of constrained terms (like the set of all constrained terms, the set of ground terms, or a set of constrained variables of the form $c \,[]\, X$ with some condition on the form of c). Other types are created out of the base ones by the mechanism of regular term grammars. This approach to constraints is rather crude, but it turned out to be useful in the context of CLP over finite

domains. An attempt of a more expressive system of regular sets of constrained terms was presented in [DP98, DP99].

Semantics of LP/CLP for descriptive typing. Descriptive typing means approximating a program's semantics with types. The first candidate is the declarative semantics, given by the least Herbrand model (or by the set of atomic logical consequences of the program). Such semantics was dealt with in typing approaches of [Mis84, JB92, FSVY91, YS91].

Unfortunately it turns out that for many programs useful approximations of their declarative semantics do not exist. A typical example is the standard append program $\{append([H|K], L, [H|M]) \leftarrow append(K, L, M).;\ \ append([\,], L, L) \leftarrow .\}$. The second and the third argument of *append* in an answer of the program may be arbitrary terms. As a consequence, the best approximation (for the declarative semantics of the program) says that the first argument is a list and the remaining ones are arbitrary terms. This is not what one would expect from a typing for this program. For instance such approximations cannot express that, speaking informally, *append* works with lists. Possibly the only practical application of this approach was for discovering that a predicate never succeeds (because the obtained approximation is empty).

The difficulty described above lead even to opinions (not widely shared) that such programs should be considered erroneous [Nai92] and should be corrected to change their declarative semantics. In the case of the append program the unary clause should be changed into $append([\,], L, L) \leftarrow list(L)$, with an appropriately defined predicate $list$, so that *append* succeeds only with all its arguments being lists.

A more reasonable way of making descriptive typing practical is to adopt another semantics. A usual solution is to use a call-success semantics. The semantics formalizes a directional view of logic programs. In this view each predicate is considered a procedure which, applied to a suitable tuple of call arguments, returns upon a success a tuple of computed values. So instead of the declarative semantics one uses operational semantics, given by SLD-resolution. Usually the Prolog selection rule is assumed (LD-resolution). One considers the procedure calls and procedure successes in LD-derivations. The *call-success* semantics of the program is the set of calls and the set of successes for the given program and a given set of initial atomic goals.

The declarative semantics and the call-success semantics are related by the "magic transformation" [BMSU86]. Given a program P and a query Q, their magic transformation is a logic program P'. Roughly speaking, the declarative semantics or P' is the call-success semantics of P. More precisely, for each predicate symbol p of P, P' contains predicates ${}^\bullet p$ and $p^\bullet$; $P' \models {}^\bullet p(\vec{t})$ iff $p(\vec{t})$ is (an instance of) a call of P (in an LD-derivation starting from Q), and $P' \models p^\bullet(\vec{t})$ iff $p(\vec{t})$ is (an instance of) a success of P. This can be generalized to a regular set of initial goals.

As a result, any method of approximating the declarative semantics can be used to approximate the call-success semantics, by applying the magic transformation.

Type checking In the context of descriptive typing, we say that a program is *type correct* w.r.t. a typing, if the typing approximates the semantics of the program. More formally, both the program meaning and the typing are sets; type correctness means that the former is a subset of the latter. (In a general case they may be tuples of sets, and we consider component-wise inclusion).

Another viewpoint may be useful. The typing can be seen as a specification; type correctness boils down to correctness of a program w.r.t. the specification. By a program P being correct

w.r.t. a specification *spec* we mean that $spec \models A$ for any answer A of the program.[4] Hence type checking (i.e. checking whether a given program is type correct w.r.t. a given typing) means proving program correctness. As specifications are regular sets, the proving can be made effective.

In order to prove program correctness for the declarative semantics, it is sufficient to show that the specification is a model of the program:

$$spec \models P\,.$$

This method was proposed by Clark [Cla79], see also [DM93, Chapter 7], [DM01] and the references therein. Soundness of the method is simple to prove: if $spec \models P$ and A is an answer of P then $P \models A$ and thus $spec \models A$. The method is also complete [Der93]; if a program P is correct w.r.t. *spec* then there exists a stronger specification $spec' \subseteq spec$ such that $spec' \models P$.

If *spec* is a Herbrand interpretation, given by a set of ground atoms, then the sufficient condition is equivalent to $T_P(spec) \subseteq spec$, where T_P is the immediate consequence operator. To check the condition one has to check that for every ground instance $(H \leftarrow \mathbf{B})\theta$ of any clause of the program P, if $\mathbf{B}\theta \subseteq spec$ then $H\theta \in spec$. This can be done by computing, for each variable X in the clause, the set of ground terms $X\theta$ such that $\mathbf{B}\theta \subseteq spec$. These sets characterize the set of clause instances with the bodies contained in *spec*. Then the set of the heads of these clause instances can be computed and checked whether it is a subset of *spec*. (See e.g. [DMP02] for missing details.) If we begin with a regular set *spec* then all the sets in this computation are regular, and the checks decidable. (Notice that the union of the computed sets of heads, for all clauses of P, is $T_P(spec)$.)

For the call-success semantics, an appropriate proof method was given in [BC89] (see also [Apt97]); it is a simplification of the method of [DM88] (the latter makes it possible to deal with sets not closed under substitution). It gives a sufficient condition for type correctness with call-success semantics. Out of this condition a type checking algorithm can be obtained, similarly as described above for the declarative semantics (cf. e.g. [DMP02]). Alternatively, one can apply first the magic transformation to the program, and then type checking for the declarative semantics. Both approaches result in the same verification conditions and type checking algorithms.

Most authors use the described above way of type checking, often without relating it to program correctness (e.g. [GdW94]). The abovementioned condition for correctness w.r.t. the call-success semantics is often used as a definition for well-typedness of a program, without mentioning its relation to the call-success semantics of LD-resolution (e.g. [AL94, CP98]).

Type inference There exist two main approaches for inferring descriptive types for (constraint) logic programs: based on abstract interpretation paradigm and employing set constraint solving.

Abstract interpretation is a general framework for computing approximations of the semantics of programs [CC77]. Roughly speaking it consists in performing computations over an *abstract domain* (instead of using the actual data values). The elements of the abstract domain correspond to sets of actual values; the domain is partially ordered, the ordering corresponds to inclusion of the corresponding sets.

[4] By an answer we mean any $Q\theta$, where Q is a query and θ a computed, or equivalently correct, answer substitution for P and Q. Notice that if *spec* is a Herbrand interpretation then correctness of P is equivalent to $M_P \subseteq spec$, where M_P is the least Herbrand model of P.

Abstract interpretation paradigm was widely applied to logic programs, see the overview [CC92a]. (For later work see e.g. the references in [Mil99] and below). In most cases abstract interpretation was used to approximate the call-success semantics. Our brief discussion will however deal with approximating the declarative semantics. (Any such approximation method may be applied to approximating the call-success semantics by employing magic transformation — the later relates both kinds of semantics as described above).

When abstract interpretation is applied to derive descriptive types [JB92, GdW94, HCC95, Mil99, Cod99, DMP02], the abstract domain is the chosen class of regular sets, ordered by inclusion. Such abstract domain has some specific properties. Its height is infinite. The abstraction function does not exist [DP99, DP98]. This is because there may not exist the closest regular approximation of a given set. Instead we have an infinite set of approximations without a greatest lower bound in the abstract domain (see [DP98] for an example).

Approximating the declarative semantics of a given program P means finding a specification *spec* from the abstract domain, such that P is correct w.r.t. *spec*. A sufficient condition is $spec \models P$. Assume that *spec* is a Herbrand interpretation; the sufficient condition is thus $T_P(spec) \subseteq spec$. The method of type checking outlined above provides a way of computing $T_P(spec)$ for a given *spec*. Abstract interpretation starts from $spec_0 = \emptyset$ and then iteratively computes $T_P(spec_{i-1})$ and $spec_i \supseteq T_P(spec_{i-1})$, for $i > 0$. The iteration terminates when $T_P(spec_i) \subseteq spec_i$; such $spec_i$ is the required result.

There are two reasons why $spec_i$ may be not equal to $T_P(spec_{i-1})$. First, $T_P(spec_{i-1})$ may not belong to the abstract domain. (If the abstract domain is the class of tuple-distributive regular sets then $T_P(spec_{i-1})$ is a union of such sets, but the class is not closed under union). Then, the height of the domain is infinite and infinite computations are possible. A usual way of assuring termination is widening [CC92a]; this means using, for some values of i, a set larger than $T_P(spec_{i-1})$ as $spec_i$ (more precisely, a set larger than the closest superset of $T_P(spec_{i-1})$ within the abstract domain).

Termination of abstract interpretation for type inference often poses difficulties. For instance the termination proof in [GdW94] is incorrect. This topic is discussed in [Mil99].

Set constraints can be used to represent a sufficient condition for program correctness. A solution of such system of set constraints is a specification for which the program is correct.

A set constraint is an inclusion $E_1 \subseteq E_2$, where E_1, E_2 are set expressions. Set expressions are built out of variables and set operations, like conjuction, intersection, construction ($f(S_1, \ldots, S_n) := \{ f(t_1, \ldots, t_n) \mid t_1 \in S_1, \ldots, t_n \in S_n \}$, where f is a function symbol), projection ($f^{-1}_{(i)}(S) := \{ t_i \mid f(t_1, \ldots, t_n) \in S \}$), generalized projection ($t^{-X}(S) := \{ X\theta \mid t\theta \in S \}$, where t is a term and X a variable), complementation and others. Various classes of set constraints were studied, complexity results for these classes and actual algorithms were presented (see [HJ90a, Aik94, PP97, TDT00] and references therein). In most cases the least solutions (of systems of set constraints) are computed. Usually the solutions are (unrestricted) regular sets of terms (while in most abstract interpretation approaches to type inference the inferred types are tuple-distributive, cf. page 6).

Applying set constraint solving to type inference of logic programs was discussed, among others, in [HJ90b, Hei92, DTT97, CP98]. As an advantage of such approach it is pointed out that it provides separation between specification of the problem (constructing constraints that represent correctness of a given program) and constraint solving [Aik99]. Algorithms for solving constraints can be studied separately, the reach existing theory can be exploited in sophisticated implementations. On the other hand most actual implementations of descriptive type inference

employ abstract interpretation.

It is interesting to compare abstract interpretation with set constraint solving. In the former the computation is in principle infinite and requires additional measures for termination. Why this problem does not occur in set constraint solving? A formal comparison is given in [CC95] and an informal discussion in [Aik99]. Roughly speaking, for each particular program only finite subset of the abstract domain is of importance. The subset is implicitely discovered during solving set constraints. Otherwise it is difficult to find; it cannot be obtained directly from the text of the program [CC92b, Section 7.1].

2.1.3 Prescriptive type systems

Prescriptive type systems address a different issue: the one of composition. By providing types to the signature of function and predicate symbols, one expresses the syntactic categories to which a function, a predicate, and in turn a complete module, is supposed to be applied. Prescriptive types are therefore an integral part of the programs and modules and constitute a discipline to compose them correctly.

The type system of Mycroft and O'Keefe [MO84] is an adaptation of the Damas-Milner [DM82] type system for ML. The type of function symbols has to be declared, while the type of the predicates may be inferred. This type system has been implemented in the language Gödel [HL94].

In this context however, parametric polymorphism is quite restrictive, and there have been several attempts to augment the expressive power of prescriptive type systems for constraint logic programming by using subtyping.

The type system of Dietrich and Hagl [DH88] adds subtyping to the Mycroft - O'Keefe type system, provided that information is given about the data-flow. This information can be provided either by modes, or by a global analysis of the program. However, Dietrich and Hagl do not study the decidability of the conditions they impose on the subtyping relation. Furthermore, each result type must be transparent, that is the type variables occurring in the type of the arguments of a function symbol must also appear in the type of the result. Smaus, Fages and Deransart [SFD00] proposed a type system with subtyping and parametric polymorphism for moded constraint logic programs, without the transparency condition.

Hanus [Han92] introduced a type system with a typed execution model. The type system allows for many-sorted, order-sorted, polymorphic and polymorphically order-sorted type structures, by using an equality theory on types.

The type system of Fages, Paltrinieri and Coquery [FP97, FC01] for constraint logic programming adding subtyping to the Mycroft - O'Keefe system. The system thus combines parametric polymorphism and subtyping. The equality constraint has the polymorphic type $\forall.\alpha \times \alpha$. Parametric polymorphism is also used for data structures, such as homogeneous lists. Subtyping is used for typing the simultaneous use of different constraint domains. For example, it is possible to share variables between boolean constraints and finite domain constraints. This is possible when the type boolean, corresponding to the boolean domain, is a subtype of the type integer, corresponding to finite domain constraints. Subtyping is also used for the typing of meta-programming predicates. The type term is introduced as a supertype of all other types, and is used for predicates that compose or decompose terms. For example, the predicate =../2, which associates to a term a list containing its head symbol and all its arguments has the type term $\times$ list(term). Using this type requires to deal with subtyping inequalities like list(α) $\leq$ term, which correspond to a non-structural non-homogeneous subtyping relation. The

system can be extended to use overloaded function symbols [CF02]. For example, the operator $-/2$ can have types expr $\times$ expr $\rightarrow$ expr for arithmetics and $\forall\alpha\beta.\alpha \times \beta \rightarrow \text{pair}(\alpha, \beta)$ for pairs. The correctness of the type system is proved w.r.t. two execution models. The first one is the CSLD resolution [JL87], which is an abstract untyped model that proceeds by constraint accumulation. The second one is a typed execution model, in which type are kept on variables and that ensure that whenever a variable X of type τ is instantiated with a term t, then t is also of type τ.

The type checking algorithm features type inference for variables. Type inference for predicates is also possible. In presence of the supertype *term* however, the type term $\times \ldots \times$ term is a possible type for any predicate. In order to obtain more informative types for undeclared predicates, a heuristic type inference algorithm is used for infering the types of predicates.

2.2 Typing rewriting based programs

This section presents a prescriptive based approach for the typing of rewrite based programs. Rewriting is a fundamental concept when dealing with computation, equality or inference. It is therefore of fundamental interest for the Semantic Web. Furthermore, the rewriting concept is well understood and many type systems have been designed for its specific purpose.

The rewriting calculus is a fine grained framework that allows us to describe rule application and to formalize the fundamental concept of strategy application. It has therefore been already used in order to give a semantics to the (rules and strategies of the) rewrite based language ELAN. We survey in more depth its main design concepts and results about its type systems.

We will not present here the order-sorted or the membership constraint approaches implemented, for example, in the rule based language Maude. Good references about them is the special issue [Smo98] as well as [BJM00].

2.2.1 The simply typed rewriting calculus

Introduced in [CK99, CK01], the rewriting calculus, also called ρ-calculus, is a framework embedding λ-calculus and rewriting capabilities, by allowing abstraction not only on variables but also on patterns. The first proposed typed version [CK00] used a quite strict type discipline and enjoyed the good properties: strong normalisation and subject reduction. More sophisticated type systems were introduced later [CKL01b, BCKL03] and the relationship with corresponding logics explored.

We briefly present here the latest simply typed rewriting calculus [CLW03] due to its close expressiveness wrt classical rewriting (languages) and thus to its potential wrt to the query languages of REWERSE.

The core mechanism of the rewriting calculus is pattern matching and thus we define first the classical notions of matching equations and matching solutions.

Definition 2.1 (Matching)
Given a theory $\mathbb{T}$ *(i.e. a set of axioms defining a congruence relation* $\overset{\mathbb{T}}{=}$*):*

1. *A matching equation is a problem* $\mathsf{T} \triangleq P \prec\!\!<_{\mathbb{T}} A$ *with* P *a pattern and* A *a term.*
2. *A substitution* θ *is a solution of the matching equation* T *if:*
   (a) $P\theta \overset{\mathbb{T}}{=} A$

(b) θ is well-typed, i.e. $\forall \Gamma$ *such that* A *and* P *are typable, if* $\Gamma \vdash X : \tau$ *then* $\Gamma \vdash X\theta : \tau\theta$.

The set of solutions of T *is denoted by* $\mathcal{S}ol(\mathsf{T})$.

Different theories and the corresponding pattern matching problems can quand vous voulez.be formally defined and solved, for example, as explained in [CKL01a]. If the underlining theory is clear from the context then it can be omitted from the notation.

The higher-order mechanisms of the λ-calculus and the pattern matching facilities of the rewriting are then both available at the same level. The syntax is presented in Figure 1. The *types* are as one would expect from a first-order type system, *i.e.* constant-types and

$$
\begin{array}{lll}
\tau ::= \iota \mid \tau \twoheadrightarrow \tau & \mathcal{T}y & \text{Types} \\
\Delta ::= \emptyset \mid \Delta, X{:}\tau \mid \Delta, f{:}\tau & & \text{Contexts} \\
P ::= X \mid \mathsf{stk} \mid f\,\overline{P} \qquad \textit{(variables occur only once in any } P\textit{)} & \mathcal{P} & \text{Patterns} \\
A ::= f \mid \mathsf{stk} \mid X \mid P \twoheadrightarrow_\Delta A \mid [P \ll_\Delta A]A \mid A \bullet A \mid A; A & \mathcal{T} & \text{Terms}
\end{array}
$$

Figure 1: Syntax of $\rho_{\twoheadrightarrow}$

arrow-types. The *patterns* are algebraic terms (*i.e.* terms constructed only with variables, constants and application) which can be used as left-hand sides of the rewrite rules; the set of patterns is obviously included in the set of terms. The well-known linearity restriction [vO90] is needed to keep the small-step semantics confluent. A *rewrite rule* of the form $(P \twoheadrightarrow_\Delta A)$ abstracting over the free variables of P is a first-class citizen of the calculus. The types of the free variables of P are declared in Δ, *i.e.* $\mathcal{FV}(P) = \mathcal{D}om(\Delta)$, resulting in a fully annotated calculus *à la* Church. An *application* is implicitly denoted by concatenation. The *delayed matching constraint* $[P \ll_\Delta A]B$ can be seen as the term B with its free variables constrained by the matching between P and A. Again, the context Δ contains the type declarations of all the free variables appearing in the pattern P. A *structure* is a collection of terms that can be seen either as a set of rewrite rules or as a set of results. The symbol stk can be considered as the special constant representing a delayed matching constraint whose matching problem is unsolvable. An alternative approach would be to omit this symbol from the syntax, but the encoding of rewriting systems presented later would be no longer possible.

Example 2.1 (Some ρ-terms)

- *a term similar to the* λ*-term* $(\lambda x.x)$ *a:* $\quad (X \twoheadrightarrow_{(X:\tau)} X)\bullet a$
- *the well-known* λ*-term* $(\omega\omega)$*:* $\quad (X \twoheadrightarrow_{(X:\tau)} X\bullet X)\bullet(X \twoheadrightarrow_{(X:\tau)} X\bullet X)$
- *the application of the rule* $a \twoheadrightarrow b$ *to the term* a*:* $\quad (a \twoheadrightarrow b)\bullet a$
- *a classical rewrite rule application:* $\quad (f(X,Y) \twoheadrightarrow_{(X:\tau_1,Y:\tau_2)} g(X,Y))\bullet f(a,b)$
- *"non-deterministic" application:* $\quad (a \twoheadrightarrow b; a \twoheadrightarrow c)\bullet a$

By now we have settled all the background necessary to describe in Figure 2 the reduction rules of the general rewriting calculus parameterized by the theory $\mathbb{T}$. When instantiating $\mathbb{T}$ with

concrete theories (e.g. theories containing axioms for associativity, associativity-commutativity, etc., for a given symbol) different versions of the calculus are obtained. When not essential or clear from the context, we will omit the theory $\mathbb{T}$ in rules and congruences.

$$
\begin{array}{rcll}
(P \twoheadrightarrow_{\Delta} A) \bullet B & \to_{\rho} & [P \ll_{\Delta} B]A & \\
[P \ll_{\Delta} B]A & \to_{\sigma} & A\theta_1, \ldots, A\theta_n & \text{with } \{\theta_1, \ldots, \theta_n\} = \mathcal{S}ol(P \lessdot_{\mathbb{T}} B) \\
(A; B) \bullet C & \to_{\delta} & A \bullet C; B \bullet C &
\end{array}
$$

Figure 2: Small step semantics of the Rewriting Calculus

Let us quickly explain these rules:

(ρ) this rule "fires" the application of an abstraction to a term, but does not immediately try to solve the associated matching equation.

(σ) this rule is applied if (and only if) the matching equation $P \lessdot_{\mathbb{T}} B$ has at least one solution: in this case the matching solutions are computed and applied to the term A. If the matching is not unitary, a structure collecting all the different results is obtained when the rule is applied. If there is no solution, this rule does not apply and thus, the term represents a matching failure. As we will see, further reductions or instantiations are likely to modify B so that the equation has a solution and the rule can be fired.

(δ) this rule distributes structures on the left-hand side of the application. This gives the possibility, for example, to apply in parallel two distinct pattern abstractions A and B to a term C.

The multi-step reduction induced by these rules is denoted by $\mapsto\!\!\!\!\!\rightarrow_{\rho\sigma\delta}$.

Example 2.2 (Some ρ-reductions)

$$
\begin{array}{l}
(X \twoheadrightarrow_{(X:\tau)} X)\bullet a \quad \mapsto\!\!\!\!\!\rightarrow_{\rho\sigma\delta} \; a \\
(X \twoheadrightarrow_{(X:\tau)} (X\bullet X))\bullet(X \twoheadrightarrow_{(X:\tau)} (X\bullet X)) \quad \mapsto\!\!\!\!\!\rightarrow_{\rho\sigma\delta} \; \omega\bullet\omega \mapsto\!\!\!\!\!\rightarrow_{\rho\sigma\delta} \; \ldots \\
(a \twoheadrightarrow b)\bullet a \quad \mapsto\!\!\!\!\!\rightarrow_{\rho\sigma\delta} \; b \\
(f(X,Y) \twoheadrightarrow_{(X:\tau_1,Y:\tau_2)} g(X,Y))\bullet(f(a,b)) \quad \mapsto\!\!\!\!\!\rightarrow_{\rho\sigma\delta} \; [f(X,Y) \ll_{(X:\tau_1,Y:\tau_2)} f(a,b)]g(X,Y) \mapsto\!\!\!\!\!\rightarrow_{\rho\sigma\delta} \; g(a,b) \\
(f(X,Y) \twoheadrightarrow_{(X:\tau_1,Y:\tau_2)} g(X,Y))\bullet(g(a,b)) \quad \mapsto\!\!\!\!\!\rightarrow_{\rho\sigma\delta} \; [f(X,Y) \ll_{(X:\tau_1,Y:\tau_2)} g(a,b)]g(X,Y) \\
(a \twoheadrightarrow b; a \twoheadrightarrow c)\bullet a \quad \mapsto\!\!\!\!\!\rightarrow_{\rho\sigma\delta} \; (a \twoheadrightarrow b)\bullet a; (a \twoheadrightarrow c)\bullet a \mapsto\!\!\!\!\!\rightarrow_{\rho\sigma\delta} \; b; c
\end{array}
$$

Example 2.3 *If we denote* $\omega \triangleq f\bullet X \twoheadrightarrow X\bullet(f\bullet X)$ *then we have*

$$
\begin{array}{lcl}
\omega\bullet(f\bullet\omega) & \equiv & (f\bullet X \twoheadrightarrow X\bullet(f\bullet X))\bullet(f\bullet\omega) \\
& \mapsto\!\!\!\!\!\rightarrow_{\rho\sigma\delta} & [f\bullet X \ll f\bullet\omega](X\bullet(f\bullet X)) \\
& \mapsto\!\!\!\!\!\rightarrow_{\rho\sigma\delta} & \omega\bullet(f\bullet\omega) \\
& \mapsto\!\!\!\!\!\rightarrow_{\rho\sigma\delta} & \ldots
\end{array}
$$

2.2.2 Typed encoding of normalizing strategies

Many type systems for the λ-calculus can be generalized to the ρ-calculus. The type system of $\rho_{\to}$ [CLW03] allows one to type (object oriented flavored) fixpoints, leading to an expressive and safe calculus. In particular, using pattern matching, one can encode and typecheck term rewriting systems in a natural and automatic way.

Figure 3 presents the typing rules of $\rho_{\to}$, which are directly inspired by the simply typed λ-calculus.

$$\frac{\alpha{:}\tau \in \Gamma \quad \tau \in \mathcal{T}y}{\Gamma \vdash \alpha : \tau} \ (Start) \qquad \frac{\Gamma, \Delta \vdash P : \tau_1 \quad \Gamma, \Delta \vdash A : \tau_2}{\Gamma \vdash P \twoheadrightarrow_{\Delta} A : \tau_1 \twoheadrightarrow \tau_2} \ (Abs)$$

$$\frac{\tau \in \mathcal{T}y}{\Gamma \vdash \mathsf{stk} : \tau} \ (Stuck) \qquad \frac{\Gamma \vdash A : \tau_1 \twoheadrightarrow \tau_2 \quad \Gamma \vdash B : \tau_1}{\Gamma \vdash A \bullet B : \tau_2} \ (Appl)$$

$$\frac{\Gamma \vdash A : \tau \quad \Gamma \vdash B : \tau}{\Gamma \vdash A; B : \tau} \ (Struct) \qquad \frac{\Gamma, \Delta \vdash P : \tau_1 \quad \Gamma \vdash B : \tau_1 \quad \Gamma, \Delta \vdash A : \tau_2}{\Gamma \vdash [P \ll_{\Delta} B]A : \tau_2} \ (Match)$$

Figure 3: The Type System for $\rho_{\to}$

- (*Start*)**:** The context determines the type of variables and constants. It cannot contain two declarations for the same variable (or constant);

- (*Abs*)**:** We impose $\mathcal{D}om(\Delta) = \mathcal{FV}(P)$. For the left-hand side of the arrow-type, we use the type of the pattern P; notice that the (*Abs*) rule allows one to hide some type informations in a pattern containing applications, *e.g.* τ_2 disappears in the final type of $(f \bullet X)$ in the judgment $f{:}\tau_2 \twoheadrightarrow \tau_1, X{:}\tau_2 \vdash f \bullet X : \tau_1$.

- (*Appl*)**:** We directly exploit the information given in the type of the function by statically checking that the given argument has the expected type τ_1;

- (*Struct*)**:** This rule states that all the members of a structure have the same type. This is important when considering structures as a collection of results; if a function can return different results, we would at least expect them to have the same type;

- (*Stuck*)**:** Since stk can appear in any structure, it can have any type;

- (*Match*)**:** This rule states that the constraint $[P \ll_{\Delta} B]A$ gets the same type as $(P \twoheadrightarrow_{\Delta} A)\bullet B$. This is sound since $(P \twoheadrightarrow_{\Delta} A) \bullet B \to_{\rho} [P \ll_{\Delta} B]A$. Once again, $\mathcal{D}om(\Delta) = \mathcal{FV}(P)$.

Example 2.4 (Simple type derivation)
The (*Appl*) *rule is effective for the typing of algebraic terms too. Let* $\Gamma \triangleq f{:}\iota \twoheadrightarrow \iota, a{:}\iota$.

$$\frac{\Gamma \vdash f : \iota \twoheadrightarrow \iota \quad \Gamma \vdash a : \iota}{\Gamma \vdash f \bullet a : \iota} \ (Appl)$$

and, let $\Gamma \triangleq f{:}\tau_1 \to \tau_2, g{:}\tau_1 \to \iota, a{:}\tau_1$.

$$\dfrac{\dfrac{\Gamma, X{:}\tau_1 \vdash f \bullet X : \tau_2 \quad \Gamma, X{:}\tau_1 \vdash g \bullet X : \iota}{\Gamma \vdash f \bullet X \to_{(X:\tau_1)} g \bullet X : \tau_2 \to \iota} (Abs) \qquad \Gamma \vdash f \bullet a : \tau_2}{\Gamma \vdash (f \bullet X \to_{(X:\tau_1)} g \bullet X) \bullet (f \bullet a) : \iota} (Appl)$$

This type system is designed as a typing discipline for a programming language: its aim is to ensure that the arguments of a function have the same types as the corresponding formal parameters. However, the notion of (well-typed) pattern used here is crucial since it guarantees that the instantiation of the variables of the pattern will be correct with respect to types (*i.e.* the substitutions obtained as result of the matching are well-typed) even if no type-checking is performed in the matching algorithm.

Example 2.5 *If we take* $f : (\alpha \to \alpha) \to \alpha$ *and* $\Gamma = X : \alpha \to \alpha$ *the term* $\omega \triangleq f \bullet X \to X \bullet (f \bullet X)$ *defined in Example 2.3 is well-typed:*

$$\dfrac{\dfrac{\Gamma \vdash f : (\alpha \to \alpha) \to \alpha \quad \Gamma \vdash X : \alpha \to \alpha}{(b) \;\; \Gamma \vdash f \bullet X : \alpha} \qquad \dfrac{\Gamma \vdash X : \alpha \to \alpha \quad \dfrac{(b)}{\Gamma \vdash f \bullet X : \alpha}}{\Gamma \vdash X \bullet (f \bullet X) : \alpha}}{(a) \vdash \omega \equiv f \bullet X \to X \bullet (f \bullet X) : \alpha \to \alpha}$$

$$\dfrac{\dfrac{(a)}{\vdash \omega : \alpha \to \alpha} \qquad \dfrac{\vdash f : (\alpha \to \alpha) \to \alpha \quad \dfrac{(a)}{\vdash \omega : \alpha \to \alpha}}{\vdash f \bullet \omega : \alpha}}{\vdash \omega \bullet (f \bullet \omega) : \alpha}$$

Starting from fixpoints like the one presented in the previous example one can encode various reduction strategies wrt a given rewrite system by a ρ-term. The encoding methodology together with its properties and some examples can be found in [CLW03]; we give here an informal example that just gives the intuition on the proposed approach.

Example 2.6 (An (ad-hoc) Object-Oriented Encoding) *We can define in the typed* $\rho_{\to}$ *a suitable self-duplicating term that allows us to simulate the* global *behavior of a TRS* $\mathcal{R}$. *For the addition, using two constants* $rec^{(lab \to \iota \to \iota) \to lab}$ *and* $add^{\iota \to \iota \to \iota}$, *the corresponding term is:*

$$plus \;\triangleq\; rec \bullet S \to \left(\begin{array}{l} add \bullet 0 \bullet Y \to Y; \\ add \bullet (suc \bullet X) \bullet Y \to suc \bullet (S.rec \bullet (add \bullet X \bullet Y)) \end{array} \right)$$

Intuitively, the variable S *acts like the meta-variable* this *in* JAVA *and thus, the recursive application of the different rules is realized explicitly by using this variable in the right-hand side of the corresponding rules. This term computes indeed the addition over Peano integers defined by the rewrite system* $add(0, Y) \to Y$, $add(suc(Y), X) \to suc(add(X, Y))$ *where the strategy is explicitly given by the rho-term rec.*

We say that a substitution θ is *well-typed in context* Γ if for any X in the domain of θ such that $\Gamma \vdash X : \tau$ we have $\Gamma \vdash X\theta : \tau$ and we have that:

Lemma 2.1 (Substitution Lemma)
If $\Gamma, \Delta \vdash A : \tau$, then for any substitution θ well-typed in Γ such that $\mathcal{D}om(\theta) = \mathcal{D}om(\Delta)$, we have $\Gamma \vdash A\theta : \tau$.

Using the previous result we can show the correctness of the type system w.r.t. the reduction:

Theorem 2.1 (Subject Reduction for $\rho_{\twoheadrightarrow}$)
If $\Gamma \vdash A : \tau$ and $A \mapsto\!\!\!\!\rightarrow_{\rho\sigma\delta}$, then $\Gamma \vdash B : \tau$.

[*** a few words about the properties below to be added here ***]

Theorem 2.2 (Type Uniqueness for $\rho_{\twoheadrightarrow}$)
If $\Gamma \vdash A : \tau_1$ and $\Gamma \vdash A : \tau_2$, and stk *is not a subterm of A, then $\tau_1 \equiv \tau_2$.*

Theorem 2.3 (Decidability of Typing for $\rho_{\twoheadrightarrow}$)
If stk *is not a subterm of A and $\mathcal{FV}(A) \subseteq \mathcal{D}om(\Gamma)$, then the following problems are decidable:*

1. *Type Reconstruction: given Γ, is there a type τ such that $\Gamma \vdash A : \tau$?*

2. *Type Checking: given Γ, and a type τ, is it true that $\Gamma \vdash A : \tau$?*

2.2.3 Dependent types ensure normalization

Nevertheless, it is important to remark that when the type discipline is enhanced with dependent types, as it was done recently by the authors [BCKL03], the example presented above is statically rejected, *i.e.* blocked by the type system. The chosen dependent type theory introduces pattern matching inside types, and matching failures significantly restrict the set of type-checked programs.

Here are recalled the two main previous typing rules, where patterns are given a context as Church-style type information (*e.g.* in $T_1 : \Delta \to T_2$) but only simple types appear in the types (*e.g.* $\sigma \twoheadrightarrow \tau$).

$$\frac{\Gamma, \Delta \vdash T_1 : \sigma \quad \Gamma, \Delta \vdash T_2 : \tau}{\Gamma \vdash T_1 : \Delta \twoheadrightarrow T_2 : \sigma \twoheadrightarrow \tau} \ (Abs)$$

$$\frac{\Gamma \vdash T_1 : \sigma \twoheadrightarrow \tau \quad \Gamma \vdash T_2 : \sigma}{\Gamma \vdash T_1 \ T_2 : \tau} \ (Appl)$$

In the "pattern-dependent" type system, patterns occur in the types, extending the usual notion of dependent type, where bound variables appear in the types. A type like $\sigma \twoheadrightarrow \tau$ may then have shape $T_1 : \Delta \twoheadrightarrow \tau$. The types for abstractions are delayed matching constraints that may be resolved later using a conversion rule.

$$\frac{\Gamma, \Delta \vdash T_2 : \tau}{\Gamma \vdash (T_1 : \Delta) \twoheadrightarrow T_2 : (T_1 : \Delta) \twoheadrightarrow \tau} \ (Abs)$$

$$\frac{\Gamma \vdash T_1 : (T_{11} : \Delta) \twoheadrightarrow \tau \quad \Gamma \vdash T_2 : \sigma \quad \Gamma, \Delta \vdash T_{11} : \sigma}{\Gamma \vdash T_1 \ T_2 : [T_{11} \ll_\Delta T_2].\tau} \ (Appl)$$

Let us quickly explain why the previous encoding of Ω is not typable in this system. Since patterns appear inside types, with the notation $\Delta \equiv X : \alpha \twoheadrightarrow \alpha$, we have:

$$\vdash \omega \stackrel{\triangle}{=} f(X) : \Delta \twoheadrightarrow X \bullet f(X) \quad : \quad f(X) : \Delta \twoheadrightarrow \alpha$$

However, the constant f must be given a type like:

$$\vdash f \ : \ Y : (Y{:}Z \twoheadrightarrow \alpha) \twoheadrightarrow \alpha$$

Thus, the term $f(\omega)$ can not be typed because of the premises of rule (*Appl*): the pattern T_{11} and the argument T_2 (here respectively Y and ω) should have a common type, but $f(X) \twoheadrightarrow \alpha$ is not convertible to $Z \twoheadrightarrow \alpha$.

Theorem 2.4 (Strong normalization of typable PPTS terms) *[Wac04]*
$\forall \Sigma, \Gamma, A, \Phi$, *if* $\Sigma, \Gamma \vdash A : \Phi$ *then* A *is strongly normalizing.*

3 Descriptive typing of RuleML with description logics

This section shows how the idea of descriptive typing can be applied to reasoning languages of the Semantic Web. At present such languages are still in discussion and development phase, so we focus on the RuleML sublanguage encoding Datalog, which seems to be a stable core for further development (inside and outside REWERSE). The section summarizes the work reported in [HM04]; for further details the reader is referred to that paper.

3.1 Typing Datalog rules

Recall that a Datalog rule is an implicitly universally quantified formula of the form

$$h \leftarrow b_1 \wedge ... \wedge b_n$$

where $n > 0$ and $h, b_1, ..., b_n$ are atomic formulae built over given alphabets of predicate symbols, constants and variables, and $\leftarrow$ is the implication connective. h is called the *head* of the rule, while the conjunction on the right hand side is called *the body*. A Datalog program P is a set of rules and ground atomic formulae (called *facts*). The least Herbrand model semantics (see e.g. [NM95])associates with each program P a set $\mathcal{M}_P$ of ground atomic formulae, which is also the set of all ground atomic logical consequences of the program.

Example 3.1 *Consider the program consisting of the rule*

$$uncle(U, X) \leftarrow father(Y, X), sibling(U, Y)$$

and facts $father(tom, john), sibling(bill, tom), sibling(john, mary)$ *and* $father(tom, mary)$.

The least Herbrand model consists of all facts of this program and of the facts $uncle(bill, john)$ *and* $uncle(bill, mary)$.

The least Herbrand model of a program is defined jointly by the rules and by the facts of the program. We expect that Datalog rules on the web will be used with different sets of facts distributed on the web. Thus for a given set of rules different sets of attached facts will result in

different least Herbrand models. For example, adding the fact $sibling(bob, tom)$ to the example program above will extend its least Herbrand model with two additional facts $uncle(bob, john)$ and $uncle(bob, mary)$.

Generally the constants of a program P represent elements of some application domain $\mathcal{D}$. Thus there exists a mapping $\mathcal{I}$ of the constants of P into objects in $\mathcal{D}$. We extend the mapping $\mathcal{I}$ to a logical interpretation over $\mathcal{D}$ where each n-ary predicate p is interpreted as the relation $p_{\mathcal{I}}$ consisting of those tuples $\langle \mathcal{I}(c_1), ..., \mathcal{I}(c_n) \rangle$ that $p(c_1, .., c_n) \in \mathcal{M}_P$. This interpretation is also a model of P; we will call it the $\mathcal{I}$-induced model of P.

For example, if $\mathcal{I}$ maps the constants of our example to some persons Bill, John, Tom and Mary then in the induced model Bill is the uncle of John and Mary.

The arguments of the relations of the $\mathcal{I}$-induced models will usually range over specific subsets of the domain, rather than over the whole domain. For example we would expect that an uncle of a person is a man and not a woman. In web applications the whole database of facts may not be accessible, or may dynamically change over time. Therefore it might be desirable to characterize a priori (a superset of) the range of each argument of the relation associated with a given predicate in the $\mathcal{I}$-induced model. We call this set the *type* of the argument. The *type of a predicate* is then the Cartesian product of the types of the respective arguments. More precisely for given program P and mapping $\mathcal{I}$ over $\mathcal{D}$ a type specification $\mathcal{T}$ associates with the i-th argument of an n-ary predicate p a subset $\mathcal{T}(p_i)$ of $\mathcal{D}$. Then by $\mathcal{T}(p)$ we denote the set $\mathcal{T}(p_1) \times ... \times \mathcal{T}(p_n)$.

Let $\mathcal{I}$ be a mapping of constants into a domain $\mathcal{D}$ and let $\mathcal{T}$ be a type specification of a program P. We say that P is correct wrt a type specification $\mathcal{T}$ iff for every predicate p in P the relation $p_{\mathcal{I}}$ in the $\mathcal{I}$-induced model is a subset of $\mathcal{T}(p)$.

Example 3.2 *Consider a domain of people including, among others, the persons Bill, John, Tom and Mary. Naturally, the universe divides into two classes:* female *and* male *which are disjoint subsets of the set* person *including all elements of the domain.*

For the program of Example 3.1 we consider the natural mapping $\mathcal{I}$ of constants into the considered domain, assigning the constant mary to person Mary, etc. We can check then that the program is correctly typed wrt the following type specification:

$$\mathcal{T}(father) = male \times person$$
$$\mathcal{T}(sibling) = person \times person$$
$$\mathcal{T}(uncle) \ = male \times person$$

This is because in any tuple of the relation *uncle* of the induced model the first argument is a male.

3.2 A technique for proving type correctness of rules

Clearly the program, may or may not be correct wrt a given type specification. Verification of type correctness of a logic program is a special case of a more general problem of verification of a definite program wrt a specification, discussed by many authors. We apply the method originating from Clark [Cla79] and discussed in Section 2.1.2 (page 8). Specialization of this method to our problem can be summarized by the following proposition.

Proposition 3.1 *Let P be a Datalog program, $\mathcal{T}$ a type specification for P over $\mathcal{D}$, and $\mathcal{I}$ be a mapping of the constants into the domain $\mathcal{D}$.*

If

- *for every ground fact* $p(c_1, ..., c_n)$ *in* P *we have* $\mathcal{I}(c_i) \in \mathcal{T}(p_i)$ *for* $i = 1, ..., n$, *and*
- *for every ground instance*

$$p0(\overline{c_0}) \leftarrow p1(\overline{c_1}) \wedge ... \wedge pk(\overline{c_k})$$

of a rule of P, *whenever* $\mathcal{I}(\overline{c_i}) \in \mathcal{T}(pi)$ *for* $i = 1, ..., k$ *then* $\mathcal{I}(\overline{c_0}) \in \mathcal{T}(p0)$ *where* $\overline{c_i}$ *denotes the respective vector of constants,*

then P *is correct wrt* $\mathcal{T}$.

This proposition holds by soundness of the inductive proof method. It can be easily proved by a structural induction on the proof trees of the program. Intuitively, we want to check that the elements of the tuples in the $\mathcal{I}$-induced model are in the sets specified by a given type specification. The model is obtained by interpretation of constants in the least Herbrand model. Thus, the induced model consists of the tuples which are either directly indicated by facts (first condition of the proposition) or are obtained by the ground instances of the rules, using the T_P operator (see e.g. [NM95]). The type correctness of the latter is guaranteed by the second condition.

Notice that Proposition 3.1 only provides a sufficient condition. The method is not complete: a program may be correct wrt a given type specification which does not satisfy the condition.

Example 3.3 *Consider the program of Example 3.1 with type specification of Example 3.2. In the following ground instance of the program rule:*

$$uncle(mary, tom) \leftarrow father(john, tom), sibling(mary, john)$$

all body atoms are correctly typed but the head atom is not. Thus the program does not satisfy the conditions of Proposition 3.1, but, as discussed in Example 3.2 it is correct wrt the type specification.

In the considered example application of the rule to the facts of the program did not make it possible to obtain incorrectly typed conclusions. However, by adding to the program the correctly typed fact $sibling(ann, tom)$ would make it possible to obtain incorrectly typed conclusion that Ann is the uncle of John. This shows that in web applications the incompleteness of the considered proof method is not a problem, since we want the rules to work properly not only with the given fixed set of facts, but with all correctly typed facts. For this, satisfaction of the verification conditions for the rule is not only sufficient but also necessary.

3.3 Specializing the method to types defined in DL

We now discuss how conditions of Proposition 3.1 can be checked in practice, without generating ground instances of clauses and looking at interpretation of constants. A ground instance of a clause is obtained by binding each variable in a clause to a constant. Consider all occurrences of a variable X in a clause c. Generally it may have m occurrences in the body and p occurrences in the head. For each of them the type specification of the program predicates determines a type $T_i, i = 1, ..., m$ and $R_j, j = 1, ..., p$. To satisfy the second condition of the proposition it is necessary that

$$T_1 \cap ... \cap T_m \subseteq R_1 \cap ... \cap R_p$$

Such a condition should be generated and checked for each variable occurring in the clause. In addition one has to check that every constant occurring in the head is of the type required for its position. Formally, it is not necessary to check the types of the constants in the body, since a constant of wrong type will result in trivial satisfaction of the second condition of the proposition. This means that such a clause would never be applicable to correctly typed atoms. In practice it is desirable to perform this check in order to identify these useless clauses, which most likely were not intended by the programmer.

In order to be able to perform type checking described above we need a language for describing sets to be used as types. Following the Semantic Web approach we propose to adopt for that purpose the language of a Description Logic [BCM+02]. A typing of a program P is thus specified as follows:

- We provide DL definitions T of concepts and roles (Tbox).
- We provide DL axioms A about individuals (Abox), so that for every constant a of P and for any given class expression C the formula $C(a)$ can be proved (or disproved) in the Description Logic from T and A.
- We assign to each n-ary predicate of P an n-tuple of class expressions built with the constructors of the DL over the primitive concepts of the ontology. Each model of T provides thus a typing of the program in the sense of Section 3.1.

Example 3.4 *In most of the Description Logics the type specifications of Example 3.2 could be formulated as follows:*

- *Tbox:*
$$person = male \sqcup female \quad male \sqcap female = \bot$$
- *Abox:*
$$male(john), male(tom), male(bill), female(mary)$$
- *predicate assignment:*
$$\begin{array}{ll} father: & (male,\ person) \\ sibling: & (person,\ person) \\ uncle: & (male,\ person) \end{array}$$

We can now perform automatic type checking wrt to every model of this specification. For this

- using the Tbox, the Abox and the assignment of types to predicates we generate the formulae of the Description Logic, corresponding to the verification conditions described above,
- we prove the verification conditions using a complete DL reasoner.

If the proof succeeds, the program is correct wrt to the typing provided by any model of the ontology.

Example 3.5 *The verification conditions for the rule of Example 3.1 and the type specification of Example3.2 give rise to the following DL formulae:*

$$person \sqsubseteq male, \quad person \sqsubseteq person$$

The first of them is not a logical consequence of the ontology, the other trivially holds.

As the research in DL resulted in several complete and efficient DL reasoners, we can choose any of them to implement a type checker for Datalog, as described above.

For example, in [HM04] we outlined a prototype implementation of a type checker for Datalog rules, encoded in RuleML with types described in OWL. The system uses the RACER[5] reasoner available on the web to perform type checking.

4 Typing query languages

In this section we review that existing work on typing web query languages which we believe is relevant for REWERSE. As the work mainly deals with descriptive typing, we first discuss formalisms to describe regular sets of unranked trees (which are a useful abstraction of XML documents).

4.1 Regular sets of semistructured data

In the context of Web applications, tree structured data are of particular importance. HTML and XML documents are tree-structured. Other kinds of data are required to be translated into trees. XML has constructs, like ID and IDREF attributes/links, for representing graphs as trees.

Recommendations of W3C provide two tools for describing sets of XML documents. A rather restricted formalism of DTD's is a part of XML [XML00]. A more general one is XML Schema [Fal01]. A substantial part of XML Schema is formalized in [XML04]. Relax NG [CM01] is possibly the most important of a few other proposed schema languages. Schema languages, particularly XML Schema, are large and complicated. Thus for the purpose of research one should first choose some abstract mathematical formalism.

There exist well-understood formalisms to deal with trees, or terms, and their sets. It is natural to represent trees as terms, and define their (regular) sets by means of tree-automata, or some equivalent formalism (cf. Section 2.1.2).[6]

There is however a substantial difference between such trees and the tree-structured data of XML. Classical approaches deal with terms, in which each symbol has a fixed arity. The data in logic programming and functional programming are of this form. Such terms correspond to labelled trees where the label of a node determines the number of its children; this number however is not fixed in XML document sets of interest. Sometimes trees of the latter kind are called *unranked*. Some classes of regular sets of unranked trees are introduced, and related to schema languages, in [MLMK03].

Usually ordered (unranked) trees are considered. This is because XML data are ordered. However it is often useful to abstract from the ordering. For instance the order of attributes of an XML element is irrelevant. A corresponding formal notion is that of *mixed tree*, in which a node has either a sequence or a set of children. Such representation of semistructured data was dealt with in [BS02, BS04, WD03, BDM04].

There are two possible approaches to deal with unranked trees. We can represent them by binary trees in a standard way [Knu73] and then apply standard formalisms (like it is done in [HVP00]). Alternatively, we can generalize the tree formalisms to deal with unranked

[5] http://www.sts.tu-harburg.de/~r.f.moeller/racer/

[6] Some researchers prefer defining sets of sequences of trees [HVP00], instead of sets of trees [BKMW01, MLMK03, WD03, BDM04]. Both approaches are equivalent.

trees. Roughly speaking, single sequences (of states in a tree automaton, or of nonterminals in a term grammar) have to be replaced by regular languages. (Thus the resulting formalism has to incorporate a formalism to define regular languages, like regular expressions or finite automata.) Out of tree automata one obtains in this way hedge automata, or unranked tree automata [BKMW01], where instead of a tuple of next states we have a regular language of next states. Similarly, a generalization of regular term grammars can be obtained [MLMK03, WD03, BDM04], where the right hand side of a production contains a regular expression over nonterminals instead of a sequence of nonterminals.

Each of the approaches has its advantages. Using binary trees makes it possible to apply known formalisms and existing tools. On the other hand, the difference between being a child and a sibling is in a sense hidden in the binary tree encoding. (The left child of a node a represents a child of a, the right child represents a sibling of a. Then algorithms working on binary trees treat both kinds of children in the same way.)

Algorithms for descriptive type checking or inference employ some basic operations on types (regular sets of trees), like check for emptiness, computing intersection, or checking inclusion. These operations should be decidable, and efficient algorithms should exist. What poses a difficulty is inclusion checking. It is EXPTIME-complete already for tree automata (but polynomial for deterministic top-down tree automata). For regular sets of unranked trees it is decidable and also EXPTIME-complete. Hosoya et al. [HVP00] present a practical algorithm, which includes some heuristic optimizations. The algorithm is reported to behave efficiently on many practical examples. The intuition is that the types encountered in practice do not use the full power of the formalism.

A formalization of this intuition is proposed in [BDM04, WD03]. That work proposes a restriction on the class of sets, which leads to an efficient, polynomial inclusion check. The restriction seems acceptable from the practical point of view, it resembles some restrictions of DTD's and XML Schema. The used formalism combines regular tree grammars with regular expressions, as outlined above, and the restriction consists of three conditions. First one requires that in a defined set all the trees have the same root label. So to each set name there corresponds a label (of the root of each term in the set). Moreover, if two distinct set names occur in a regular expression in a grammar, then the corresponding labels are distinct. The third condition is that the regular expressions are 1-unambiguous in the sense of [BKW98].

The first two conditions correspond to the class of single-type tree grammars of [MLMK03]; this class includes DTD and "the expressiveness of W3C XML Schema is mostly within" this class [MLMK03]. The third condition seems equivalent to an (informal) restriction on regular expressions in DTD's, from Appendix E of [XML00]. Imposing the first two conditions is similar to requiring a tree automaton to be top-down deterministic (the latter leads to polynomial inclusion checking for tree languages, cf. Section 2.1.2). The third condition assures a linear time transformation (instead of exponential) of regular expressions into deterministic finite automata. For the formalism restricted in such way, efficient and simple polynomial time algorithms exist for the primitive operations of checking inclusion and emptiness, and of computing intersection, needed in descriptive type checking / inference.

4.2 Typing for Web query languages

Here we present a brief overview of some work on typing for queries for XML documents.

XQuery is a W3C proposal for a query language for XML data. It is a typed functional language, with prescriptive typing. There exists a formal definition of XQuery [XML04], it

includes typing rules.

Suciu [Suc02] discusses type checking for a restriction of the transformation language XSLT. The question is whether a given XSLT program applied to XML data from a given type will provide results of a given type. One may use type inference to check this (derive the type of the results and check its inclusion in the given type). It turns out however that methods not employing type inference are stronger. This is due to the fact that the type, say T, provided by an approximation algorithm may be a too rough approximation of the actual set T_0 of generated data. T may be not a subset of the given type U, despite $T_0 \subseteq U$. A method that does not employ type inference can be stronger, as it can take U into account, in contrast to type inference algorithms.

An interesting result (from [AMN+01]) is that such type checking problem is, in general case, undecidable. Thus for any (correct) type checking algorithm there exist counterexamples, where all the results of an XSLT program are in a given type but the algorithm is unable to find this. Some subclasses are known, for which the problem is decidable.

The functional language CDuce [BCF03] is adapted to the manipulation of XML documents. This language provides powerful pattern matching, first class functions, overloaded functions, a very rich type system (arrows, sequences, pairs, records, intersections, unions, differences), precise type inference for patterns and error localization, and a natural interpretation of types as sets of values. Additionally, static type information can be used to obtain very efficient compilation schemas.

XDuce [HP03] is a statically typed programming language for XML processing. Its basic data values are XML documents, and its types (so-called regular expression types) directly correspond to document schemas. XDuce also provides a flexible form of regular expression pattern matching, integrating conditional branching, tag checking, and subtree extraction, as well as dynamic typechecking.

Regular types have been proposed as a foundation for statically typed processing of XML and other forms of tree-structured data. and they have been mainly explored in special-purpose languages (e.g., XDuce, CDuce, and XQuery). The XTATIC language [GP03] is a combination of the tree-structured data model of XDuce and the classes-and-objects data model of a conventional object-oriented language and its goal is to bring regular types to a broad audience by offering them as a lightweight extension of a popular object-oriented language, C#.

Descriptive typing for Xcerpt (cf. Section 1.3.1) rules was discussed in [WD03]. That approach follows the work on descriptive typing for logic programming with declarative semantics (cf. page 7, and [DMP02] together the references therein). The data is modelled as unranked mixed trees (cf. p. 21 above), called *data terms*. Types are defined by the formalism outlined in the previous section (p. 22). The formalism is a generalization of regular tree grammars to unranked terms, with some restrictions guaranteeing polynomial inclusion checking and other basic operations.

An Xcerpt program is a set of rules. The *body* (right hand side) of a rule is a conjunction of *query terms*. Each query term is to be matched against a given database, by means of so called *simulation unification*. As a result a (possibly non unique) binding of data terms to variables is obtained. Applying this binding to the *head* (left hand side) of the rule results in a data term that is a *result* of the rule. In a general case the computation resembles bottom-up evaluation of logic programs: the body of a rule is matched against the database augmented by the already obtained results. Thus chaining of rules and recursion is possible.

Type checking of an Xcerpt program consists of independent type checking of its rules. For each rule one computes (a superset approximation of) its results, under assumption that the data matched against are in the specified type. Then one checks whether the obtained set is a subset of the specified type. The core of the approach is approximating the set of the results of an Xcerpt rule.

To compute the set of rule results, one first approximates the results of simulation unifications. For each variable and each query term in the rule body one computes the set of possible bindings of the variable, resulting from the unification. Then the sets for each variable are intersected. In this way the set of possible values for each variable is found, out of these sets the set of possible rule results is obtained. Notice that this method can be employed for type inference for non-recursive sets of rules.

The work described in [WD03] deals with a simplified version of Xcerpt; the main restriction is that data terms representing non-tree graphs are excluded. The computed approximations of the sets of rule results are shown to be exact (i.e. the obtained set is the set of the results of the given rule applied to data from the given type), provided the rule does not contain the "at" ($\rightsquigarrow$) construct of Xcerpt. (This construct is treated in a simplified way). The computed approximations are in the form of a finite union of types (i.e. of sets specified by the above mentioned formalism). The union may be not expressible by a single type, as the formalism is not closed under union.

The decidability of type checking for Xcerpt is not discussed in [WD03]. For single rules for which the set of results is computed exactly by the presented method, the problem is obviously decidable. In a general case a negative answer of type checking does not imply that the Xcerpt program is not type correct. A question whether the problem is undecidable, similarly to type checking for XSLT, is open.

5 Conclusions and future work

The objective of this report was to survey the existing work on typing relevant for the emerging rule layer of the Semantic Web, in particular for the ongoing work in REWERSE. As REWERSE is to develop a collection of reasoning languages and prototype processors for these languages, a natural question is whether these languages are to be typed, and if yes, what are the suitable type systems. The ongoing work on defining the REWERSE languages can be linked to two different initiatives:

- RuleML, aiming at definition of a family of web reasoning languages,
- Xcerpt, developing a declarative query and transformation language for XML databases.

Both have some connections to logic programming and their computations can be seen as rewritings. Therefore the report focuses on selected work on typing done in constraint logic programming and in the theory of rewriting.

The initial research done in REWERSE, reported therein, shows that the concept of descriptive types originating from the field of logic programming is relevant for the emerging REWERSE languages. In particular, the following ideas seem to be worth further investigation:

- Using extensions of regular tree grammars for specifying descriptive types for the Semantic Web query language developed on the basis of Xcerpt by the REWERSE WG I4. This work should include the following steps:

  - Integration of the type checking algorithms for Xcerpt described in [BDM04, WD03] with the prototype of the Xcerpt system.
  - Investigation of the use of typing information in query evaluation for improvements of the Xcerpt system.
  - Investigation of the ways of using ontologies in Xcerpt; while an ontology provides a conceptualization of the application domain, present version of Xcerpt deals with pure XML data. Relating types of Xcerpt to classes of the ontology could provide a link between the concepts defined by the ontology and the representation of their instances.

- Using ontology languages for specifying descriptive types for web rule languages, while type checking of rules reduces to ontology reasoning. An instance of this approach is a typechecker for RuleML with types specified in OWL, described in [HM04] (see Section 3). A continuation of this work should investigate application of this technique to SWRL, and to the rule languages to be defined by the REWERSE WG I1. Extension of the approach to the case of dynamic typing, when types of the variables are explicitly included as constraints in the rules should also be considered.

The use of prescriptive types in REWERSE languages addresses the complementary issue of composition: how ontologies can be combined, how Datalog rules or different reasoning rules can be composed? The following workplan is proposed in this direction:

- Study the adaptation of a prescriptive type system for constraint logic programs to Xcerpt/RuleML, in particular revisit the structure of signatures in order to treat non-fixed arity extensible structures,

- Design a prescriptive type system for Xcerpt/RuleML,

- Experiment the use of typed Xcerpt/RuleML for complex queries involving different kinds of ontologies or reasoning.

The typed rewriting calculus offers a conceptually simple way to describe uniformly three fundamental components of any rule based language: (1) computation rules described by a confluent and terminating rewriting relation, (2) deduction or search rules whose application should be directed by (3) strategies. Since these paradigms are intensively used in the reasoning and query languages under development in WG I1 and WG I4, the prescriptive type systems proposed in this context [CKLW03, CLW03, Wac04] could be good candidates for typing the language constructions of the REWERSE project. In seems therefore useful to further investigate:

- The applicability of this (term) rule based model to the entities of interest in REWERSE, in particular the ruleML and Xcerpt languages,

- The kind of type system enforcing a strongly terminating behavior of the typed ρ-terms,

- The meta-properties of the type systems of interest, in particular type inference.

These investigations are intended to be done in parallel. One way to combine their benefits in the future would be to define descriptive types for a prescriptively typed language, with a simple relation of abstraction between both type hierarchies.

The progress of the above mentioned work will be reported in the deliverables I3-D4 (month 12) and I3-D6 (month 24).

Acknowledgements Hans Jürgen Ohlbach served as a second reader on this report and we are very grateful for his useful and pertinent comments and suggestions.

References

[Aik94] A. Aiken. Set constraints: Results, applications, and future directions. In *Second Workshop on the Principles and Practice of Constraint Programming*, volume 874 of *LNCS*, pages 171–179. Springer-Verlag, May 1994.

[Aik99] A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35:79–111, 1999.

[AL94] A. Aiken and T. K. Lakshman. Directional type checking of logic programs. In B. LeCharlier, editor, *Proc. of the 1st Int. Symposium on Static Analysis*, volume 864 of *LNCS*, pages 43–60. Springer-Verlag, 1994.

[AMN+01] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Typechecking XML views of relational data. In *LICS*, pages 138–149, 2001.

[Apt97] K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[BC89] A. Bossi and N. Cocco. Verifying correctness of logic programs. In J. Díaz and F. Orejas, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development TAPSOFT '89*, volume 352 of *LNCS*, pages 96–110. Springer-Verlag, 1989.

[BCF03] V. Benzaken, G. Castagna, and A. Frisch. Cduce: An XML-centric general-purpose language. In C. Runciman and O. Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003*, pages 51–63. ACM Press, 2003.

[BCKL03] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Patterns Type Systems. In *Proc. of POPL*. ACM, 2003.

[BCM+02] F. Baader, D. Calvanese, D. McGuiness, D. Nardi, and P. P.-S. (eds.). *The Description Logic Handbook*. Cambridge University Press, 2002.

[BDM04] F. Bry, W. Drabent, and J. Maluszynski. On subtyping of tree-structured data: A polynomial approach. In *Proceedings PPSWR*, 2004.

[BJM00] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[BKK+98] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 329–344. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[BKMW01] A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets. Technical report, The Hong Kong University of Science and Technology, April 2001. `http://hdl.handle.net/1783.1/738`.

[BKW98] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, May 1998.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM SIGMOD-SIGACT Symposium on Principles of Datebase Systems*, pages 1–15, 1986.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.

[BS02] F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *Proc. of the International Conference on Logic Programming*, LNCS. Springer-Verlag, 2002.

[BS04] F. Bry and S. Schaffert. Querying the web reconsidered: A practical introduction to Xcerpt. Technical Report PMS-FB-2004-7, Ludwig Maximilians Universität München, April 2004.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM POPL*, pages 238–252, 1977.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. (For a correct version of this paper, see `http://www.di.ens.fr/~cousot`.).

[CC92b] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Proceedings of the Fourth International Symposium, PLILP'92*, number 631 in LNCS, pages 269–295. Springer-Verlag, 1992.

[CC95] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Conference Record of FPCA '95 SIGPLAN/SIGARCH/WG2.8 Conference on Functional Programming and Computer Architecture*, pages 170–181. ACM Press, 1995.

[CDE+99] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. The Maude system. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications, 10th International Conference, RTA'99, Trento, Italy, July 2–4, 1999, Proceedings*, volume 1631 of *Lecture Notes in Computer Science*, pages 240–243. Springer-Verlag, 1999.

[CDG+02] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. http://www.grappa.univ-lille3.fr/tata/, 2002.

[CF02] E. Coquery and F. Fages. Tclp: overloading, subtyping and parametric polymorphism made practical for constraint logic programming. Technical report, INRIA Rocquencourt, 2002.

[CK99] H. Cirstea and C. Kirchner. Combining Higher-Order and First-Order Computation Using ρ-calculus: Towards a Semantics of ELAN. In *Frontiers of Combining Systems 2*, pages 95–120. Wiley, 1999.

[CK00] H. Cirstea and C. Kirchner. The Simply Typed Rewriting Calculus. In *Proc. of WRLA*. Electronic Notes in Theoretical Computer Science, 2000.

[CK01] H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

[CKL01a] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In *Proc. of RTA*, volume 2051 of *Lecture Notes in Computer Science*, pages 77–92. Springer-Verlag, 2001.

[CKL01b] H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In *Proc. of FOSSACS*, volume 2030 of *Lecture Notes in Computer Science*, pages 166–180, 2001.

[CKLW03] H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. Rewrite strategies in the rewriting calculus. In B. Gramlich and S. Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.

[CL96] Y. Caseau and F. Laburthe. *Introduction to the CLAIRE Programming Language*. Département Mathématiques et Informatique, Ecole Normale Supérieure, September 1996.

[Cla79] K. L. Clark. Predicate logic as computational formalism. Technical Report Technical Report 79/59, Imperial College, London, December 1979.

[CLW03] H. Cirstea, L. Liquori, and B. Wack. Rewriting calculus with fixpoints: Untyped and first-order systems. In *Post-procededings of TYPES'03*, 2003.

[CM01] J. Clark and M. Murata (editors). RELAX NG specification, December 2001. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

[Cod99] M. Codish. Efficient goal directed bottom-up evaluation of logic programs. *Journal of Logic Programming*, 38(3):355–370, 1999.

[Com91] H. Comon. Rewriting with membership constraints. Research report xx, Laboratoire de Recherche en Informatique, Orsay, France, 1991.

[Cou90] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, pages 193–242. North-Holland, Amsterdam, 1990.

[CP98] W. Charatonik and A. Podelski. Directional type inference for logic programs. In G. Levi, editor, *Proceedings of the 5th Static Analysis Symposium (SAS'98)*, volume 1503 of *LNCS*. Springer-Verlag, 1998.

[CR03] C. Culbert and G. Riley. *Basic Programming Guide*, June 2003.

[Der93] P. Deransart. Proof methods of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.

[DH88] R. Dietrich and F. Hagl. A polymorphic type system with subtypes for Prolog. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming ESOP'88*, LNCS, pages 79–93. Springer-Verlag, 1988.

[DHK96] A. Deursen, J. Heering, and P. Klint. *Language Prototyping*. World Scientific, 1996. ISBN 981-02-2732-9.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 207–212, 1982.

[DM88] W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59(1):133–155, 1988.

[DM93] P. Deransart and J. Małuszynski. *A Grammatical View of Logic Programming*. MIT Press, 1993.

[DM01] W. Drabent and M. Miłkowska. Proving correctness and completeness of normal programs — a declarative approach. In P. Codognet, editor, *Logic Programming, 17th International Conference, ICLP 2001, Proceedings*, volume 2237 of *LNCS*, pages 284–299. Springer-Verlag, 2001.

[DMP02] W. Drabent, J. Maluszynski, and P. Pietrzak. Using parametric set constraints for locating errors in CLP programs. *Theory and Practice of Logic Programming*, 2(4–5):549–610, 2002.

[DP98] W. Drabent and P. Pietrzak. Inferring call and success types for CLP programs. ESPRIT DiSCiPl deliverable, Linköpings universitet, 1998. Available at URL: `http://discipl.inria.fr/deliverables2.html`.

[DP99] W. Drabent and P. Pietrzak. Type Analysis for CHIP. In A. Haeberer, editor, *Proc. of the Seventh International Conference on Algebraic Methotology and Software Technology (AMAST'98)*, volume 1548 of *LNCS*, pages 389–405. Springer-Verlag, 1999.

[DTT97] P. Devienne, J.-M. Talbot, and S. Tison. Set-based analysis for logic programming and tree automata. In *Proceedings of the Static Analysis Symposium, SAS'97*, volume 1302 of *LNCS*, pages 127–140. Springer-Verlag, 1997.

[DZ92] P. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. The MIT Press, 1992.

[Fal01] D. C. Fallside (ed.). XML Schema Part 0: Primer. `http://www.w3.org/TR/xmlschema-0/`, 2001.

[FC01] F. Fages and E. Coquery. Typing constraint logic programs. *Theory and Practice of Logic Programming*, 1(6):751–777, November 2001.

[For81] C. Forgy. Ops5 user's manual. Technical Report CMU-CS-81-135, Carnegie-Mellon University, Pittsburgh, USA, July 1981. 62 pages.

[FP97] F. Fages and M. Paltrinieri. A generic type system for CLP($\mathcal{X}$). Technical report, Ecole Normale Supérieure LIENS 97-16, December 1997.

[Frü95] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1995.

[FSVY91] T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In G. Kahn, editor, *Annual IEEE Symposium on Logic in Computer Science*, pages 300–309, Amsterdam, 1991. Corrected version available from `http://WWW.pst.informatik.uni-muenchen.de/~fruehwir`.

[GdW94] J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In P. V. Hentenryck, editor, *Proc. of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.

[GHRS03] B. Grosof, I. Horrocks, R.Volz, and S.Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *Proc. 12th International World Wide Web Conference*, pages 48–57. ACM Press, 2003. http://www2003.org/cdrom/papers/refereed/p117/p117-grosof.html.

[GP03] V. Gapeyev and B. Pierce. Regular object types. In L. Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*, volume 2743 of *Lecture Notes in Computer Science*, pages 151–175. Springer, 2003.

[Han92] M. Hanus. *Logic Programming with Type Specifications*, chapter 3, pages 91–140. MIT Press, 1992. In [Pfe92].

[HCC95] P. V. Hentenryck, A. Cortesi, and B. L. Charlier. Type analysis of Prolog using type graphs. *Journal of Logic Programming*, 22(3):179–209, March 1995.

[Hei92] N. Heintze. Practical aspects of set based analysis. In M. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming* , pages 765–779. The MIT Press, 1992.

[HJ90a] N. Heintze and J. Jaffar. A decision procedure for a class of set constraints (extended abstract). In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 42–51, 1990.

[HJ90b] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 197–209. ACM Press, 1990.

[HKK98] C. Hintermeier, C. Kirchner, and H. Kirchner. Dynamically-typed computations for order-sorted equational presentations. *Journal of Symbolic Computation*, 25(4):455–526, 98. Also report LORIA 98-R-157.

[HL94] P. Hill and J. Lloyd. *The Gödel programming language*. MIT Press, 1994.

[HM04] J. Henriksson and J. Maluszynski. Static type-checking of datalog with ontologies. In *Principle and Practice of Semantic Web Reasoning*. Springer-Verlag, 2004. Proc. of the Int. Workshop PPSWR04.

[HP03] H. Hosoya and B. C. Pierce. Xduce: A statically typed XML processing language. *ACM Trans. Inter. Tech.*, 3(2):117–148, 2003.

[HPSB+04] I. Horrocks, P. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. http://www.daml.org/rules/proposal/, April 2004.

[HVP00] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proc. of the International Conference on Functional Programming*, pages 11–22. ACM Press, 2000.

[Ilo02] Ilog. Business Rules: ILOG Rules White Paper, October 2002.

[JB92] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2 & 3):205–258, 1992.

[JL87] J. Jaffar and J. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL'87*, pages 111–119. ACM SIGACT-SIGPLAN, 1987.

[Knu73] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 10 January 1973.

[LR91] T. Lakshman and U. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 202–217. MIT Press, 1991.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[Mes98] J. Meseguer. Membership algebra as a semantic framework for equational specification. In *Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1998.

[Mil99] P. Mildner. *Type Domains for Abstract Interpretation, A Critical Study*. PhD thesis, Uppsala University, 1999.

[Mis84] P. Mishra. Towards a theory of types in Prolog. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 289–298, 1984.

[MLMK03] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. Submitted, 2003.

[MO84] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[MRV03] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In G. Hedin, editor, *International Conference on Compiler Construction - CC'2003, Varsovie, Pologne*, volume 2622 of *Lecture notes in Computer Science*, pages 61–76, Apr 2003.

[Nai92] L. Naish. *Types and the intended meaning of logic programs*, chapter 6, pages 189–216. MIT Press, Cambridge, Massachusetts, 1992.

[Nev02] F. Neven. Automata, logic, and XML. In *CSL*, 2002. Invited talk.

[NM95] U. Nilsson and J. Małuszyński. *Logic, Programming and Prolog*. John Wiley, 2 edition, 1995. Second Edition.

[Pfe92] F. Pfenning, editor. *Types in Logic Programming*. MIT Press, 1992.

[PP97] A. Podelski and L. Pacholski. Set constraints – a pearl in research on constraints. In *Proceedings of the Third International Conference on the Principles and Practice of Constraint Programming*, number 1330 in LNCS. Springer-Verlag, October 1997.

[SFD00] J.-G. Smaus, F. Fages, and P. Deransart. Using modes to ensure subject reduction for typed logic programs with subtyping. In *Proceedings of FSTTCS '2000*, number 1974 in LNCS. Springer-Verlag, 2000.

[SHC96] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):14–64, 1996.

[Smo98] G. Smolka. Special issue on order-sorted rewriting foreword by the Guest Editor. *Journal of Symbolic Computation*, 25(4):395–395, April 1998.

[Sta03] S. Staab (ed.). Where Are the Rules. *IEEE Intelligent Systems*, pages 76–83, September/October 2003.

[Suc02] D. Suciu. The XML typechecking problem. *SIGMOD Record*, 31(1):89–96, 2002.

["T02] "Terese" (M. Bezem, J. W. Klop and R. de Vrijer, eds.). *Term Rewriting Systems*. Cambridge University Press, 2002.

[TDT00] J.-M. Talbot, P. Devienne, and S. Tison. Generalized definite set constraints. *Constraints: An International Journal*, 5(1-2):161–202, 2000.

[Tho97] W. Thomas. Languages, automata and logic. In *Handbook of Formal Languages*, volume 3, chapter 7. Springer Verlag, 1997.

[Vis99] E. Visser. *The Stratego Tutorial*. Department of Computer Science, Universiteit Utrecht, Utrecht, The Netherlands, 1999.

[vO90] V. van Oostrom. Lambda calculus with patterns. Technical report, Vrije Universiteit, Amsterdam, November 1990.

[Wac04] B. Wack. The simply-typed pure pattern type system ensures strong normalization. In J.-J. Levy, editor, *Proc. of TCS'04*, Toulouse (France), August 2004.

[WD03] A. Wilk and W. Drabent. On types for XML query language Xcerpt. In *International Workshop, PPSWR 2003, Mumbai, India, December 8, 2003, Proceedings*, number 2901 in LNCS, pages 128–145. Springer Verlag, 2003.

[X-T88] X-tra 1.0 - manuel de reference, 1988.

[XML00] Extensible markup language (XML) 1.0 (second edition), W3C recommendation, 6 October 2000. `http://www.w3.org/TR/REC-xml`.

[XML04] XQuery 1.0 and XPath 2.0 formal semantics, 20 February 2004. W3C Working Draft, `http://www.w3.org/TR/xquery-semantics/`.

[YS91] E. Yardeni and E. Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–153, 1991.