A Descriptive Type Foundation for RDF Schema [1]

Gabriel Ciobanu[a], Ross Horne[a,b,c], Vladimiro Sassone[d]

[a] *Romanian Academy, Institute of Computer Science, Blvd. Carol I, no. 8, Iaşi, Romania*
[b] *Nanyang Technological University, School of Computer Engineering, Singapore*
[c] *Kazakh-British Technical University, Faculty of Information Technology, Almaty, Kazakhstan*
[d] *University of Southampton, Electronics and Computer Science, Southampton, UK*

Abstract

This paper provides a type theoretic foundation for descriptive types that appear in Linked Data. Linked Data is data published on the Web according to principles and standards supported by the W3C. Such Linked Data is inherently messy: this is due to the fact that instead of being assigned a strict a priori schema, the schema is inferred a posteriori. Moreover, such a posteriori schema consists of opaque names that guide programmers, without prescribing structure. We employ what we call a descriptive type system for Linked Data. This descriptive type system differs from a traditional type system in that it provides hints or warnings rather than errors and evolves to describe the data while Linked Data is discovered at runtime. We explain how our descriptive type system allows RDF Schema inference mechanisms to be tightly coupled with domain specific scripting languages for Linked Data, enabling an interactive feedback to Web developers.

Keywords: Linked Data, RDF, schema, type systems, operational semantics

1. Introduction

This paper is the second of two journal papers that address RDF Schema [9] from a type theoretic perspective. RDF Schema is a data-modelling vocabulary for the Resource Description Framework (RDF) [16], where RDF is a W3C recommended data format for publishing data on the Web.

This work spans two journal papers, since certain aspects of RDF Schema are best treated using a conventional approach to typing, whereas other aspects are quite unconventional even to the seasoned type theorist. The previous paper in the series [14] treated the conventional typing aspects of RDF Schema that concern familiar simple datatypes [43] such as integers and strings. As we know, if a type system guarantees that a variable is a string, but the same variable appears in an expression for integers, then a type error arises. We call such conventional type systems *prescriptive* type systems, since the type system prescribes that the variable concerned *must* be a particular type, hence can only be used in the manner prescribed. A prescriptive type system is appropriate for aspects of RDF Schema concerning simple data types. However prescriptive typing is less appropriate for other aspect of RDF Schema.

In this second paper in the series, that may be read independently of the first, we address less conventional aspects of RDF Schema types. The aspects we model concern opaque names, where there is no difference in the underlying structure of names that inhabit distinct types. In this paper, all resources are named by a URI — a Web address such as *res:Vitali_Klitschko* or *res:Udar* . Since all URIs are URIs, no runtime type error would arise if one URI is accidentally used in place of another URI. However, these URIs are intended to represent resources that are understandable to human beings. If resources are used in the wrong place in data, then the data may not make sense.

The descriptive type system we introduce enables simple routine data-modelling slips to be detected. RDF is based on triples of URIs that represent how two URIs are related to each other. Here the property that relates the two

 Article available on ScienceDirect at DOI:10.1016/j.jlamp.2016.02.006. Corresponding author: Ross Horne.
Email address: rhorne@ntu.edu.sg (Ross Horne)

URIs is also a URI, e.g. *free:government/politician/party* . The programmer may simply have accidentally switched the expected order of the two URIs related to imply something nonsensical, such as: " *res:Udar* is a member of the political party *res:Vitali_Klitschko* ". Since our descriptive type system would describe that the above property relates politicians to political parties, then our type system issues a warning suggesting that either *res:Udar* is politician, or there is some problem with the data. In another scenario, the wrong person may be used. For example, the statement " *res:Wladimir_Klitschko* is a member of the political party *res:Udar* " would result in a warning, since Wladimir Klitschko, Vitali's brother, is not a politician. From meaningful warnings a human is likely to spot the problem with the data. Note that types are themselves named using URIs that do not impose any structure on the data itself. For example, the type politician can be represented by the URI *free:government.politician* .

The descriptive type approach illustrated above differers from the standard approach to RDF Schema inference [25]. In both of the above examples, standard RDF Schema inference would wrongly infer that *res:Udar* and *res:Wladimir_Klitschko* are politicians. In the descriptive typing approach a warning that presents a menu of options is generated. From the menu, the human reading the warning can select the best option, where the options include the standard RDF Schema inference along with several other possible courses of action. Furthermore, since, unlike errors, warnings may be ignored, the choice of inference may be suspended while the program continues. At a later point more illuminating data may be obtained that helps resolve the warnings; or, perhaps, the warning can be ignored indefinitely citing imperfect schema information.

This line of work also considers how descriptive types can be of assistance to programming languages that consume Linked Data [19, 24, 28]. Linked Data is data published on the Web according to certain principles and standards. The main principle laid down by Berners-Lee in a note [6] is to use HTTP URIs to identify resources in data. By using HTTP URIs, anyone can use the HTTP protocol to look up (dereference) resources that appear in data in order to obtain more data. All URIs that appear in this paper are real dereferenceable URIs that you can dereference by following the links in the electronic version of this article.

The descriptive type system introduced in this work can be used for typing programs, as well as data. For example, the descriptive type system can raise warnings when a query over RDF data involves properties that make no sense according the their schema, for example the subject and object of a statement are accidentally reversed. When a program is well typed, the program can be used in confidence that there will be no warnings and hence unwanted RDF Schema inferences will never be applied.

If you ask the Linked Data scientist whether there is any link between types in RDF and type systems, they will explain that there is almost no connection. Traditionally, type systems are used for static analysis to prescribe a space of constraints on data and programs. In contrast, types in RDF change to describe the data instead of prescribing constraints on the data. In this work, we provide a better answer to the question of the type-theoretic nature of types in Linked Data, by distinguishing between *prescriptive type systems* and *descriptive type systems*. The idea of descriptive types arose in joint work with Giuseppe Castagna and Giorgio Ghelli. Here we instantiate it for our Linked Data scripting language [14]. Descriptive type systems, not formally related to this work, appear in work on logic programs, tree data structures and dynamically typed objects [22, 33, 15, 5, 17, 26].

This work is an extended version of the invited conference version presented at PSI 2014 [13]. This version of the paper closes further the gap between the descriptive type system in the conference version and W3C standards. The new contributions compared to the conference version are:

- An extended introduction that presents the W3C standard RDF Schema inference mechanism called *simple entailment* and, by intuitive examples, compares the standards to the approach enabled by the descriptive type system in this work.
- An extended syntax and type system covering a larger subset of the RDF Schema standard, with features corresponding to not only *rdf:type* triples but also triples with *rdfs:subClassOf* , *rdfs:domain* and *rdfs:range* as the property.
- A proposition formally relating W3C standard simple entailment to inference for descriptive types; accompanied by common-sense recommendations about good practice for designing ontologies to work well with both descriptive type systems and the W3C standards.

This version also expands considerably the discussion and deals more carefully with algorithmic issues regarding generating and solving subtype constraints. Hence this version fully supersedes the invited conference version.

In Section 2, we provide a self-contained section that explains this work in the context of existing work on type systems for semi-structured data and RDF Schema. We present a motivating example of a scenario where descriptive typing can be applied to Linked Data to present meaningful warnings to a programmer that would like to interact with Linked Data. This section can be read separately without going into details of the type system or the scripting language.

In Section 3, we develop technical prerequisites for our descriptive type system. In particular, we require a notion of type and a consistent notion of subtyping. We develop these notions and present supporting results.

In Section 4, we continue the technical development of the type system. We introduce a simple scripting language for dereferencing resources over the Web and querying Linked Data in a local store. We devise an algorithmic type system that we use as part of our typing and inference mechanism. This section formally connects the descriptive type system developed with the standard RDF Schema inference mechanisms and justifies differences between the approaches.

In Section 5, we specify the behaviour of scripts using a novel operational semantics. This section formalises the real novelty of descriptive types as a means to support the inference of schema at runtime. The operational semantics allows us to refine the type system during execution in response to warnings about potential mismatches between the data and schema. We describe an algorithm for deriving warnings based on constraints generated by the operational semantics and algorithmic type system. We conclude with a type reduction result that proves that, if the type system is sufficiently refined, then the script will run without unnecessary warnings; and hence no new inferences need be applied.

2. Type Systems for Semi-Structured Data

Schema for the Resource Description Framework (RDF) are considerably different from schema for other popular forms of data representation including XML and relational databases. We firstly review the state of the art of XML Schema as a type system for XML and also types for relational databases, before explaining why such approaches do not directly translate across to RDF Schema.

Types for XML. XML is essentially a standardised syntax for serialising abstract syntax trees [45]. XML Schema [23] is used to describe the permitted structure of abstract syntax trees as well as the primitive datatypes that may appear inside the tree.

XML Schema are types for data, as clarified and formalised in a line of work where type systems for programming languages have been extended with types inspired by XML Schema. Notable contributions include the prototype functional programming languages XDuce [29] and CDuce [4]. In these prototype languages, it is possible to statically type check functions that manipulate XML. To consider an example, suppose also that we would like to transform SPARQL Query XML Results into GraphML format[2]. When defining a function performing the transformation, CDuce will infer the type of the function. If there is a problem with the function, then either a type checking error will occur, or the wrong type will be inferred, suggesting to the programmer how the function may be fixed.

The investigation into XML Schema and type systems have lead to non-trivial developments in the theory of types. The theory of semantic subtyping [21] was originally motivated by the problem of developing a subtype system for XML Schema. Semantic subtyping employs a denotational model of types to derive a subtype relation.

Types for relational data. Languages for relational databases have long been endowed with type systems [1]. Rows in a table, or relations, can be typed using record types [12]. A combination of record types and type constructors for lists, bags and sets can be used to type complex query languages and database programming languages as demonstrated in [10]. For example the language Kleisli [49] incorporates a type system and has inspired the development of modern query languages. Related work on types for RDF [36] suggested a distinct approach to ours based on record types.

[2]For SPARQL XML results schema see `http://www.w3.org/2007/SPARQL/result.xsd`, for GraphML see `http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd`.

Why Types for RDF Schema are Different. This work proposes quite a different use of type systems compared to existing systems for XML and relational databases mentioned above. In the context of XML Schema and relational databases, the method for integrating the type system with programming languages is conventional, in the sense that the types are used to prescribe a space in which only certain structures are permitted to exist, and if a structure is not inhabited by the given type, then a type error occurs.

RDF Schema is designed for messy data that is combined from multiple heterogeneous datasets published on the Web. RDF Schema is only intended to be used as a guideline for how data may be used, rather than a prescription for how data must be used. For example, suppose that data is discovered from one source stating that a resource has type boxer, while in another source statements are made about the resource that can only apply to politicians. In this scenario, a type error need not be thrown. Instead of generating an error, RDF Schema is used to infer that person is both a boxer and a politician. The fundamental difference to the type systems described above is that the type of the resource itself (the person) has been refined at runtime; in contrast, type information in traditional prescriptive type systems is static.

This work builds on the conference version of this paper [13] to establish a formal system that respects RDF Schema inference [9]. Further to respecting standard RDF Schema inference, the system enables some new forms of inference that are useful for completing missing schema information. This paper formalises and illustrates the underlying system, called a descriptive type system, that reflects the requirements of RDF Schema inference as well as enabling tight integration with domain specific scripting languages.

A triple is the basic unit for representing data in RDF. It consists of a subject, a property and an object. The subject is a URI that names the resource being described, the property is a URI indicating how the subject is related to another resource that appears in the object position.

Suppose that we have the following triple that states that Vitali Klitschko has the boxing category Heavyweight.

res:Vitali_Klitschko dbp:boxerCategory res:Heavyweight .

A property can be assigned type information that indicates the type of resource that should appear in the subject and object position when the resource appears as the property. Suppose that *dbp:boxerStyle* is a property with domain *dbp:Boxer* and range *dbp:BoxingCategory* . In RDF Schema notation this may be represented by the following two triples.

dbp:boxerCategory rdfs:domain dbp:Boxer .
dbp:boxerCategory rdfs:range dbp:BoxingCategory .

The URIs *rdfs:domain* and *rdfs:range* are special keywords from the RDF Schema vocabulary [9]. According to the specification, a triple of the form *p rdfs:domain t* indicates that the URI *t* is a type and resources that appear as *subject* of triples with property *p* have the type *t*. Similarly, a triple of the form *p rdfs:range t* indicates that the URI *t* is a type and resources that appear as the *object* of triples with property *p* have the type *t*.

Now consider that the type of *res:Vitali_Klitschko* is currently unknown, so assumed to be only the top type *rdfs:Resource* that is a special top type that can be assigned to all URIs. By RDF Schema inference described informally above we can deduce that, since *res:Vitali_Klitschko* appears as the subject of a triple with property *dbp:boxerCategory* and furthermore the domain of that property is *dbp:Boxer*, we can infer that *res:Vitali_Klitschko* is of type *dbp:Boxer*. In RDF this may be represented by the following inferred triple.

res:Vitali_Klitschko a dbp:Boxer .

The notation we use for representing RDF triples is inspired by the W3C standard notation Turtle [3]. In this notation the keyword 'a' is a synonym for the URI *rdf:type* from the RDF vocabulary [16] which is used to indicate that the resource that appears in the subject position is an instance of the type that appears in the object position.

Triples can be of the following forms, where the tokens ranging over *type* form a distinguished finite subset of the tokens ranging over *uri*.

- Simple triples consisting of three URIs: *uri uri uri*.
- Type declarations that indicate a type associated with a URI: *uri a type*.
- Subclass assertions relating two classes: *type rdfs:subClassOf type*.

- Assertions about the domain of URI naming a property: *uri rdfs:domain type*.
- Assertions about the range of URI naming a property: *uri rdfs:range type*.

Notice that to avoid circular definitions, we treat *rdfs:subClassOf* , *rdfs:domain* and *rdfs:range* like keywords. Only in the ontology for RDF Schema itself, do we need to make statements about these properties as if they are any other URI; hence the price paid is small.

2.1. W3C Recommended RDF Schema inference

In Figure 1, we present a version of the W3C recommended semantics for RDF Schema. Similarly to the work of Munoz et al. [37] on minimal deductive systems for RDF Schema, rules are presented as a deductive system. In such deductive systems, if all the triples that appear above the horizontal line hold, then the rules below the horizontal line hold. If there are no rules above the horizontal line, then an axiom is defined that always holds.

$$\frac{uri_1 \;\; rdfs{:}domain \;\; type \qquad uri_0 \;\; uri_1 \;\; uri_2}{uri_0 \;\; a \;\; type} \text{ (rdfs2)}$$

$$\frac{uri_1 \;\; rdfs{:}range \;\; type \qquad uri_0 \;\; uri_1 \;\; uri_2}{uri_2 \;\; a \;\; type} \text{ (rdfs3)}$$

$$\frac{}{uri \;\; a \;\; rdfs{:}Resource} \text{ (rdfs8)}$$

$$\frac{uri \;\; a \;\; type_0 \qquad type_0 \;\; rdfs{:}subClassOf \;\; type_1}{uri \;\; a \;\; type_1} \text{ (rdfs9)}$$

$$\frac{}{type \;\; rdfs{:}subClassOf \;\; type} \text{ (rdfs10)}$$

$$\frac{type_0 \;\; rdfs{:}subClassOf \;\; type_1 \qquad type_1 \;\; rdfs{:}subClassOf \;\; type_2}{type_0 \;\; rdfs{:}subClassOf \;\; type_2} \text{ (rdfs11)}$$

Figure 1: A deductive system for RDF Schema inference. Rule names correspond to the names of the equivalent patterns of entailment in the W3C recommendation [25].

There are two axioms in the deductive system defined in Fig. 1. The axiom *rdfs8* defines that all resources have the supertype *rdfs:Resource*. The axiom *rdfs10* states that all types are a subtype of themselves; thereby *rdfs:subClassOf* is a reflexive relation. The relation *rdfs:subClassOf* is also a transitive relation, as defined by the rule *rdfs11*.

The *rdfs9* rule in Fig. 1 performs subsumption, which allows a weaker type for a resource to be inferred, by using the *rdfs:subClassOf* relation. Subsumption is a common feature of subtype systems [11, 35, 2] — an observation key to the formal system developed in the body of this work. The remaining rules *rdfs2* and *rdfs3* formalise that if the given URI is used as a property in a triple then the object of the triple, respectively the subject of the triple, has the type indicated. See above for the examples regarding boxer categories.

Notice that rules involve two sorts of variables: classes that range over the variables *type*, $type_0$, $type_1$, $type_2$; and URIs that range over the variables *uri*, uri_0, uri_1. Classes form a subset of all URIs, thus a variable for a class may appear as a URI. This allows triples that describe classes to be expressed, such as the following triples.

dbp:Boxer prov:wasDerivedFrom mapping:Boxer .
dbp:Boxer a prov:Entity .

The above triples state that type information about the class Boxer was derived from a particular ontology wiki page, and is also of type *prov:Entity*. This metadata is taken from the W3C recommended provenance ontology [32] that is used to express the origin of resources, including classes.

By treating classes as an explicit finite subset of URIs, several rules become more concise than in the recommendation, and two rules called *rdfs4a* and *rdfs4b* become redundant. The remaining rules in the recommendation *rdfs5*, *rdfs6*, *rdfs7* concern a relation called *rdfs:subPropertyOf* , which can be added straightforwardly to this work, however contribute little to the type system. The final remaining rule *rdfs1* concerns literals, such as strings and integers which are treated thoroughly by the prescriptive type system in [14]. The prescriptive type system may co-exist with the descriptive type system in this work: the descriptive type system generating warnings; the prescriptive type system generating errors. Literals must be treated prescriptively, since for example adding an integer to a string can cause serious runtime errors.

Several papers already provide related formal models of RDF Schema addressing known issues with the W3C recommendations. The inference mechanisms are analysed from a model theoretic perspective by ter Horst [48]. The main result of the model theoretic analysis is that RDF Schema inference, as defined in the W3C recommendation is incomplete. The incompleteness is caused by an interaction between two features of RDF Schema – the property *rdfs:subPropertyOf* and blank nodes. Blank nodes are local identifiers that appear in RDF in place of URIs, but not in the property position in a triple. With a more complete reasoning mechanism blank nodes should be allowed to appear in the property position. In the latest W3C recommendation [25], the notion of a generalised triple is proposed that allows blank nodes to appear in the property position, addressing the concerns of ter Horst. In this work, we cover neither blank nodes nor *rdfs:subProperty*, since the associated problems are perpendicular to those in this work. For a discussion on blank nodes and their associated problems we refer to Mallea et al. [34].

Pan et al. [40] address issues with the higher-order nature of the official W3C recommendation for RDF Schema. A major point of contention is that the top level types are defined in a circular fashion. In particular, there is a type *rdfs:Class* which is a type of itself meaning making the system higher-order. Fortunately, there is little descriptive power to gain by treating *rdfs:Class* like any other URI, since its main use is to describe the RDF Schema vocabulary itself. For example, *rdfs:subClassOf* has domain *rdfs:Class* and range *rdfs:Class* – a fact that can be implicitly built into the deductive system itself by using a separate sort for classes, as in Fig. 1. In this work, *rdfs:Class* is built into the formal system implicitly, by maintaining a finite set of known URIs that can be used as classes. In this way, we are able to avoid the descriptive type system in this work becoming higher-order.

2.2. Extending inference to infer the RDF Schema itself.

In the body of this work, we formally define a type system that supports RDF Schema inference. RDF Schema inference is one of several other modes of inference enabled by our type system. We argue that firstly these new modes of inference are no less natural than the recommended RDF Schema inference; and, in common scenarios, an alternative mode of inference is more appropriate. Unlike OWL [27] we are not extending the vocabulary used to describe schema; we are simply providing an alternative method for using the existing vocabulary, that furthermore, remains sound with respect to the W3C recommendation.

Consider the following triple that states that Vitali Klitschko was born in the Kyrgyz SSR.

res:Vitali_Klitschko dbp:birthPlace res:Kyrgyz_SSR .

We know that Vitali Klitschko has the type *dbp:Boxer* . Furthermore, from the DBpedia ontology, we obtain the following triples.

dbp:birthPlace rdfs:domain dbp:Person .
dbp:birthPlace rdfs:range dbp:Place .

From the above RDF Schema information, the standard behaviour is to apply RDF Schema inference, whence we infer that Klitschko has the types *dbp:Boxer* and *dbp:Person*.

In this scenario, there is an alternative stronger mode of inference. Consider the following subclass assumption.

dbp:Boxer rdfs:subClassOf dbp:Person .

If we add the above assumption to our dataset, then every boxer can be used as a person. Therefore, by the *rdfs9* rule in Fig. 1, we can infer that Klitschko is person, without using the *rdfs2* rule from Fig. 1. Thus, no new type information regarding Klitschko need be added to the dataset.

In the example, more general schema information has been extracted that can be applied to other resources of the same type. So, for example, suppose that the following triples are retrieved about Wladimir the brother of Vitali.

res:Wladimir_Klitschko a dbp:Boxer .
dbp:Kazakh_SSR a dbp:Place .
res:Wladimir_Klitschko dbp:birthPlace dbp:Kazakh_SSR .

Since Wladimir is a boxer and all boxers are people, no further type information need to be added to the dataset. The type system developed in this work identifies such well-typed systems where no further inference is required.

Now suppose that by using freebase [8] we obtain the following triple, where *free:m/0j1b9hc* is name in freebase for the political party Ukrainian Democratic Alliance for Reform.

res:Vitali_Klitschko free:government/politician/party free:m/0j1b9hc .

Suppose furthermore that, from the freebase ontology, we obtain the following triples indicating the usage of the above property.

free:government/politician/party rdfs:domain free:government/politician .
free:government/politician/party rdfs:range free:government/political_party .

Now, since Klitschko is a boxer and appears as the subject of a triple with property *free:government/politician/party*, we can apply one of the two modes of inference described.

- Add the triple *dbp:boxer rdfs:subClassOf free:government/politician*.
- Infer that Vitali Klitschko has types *free:government/politician* and *dbp:boxer*.

By adding the above subclass assumption to our dataset, every boxer would also be a politician. Clearly, this is not what is intended since most boxers have no political ambitions. In this scenario, what is intended is clearly the second case above.

A question that a machine cannot answer easily is which option is best. In both the examples in this section, to a human, it is immediately clear that all boxers are people but not all boxers are politicians. The descriptive type system formalised and explained in the rest of this paper provides a mechanism for involving humans in the decision process. Whenever inference may be performed, the required inference is presented to the user of a program as a warning. The warning presents the option of applying standard RDF Schema inference or possibly a stronger inference mode as described in this section. Furthermore, since the message is a warning about a type violation, rather than an error, the user may choose to ignore the warnings and only resolve warnings that they believe are necessary. Thereby, we introduce the basis of a tool that assists rather than restricts programmers.

3. Types and Subtyping for the Descriptive Type System

In this section, we introduce the types that are used in our type system. We explain the intuition behind each construct and how they are useful for describing resources. We also define how types are arranged into a hierarchy by using a subtype system.

3.1. Types for Classifying Resources

Many type systems are intimately connected to the form of data. For example, in XML Schema, the lexeme 3 has the type *xsd:integer*, whereas the lexeme `"Ershov"` has the type *xsd:string*. RDF does allow XML Schema datatypes [43] to appear as objects in triples. Such literals should be typed prescriptively, since it should be forbidden to add a string to an integer or evaluate an integer using a regular expression.

Now consider the types of resources. Resources in RDF are represented by a URI that identifies the resource. The simplest answer is to say that the type of a resource is *xsd:anyURI* , in which case a prescriptive type system is sufficient, as defined in [14].

In contrast, this work concerns types that classify resources represented as URIs. Using RDF types, one resource can be classified using the type *dbp:IceHockeyPlayer* and another using the type *yago:SovietComputerScientists*. Both resources are represented as URIs, so there is nothing about the syntax of the resources that distinguishes their type. Classes themselves are a distinguished finite set of URIs that form the atomic types of our type system. The full syntax for types is presented in Fig. 2.

Type ::=	*class*	*atomic type*
	rdfs:Resource	*top type*
	`IntersectionOf`(*Type*, *Type*)	*intersection type*
	`UnionOf`(*Type*, *Type*)	*union type*
	`Property`(*Type*, *Type*)	*property type*

Figure 2: The syntax of descriptive types. Variables C, D, E are used to range over types.

In Fig. 2, there are three type constructors, namely intersection, union, and property types. There is also a top type *rdfs:Resource* that represents the type of all resources. This type for all resources includes classes; hence all classes are also resources as in the W3C recommendation [25].

Intersection types. The intersection type constructor is used to combine several types. For example, according to DBpedia, *res:Andrey_Ershov* has several types, including *yago:SovietComputerScientists* and *yago:FellowsOfTheBritishComputerSociety* . In this case, the following intersection type can be assigned to *res:Andrey_Ershov*.

`IntersectionOf` (*yago:SovietComputerScientists* ,
yago:FellowsOfTheBritishComputerSociety)

Intuitively, the resource *res:Andrey_Ershov* is a member of the intersection of the set of all resources that have the type *yago:SovietComputerScientists* and the set of all resources of type *yago:FellowsOfTheBritishComputerSociety*.

Note that intersections and unions form part of the OWL [27] recommendation. The semantics for intersections and unions in OWL are the same as the semantics for intersection and union types, in the sense that they are the greatest lower bounds and least upper bounds respectively in a preorder.

Property types. The property type is inspired by the *rdfs:domain* and *rdfs:range* properties in RDF Schema [9], which declare the type of data that may appear in the respective subject and object position of a triple. In RDF [16], the basic unit of data is a triple, such as:

res:Vitali_Klitschko dbp:birthPlace res:Kyrgyz_SSR .

The elements of a triple are the subject, property and object respectively. Here the subject is expected to be of type *dbp:Person*, the object is expected to be of type *dbp:Settlement*, while the property *dbp:birthPlace* is assigned the following type.

`Property`(*dbp:Person*, *dbp:Settlement*)

In RDF Schema, this type represents two statements: one that declares that the domain of the property is *dbp:Person* ; and another that declares that the range of the property is *dbp:Settlement*.

Union types. If we inspect the data in DBpedia, we discover that the following triple also appears.

res:Vitali_Klitschko dbp:birthPlace res:SovietUnion .

Observe that *res:SovietUnion* is not a settlement. We can use the union type to refine the above type so that the range of the property is either a settlement or a country. The refined type for *dbp:birthPlace*, involving union, is as follows.

$$\texttt{Property}(\textit{dbp:Person},\ \texttt{UnionOf}(\textit{dbp:Settlement},\ \textit{dbp:Country}))$$

Notice that intersection would not be appropriate above. If we replace `UnionOf` with `IntersectionOf` in the above example, the range of the property is restricted to resources that are both a settlement and a country (e.g. Singapore), which is not the intended semantics of *dbp:birthPlace*. We return to this common modelling slip when we discuss properties with multiple domains and ranges, towards the end of the next section.

Top type. Intuitively the top type ranges over the set of all resources. If a resource has no descriptive type information, then it can be assigned the top type. The resource *yago:Random_access_machine* in the Yago dataset [47] has no type other than *rdfs:Resource*.

In Yago the following triple relates Ershov to the random access machine.

$$\textit{yago:Andrei\_Ershov}\ \ \textit{yago:linksTo}\ \ \textit{yago:Random\_access\_machine}\ .$$

The property *yago:linksTo* is very general, relating any resource to any resource, as indicated by the property type $\texttt{Property}(\textit{rdfs:Resource},\ \textit{rdfs:Resource})$.

Notice that the syntax of types is liberally expressive. We can express types that are both resources and property types, allowing multiple uses of one URI. This design decision accommodates the subjective nature of human knowledge and data representation, without the system becoming higher order. A descriptive type system is expected to evolve, hence we do not want to restrict unforeseeable developments in its evolution.

3.2. A Subtype Relation over Descriptive Types

Types form a lattice defined by a subtype relation. The subtype relation, defined in Fig. 3, determines when a resource of one type can be used as a resource of another type. In subsequent sections, this relation is important for both the type system for data and scripts that is introduced, and also for refining the type system itself at runtime in response to warnings.

$$\frac{}{\vdash C \leq \textit{rdfs:Resource}}\ \textit{top} \qquad \frac{\textit{class}_0 \leq \textit{class}_1 \in \mathrm{SC}^*}{\vdash \textit{class}_0 \leq \textit{class}_1}\ \textit{subclass}$$

$$\frac{\vdash C \leq D}{\vdash C \leq \texttt{UnionOf}(D, E)}\ \textit{left injection} \qquad \frac{\vdash C \leq E}{\vdash \texttt{IntersectionOf}(C, D) \leq E}\ \textit{left projection}$$

$$\frac{\vdash C \leq E}{\vdash C \leq \texttt{UnionOf}(C, E)}\ \textit{right injection} \qquad \frac{\vdash D \leq E}{\vdash \texttt{IntersectionOf}(C, D) \leq E}\ \textit{right projection}$$

$$\frac{\vdash C \leq E \qquad \vdash D \leq E}{\vdash \texttt{UnionOf}(C, D) \leq E}\ \textit{least upper bound} \qquad \frac{\vdash C \leq D \qquad \vdash C \leq E}{\vdash C \leq \texttt{IntersectionOf}(D, E)}\ \textit{greatest lower bound}$$

$$\frac{\vdash C_0 \leq C_1 \qquad \vdash D_0 \leq D_1}{\vdash \texttt{Property}(C_1, D_1) \leq \texttt{Property}(C_0, D_0)}\ \textit{property}$$

Figure 3: Axioms and rules of the subtype system.

Axioms. We assume that there are a number of subtype inequalities that relate atomic types to each other. For example, we may assume that the following subtype inequalities hold.

$$\textit{dbp:Settlement} \leq \textit{dbp:PopulatedPlace} \qquad \textit{dbp:Country} \leq \textit{dbp:PopulatedPlace}$$

$$\textit{yago:CitiesAndTownsInMoscowOblast} \leq \textit{dbp:Settlement}$$

Clearly a settlement is a populated places, as is a country. Cities and towns in Moscow oblast are also populated places, but are more specifically settlements.

These inequalities are inspired by the *rdfs:subClassOf* property from RDF Schema, which defines a reflexive transitive relation. We call a subclass relation SC the set of *subtype assumptions*. We denote the reflexive transitive closure of SC as SC*. Notice that SC is a relation over a finite number of atomic types, hence SC* can be calculated in cubic time [30]. The relation SC* is used in the *subclass* rule in Fig. 3.

The *top* axiom states that every resource is of type *rdfs:Resource*.

Rules for union, intersection and properties. Suppose that a hint leads to the type of the property *dbp:birthPlace* to be refined further. The hint suggests that the range of the property should include *dbp:PopulatedPlace*. From the subtype rules, we can derive the following inequality.

$$\vdash \texttt{Property}(\ \textit{dbp:Person}\ ,\ \textit{dbp:PopulatedPlace}\ ) \leq \texttt{Property}(\ \textit{dbp:Person}\ ,\ \texttt{UnionOf}(\ \textit{dbp:Settlement}\ \ \textit{dbp:Country}\ )\ )$$

The derivation of the above subtype relation follows from applying first the *property* rule, that swaps the direction of subtyping in each component, generating the following subtype constraint, and an axiom.

$$\vdash \texttt{UnionOf}(\textit{dbp:Settlement},\ \textit{dbp:Country}) \leq \textit{dbp:PopulatedPlace}$$

The above subtype inequality between properties suggests that a property with range *dbp:PopulatedPlace* can also range over resources that are settlements or countries.

The above constraint is solved by applying the *least upper bound* rule. The least upper bound rule generates two inequalities between classes that were declared to be in SC* above, hence the following hold by the *subclass* rule.

$$\textit{dbp:Settlement} \leq \textit{dbp:PopulatedPlace} \quad \text{and} \quad \textit{dbp:Country} \leq \textit{dbp:PopulatedPlace}$$

Now suppose that the following inequality is added to the relation SC.

$$\textit{yago:SovietComputerScientists} \leq \textit{dbp:Person}$$

By the *left projection* rule and the above subtype inequality, we can derive the following subtype inequality.

$$\vdash \texttt{IntersectionOf}(\ \textit{yago:SovietComputerScientists}\ ,\ \textit{yago:FellowsOfTheBritishComputerSociety}\ ) \leq \textit{dbp:Person}$$

The above inequality suggests that Ershov can be treated as a person, although his type does not explicitly mention the class *dbp:Person*.

The cut rule. A subtype relation is expected to be a transitive relation. To prove that subtyping is transitive, we assume that there is a rule called the *cut* rule in our subtype system and then show that any subtype inequality that is derived using cut can also be derived without using cut. The cut rules is defined as follows.

$$\frac{\vdash C \leq D \quad \vdash D \leq E}{\vdash C \leq E}\ \textit{cut}$$

In proof theory, the result that show that the cut rule is redundant is known as *cut elimination*, and is central to establishing desirable properties of systems including consistency.

Theorem 1 (Cut elimination)**.** *The cut rule can be eliminated from the subtype system in Fig. 3 using an algorithmic procedure.*

Proof. The proof works by transforming the derivation tree for a subtype judgement into another derivation tree with the same conclusion. The transformation is indicated by $[\![\cdot]\!]$. The symbol π_i above a subtype inequality represents a proof tree with the subtype inequality as its conclusion.

Without loss of generality, assume that the rule applied at the base of the proof tree is the cut rule. The proof proceeds by induction over the structure of the proof tree.

Consider the case of cut applied across two *subclass* rules. Since SC* is transitively closed, if $a \leq b \in \mathrm{SC}^*$ and $b \leq c \in \mathrm{SC}^*$, then we know that $a \leq c \in \mathrm{SC}^*$. Hence the following transformation simplifies *subclass* rules.

$$\left[\!\!\left[\dfrac{\dfrac{a \leq b \in \mathrm{SC}^*}{\vdash a \leq b} \quad \dfrac{b \leq c \in \mathrm{SC}^*}{\vdash b \leq c}}{\vdash a \leq c} \right]\!\!\right] \longrightarrow \dfrac{a \leq c \in \mathrm{SC}^*}{\vdash a \leq c}$$

The above case is a base case for the induction. The other base case is when the top rule is applied on the right branch of the cut rule. In this case, the cut can be absorbed by the top type axiom, as follows.

$$\left[\!\!\left[\dfrac{\begin{array}{c}\pi \\ \vdash C \leq D\end{array} \quad \dfrac{}{\vdash D \leq \mathit{rdfs{:}Resource}}}{\vdash C \leq \mathit{rdfs{:}Resource}} \right]\!\!\right] \longrightarrow \dfrac{}{\vdash C \leq \mathit{rdfs{:}Resource}}$$

The result of the above transformation step is clearly cut free.

Consider the case where the left branch of a cut is another *cut* rule. The nested cut rule can be normalised first, as demonstrated by the transformation bellow.

$$\left[\!\!\left[\dfrac{\dfrac{\begin{array}{c}\pi_0 \\ \vdash C \leq D\end{array} \quad \begin{array}{c}\pi_1 \\ \vdash D \leq E\end{array}}{\vdash C \leq E} \quad \begin{array}{c}\pi_2 \\ \vdash E \leq F\end{array}}{\vdash C \leq F} \right]\!\!\right] \longrightarrow \left[\!\!\left[\dfrac{\left[\!\!\left[\dfrac{\begin{array}{c}\pi_0 \\ \vdash C \leq D\end{array} \quad \begin{array}{c}\pi_1 \\ \vdash D \leq E\end{array}}{\vdash C \leq E} \right]\!\!\right] \quad \begin{array}{c}\pi_2 \\ \vdash E \leq F\end{array}}{\vdash C \leq F} \right]\!\!\right]$$

By induction, the resulting nested tree is transformed into a cut free derivation tree; hence another case applies. This induction step is symmetric when a nested cut appears on the right branch of a cut.

Consider the case where the *least upper bound* rule appears on the left branch of a cut. In this case, the transformation can be applied separately to each of the premises of the union introduction rule, as demonstrated below.

$$\left[\!\!\left[\dfrac{\dfrac{\dfrac{\begin{array}{c}\pi_0 \\ \vdash C_0 \leq D\end{array} \quad \begin{array}{c}\pi_1 \\ \vdash C_1 \leq D\end{array}}{\vdash \mathtt{UnionOf}(C_0, C_1) \leq D} \quad \begin{array}{c}\pi_2 \\ \vdash D \leq E\end{array}}{\vdash \mathtt{UnionOf}(C_0, C_1) \leq E} \right]\!\!\right] \longrightarrow \left[\!\!\left[\dfrac{\begin{array}{c}\pi_0 \\ \vdash C_0 \leq D\end{array} \quad \begin{array}{c}\pi_2 \\ \vdash D \leq E\end{array}}{\vdash C_0 \leq E} \right]\!\!\right] \quad \left[\!\!\left[\dfrac{\begin{array}{c}\pi_1 \\ \vdash C_1 \leq D\end{array} \quad \begin{array}{c}\pi_2 \\ \vdash D \leq E\end{array}}{\vdash C_1 \leq E} \right]\!\!\right]$$
$$\vdash \mathtt{UnionOf}(C_0, C_1) \leq E$$

By induction, the result of the transformation is a cut free proof. The case for the *greatest lower bound* is symmetric, with the order of subtyping exchanged and union exchanged for intersection.

Consider the case of the injection rules. Without loss of generality, consider the *left injection* rule. In this case, the cut is pushed up the proof tree, as demonstrated below.

$$\left[\!\!\left[\dfrac{\begin{array}{c}\pi_0 \\ \vdash C \leq D\end{array} \quad \dfrac{\begin{array}{c}\pi_1 \\ \vdash D \leq E_0\end{array}}{\vdash D \leq \mathtt{UnionOf}(E_0, E_1)}}{\vdash C \leq \mathtt{UnionOf}(E_0, E_1)} \right]\!\!\right] \longrightarrow \left[\!\!\left[\dfrac{\begin{array}{c}\pi_0 \\ \vdash C \leq D\end{array} \quad \begin{array}{c}\pi_1 \\ \vdash D \leq E_0\end{array}}{\vdash C \leq E_0} \right]\!\!\right]$$
$$\vdash C \leq \mathtt{UnionOf}(E_0, E_1)$$

By induction, the result is a cut free proof. The cases for *right injection*, *left projection* and *right projection* are similar.

Consider when the *injection* rule is applied on the left of a cut, and *least upper bound* rule is applied on the right of a cut. This is a *principal case* of the cut elimination procedure. Without loss of generality, consider the left projection.

The result of the transformation is that only the left premise of the union introduction rule is required; the irrelevant branch is removed by the elimination step, as demonstrated below.

$$\left[\!\!\left[\dfrac{\dfrac{\begin{matrix}\pi_0\\ \vdash C \leq D_0\end{matrix}}{\vdash C \leq \texttt{UnionOf}(D_0, D_1)} \quad \dfrac{\begin{matrix}\pi_1 & \pi_2\\ \vdash D_0 \leq E & \vdash D_1 \leq E\end{matrix}}{\vdash \texttt{UnionOf}(D_0, D_1) \leq E}}{\vdash C \leq E}\right]\!\!\right] \longrightarrow \left[\!\!\left[\dfrac{\begin{matrix}\pi_0 & \pi_1\\ \vdash C \leq D_0 & \vdash D_0 \leq E\end{matrix}}{\vdash C \leq E}\right]\!\!\right]$$

By induction, the result of the transformation is a cut-free proof. The principal case for intersection is similar to union.

Consider the case of cut applied to two predicate subtype rules. In this case, the contravariant premises of each subtype rule are cut individually, as follows.

$$\left[\!\!\left[\dfrac{\dfrac{\begin{matrix}\pi_0 & \pi_0'\\ \vdash D_0 \leq C_0 & \vdash D_1 \leq C_1\end{matrix}}{\vdash \texttt{Property}(C_0, C_1) \leq \texttt{Property}(D_0, D_1)} \quad \dfrac{\begin{matrix}\pi_1 & \pi_1'\\ \vdash E_0 \leq D_0 & \vdash E_1 \leq D_1\end{matrix}}{\vdash \texttt{Property}(D_0, D_1) \leq \texttt{Property}(E_0, E_1)}}{\vdash \texttt{Property}(C_0, C_1) \leq \texttt{Property}(E_0, E_1)}\right]\!\!\right]$$

$$\longrightarrow \dfrac{\left[\!\!\left[\dfrac{\begin{matrix}\pi_1 & \pi_0\\ \vdash E_0 \leq D_0 & \vdash D_0 \leq C_0\end{matrix}}{\vdash E_0 \leq C_0}\right]\!\!\right] \left[\!\!\left[\dfrac{\begin{matrix}\pi_0' & \pi_1'\\ \vdash E_1 \leq D_1 & \vdash D_1 \leq C_1\end{matrix}}{\vdash E_1 \leq C_1}\right]\!\!\right]}{\vdash \texttt{Property}(C_0, C_1) \leq \texttt{Property}(E_0, E_1)}$$

By induction, each of the new transformations on the right above have a cut-free proof, so the result of original transformation on the left above has a cut-free proof.

For every cut one of the above cases applies. Furthermore, in each transformation a finite number of proof trees are considered after a transformation step, each of which has a smaller depth than the original proof tree; hence by a standard multiset ordering argument [18] it is easy to see that the procedure terminates. Therefore, by structural induction on the derivation tree, a cut free derivation tree with the same conclusion can be constructed for any derivation. □

Notice that the base case in the proof above would not be possible without taking the transitive closure of the subclass relation. Pre-computing the transitive closure is equivalent to applying in advance the cut rule exhaustively over atomic types, which works since there are finitely many atomic classes at any time and the resulting relation is finite.

Cut elimination proves that the subtype system is transitive. It is straightforward to prove that the subtype system is reflexive, by structural induction. Also, the direction of subtyping is preserved (monotonicity) by conjunction and disjunction, while the direction of subtyping is reversed (antitonicity) for property types. Monotonicity and antitonicity can be established by a direct proof.

Proposition 2. *For any type* $\vdash C \leq C$ *is derivable. Also, if* $\vdash C_0 \leq D_0$ *and* $\vdash C_1 \leq D_1$ *then the following hold:*

- $\vdash \texttt{IntersectionOf}(C_0, C_1) \leq \texttt{IntersectionOf}(D_0, D_1)$ *is derivable.*
- $\vdash \texttt{UnionOf}(C_0, C_1) \leq \texttt{UnionOf}(D_0, D_1)$ *is derivable.*
- $\vdash \texttt{Property}(D_0, D_1) \leq \texttt{Property}(C_0, C_1)$ *is derivable.*

Proof. Reflexivity of subtyping follows by induction on the structure of types. The base cases are as follows.

There are two base cases. Consider the case for class types a. Since SC^* is reflexively closed, $a \leq a \in \mathrm{SC}^*$; hence by the *subclass* rule $\vdash a \leq a$ holds. Consider the case for *rdfs:Resource*. By the *top* rule, $\vdash$ *rdfs:Resource* $\leq$ *rdfs:Resource* as required.

The remaining cases follow by induction. Assume that $\vdash C \leq C$ and $\vdash D \leq D$, then by the *property* rule, $\vdash \mathit{range}(C)\, D$ holds. Also by the *left projection*, *right projection* and *greatest lower bound* rules, we can derive the following.

$$\dfrac{\dfrac{\vdash C \leq C}{\texttt{IntersectionOf}(C, D) \leq C} \quad \dfrac{\vdash D \leq D}{\vdash \texttt{IntersectionOf}(C, D) \leq D}}{\vdash \texttt{IntersectionOf}(C, D) \leq \texttt{IntersectionOf}(C, D)}$$

A symmetric argument holds for union types and similar argument or property types. Therefore, by induction on the structure of the type, the subtype relation is reflexive.

The antitonicity of property types and monotonicity of intersection and union types follow directly from the rules for each type constructor. □

Theorem 1 and Proposition 2 are sufficient to establish the consistency of the subtype system, i.e. subtyping is a well defined preorder.

Corollary 3. *The relation defined by subtyping is a preorder over types.*

Our subtype system is closely related to the functional programming language with intersection and union types presented by Barbanera et al. [2]. Our subtype system without properties coincides with the subtype system of Barbanera et al. without implication. Implication corresponds to fully fledged function types, which, of course, is much more expressive than our property types. Properties can be encoded using implication, so our system is a restriction of the system presented in [2]. However, the system of Barbanera et al. types all terminating functions; hence is undecidable, and therefore inappropriate to use directly.

4. An Algorithmic Type System for Scripts and Data

We introduce a simple scripting language for interacting with Linked Data. The language enables resources to be dereferenced and for data to be queried. This language is a restriction of the scripting language presented in [14]; which is based on process calculi presented in [19, 28]. We keep the language here simple to maintain the focus on descriptive types.

4.1. The Syntax of a Simple Scripting Language for Linked Data

The syntax of scripts is presented in Fig. 4. Terms in the language are *URIs* which are identifiers for resources, or *variables* of the form `$x`. RDF triples [16] and triple patterns are represented as three terms separated by spaces. The `where` keyword prefixes a triple pattern. The keyword `ok`, representing a successfully terminated script, is dropped in examples. Data is simply one or more triples, representing an RDF graph.

term ::=	*variable* | *uri*	*data* ::=	*term term term*
		|	*term a term*
script ::=	`ok`	|	*class rdfs:subClassOf class*
|	`where` *term term term script*	|	*term rdfs:domain class*
|	`from` *term script*	|	*term rdfs:range class*
|	`select` *variable* : *type script*	|	*data data*

Figure 4: The syntax of scripts and data.

The keyword `from` represents dereferencing the given resource. The HTTP protocol is used to obtain some data from the URI, and the data obtained is loaded into a graph local to the script (see [14] for an extensive discussion of the related `from named` construct).

The keyword `where` represents executing a query over the local graph that was populated by dereferencing resources. The query can execute only if the data in the local graph matches the pattern. Variables representing resources to be discovered by the query are bound using the `select` keyword (see [28] for the analysis of more expressive query languages based on SPARQL [24, 42]).

4.2. An Algorithmic Type System for Scripts and Data

We type scripts for two purposes. Firstly, if the script is correctly typed, the script is consistent with the schema information currently known by the type system. Therefore, when the script is executed, no warning will be thrown, where a warning indicates that there is an inconsistency between the current schema information and the data or script hence some form of inference should be applied. Secondly, if the script is not well typed, we can use the type system as the basis of an algorithm for generating the warnings and inferences themselves. Scripts and data are typed using the system presented in Fig. 5. There are typing rules for each form of term, script and data.

$$\frac{\vdash C \leq D}{\text{Env}, \texttt{\$x}\colon C \vdash \texttt{\$x}\colon D}\ \textit{variable} \qquad \frac{\vdash \text{Ty}(\textit{uri}) \leq C}{\text{Env} \vdash \textit{uri}\colon C}\ \textit{resource} \qquad \frac{}{\text{Env} \vdash \texttt{ok}}\ \textit{success}$$

$$\frac{\text{Env}, \texttt{\$x}\colon C \vdash \textit{script}}{\text{Env} \vdash \texttt{select \$x}\colon C\ \textit{script}}\ \textit{select} \qquad \frac{\text{Env} \vdash \textit{script}}{\text{Env} \vdash \texttt{from}\ \textit{uri}\ \textit{script}}\ \textit{from}$$

$$\frac{\text{Env} \vdash \textit{term}_0\colon C \quad \text{Env} \vdash \textit{term}_1\colon \texttt{Property}(C, D) \quad \text{Env} \vdash \textit{term}_2\colon D \quad \text{Env} \vdash \textit{script}}{\text{Env} \vdash \texttt{where}\ \textit{term}_0\ \textit{term}_1\ \textit{term}_2\ \textit{script}}\ \textit{where}$$

$$\frac{\text{Env} \vdash \textit{term}_0\colon C \quad \text{Env} \vdash \textit{term}_1\colon \texttt{Property}(C, D) \quad \text{Env} \vdash \textit{term}_2\colon D}{\text{Env} \vdash \textit{term}_0\ \textit{term}_1\ \textit{term}_2}\ \textit{triple}$$

$$\frac{\text{Env} \vdash \textit{data}_0 \quad \text{Env} \vdash \textit{data}_1}{\text{Env} \vdash \textit{data}_0\ \textit{data}_1}\ \textit{compose} \qquad \frac{\text{Env} \vdash \textit{term}\colon \textit{class}}{\text{Env} \vdash \textit{term}\ \ a\ \ \textit{class}}\ \textit{ascription}$$

$$\frac{\text{Env} \vdash \textit{class}_0 \leq \textit{class}_1}{\text{Env} \vdash \textit{class}_0\ \ \textit{rdfs:subClassOf}\ \ \textit{class}_1}\ \textit{subclass}$$

$$\frac{\text{Env} \vdash \textit{term}\colon \texttt{Property}(\textit{class}_0, \textit{class}_1)}{\text{Env} \vdash \textit{term}\ \ \textit{rdfs:domain}\ \ \textit{class}_0}\ \textit{domain} \qquad \frac{\text{Env} \vdash \textit{term}\colon \texttt{Property}(\textit{class}_0, \textit{class}_1)}{\text{Env} \vdash \textit{term}\ \ \textit{rdfs:range}\ \ \textit{class}_1}\ \textit{range}$$

Figure 5: The type system for scripts and data.

Typing data. To type resources we require a partial function Ty from resources to types. This represents the current type of resources assumed by the system. We write Ty(*uri*) for the current type of the resource, and call Ty the *type assumptions*. The type rule for resources states that a resource can assume any supertype of its current type. For example Ershov and Klitschko are people even though their type is the intersection of several professions.

The type rule for triples states that a triple is well typed as long as the subject and object of a triple can match the type of the property type assumed by the predicate. Well typed triples are then composed together.

Triples with the reserved keywords in the property position are typed differently from other triples. In the *ascription* rule, the object is an atomic type and the subject is a term of the given atomic type. This rule is used to extract type information from data during inference, and can be viewed as a form of *type ascription* or *casting* [44]. Recall that a type ascription indicates that we expect a resource to be a particular type, when it could possibly take on several other types.

The type rule for triples where *rdfs:subClassOf* appears in the predicate position forces the corresponding assertion to appear in the subclass relation. The type rules for triples with *rdfs:domain* and *rdfs:range* in the property position, force the domain and range of a property respectively to permit the type indicated. Notice that for the domain rule the corresponding range is not specified, and similarly for the range rule. This leaves considerable flexibility in determining the type of a property. Further rules could be added to extract more refined types based on OWL [27] including rules covering intersection and union types.

Typing scripts. Variables may appear in scripts. The type rule for variables is similar to the type rule for resources, except that the type of a variable is drawn from the *type environment*, which appears on the left of the turnstile in a judgement. A type environment consists of a set of type assignments of the form $\texttt{\$x}\colon C$. As standard in type systems, a variable is assigned a unique type in a type environment. Type assumptions are introduced in the type environment using the type rule for `select`.

The rule for `where` is similar to the type rule for triples, except that there is a continuation script. A script prefixed with `from` is always well typed as long as the continuation script is well typed, since we work only with dereferenceable resources. A prescriptive type system involving data, such as numbers which cannot be dereferenced as in [14], takes more care at this point. The terminated script is always well typed.

The subsumption rule. Derivation trees in an algorithmic type system are linear in the size of the syntax. Suppose that we included the following *subsumption* rule that relaxes the type of a term at any point in a typing derivation, thereby explicitly applying subtyping at any point.

$$\frac{\text{Env} \vdash \mathit{term}\colon C \quad \vdash C \leq D}{\text{Env} \vdash \mathit{term}\colon D}\ \mathit{subsumption}$$

The type system extended with the subsumption rule, gives rise to type derivations of an unbounded size. By showing that subsumption is not necessary hence can be eliminated from any type derivation, as with the cut rule in the previous section, we establish that the type system is *algorithmic* [44].

Proposition 4 (Algorithmic typing)**.** *For any type assumption that can be derived using the type system in Fig. 5 plus the subsumption rule, we can construct a type derivation with the same conclusion where the subsumption rule has been eliminated from the derivation tree.*

Proof. There are three similar cases to consider, namely when a subsumption immediately follows: another subsumption rule; or a type rule for resources, or a type rule for variables. In each case notice that, by Theorem 1, if $\vdash C \leq D$ and $\vdash D \leq E$, then we can construct a cut-free derivation for $\vdash C \leq E$. Hence, in each of the following, the type derivation of the left can be transformed into the type derivation on the right.

1. $$\dfrac{\dfrac{\text{Env} \vdash \mathit{term}\colon C \quad \vdash C \leq D}{\text{Env} \vdash \mathit{term}\colon D} \quad \vdash D \leq E}{\text{Env} \vdash \mathit{term}\colon E} \qquad \text{yields} \qquad \dfrac{\text{Env} \vdash \mathit{term}\colon C \quad \vdash C \leq E}{\text{Env} \vdash \mathit{term}\colon E}$$

2. $$\dfrac{\dfrac{\vdash \text{Ty}(\mathit{uri}) \leq D}{\text{Env} \vdash \mathit{uri}\colon D} \quad \vdash D \leq E}{\text{Env} \vdash \mathit{uri}\colon E}\ \text{where Ty}(\mathit{uri}) = C \qquad \text{yields} \qquad \dfrac{\vdash \text{Ty}(\mathit{uri}) \leq E}{\text{Env} \vdash \mathit{term}\colon E}$$

3. $$\dfrac{\dfrac{\vdash C \leq D}{\text{Env}, \texttt{\$x}\colon C \vdash \texttt{\$x}\colon D} \quad \vdash D \leq E}{\text{Env}, \texttt{\$x}\colon C \vdash \texttt{\$x}\colon E} \qquad \text{yields} \qquad \dfrac{\vdash C \leq E}{\text{Env}, \texttt{\$x}\colon C \vdash \texttt{\$x}\colon E}$$

For other type rules subsumption cannot be applied, so the induction step follows immediately. Hence, by induction on the structure of a type derivation, all occurrences of the subsumption rule can be eliminated. □

Since the type system is algorithmic, we can use it efficiently as the basis for inference algorithms that we will employ in Section 5.

Monotonicity. We define an ordering over type assumptions and subtype assumptions. This ordering allows us to *refine* our type system by enlarging the subtype assumptions; by enlarging the domain of the type assumptions; and by tightening the types of resources with respect to the subtype relation. Refinement can be formalised as follows.

Definition 5. *When we would like to be explicit about the subtype assumptions SC and type assumptions Ty used in a type judgement Env ⊢ script and subtype judgement ⊢ C ≤ D, we use the following notation:*

$$\mathit{Env} \vdash^{\mathit{Ty}}_{\mathit{SC}} \mathit{script} \qquad\qquad \vdash_{\mathit{SC}} C \leq D$$

We define a refinement relation $(\mathit{Ty}', \mathit{SC}') \leq (\mathit{Ty}, \mathit{SC})$*, such that:*

1. $SC \subseteq SC'$.

2. *For all uri such that* $Ty(uri) = D$, *there is some* C *such that* $Ty'(uri) = C$ *and* $\vdash_{SC'} C \leq D$.

We say that (Ty', SC') *is a refinement of* (Ty, SC).

In a descriptive type system, we give the option to refine the type system in response to warnings that appear. The following two lemmas are steps towards establishing soundness of the type system in the presence of refinements of subtype and type assumptions. These lemmas establish that anything that is well typed remains well typed in a refined type system.

Lemma 6. *If* $\vdash_{SC} C \leq D$ *and* $SC \subseteq SC'$, *then* $\vdash_{SC'} C \leq D$.

Proof. Observe that only the atom rule uses SC. Also notice that if $a \leq b \in \mathrm{SC}^*$, and $\mathrm{SC} \subseteq \mathrm{SC}'$, then $a \leq b \in \mathrm{SC}'^*$. Hence if the subtype axiom on the left below holds, then the subtype axiom on the right below holds.

$$\frac{a \leq b \in \mathrm{SC}^*}{\vdash_{\mathrm{SC}} a \leq b} \qquad \text{yields} \qquad \frac{a \leq b \in \mathrm{SC}'^*}{\vdash_{\mathrm{SC}'} a \leq b}$$

All other cases do not involve SC, hence the induction steps are immediate. Hence, by structural induction, the set of subtype assumptions can be enlarged while preserving the subtype judgements. □

Lemma 7. *The following monotonicity properties hold for scripts and data respectively.*

1. *If* $\vdash^{Ty}_{SC}$ *script and* $(Ty', SC') \leq (Ty, SC)$, *then* $\vdash^{Ty'}_{SC'}$ *script.*

2. *If* $\vdash^{Ty}_{SC}$ *data and* $(Ty', SC') \leq (Ty, SC)$, *then* $\vdash^{Ty'}_{SC'}$ *data.*

Proof. For type assumptions, observe that the only rule involving Ty is the rule for typing resources. Assume that $(\mathrm{Ty}', \mathrm{SC}') \leq (\mathrm{Ty}, \mathrm{SC})$. By definition, if $\mathrm{Ty}(uri) = D$ then $\mathrm{Ty}'(uri) = C$ and $\vdash_{\mathrm{SC}'} C \leq D$ where $\mathrm{SC} \subseteq \mathrm{SC}'$. Hence if $\vdash_{\mathrm{SC}} D \leq E$, by Lemma 6, $\vdash_{\mathrm{SC}'} D \leq E$. Hence, by Theorem 1, we can construct a cut free proof of $\vdash_{\mathrm{SC}'} C \leq E$. Therefore if the type axiom on the left below holds, then the type axiom on the right also holds.

$$\frac{\vdash_{\mathrm{SC}} \mathrm{Ty}(uri) \leq E}{\mathrm{Env} \vdash^{\mathrm{Ty}}_{\mathrm{SC}} uri\colon E} \qquad \text{yields} \qquad \frac{\vdash_{\mathrm{SC}'} \mathrm{Ty}'(uri) \leq E}{\mathrm{Env} \vdash^{\mathrm{Ty}'}_{\mathrm{SC}'} uri\colon E}$$

Consider the type rule for variables. By Lemma 6, if $\vdash_{\mathrm{SC}} C \leq D$ then $\vdash_{\mathrm{SC}'} C \leq D$. Therefore if the type axiom on the left below holds, then the type axiom on the right also holds.

$$\frac{\vdash_{\mathrm{SC}} C \leq D}{\mathrm{Env}, \mathtt{\$x}\colon C \vdash \mathtt{\$x}\colon D} \qquad \text{yields} \qquad \frac{\vdash_{\mathrm{SC}'} C \leq D}{\mathrm{Env}, \mathtt{\$x}\colon C \vdash \mathtt{\$x}\colon D}$$

All other rules do not involve Ty or SC, hence follow immediately. Therefore, by structural induction, refining the type system preserves well typed scripts and data. □

4.3. Simple Entailment as a Mode of Inference

This work presents a mechanism that complementary RDF Schema in a more flexible fashion, rather than conflicting with existing standards. Here we clarify the connection between conventional RDF Schema inference defined in Section 2 and the inference problem in terms of descriptive types. The inference problem is: given some data D can we calculate type assignments and subtype assumptions Ty and SC such that $\vdash^{\mathrm{Ty}}_{\mathrm{SC}} D$.

A solution to the inference problem for descriptive types is based on applying standard RDF Schema inference as defined in Fig. 1. To use the Fig. 1 formally, we introduce the following standard terminology regarding RDF Schema.

Definition 8. *Given data D, as defined in Fig. 4, assume that any triple in D can be taken to be axioms in the inference system defined in Fig. 1. If a triple, say T, can be derived using that system, then we say that D simply entails T.*

The formal statement relating simple entailment and inference is provided in Proposition 9. Intuitively, given any data, we can construct *some* type assignment and *any* subclass relation such that the data is well-typed; and furthermore, if a triple may be inferred using simple entailment, then that triple is also well typed. Notice that the statement is stronger for subclass relations than type assignments, since any subclass relation that allows the data to be typed also respects simple entailment; in contrast, there may exist other type assignments that type the data but do not respect simple entailment. Further discussion of this discrepancy follows after the proof below.

Proposition 9. *For any data D, there exists Ty such that, for any SC such that* $\vdash^{Ty}_{SC} D$ *and, for any triple T such that D simply entails T, it holds that* $\vdash^{Ty}_{SC} T$.

Proof. Firstly, consider triples of the form $class_0$ *rdfs:subClassOf* $class_1$ simply entailed by D, for any SC such that $\vdash^{\text{Ty}}_{\text{SC}} D$. Intuitively, simply entailed triples of the given form must either appear in D, follow by reflexivity, or follow by applying transitivity to *rdfs:subClassOf* triples in D. This is formally established by the induction in the following paragraph, where we establish the stronger invariant that, if $class_0$ *rdfs:subClassOf* $class_1$ is simply entailed by D such that $\vdash^{\text{Ty}}_{\text{SC}} D$, then it holds that $class_0 \leq class_1 \in \text{SC}^*$.

The base case is that the triple is in D. In this case, by the *subclass* rule in Fig. 5, this triple is well typed only if $\vdash_{\text{SC}} class_0 \leq class_1$, which by the subclass subtype rule in Fig. 3 holds only if $class_0 \leq class_1 \in \text{SC}^*$. In the base case for reflexivity (rule rdfs10 in Fig. 1), *class rdfs:subClassOf class* is simply entailed and, since SC^* is reflexively closed over all atomic types, $class \leq class \in \text{SC}^*$ holds by definition. Consider the case of transitivity (*rdfs11* from Fig. 1), where $class_0$ *rdfs:subClassOf* $class_2$ follows from $class_0$ *rdfs:subClassOf* $class_1$ and $class_1$ *rdfs:subClassOf* $class_2$. By the induction hypothesis, $class_0 \leq class_1 \in \text{SC}^*$ and $class_1 \leq class_2 \in \text{SC}^*$ hence, since SC^* is transitively closed, $class_0 \leq class_2 \in \text{SC}^*$. By induction, for any inferred triple of the form $class_0$ *rdfs:subClassOf* $class_2$, it holds that $class_0 \leq class_2 \in \text{SC}^*$, as required.

Now consider the construction of the subtype assumptions. For some data D, construct Ty such that Ty(*uri*) is defined to be the intersection of all atomic types *class* such that one of the following hold:

- *uri a class* appears in D,
- *uri* uri_1 uri_2 and uri_1 *rdfs:domain class* appear in D,
- uri_0 uri_1 *uri* and uri_1 *rdfs:range class* appear in D;

and, furthermore, if uri_0 *uri* uri_1 or *uri rdfs:domain class* or *uri rdfs:range class* appears in D, then Ty(*uri*) should also intersect with `Property`(*rdfs:Resource*, *rdfs:Resource*).

Now consider triples of the form *uri a class* and any SC such that $\vdash^{\text{Ty}}_{\text{SC}} D$. There are four base cases. Assume that *uri a class* is in D. By the *ascription* rule in Fig. 5, this hold only if $\vdash^{\text{Ty}}_{\text{SC}} uri\colon class$, which holds only if $\vdash_{\text{SC}}$ Ty(*uri*) $\leq$ *class* which holds by construction of Ty. Assume that *uri a class* follows from *uri rdfs:domain class* and *uri* uri_1 uri_2 which are in D by *rdfs2*. To force $\vdash^{\text{Ty}}_{\text{SC}}$ *uri a class* to hold, as in the previously case, it must be the case that Ty(*uri*) $\leq$ *class*. Furthermore, since $\vdash^{\text{Ty}}_{\text{SC}}$ uri_1 *rdfs:domain class* and $\vdash^{\text{Ty}}_{\text{SC}}$ *uri* uri_1 uri_2, it must be the case that the following hold.

$$\vdash^{\text{Ty}}_{\text{SC}} uri_1\colon \texttt{Property}(class, rdfs{:}Resource) \qquad \vdash^{\text{Ty}}_{\text{SC}} uri_2\colon rdfs{:}Resource$$

In the above, the former holds since Ty(uri_1) $\leq$ `Property`(*class*, *rdfs:Resource*), by construction of Ty, and the latter holds since Ty(uri_2) $\leq$ *rdfs:Resource*, by the *top* rule in the subtype system. The inductive case is when *uri a* $class_1$ follows from rule *rdfs9*. For the induction hypothesis, assume that *uri a* $class_0$ and $class_0$ *rdfs:subClassOf* $class_1$ are well typed. By the argument in the first paragraph of this proof, $\vdash_{\text{SC}} class_0 \leq class_1$ and also $\vdash^{\text{Ty}}_{\text{SC}} uri\colon class_0$. Therefore, by subsumption, $\vdash^{\text{Ty}}_{\text{SC}} uri\colon class_1$, as required.

Finally, `Property`(*rdfs:Resource*, *rdfs:Resource*) is the bottom property type since it is a lower bound with respect to subtyping for `Property`($class_0$, $class_1$), for any classes $class_0$ and $class_1$. By the above construction of Ty, Ty(*uri*) $\leq$ `Property`(*rdfs:Resource*, *rdfs:Resource*) for any *uri* used as a property. Hence, by Theorem 1, Ty(*uri*) $\leq$ `Property`(*class*, *rdfs:Resource*). Thereby by construction of Ty, $\vdash^{\text{Ty}}_{\text{SC}}$ *uri rdfs:domain class*. A similar argument holds for *rdfs:range*.

If uri_0 uri_1 uri_2 appears in D, then by construction of Ty, $\vdash^{\text{Ty}}_{\text{SC}}$ $uri_1\colon$ `Property`(*rdfs:Resource*, *rdfs:Resource*). Since also $\vdash^{\text{Ty}}_{\text{SC}}$ $uri_0\colon$ *rdfs:Resource* and $\vdash^{\text{Ty}}_{\text{SC}}$ $uri_2\colon$ *rdfs:Resource*, $\vdash^{\text{Ty}}_{\text{SC}}$ uri_0 uri_1 uri_2. □

To the RDF Schema purist, the above result is likely to be much weaker than expected. As expected, *every* subclass assumption that well types data will type the subclass triples entailed by the data by standard simple entailment. However, Proposition 9 only states that there exists *some* type assignment that types all simply entailed triples. There are many type assignments that type the data but do not type all simply entailed triples. In particular, triples with properties *rdfs:domain* and *rdfs:range* need not necessarily force the URIs appearing as the subject and object respectively to take on exactly the type indicated.

Furthermore, the assignment constructed in the proof is not the most general assignment, for which standard RDF Schema inferences are well-typed. In the construction in the proof, any URI used as a property is intersected with the type `Property`(*rdfs:Resource*, *rdfs:Resource*), which is the *bottom* property type with respect to the subtype relation, i.e. the least property type with respect to the subtype relation. A property type that is more explicit about the range and domain are more general in the sense that they are greater in the subtype hierarchy.

A justification for the more flexible choice of Ty is that standard RDF Schema inference is only one of many natural inference *modes*. Another more technical issue is that there are differences in the handling of properties with multiple domains and ranges that we explain thoroughly in the next section.

4.4. Note on Recommendations for Multiple Domain and Range Properties

Restricting ourselves strictly to the recommended RDF Schema inference, called simple entailment, introduces difficulties for systems with dynamically changing data. The problem arises due to the handling of multiple domain and range triples. The W3C recommendation [9] states the following.

> Where a property P has more than one *rdfs:domain* property, then the resources denoted by subjects of triples with predicate P are instances of all the classes stated by the *rdfs:domain* properties.

A similar quote appears for *rdfs:range*.

In light of the above statement, suppose that we have a property with domain t_0 and range t_1. Now suppose that a further *rdfs:domain* and *rdfs:range* triple are discovered for the same property, indicating types t_2 and t_3 respectively. Under a strict interpretation of RDF Schema inference, the type of the property should be as follows.

$$\texttt{Property}(\texttt{IntersectionOf}(t_0, t_2), \texttt{IntersectionOf}(t_1, t_3))$$

Unfortunately, (assuming that t_0 is not a subclass of t_2, or t_1 is not a subclass of t_3) it is not the case that the above type is a subtype of the original type for the property. Since the above type is a supertype of $\texttt{Property}(t_0, t_1)$, then all type judgements made earlier involving the property concerned may be invalidated and hence must be checked again, i.e. we cannot apply the monotonicity condition in Lemma 7. This would break the static aspects of the type checking mechanisms, making RDF Schema inference and descriptive type checking for a dynamically growing dataset prohibitively expensive. At each discovery of new *rdfs:domain* or *rdfs:range* triples, inferences would need to be recalculated for the entire dataset and all scripts that interact with the dataset.

There are a number of approaches to avoiding the above problem, all of which are within the scope of our descriptive type system. We can instead interpret multiple domains in the following disjunctive manner.

> Where a property P has more than one *rdfs:domain* property, then the resources denoted by subjects of triples with predicate P are instances of one of the classes stated by the *rdfs:domain* properties.

In practice, many real life data sets imply our more relaxed interpretation above. For example, the Bioportal [38] contains many properties with multiple domains and ranges that make no sense when they are intersected, or are even unsatisfiable. In the Bioportal, the property *bpo:has_event* has three classes indicated as its domain: *bpo:person* , *bpo:event* and *bpo:disease_or_disorder*. It is clearly not the intention of the publishers of the ontology that any resource that appears in the subject position of a triple with property *bpo:has_event*, has a type bounded above by the intersection of all three of these types. Common sense says that the three classes are intended to be disjoint. Instead the intended meaning is clearly that *bpo:has_event* has one domain that is the union of the three types. Elsewhere, for example the provenance ontology [32], the above interpretation is made explicit by using *owl:unionOf* frequently for the domain and range triples, for example property *prov:wasInfluencedBy* has the following range:

`UnionOf` (*prov:Activity*, *prov:Agent*, *prov:Entity*)

Many ontologies, such as the DBpedia ontology [7] avoid discrepancies with respect to the W3C recommendation by only ever stating one domain or range for any property. The reader may observe for themselves the real usage of multiple domains for a property by using the following query on various SPARQL endpoints exposing ontologies.

```
select ?p ?x ?y where { ?p rdfs:domain ?x . ?p rdfs:domain ?y . filter (?x != ?y) }
```

We recommend that ontology designers use the explicit approaches to domains and ranges where either an explicit union is used, as in the provenance ontology; or only one range or property appears, as in the DBpedia ontology. Multiple domain or range properties discovered from separate sources about the same property are best treated as a single domain and range with a union type, rather than an intersection type. This permits a more accommodating approach to inference and a monotonic approach to descriptive typing, where the time complexity of inference is proportional to size of the new data added rather than the entire dataset. Note that this problem means that standard RDF Schema inference would cope poorly with big evolving datasets, as typical of Linked Data applications.

In related work [41] that explores a model theoretic semantics for `schema.org`, the interpretation for handing multiple ranges does not match the RDF Schema standard for *rdfs:range* . As explained in that work: "domains for a property are constructed by taking all the types that mention the property and producing a disjunctive property domain for the property from them." This matches the alternative handing of multiple ranges discussed in this section; and suggests that an alternative model theoretic semantics could be provided for descriptive types following [41]. Furthermore, this suggests that `schema.org`, who have deliberately not fully adopted W3C standards, have found that a disjunctive semantics for multiple ranges suits their needs. Since descriptive types produce warnings with suggestions, a combination of heuristics and human responses to options can enable a consumer to draw from both W3C standard compliant datasets and datasets conforming to a more accommodating semantics for multiple ranges, as discussed in this section.

5. An Operational Semantics using Descriptive Types at Runtime

This section is the high point of this paper. We illustrate how descriptive typing is fundamentally different from prescriptive typing.

In a prescriptive type system, we only permit the execution of programs that are well typed. In contrast, in this descriptive type system, if a program is not well typed, then the program can still be executed. During the execution of an ill-typed program, warnings are generated. At runtime, the program provides the *option* to, at any point during the execution of the program, *address the warnings* and *refine the type system* to resolve the warnings.

In this section, we informally motivate the problems descriptive types address at runtime using examples, we then introduce the formal operational semantics, and revisit the same examples formally using the operational semantics. Finally, we refine the example further to explain how an algorithm can be used to generate warnings.

5.1. Descriptive Typing for Linked Data Scripting Languages

We first intuitively illustrate a scenario involving descriptive typing for scripts that interact with Linked Data. Descriptive typing generates meaningful warnings during the execution of a script, that can assist programmers without imposing obligations.

Suppose that at some point we would like to obtain data about Andrei Ershov. Our script firstly dereferences the URI *dbp:Andrei_Yershov* (in Russian Ershov and Yershov are transliterations of the same Cyrilic characters). From this we obtain some data including the following triples.

res:Andrei_Yershov dbp:birthPlace res:Soviet_Union .
res:Andrei_Yershov dbp:league res:Kazakhstan_Major_League .
res:Andrei_Yershov a dbp:IceHockeyPlayer .

The reader familiar with Ershov the academician will find the above data strange, but the script that performs the dereferencing has no experience to judge the correctness of the data.

The script then tries to query the data that has just been obtained, as follows:

```
select $place
where res:Andrei_Yershov dbp:birthPlace $place .
```

The above query uses a property *dbp:birthPlace* that can relate any person to any location. The database is aware, due to the DBPedia ontology [7], that *dbp:IceHockeyPlayer* is a subtype of *dbp:Person*. Hence the query is considered to be well typed. The query produces the result `$place` ↦ *res:Soviet_Union*, which appears to be correct.

Next the script tries a different query.

```
select $book
where res:Andrei_Yershov free:book.author.works_written $book .
```

Before the query is executed, it is type checked. The type system knows that the property *free:book.author.works_written* relates authors to books. The type system also knows, from the data obtained earlier, that *res:Andrei_Yershov* is an ice hockey player, which is not a subtype of author. The subject of the triple and the property appear not to match.

In a prescriptive type system the query would be automatically rejected as being wrong. In contrast, a *descriptive type system* provides warnings at runtime with several options to choose from other than outright rejection.

1. Change the type of *res:Andrei_Yershov* so that the resource is both an ice hockey player and an author.
2. Change the type of the property *free:book.author.works_written* so that ice hockey players can author books.
3. Change the subtype relations so that ice hockey player is a subtype of author, hence all ice hockey players are automatically inferred to be authors.
4. Change the data so that a different resource or property is used.

The default option for RDF Schema [37] is to infer that, because the subject of *free:book.author.works_written* is an author and *res:Andrei_Yershov* appears as the subject, then *res:Andrei_Yershov* must also be an author. The type of *res:Andrei_Yershov* would be refined to the following intersection of types.

$$\texttt{IntersectionOf}(\textit{dbp:IceHockeyPlayer}, \textit{free:book.author})$$

Academics can have colourful lives, so the above type may appear plausible to an observer unfamiliar with Ershov's life. However, this is a case of mistaken identity. Ershov the academician was never a professional ice hockey player.

The correct way to resolve the above conflict is instead to change the data. A query to freebase [8] and DBpedia [7] asking resources with name Ershov in Cyrillic that are book authors reveals that the intended Ershov was identified by *res:Andrey_Ershov* in DBpedia and *free:m.012s3l* in freebase.

We return to this example after developing introducing the rules of the operational semantics.

5.2. The Operational Semantics

The rules of the operational semantics are presented in Fig. 6. The first three are the operational rules for `select`, `where` and `from` respectively. The fourth rule is the *optional* rule that refines the type system in response to warnings. We quotient data by the relation ≡ defined such that the composition of data is associative and commutative, i.e. ≡ is a congruence relation such that $(data_1\ data_2)\ data_3 \equiv data_1\ (data_2\ data_3)$ and $data_1\ data_2 \equiv data_2\ data_1$.

Configurations. A configuration (*script*, *data*, Ty, SC) represents the state of the system, which can change during the execution of a script. It consists of four components:

- The script *script* that is currently being executed.
- The data *data* representing triples that are currently stored locally.
- A partial function Ty from resources to types, representing the current type assumptions about resources.
- A relation over atomic types SC, representing the current subtype assumptions.

$$\frac{\vdash^{\mathrm{Ty}}_{\mathrm{SC}} \mathit{uri} : C}{(\texttt{select}\ \mathit{uri} : C\ \mathit{script}, \mathit{data}, \mathrm{Ty}, \mathrm{SC}) \longrightarrow (\mathit{script}\{^{\mathit{uri}}/_{\$\mathtt{x}}\}, \mathit{data}, \mathrm{Ty}, \mathrm{SC})}\ \mathit{select}$$

$$\frac{\mathit{data} \equiv \mathit{term}_0\ \mathit{term}_1\ \mathit{term}_2\ \mathit{data}'}{(\texttt{where}\ \mathit{term}_0\ \mathit{term}_1\ \mathit{term}_2\ \mathit{script}, \mathit{data}, \mathrm{Ty}, \mathrm{SC}) \longrightarrow (\mathit{script}, \mathit{data}, \mathrm{Ty}, \mathrm{SC})}\ \mathit{where}$$

$$\frac{(\mathrm{Ty}', \mathrm{SC}') \leq (\mathrm{Ty}, \mathrm{SC}) \quad \vdash^{\mathrm{Ty}'}_{\mathrm{SC}'} \mathit{data}_1}{(\texttt{from}\ \mathit{uri}\ \mathit{script}, \mathit{data}_0, \mathrm{Ty}, \mathrm{SC}) \longrightarrow (\mathit{script}, \mathit{data}_0\ \mathit{data}_1, \mathrm{Ty}', \mathrm{SC}')}\ \mathit{from}$$

$$\frac{(\mathrm{Ty}', \mathrm{SC}') \leq (\mathrm{Ty}, \mathrm{SC}) \quad \vdash^{\mathrm{Ty}'}_{\mathrm{SC}'} \mathit{script}}{(\mathit{script}, \mathit{data}, \mathrm{Ty}, \mathrm{SC}) \longrightarrow (\mathit{script}, \mathit{data}, \mathrm{Ty}', \mathrm{SC}')}\ \mathit{optional}$$

Figure 6: The operational semantics for scripts. Note that, in the *from* rule, data_1 is the data obtained at runtime by dereferencing the resource *uri*.

Type assumptions and subtype assumptions can be changed by the rules of the operational semantics, since they are part of the runtime state.

The rules are best explained by the examples in the rest of the section. Notice that a novel feature of this operational semantics are the type assertions in the premise of the rules *select*, *from* and *optional*. We briefly highlight these novelties below.

For the rule *select*, the type assertion in the premise represents a dynamic type check that cannot always be checked statically, for example assume that the type of the bound variable is *dbp:Boxer*, that the variable appears in a subsequent *where* clause as the subject or object of a property *foaf:knows* that relates two resources of type *foaf:Person*, and also that *dbp:Boxer* ≤ *foaf:Person*. In this case, the well-typed data involving *foaf:knows* may match the pattern in the `where` clause, but the people related may not necessarily be boxers, hence a dynamic type check needs to be performed. Notice that static analysis could eliminate the dynamic type check when the type of the variable is of the greatest type that the variable may assume according to how the variable is used in subsequent `where` clauses.

For the rule *from*, the type assertion in the premise is required to deal with potential new type assumption and subtype assertions that accommodate the new data. For example, there may be no type assumption for a *uri* in the newly discovered data that is used as the subject in a triple that may only refer to authors, hence the type assumptions should be refined in one of a number of ways to accommodate this new schema information. The premise is a declarative specification from which constraints can be generated which are then solved algorithmically, according a method explained and illustrated later in this section. Scenarios where it is useful to have multiple options, rather than selecting default RDF Schema inference, are presented by examples in this section.

For the *optional* rule, satisfying the type assertion in the premise ensures that the remaining parts of the script are well typed. By unfolding the rules of the algorithmic type system, constraints are generated for which several options are algorithmically generated, using the same approach as for the *from* rule as explained in the rest of this section through examples and algorithms. For example, a query may involve triple with property *foaf:knows* which forces constraints on the types of variables and resources that appear in the context of the triple that cannot be resolved without refining the type system. Thus the rule *optional* anticipates refinements that may be applied later in the script and also assists the human in spotting data modelling errors in the queries in scripts themselves.

5.3. A Worked Example of a Good Script

We explain the interplay between the operational rules using a concrete example. Suppose that initially we have a configuration consisting of:

- a script:

```
from res:Andrey_Ershov
select $place: dbp:PopulatedPlace
where res:Andrey_Ershov dbp:birthPlace $book
```

- some data $data_0$ including triples such as the following:

```
dbp:birthPlace rdfs:domain dbp:Person .
dbp:birthPlace rdfs:range dbp:PopulatedPlace .
res:SovietUnion a dbp:PopulatedPlace .
```

- some type assumptions Ty such that:

$$\mathrm{Ty}(\textit{dbp:birthPlace}) = \texttt{Property}(\textit{dbp:Person}, \textit{dbp:PopulatedPlace})$$
$$\mathrm{Ty}(\textit{res:Andrey\_Ershov}) = \textit{rdfs:Resource}$$
$$\mathrm{Ty}(\textit{res:SovietUnion}) = \textit{dbp:PopulatedPlace}$$

- an empty set of subtype assumptions.

The above script is not well typed with respect to the type assumptions, since the strongest type for *res:Andrey_Ershov* is the top type, which is insufficient to establish that the resource represents a person.

There are several options other than rejecting the ill typed script. We can inspect the warning, which provides a menu of options to refine the type system so that the script is well typed. At this stage of execution, there are two reasonable solutions: either we can refine the type of *res:Andrey_Ershov* , so that he is of type *dbp:Person* ; or we can refine the type of *dbp:birthPlace* so that it can relate any resource to a populated place.

A further option is available. Since these are warnings, we can ignore them and continue executing the script. Assuming we choose to ignore the warnings at this stage, we apply the operational rule for `from`.

The rule involves some new data $data_1$ that is obtained by dereferencing resource with URI *dbp:Andrey_Ershov* . This includes triples such as:

```
res:Andrei_Ershov a yago:FellowsOfTheBritishComputerSociety .
res:Andrei_Ershov dbp:birthPlace res:SovietUnion .
```

The rule must calculate $(\mathrm{Ty}', \mathrm{SC}')$ such that $(\mathrm{Ty}', \mathrm{SC}') \leq (\mathrm{Ty}, \mathrm{SC})$ and $\vdash^{\mathrm{Ty}'}_{\mathrm{SC}'} data_1$. Again there are several options for resolving the above constraints, presented below.

1. Refine the type assumptions such that the resource *res:Andrey_Ershov* is assigned the intersection of the types *yago:FellowsOfTheBritishComputerSociety* and *dbp:Person* as its type.
2. Refine the type of Ershov to the type *yago:FellowsOfTheBritishComputerSociety* and refine the type of property *dbp:birthPlace* such that it is of the following type:

```
IntersectionOf( Property( dbp:Person,
                          dbp:PopulatedPlace ),
                Property( yago:FellowsOfTheBritishComputerSociety,
                          dbp:PopulatedPlace ) )
```

3. Refine the subtype assumptions to SC' such that it contains the following subtype inequality:

$$\textit{yago:FellowsOfTheBritishComputerSociety} \leq \textit{dbp:Person}$$

The first option above is the default option taken by RDF Schema [9]. It assumes that, since the domain of the property was *dbp:Person*, Ershov must be a person. The second option above makes the property more accommodating, so that it can also be used to relate fellows of the British Computer Society to populated places. The third option is the most general solution, since it allows any fellow of the British Computer Society to be used as a person in all circumstances.

The appropriate option is subjective, so the choice is delegated to a human. Suppose that the programmer selects the third option. This results in the following configuration:

- a script where the leading `from` keyword has been removed:

$$\texttt{select } \texttt{\$place}\texttt{:}\ \textit{dbp:PopulatedPlace}$$
$$\texttt{where } \textit{res:Andrey\_Ershov dbp:birthPlace } \texttt{\$place}$$

- some data $data_0$ $data_1$ including the new data obtained by dereferencing the resource *dbp:Andrey_Ershov* :

$$\textit{dbp:birthPlace rdfs:domain dbp:Person .}$$
$$\textit{dbp:birthPlace rdfs:range dbp:PopulatedPlace .}$$
$$\textit{res:SovietUnion a dbp:PopulatedPlace .}$$
$$\textit{res:Andrei\_Ershov a yago:FellowsOfTheBritishComputerSociety .}$$
$$\textit{res:Andrei\_Ershov dbp:birthPlace res:SovietUnion .}$$

- a refined type assumption Ty', such that:

$$\mathrm{Ty}'(\textit{dbp:birthPlace}) = \texttt{Property}(\textit{dbp:Person}, \textit{dbp:PopulatedPlace})$$
$$\mathrm{Ty}'(\textit{res:Andrey\_Ershov}) = \textit{yago:FellowsOfTheBritishComputerSociety}$$
$$\mathrm{Ty}'(\textit{res:SovietUnion}) = \textit{dbp:PopulatedPlace}$$

- the refined subtype assumptions SC' suggested in the third option above, such that:

$$\textit{yago:FellowsOfTheBritishComputerSociety} \leq \textit{dbp:Person}$$

Having resolved the warning we are now in the fortunate situation that the remainder of the script is also well typed with respect to the new type and subtype assumptions. Thus we can continue executing without further warnings.

We apply the operational rule for `select`. This rule dynamically checks that the following holds.

$$\vdash^{\mathrm{Ty}'}_{\mathrm{SC}'} \textit{res:SovietUnion}\texttt{:}\ \textit{dbp:PopulatedPlace}$$

Since the above type judgement holds, the substitution is applied to obtain a configuration with the following script.

$$\texttt{where } \textit{res:Andrey\_Ershov dbp:birthPlace res:SovietUnion}$$

Finally, since the triple in the `where` clause matches a triple in the data, we can apply the operational rule for `where`. This successfully completes the execution of the script.

5.4. A Worked Example of a Bad Script

Now consider the motivating example at the beginning of this section. We begin with the following configuration, where the wrong URI has been used for Ershov, mistaking Andrei Yershov and Andrei Ershov for the same person:

- the following script:

$$\texttt{from } \textit{res:Andrei\_Yershov}$$
$$\texttt{select } \texttt{\$book}\texttt{:}\ \textit{free:book}$$
$$\texttt{where } \textit{res:Andrei\_Yershov free:book.author.works\_written } \texttt{\$book} \ .$$

- some initial data $data_0$ including triples such as:

$$\textit{free:book.author.works\_written rdfs:domain free:book.author}$$
$$\textit{free:book.author.works\_written rdfs:range free:book}$$

- initial type assumptions Ty such that:

$$\mathrm{Ty}(\textit{free:book.author.works\_written}) = \texttt{Property}(\ \textit{free:book.author}, \ \textit{free:book})$$

- an empty set of subtype assumptions.

The programmer has not yet realised that *res:Andrei_Yershov* represents an ice hockey player who is not the intended scientist. At runtime, the programmer initially ignores a menu of warnings that would enable the *optional* rule to be applied. One option suggests that the type of *res:Andrei_Yershov* should be *free:book.author*; another option suggest refining the type of *free:book.author.works_written* to the following type.

$$\texttt{Property}(\textit{rdfs:Resource}, \textit{free:book})$$

The programmer decides to ignore the warnings and continue executing the script. As in the previous example, we apply the `from` rule. This dereferences the resource *res:Andrei_Yershov* obtaining some new data $data_1$ including the following triple.

$$\textit{res:Andrei\_Yershov}\ \ a\ \ \textit{dbp:IceHockeyPlayer}\ \ .$$

There is only one good option in this case, which that script automatically selects. It sets a refined type assumption Ty' such that the following holds.

$$\mathrm{Ty}'(\textit{res:Andrei\_Yershov}) = \textit{dbp:IceHockeyPlayer}$$

In the new configuration, there are still warnings that are induced by attempting to apply the *optional* rule. The following menu of options is presented to the programmer.

1. Refine the type assumptions such that the resource *res:Andrei_Yershov* is assigned the intersection of the classes *yago:IceHockeyPlayer* and *free:book.author* as its type.
2. Refine the type of the property *free:book.author.works_written* such that it is of the following type.

$$\begin{array}{lll}\texttt{IntersectionOf(} & \texttt{Property(} & \textit{free:book.author},\\ & & \textit{free:book}\ \texttt{),}\\ & \texttt{Property(} & \textit{dbp:IceHockeyPlayer},\\ & & \textit{free:book}\ \texttt{)}\ \texttt{)}\end{array}$$

3. Refine the subtype assumption to SC' such that it contains the following subtype inequality.

$$\textit{dbp:IceHockeyPlayer} \leq \textit{free:book.author}$$

The three options are similar to the options in the previous examples. The difference is that the programmer should be suspicious. The first option above may be plausible, but the programmer will be asking whether Ershov was really both an author and a professional ice hockey player. The second option above, which allows all ice hockey players to author books, is highly questionable. It certainly does not make sense to take the third option above and make every ice hockey player a book author.

A further reason to be alarmed is that, if the programmer attempts to ignore the strange warnings, then the script cannot be executed further. There is no resource that can be selected that enables the `where` clause to be matched.

Given the evidence, the programmer can conclude that there was a mismatch between the query and the resource dereferenced. The solution is therefore to change the scripts. By inspecting the data it is clear that the resource represents the wrong Ershov, hence the programmer decides to change all appearances of the troublesome resource.

5.5. Calculating the Options in Warnings Algorithmically

The *optional* operational rule and the operational rule for `from` are specified declaratively in the operational semantics. These rules permit any refined type system that satisfies the constraints to be chosen. By exploiting the algorithmic type system defined in previous section, we can algorithmically generate a menu of good solutions that fix some types while maximising others.

Firstly, we explain how the algorithm can be applied to generate the options in the examples above. Secondly, we present the generalised algorithm.

Example constraints. Consider a system of constraints from the running examples. Assume that SC is empty and we have that Ty is such that:

$$\begin{aligned} \mathrm{Ty}(\textit{res:Andrei\_Yershov}) &= \textit{dbp:IceHockeyPlayer} \\ \mathrm{Ty}(\textit{free:book.author.works\_written}) &= \texttt{Property}(\ \textit{free:book.author}, \\ &\qquad\qquad\qquad\qquad\quad \textit{free:book}\ ) \end{aligned}$$

The aim is to calculate Ty' and SC' such that $(\mathrm{Ty}', \mathrm{SC}') \leq (\mathrm{Ty}, \mathrm{SC})$ and the following type assumption holds.

$$\vdash^{\mathrm{Ty}'}_{\mathrm{SC}'} \ \texttt{select\$book}\colon \textit{free:book} \quad \texttt{where}\ \textit{res:Andrei\_Yershov}\ \ \textit{free:book.author.works\_written}\ \ \texttt{\$book}\ .$$

We then unfold the algorithmic type system, using type variables X and Y for types that could take several values, as follows.

$$\dfrac{\dfrac{\vdash \mathrm{Ty}'(\textit{res:Andrei\_Yershov}) \leq X}{\vdash \textit{res:Andrei\_Yershov}\colon X} \quad \dfrac{\vdash \mathrm{Ty}'(\textit{free:book.author.works\_written}) \leq \texttt{Property}(X, Y)}{\vdash \textit{free:book.author.works\_written}\colon \texttt{Property}(X, Y)} \quad \dfrac{\vdash \textit{free:book} \leq Y}{\texttt{\$book}\colon \textit{free:book} \vdash \texttt{\$book}\colon Y}}{\dfrac{\texttt{\$book}\colon \textit{free:book} \vdash \texttt{where}\ \textit{res:Andrei\_Yershov}\ \ \textit{free:book.author.works\_written}\ \ \texttt{\$book}}{\vdash \texttt{select\$book}\colon \textit{free:book}\ \texttt{where}\ \textit{res:Andrei\_Yershov}\ \ \textit{free:book.author.works\_written}\ \ \texttt{\$book}}}$$

From the above we generate the following constraints on Ty', where X and Y are variables for types to be solved.

$$\mathrm{Ty}'(\textit{res:Andrei\_Yershov}) \leq X \qquad \textit{free:book} \leq Y \qquad \mathrm{Ty}'(\textit{free:book.author.works\_written}) \leq \texttt{Property}(X, Y)$$

Also, since $(\mathrm{Ty}', \mathrm{SC}') \leq (\mathrm{Ty}, \mathrm{SC})$, we have the following constraints, by definition.

$$\begin{aligned} \mathrm{Ty}'(\textit{res:Andrei\_Yershov}) &\leq \textit{dbp:IceHockeyPlayer} \\ \mathrm{Ty}'(\textit{free:book.author.works\_written}) &\leq \texttt{Property}(\textit{free:book.author}, \textit{free:book}) \end{aligned}$$

Furthermore, $\mathrm{SC} \subseteq \mathrm{SC}'$.

From the above, we can generate the following scheme for upper bounds on Ty'.

$$\begin{aligned} \mathrm{Ty}'(\textit{res:Andrei\_Yershov}) &\leq \texttt{IntersectionOf}(\textit{dbp:IceHockeyPlayer}, X) \\ \mathrm{Ty}'(\textit{free:book.author.works\_written}) &\leq \texttt{IntersectionOf}(\\ &\qquad \texttt{Property}(\textit{free:book.author}, \textit{free:book}), \\ &\qquad \texttt{Property}(X, Y)\ ) \end{aligned}$$

We use these upper bounds to generate the options that appear in warnings, by varying X and Y within the bounds set by the constraints.

Notice that immediately the mapping for Y can be chosen such that both of the upper bounds are maximised and the constraints are satisfied. In particular, the mapping $Y \mapsto \textit{free:book}$ will maximise the upper bound on *free:book.author.works_written* and has no impact on the upper bound for *res:Andrei_Yershov*. We can therefore immediately apply this mapping for Y to refine the upper bound on the type of the property as follows.

$$\begin{aligned} \mathrm{Ty}'(\textit{free:book.author.works\_written}) &\leq \texttt{IntersectionOf}(\\ &\qquad \texttt{Property}(\textit{free:book.author}, \textit{free:book}), \\ &\qquad \texttt{Property}(X, \textit{free:book})\ ) \end{aligned}$$

In contrast to Y, there is no optimal solution for X, since $X \mapsto \textit{rdfs:Resource}$ will maximise the upper bound for *res:Andrei_Yershov*, but $X \mapsto \textit{free:book.author}$ will maximise the upper bound for *free:book.author.works_written*. Any type between these two types are potential solutions for X for which there are infinitely many possible solutions.

Option no.1: Maximise type of property. To generate the first option Ty_1, we maximise the type of properties by ensuring that $\mathrm{Ty}_1(uri) = \mathrm{Ty}(uri)$ for any property *uri*. Hence we assume:

$$\mathrm{Ty}_1(\textit{free:book.author.works\_written}) = \texttt{Property}(\textit{free:book.author}, \textit{free:book})$$

This assumption yields the following type inequality.

$$\begin{aligned}\texttt{Property}(\textit{free:book.author}, \textit{free:book}) \leq \quad & \texttt{IntersectionOf}(\\ & \texttt{Property}(\textit{free:book.author}, \textit{free:book}), \\ & \texttt{Property}(X, \textit{free:book}) \ )\end{aligned}$$

We use the cut-free subtype system to analyse the above constraints. We apply the *greatest lower bound* rule, then the *property* rule to obtain the following subtype constraint: $X \leq \textit{free:book.author}$.

We then maximise the remaining type assignments with respect this these constraints. We seek the mapping for X such that the constraints are satisfied and the new type assignment is maximised, with respect to the subtype relation. Recall that the upper bound on the subject of the example triple is of the following form:

$$\mathrm{Ty}_1(\textit{res:Andrei\_Yershov}) \leq \texttt{IntersectionOf}(\textit{dbp:IceHockeyPlayer}, X)$$

From this we derive the mapping $X \mapsto \textit{free:book.author}$, as an optimal solution for X that satisfies the constraints and maximises the above upper bound on the type assignment. By substituting the variable for the optimal solution we obtain a refined type system such that:

$$\mathrm{Ty}_1(\textit{res:Andrei\_Yershov}) = \texttt{IntersectionOf}(\textit{dbp:IceHockeyPlayer}, \textit{free:book.author})$$

The above type assignment is exactly what standard RDF Schema inference would infer [37].

Option no.2: Maximise type of subject/object. To generate the second option Ty_2, we maximise the type of the subject of our example triple be setting $\mathrm{Ty}_2(\textit{res:Andrei\_Yershov}) = \mathrm{Ty}(\textit{res:Andrei\_Yershov})$, hence the following assumption is made.

$$\mathrm{Ty}_2(\textit{res:Andrei\_Yershov}) = \textit{dbp:IceHockeyPlayer}$$

This yields the following subtype inequality, due to the upper bound on the resource.

$$\textit{dbp:IceHockeyPlayer} \leq \texttt{IntersectionOf}(\textit{dbp:IceHockeyPlayer}, X)$$

As in option no.1, we unfold the rules of the algorithmic type system to derive the constraint $\textit{dbp:IceHockeyPlayer} \leq X$. We then maximise the type of the property with respect to this constraint. Recall that the upper bound on the property is as follows.

$$\begin{aligned}\texttt{Property}(\textit{free:book.author}, \textit{free:book}) \leq \texttt{IntersectionOf}(& \\ \texttt{Property}(\textit{free:book.author}, \textit{free:book}), & \\ \texttt{Property}(X, \textit{free:book}) \ ) &\end{aligned}$$

Hence, due to the contravariance of property types, the solution for X that maximises the above upper bound on the type assignment is $X \mapsto \textit{dbp:IceHockeyPlayer}$. By applying this mapping to variable X, we obtain a refined type system such that.

$$\begin{aligned}\mathrm{Ty}_2(\textit{free:book.author.works\_written}) = \quad & \texttt{IntersectionOf}(\\ & \texttt{Property}(\textit{free:book.author}, \textit{free:book}) \ , \\ & \texttt{Property}(\textit{dbp:IceHockeyPlayer}, \textit{free:book}))\end{aligned}$$

This is more general than standard RDF Schema inference, since RDF Schema inference does not have the ability to modify the types of properties according to the types of data related by the properties.

Option no.3: Extend the subtype relation. The final option is to add subtype assumptions to the type system. We can calculate these subtype assumptions algorithmically, by calculating the conditions under which the above two options are equal, i.e. $\mathrm{Ty}_1 = \mathrm{Ty}_2$. Now $\mathrm{Ty}_1 = \mathrm{Ty}_2$ if and only if the following inequalities hold.

$$\textit{dbp:IceHockeyPlayer} \leq \texttt{IntersectionOf}(\textit{dbp:IceHockeyPlayer}, \textit{free:book.author})$$

$$\texttt{Property}(\ \textit{free:book.author}\ ,\ \textit{free:book}\ ) \leq \begin{array}{l}\texttt{IntersectionOf}(\\ \quad \texttt{Property}(\textit{free:book.author}, \textit{free:book}),\\ \quad \texttt{Property}(\textit{dbp:IceHockeyPlayer}, \textit{free:book}))\end{array}$$

By unfolding the rules of the cut-free subtype system in Fig. 3, we can calculate that the above inequalities hold only if the following subtype inequality holds.

$$\textit{dbp:IceHockeyPlayer} \leq \textit{free:author}$$

Thus, if we include the above constraint in SC', then the original Ty satisfies the necessary constraints to enable the *optional* rule.

Note that the above algorithm does not always find a suitable set of constraints. For example, if we attempt to apply the *optional* rule before executing `from` in the above example of a bad script, the algorithm gets stuck at the following constraint.

$$\textit{rdfs:Resource} \leq \textit{free:book.author}$$

Since *rdfs:Resource* is not an atomic type, the above inequality cannot be induced by extending the set of subtype assumptions, so there is no solution to modifying SC. This is a positive feature since, in an open world of knowledge like the Web, it makes no sense to state that every resource is an author.

Summary of all three options. The general algorithm works as follows.

1. We use the algorithmic type system and the constraint $(\mathrm{Ty}', \mathrm{SC}') \leq (\mathrm{Ty}, \mathrm{SC})$ to generate a scheme (types with type variables) for upper and lower bounds on Ty'. We consider the type assumption appearing in the premise of the *from* and *optional* rules: the script — in the case of the *optional* rule; or the data — in the case of the *from* rule. We unfold the rules of the algorithmic type system, in Fig. 5, using variables for any undetermined value and thereby obtaining a set of subtype constraints, which include bounds on the types of URIs in Ty'. From the assumption $(\mathrm{Ty}', \mathrm{SC}') \leq (\mathrm{Ty}, \mathrm{SC})$, by definition, we deduce that all of the constraints in SC must also hold in SC' and also, for any *uri* in the domain of Ty, $\mathrm{Ty}'(\textit{uri}) \leq \mathrm{Ty}(\textit{uri})$. For each URI in the domain of Ty', we take the intersection of all upper bounds and the union of all lower bounds generated to obtain the scheme for upper and lower bounds.

2. We generate the first option $(\mathrm{Ty}_1, \mathrm{SC})$ by, for every property *uri* in the domain of Ty, setting the type assignment such that $\mathrm{Ty}_1(\textit{uri}) = \mathrm{Ty}(\textit{uri})$. This generates a subtype inequality based on the upper bounds on Ty' calculated in the first step. We unfold the cut-free subtype system, in Fig. 3, to generate constraints on other URIs that must be assigned types by Ty_1. We then find a mapping from type variables to types that maximises the type assigned to URIs that have not yet been assigned a type in Ty_1, while respecting all constraints generated (this can be done on a random or user/heuristic guided URI-by-URI basis). By applying the mapping generated to the upper bounds from the first step above, we obtain the type assignment Ty_1.

3. We generate the second option $(\mathrm{Ty}_2, \mathrm{SC})$ by, for every subject and object *uri* in the domain of Ty, setting the type assignment such that $\mathrm{Ty}_2(\textit{uri}) = \mathrm{Ty}(\textit{uri})$. We then proceed as in the previous step to maximise the type assignments for the URIs representing properties.

4. To generate the third option $(\mathrm{Ty}, \mathrm{SC}')$, we set $\mathrm{Ty}_1 = \mathrm{Ty}_2$ to obtain a system of subtype inequalities. If there is a solution then, by unfolding the rules of the cut-free subtype system, we obtain a set of subtype inequalities over atomic types and we extend SC with these constraints to obtain SC'. If there is no solution, then the unfolding of the rules of the cut-free system will halt with neither a subtype inequality over atomic types nor an axiom, in which case no third option is presented.

The second point above generates classes for resources as expected by RDF Schema [9]; with the exception of the handling of multiple domains and ranges as discussed in Section 4.4. The third and fourth points above provide alternative, more general modes of inference. Thus the above algorithm extends RDF Schema inference. Note that steps 2 and 3 above can be interleaved allowing the type of any URI that appears to be maximised in any order, thereby mixing RDF Schema inference with more general inferences. Further analysis of the above algorithm is we believe is best conducted by an implementation.

5.6. Subject Reduction

There are two reasons why a system is well typed. Either *a priori* the script was well typed before refining the type system, or at some point during the execution the programmer acted to resolve all warnings. In either case, once the script is well typed it can be executed to completion without generating any warnings other than unavoidable warnings that occur from reading data from the Web.

The following proposition characterises the relationship between the type system and script after choosing to resolve warnings, by selecting the optional rule. Recall that warnings are resolved by selecting one or more options for refining the type system from a menu generated based on constraints imposed by the premise of the *optional* rule. In particular, after choosing to resolve all warnings the script is also well typed with respect to the refined type system.

Proposition 10. *If* $\vdash^{Ty}_{SC}$ *data and the optional rule is applied, such that*

$$(script, data, Ty, SC) \longrightarrow (script, data, Ty', SC'),$$

then $\vdash^{Ty'}_{SC'}$ *script and* $\vdash^{Ty'}_{SC'}$ *data.*

Proof. Assume that $\vdash^{\mathrm{Ty}}_{\mathrm{SC}}$ *data* and $(script, data, \mathrm{Ty}, \mathrm{SC}) \longrightarrow (script, data, \mathrm{Ty}', \mathrm{SC}')$ due to the *optional* rule. Hence it must be the case that $(\mathrm{Ty}', \mathrm{SC}') \leq (\mathrm{Ty}, \mathrm{SC})$ and $\vdash^{\mathrm{Ty}'}_{\mathrm{SC}'}$ *script*. Hence, by Lemma 7, $\vdash^{\mathrm{Ty}'}_{\mathrm{SC}'}$ *data* holds, as required. □

We require the following substitution lemma. It states that if we assume that a variable is of a certain type, then we can substitute a URI of that type for the variable and preserve typing.

Lemma 11. *Assume that* $\vdash uri : C$*. Then the following statements hold:*

1. *If* $Env, \$\mathtt{x} : C \vdash script$*, then* $Env \vdash script\left\{^{uri}/_{\$\mathtt{x}}\right\}$.
2. *If* $Env, \$\mathtt{x} : C \vdash term : D$*, then* $Env \vdash term\left\{^{uri}/_{\$\mathtt{x}}\right\} : D$.

Proof. Assume that $\vdash uri : C$. The proof proceeds by structural induction on the type derivation tree.

Consider the case of the type rule for variables, where the variable equals `$x`. In this case, the type tree on the left can be transformed into the type tree on the right.

$$\frac{\vdash C \leq D}{\mathrm{Env}, \$\mathtt{x} : C \vdash \$\mathtt{x} : D} \qquad \text{yields} \qquad \frac{\vdash uri : C \quad \vdash C \leq D}{\mathrm{Env} \vdash uri : D}$$

Hence, by Proposition 4, $\mathrm{Env} \vdash uri : D$ holds in the algorithmic type system and clearly $\$\mathtt{x}\left\{^{uri}/_{\$\mathtt{x}}\right\} = uri$ as required. All other cases for terms are trivial.

Consider the case of the select rule. Assume that $\mathrm{Env}, \$\mathtt{x} : C \vdash \mathtt{select}\ \$\mathtt{y} : D\ script$ holds. If `$x` = `$y`, then `$x` does not appear free in `select $x : D`, hence $\mathrm{Env} \vdash \mathtt{select}\ \$\mathtt{x} : D\ script$ as required. If `$x` ≠ `$y`, then, by the induction hypothesis, if $\mathrm{Env}, \$\mathtt{x} : C, \$\mathtt{y} : D \vdash script$ holds then $\mathrm{Env}, \$\mathtt{y} : D \vdash script\left\{^{uri}/_{\$\mathtt{x}}\right\}$ holds. Hence the proof tree on the left below can be transformed into the proof tree on the right below.

$$\frac{\mathrm{Env}, \$\mathtt{x} : C, \$\mathtt{y} : D \vdash script}{\mathrm{Env}, \$\mathtt{x} : C \vdash \mathtt{select}\ \$\mathtt{y} : D\ script} \qquad \text{yields} \qquad \frac{\mathrm{Env}, \$\mathtt{y} : D \vdash script\left\{^{uri}/_{x}\right\}}{\mathrm{Env} \vdash \mathtt{select}\ \$\mathtt{y} : D\ script\left\{^{uri}/_{x}\right\}}$$

Furthermore, since `$x` ≠ `$y`, by the standard definition of substitution the following holds as required.

$$\mathtt{select}\ \$\mathtt{y} : D\ script\left\{^{uri}/_{x}\right\} = (\mathtt{select}\ \$\mathtt{y} : D\ script)\left\{^{uri}/_{x}\right\}$$

The cases for other rules follow immediately by induction. □

The property that a well-typed script will not raise unnecessary warnings, is formulated as the following subject reduction result.

Theorem 12 (Subject reduction)**.** *If* $\vdash_{SC}^{Ty}$ *script and* $\vdash_{SC}^{Ty}$ *data, then if*

$$(script, data, Ty, SC) \longrightarrow (script', data', Ty', SC'),$$

then $\vdash_{SC'}^{Ty'}$ *script*′ *and* $\vdash_{SC'}^{Ty'}$ *data*′.

Proof. The proof is by case analysis, over each operational rule.

Consider the operational rule for `select`. Assume that the following hold.

$$\vdash \texttt{select}\ \texttt{\$x}\colon C\ script \qquad \vdash data \qquad \vdash uri\colon C$$

The above holds only if $\texttt{\$x}\colon C \vdash script$, by the type rule for `select`. By Lemma 11, since $\texttt{\$x}\colon C \vdash script$ and $\vdash uri\colon C$, it holds that $\vdash script\{^{uri}/_{x}\}$. Therefore the `select` rule preserves types.

Consider the operational rule for `where`. Assume that the following type assumption holds.

$$\vdash \texttt{where}\ term_0\ term_1\ term_2\ script \qquad \vdash term_0\ term_1\ term_2\ data$$

The above holds only if $\vdash script$ holds, hence the operational rule for `where` preserves well-typed scripts.

Consider the operational rule for `from`. Assume that the following assumptions hold.

$$\vdash_{\mathrm{SC}}^{\mathrm{Ty}} \texttt{from}\ uri\ script \qquad \vdash_{\mathrm{SC}}^{\mathrm{Ty}} data_0 \qquad \vdash_{\mathrm{SC}'}^{\mathrm{Ty}'} data_1 \qquad (\mathrm{Ty}', \mathrm{SC}') \leq (\mathrm{Ty}, \mathrm{SC})$$

The first assumption above holds only if $\vdash_{\mathrm{SC}}^{\mathrm{Ty}} script$ holds, by the type rule for `from`. Since $(\mathrm{Ty}', \mathrm{SC}') \leq (\mathrm{Ty}, \mathrm{SC})$, by Lemma 7, $\vdash_{\mathrm{SC}'}^{\mathrm{Ty}'} script$ holds. By Lemma 7 again, $\vdash_{\mathrm{SC}'}^{\mathrm{Ty}'} data_0$ holds. Hence $\vdash_{\mathrm{SC}'}^{\mathrm{Ty}'} data_0\ data_1$ holds. Therefore the `from` rule preserves types.

Consider the case of the *optional* operational rule. For some initial configuration $(script, data, \mathrm{Ty}, \mathrm{SC})$, we assume that $\vdash_{\mathrm{SC}}^{Ty} data$. The result then follows from Prop. 10. □

6. Future Work

This paper provides the foundational definitions and results required to introduce a descriptive type system. This approach can be adapted to more expressive languages including the full W3C SPARQL recommendation [24] and extensions of scripting languages for Linked Data developed by the authors [28, 14]. We now consider our line of work to be mature, having enough supporting results to be confident in the correctness of the approach and the realistic computational resources required. In related work, prescriptive type systems have been proposed for ensuring privacy guarantees [31] and for provenance-based access control [19] in Linked Data. Observe that such type systems for security guarantees are, necessarily, prescriptive.

Further investigations in the direction of descriptive types would best be pursued through prototype implementations. A prototype implementation would enable questions to be investigated regarding the presentation of type information and warnings to users and also machine learning support to reduce, but not eliminate, decisions made by humans. We highlight some challenges in the remaining paragraphs in this subsection.

How would type information best be presented to the user? RDF Schema and the descriptive types in this work are essentially graphical, hence a graphical representation of type information would be appropriate. It makes sense to use representations familiar to software engineers similar to entity-relationship diagrams. There is a wealth of tool support for graphical modelling [39, 20] that can be adapted for this purpose.

How would the menu of options to resolve warnings be presented? The algorithm for generating warnings, particularly when larger queries and scripts are supported, is capable of generating many options for resolving warnings about mismatches between the schema and data. Indeed in some cases there are infinitely many potential solutions. This problem may require heuristics to select a small number of options, or perhaps present options visually as a lattice, thereby assisting further the user in pinning down the appropriate schema.

A third line of enquiry, that can be evaluated by using a prototype implementation, is to investigate the potential of machine learning in the support for choosing correct inferences. The programmer is only asked about a minimal number of inferences where only human knowledge of the context can distinguish the correct inference; most other inferences are applied automatically. Furthermore, the schema for a dataset is generally much smaller than the dataset itself. In many scenarios, ideally, once a few queries have been asked over a dataset, enough schema information will have been inferred to proceed without applying many more inferences. However, in some use cases, the number of warnings presented to the programmer by the type system may become prohibitively large. To support this scenario, machine learning could be applied to support selecting or suggesting the best options for resolving warnings on behalf of the programmer. We imagine that a collaborative or social approach would be most effective, assuming that programmers are willing to share their warnings and any chosen resolutions.

We do not believe that human input to resolving warnings thrown by the descriptive type system should be replaced entirely by machine learning. A philosophical argument from the field of semiotics [46] is that descriptive types provide support for abductive reasoning to assist with moving between the levels of syntax and semantics. In the philosophy of semiotics, syntax is the data which is typically intended for machine processing, while semantics refers to the ontology that is intended for humans to make sense of the data. Semiotics emphasises the subjective nature of the relationship between syntax and semantics by recording the interpretant. In this case, the interpretants are the users resolving warnings to infer refinements to the ontology represented by the type system. The community who originally introduced the RDF recommendations are directly inspired by semiotics, hence an additional result of our work on descriptive types is to formally bring closer the fields of type theory in the tradition of Curry, and semiotics in the tradition of Pierce. Future work emphasising the interpretant can draw from techniques in the field of semiotics. For instance, the consequences of each refinement made by users can be recorded; thereby enabling decisions to be reversed when desired changes to an ontology are not a monotonic refinement of the current ontology.

7. Conclusion

The contribution of this paper is motivation for and the technical development of a novel descriptive type system for Linked Data. This descriptive type system is a seamless combination of run-time schema inference and scripting languages that interact with Linked Data. The schema inference mechanism permits RDF Schema inference as one of several modes of inference, where inference is performed by refining the type system itself at run-time. This descriptive type system is quite different from a traditional prescriptive type system, since, in this descriptive type system, types change to describe the data, whereas traditionally types fix static information about data.

Changes are made to data in a controlled fashion such that the descriptive type system can still benefit from many properties enjoyed by a traditional prescriptive type system: in particular, when part of a system is well typed, it remains well typed, and hence no more inferences need be applied to that part of the system. Furthermore, when part of the system is not well-typed, instead of blindly applying one mode of inference, a menu of options is generated and the options are presented as a warning rather than an error. This gives the ability to the programmer to inspect the suggestions to decide whether they make sense conceptually, thereby possibly identifying mistakes in the data or more general inferences than assumed by RDF Schema by default.

We bring a number of type theoretic results to the table. We establish the consistency of subtyping through a cut elimination result (Theorem 1). We are able to tightly integrate RDF schema with executable scripts that dereference and query Linked Data. This is formalised by a type system that we prove is algorithmic (Proposition 4) — hence suitable for inference — and monotonic (Lemma 7) — hence permissive of refinements to the type system itself. We specify the run-time behaviour of scripts using an operational semantics, and prove a subject reduction result (Theorem 12) that proves that well-typed scripts do not raise unnecessary warnings; and hence no new inference need be applied. We also provide an algorithm for solving systems of constraints to generate warnings at run-time.

A subtle point regarding the W3C recommendations themselves is discussed in Section 4.4. In particular, we make recommendations for handling scenarios where multiple domain and range properties appear. The handling of multiple domains and ranges is the only significant technical discrepancy between the standard and descriptive type based approaches. We highlight how three major real life ontologies use three different approaches to handling this discrepancy, which should cover the needs of ontology engineers.

The Web is an open world of subjective knowledge, where it is impossible to globally agree schema a priori. Our descriptive type system assists with subjective decisions that resolve inconsistencies between the data and schema information a posteriori.

Acknowledgements. The first and second authors were supported by a grant of the Romanian National Authority for Scientific Research, project number PN-II-ID-PCE-2011-3-0919. The first author also received support from Ministry of Education Singapore Tier 2 grant MOE2014-T2-2-076 and British Council and Newton–Al-Farabi Partnership Programme travel grant 165901157. We thank the anonymous reviewers for their careful reading and comments.

References

[1] Malcolm P Atkinson and O Peter Buneman. Types and persistence in database programming languages. *ACM Computing Surveys (CSUR)*, 19(2):105–170, 1987.

[2] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: Syntax and semantics. *Inf. Comput.*, 119(2):202–230, 1995.

[3] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. RDF 1.1 turtle: Terse RDF triple language. Recommendation REC-turtle-20140225, W3C, 2014.

[4] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. *ACM SIGPLAN Notices*, 38(9):51–63, 2003.

[5] Sacha Berger, Emmanuel Coquery, Włodzimierz Drabent, and Artur Wilk. Descriptive typing rules for Xcerpt. In *Principles and Practice of Semantic Web Reasoning*, pages 85–100. Springer, 2005.

[6] Tim Berners-Lee. Linked data. *International Journal on Semantic Web and Information Systems*, 4(2):1, 2006.

[7] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia – a crystallization point for the Web of Data. *Web Semantics: science, services and agents on the world wide web*, 7(3):154–165, 2009.

[8] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1247–1250. ACM, 2008.

[9] Dan Brickley and Ramanathan V. Guha. RDF Schema 1.1. Recommendation REC-rdf-schema-20140225, W3C, 2014.

[10] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsson Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[11] Luca Cardelli, Simone Martini, John C Mitchell, and Andre Scedrov. An extension of system F with subtyping. In *Theoretical Aspects of Computer Software*, pages 750–770. Springer, 1991.

[12] Luca Cardelli and John C Mitchell. Operations on records. *Mathematical structures in computer science*, 1(01):3–48, 1991.

[13] Gabriel Ciobanu, Ross Horne, and Vladimiro Sassone. Descriptive types for linked data resources. In Andrei Voronkov and Irina Virbitskaite, editors, *Perspectives of Systems Informatics, 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27*, volume 8974 of *LNCS*, 2015.

[14] Gabriel Ciobanu, Ross Horne, and Vladimiro Sassone. Minimal type inference for linked data consumers. *Journal of Logical and Algebraic Methods in Programming*, 84(4):485–504, 2015.

[15] Sylvain Conchon and François Pottier. JOIN(X): Constraint-based type inference for the join-calculus. In *Programming Languages and Systems*, pages 221–236. Springer, 2001.

[16] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 concepts and abstract syntax. Recommendation REC-rdf11-concepts-20140225, W3C, 2014.

[17] Oege de Moor, Damien Sereni, Pavel Avgustinov, and Mathieu Verbaere. Type inference for datalog and its application to query optimisation. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 291–300. ACM, 2008.

[18] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[19] Mariangiola Dezani-Ciancaglini, Ross Horne, and Vladimiro Sassone. Tracing where and who provenance in Linked Data: A calculus. *Theor. Comput. Sci.*, 464:113–129, 2012.

[20] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz–open source graph drawing tools. In *Graph Drawing*, pages 483–484. Springer, 2002.

[21] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 137–146. IEEE, 2002.

[22] Thom Fruhwirth, Ehud Shapiro, Moshe Y Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Logic in Computer Science, 1991. LICS'91., Proceedings of Sixth Annual IEEE Symposium on*, pages 300–309. IEEE, 1991.

[23] Shudi Gao, C. M. Sperberg-McQueen, and Henry S. Thompson. W3C XML Schema Definition Language (XSD) 1.1 part 1: Structures. Recommendation REC-xmlschema11-1-20120405, W3C, 2012.

[24] Steve Harris and Andy Seaborne. SPARQL 1.1 query language. Recommendation REC-sparql11-query-20130321, W3C, MIT, MA, 2013.

[25] Patrick J. Hayes and Peter F. Patel-Schneider. RDF 1.1 Semantics. Recommendation REC-rdf11-mt-20140225, W3C, 2014.

[26] Phillip Heidegger and Peter Thiemann. Recency types for analyzing scripting languages. In *ECOOP 2010–Object-Oriented Programming*, pages 200–224. Springer, 2010.

[27] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. OWL 2 Web Ontology Language primer (second edition). Recommendation REC-owl2-primer-20121211, W3C, 2012.

[28] Ross Horne and Vladimiro Sassone. A verified algebra for read-write Linked Data. *Science of Computer Programming*, 89(A):2–22, 2014.
[29] Haruo Hosoya and Benjamin C Pierce. Xduce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.
[30] Yannis E. Ioannidis and Raghu Ramakrishnan. Efficient transitive closure algorithms. In *VLDB*, volume 88, pages 382–394, 1988.
[31] Svetlana Jakšić, Jovanka Pantović, and Silvia Ghilezan. Linked data privacy. *Mathematical Structures in Computer Science*, pages 1–21, 2015.
[32] Timothy Lebo, Satya Sahoo, Deborah McGuinness, Khalid Belhajjame, James Cheney, David Corsar, Daniel Garijo, Stian Soiland-Reyes, Stephan Zednik, and Jun Zhao. PROV-O: The PROV Ontology. Recommendation REC-prov-o-20130430, W3C, 2013.
[33] Lunjin Lu. Type analysis of logic programs in the presence of type definitions. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based program manipulation*, pages 241–252. ACM, 1995.
[34] Alejandro Mallea, Marcelo Arenas, Aidan Hogan, and Axel Polleres. On blank nodes. In *The Semantic Web–ISWC 2011*, pages 421–437. Springer, 2011.
[35] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(03):245–285, 1991.
[36] Rod Moten. Modelling the semantic web using a type system. In *Proceedings of Semantic Web Information Management on Semantic Web Information Management*, SWIM'14, pages 21:1–21:4, New York, NY, USA, 2014. ACM.
[37] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez. Simple and efficient minimal RDFS. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):220–234, 2009.
[38] Natalya F. Noy, Nigam H. Shah, Patricia L. Whetzel, Benjamin Dai, Michael Dorf, Nicholas Griffith, Clement Jonquet, Daniel L. Rubin, Margaret-Anne Storey, Christopher G. Chute, et al. Bioportal: ontologies and integrated data resources at the click of a mouse. *Nucleic acids research*, page gkp440, 2009.
[39] Natalya F. Noy, Michael Sintek, Stefan Decker, Monica Crubézy, Ray W. Fergerson, and Mark A. Musen. Creating semantic web contents with protege-2000. *IEEE intelligent systems*, (2):60–71, 2001.
[40] Jeff Z. Pan and Ian Horrocks. RDFS (FA) and RDF MT: Two semantics for RDFS. In *The Semantic Web-ISWC 2003*, pages 30–46. Springer Berlin Heidelberg, 2003.
[41] Peter F. Patel-Schneider. Analyzing schema.org. In *The Semantic Web–ISWC 2014*, pages 261–276. Springer, 2014.
[42] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009.
[43] David Peterson, Shudi Gao, Ashok Malhotra, C. M. Sperberg-McQueen, and Henry S. Thompson. W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. Recommendation REC-xmlschema11-2-20120405, W3C, 2012.
[44] Benjamin C. Pierce. *Types and programming languages*. MIT press, 2002.
[45] Jérôme Siméon and Philip Wadler. The essence of XML. *ACM SIGPLAN Notices*, 38(1):1–13, 2003.
[46] John F. Sowa. Conceptual structures: information processing in mind and machine. 1983.
[47] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of 16th WWW conference*, pages 697–706. ACM, 2007.
[48] Herman J. ter Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):79–115, 2005.
[49] Limsoon Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10(01):19–56, 2000.