Descriptive Types for Linked Data Resources

Horne, Ross; Sassone, Vladimiro; Ciobanu, Gabriel

2015

Ciobanu, G., Horne, R., & Sassone, V. (2015). Descriptive Types for Linked Data Resources. Lecture Notes in Computer Science, 8974, 1-25.

https://hdl.handle.net/10356/80953

https://doi.org/10.1007/978-3-662-46823-4_1

Descriptive Types for Linked Data Resources

Gabriel Ciobanu[1], Ross Horne[1,2], and Vladimiro Sassone[3]

[1] Romanian Academy, Institute of Computer Science, Blvd. Carol I, no. 8, Iaşi, Romania
gabriel@info.uaic.ro
[2] Kazakh-British Technical University, Faculty of Information Technology, Almaty, Kazakhstan
ross.horne@gmail.com
[3] University of Southampton, Electronics and Computer Science, Southampton, UK
vs@ecs.soton.ac.uk

Abstract. This work introduces the notion of descriptive typing. Type systems are typically prescriptive in the sense that they prescribe a space of permitted programs. In contrast, descriptive types assigned to resources in Linked Data provide useful annotations that describe how a resource may be used. Resources are represented by URIs that have no internal structure, hence there is no a priori type for a resource. Instead of raising compile time errors, a descriptive type system raises runtime warnings with a menu of options that make suggestions to the programmer. We introduce a subtype system, algorithmic type system and operational semantics that work together to characterise how descriptive types are used. The type system enables RDF Schema inference and several other modes of inference that are new to Linked Data.

1 Introduction

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 [2] is to use HTTP URIs to identify resources in data, e.g. *res:Andrey_Ershov*. 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.

To facilitate interoperability, Web standards are employed. Data is represented in a simple graph-based format called the Resource Description Framework (RDF) [7]. In RDF, there is a notion of a *type*, for example *res:Andrey_Ershov* may be assigned the type *dbp:SovietComputerScientist*. RDF Schema [5] provides some simple mechanisms for organising these types into *hierarchies*. RDF Schema also enables the types of resources to be *inferred* in certain contexts.

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 a system. In contrast, types in RDF change to describe the system instead of prescribing constraints on the system.

In this work, we provide a better answer to the question of the type-theoretic nature of types in Linked Data. We distinguish between a *prescriptive type system* and *descriptive type system*. In a traditional prescriptive type system, if a program is not well typed, then the program is rejected. In a descriptive type system [4], if a script is not well typed then the script can be executed but warnings are produced at runtime. The warnings present the user of the script with several options including expanding the type system itself to accommodate the script. Options can be selected interactively at any point during the execution of a script. Note that the user of a script is usually the programmer in this setting.

In Section 2, 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.

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.

In Section 5, we specify the behaviour of scripts using a novel operational semantics. The operational semantics allows us to refine the type system during execution in response to warnings. The operational semantics also enables us to formalise the examples in the motivating section. 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.

2 Motivation for Descriptive Typing for Linked Data

We 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 rdf:type dbp:IceHockeyPlayer .

In the electronic version of this paper the above URIs are dereferenceable, allowing the reader to follow the links. 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.

[4] 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 [6]. The development of descriptive types for the full SPARQL 1.1 specification will be given in a forthcoming paper by the above authors.

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 [3], that *dbp:IceHockeyPlayer* is a subtype of *dbp:Person*. Hence the query is considered to be well typed. The query produces the result `$place` $\mapsto$ *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 [14] 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.

`IntersectionOf`(*dbp:IceHockeyPlayer*, *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 [4] and DBpedia [3] 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 in Section 5, after developing the necessary technical apparatus.

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 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 [6].

In this work, we draw inspiration from RDF Schema [5], which concerns types that classify resources. In RDF Schema, 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.

Notice that basic types themselves are also URIs. We use these basic types as the *atomic types* of our type system. The full syntax for types is presented in Fig. 1.

$$\begin{array}{rlr}
\mathit{Type} ::= & \mathit{atom} & \mathit{atomic\ type} \\
& \mathit{owl{:}Thing} & \mathit{top\ type} \\
& \mathtt{IntersectionOf}(\mathit{Type}, \mathit{Type}) & \mathit{intersection\ type} \\
& \mathtt{UnionOf}(\mathit{Type}, \mathit{Type}) & \mathit{union\ type} \\
& \mathtt{Property}(\mathit{Type}, \mathit{Type}) & \mathit{property\ type}
\end{array}$$

Fig. 1. The syntax of descriptive types.

In Fig. 1, there are three type constructors, namely intersection, union, and property types. There is also a top type *owl:Thing* that represents the type of all resources.

Intersection types. The intersection type constructor is used to combine several types. For example, acording 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* .

$$\mathtt{IntersectionOf}\,(\,\mathit{yago{:}SovietComputerScientists}\,,\ \mathit{yago{:}FellowsOfTheBritishComputerSociety}\,)$$

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* .

Property types. The property type is inspired by RDF Schema [5], which enables the domain and range of a property to be declared. In RDF [7], the basic unit of data is a triple, such as:

res:Andrei_Yershov dbp:birthPlace res:Voskresensk .

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 is assigned the following type.

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

In RDF Schema, this type represent 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:Andrei_Yershov 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.

`Property`(*dbp:Person*, `UnionOf`(*dbp:Settlement*, *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*.

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 [16] has no type other than *owl:Thing*.

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

yago:Andrei_Ershov yago:linksTo yago:Random_access_machine

The property *yago:linksTo* is very general, relating any resource to any resource, as indicated by the property type `Property`(*owl:Thing*, *owl:Thing*).

Notice that the syntax of types is liberally expressive. We can express type that are both atomic types 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. 2, determines when a resource of one type can be used as a resource of another type. This relation is important for both typing systems and for refining the type system itself, in response to warnings.

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

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

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

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

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

$$\frac{}{\vdash C \leq \textit{owl:Thing}}\ \textit{top} \qquad \frac{a \leq b \in \mathrm{SC}^*}{\vdash a \leq b}\ \textit{atoms} \qquad \frac{\vdash C \leq D \quad \vdash D \leq E}{\vdash C \leq E}\ \textit{cut}$$

Fig. 2. 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.

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

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

These inequalities are inspired by the *rdfs:subClassOf* property from RDF Schema, which defines a reflexive transitive relation [14]. 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^* is efficient to calculate using techniques such as the Dedekind-MacNeille completion [13].

The relation SC^* is used in the axiom *atoms* in Fig. 2. The *top* axiom states that every resource is of type *owl:Thing* .

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

$$\vdash \begin{array}{l}\texttt{Property(}\ \textit{dbp:Person}\ ,\\ \qquad\qquad \textit{dbp:PopulatedPlace}\ \texttt{)}\end{array} \leq \begin{array}{l}\texttt{Property(}\ \textit{dbp:Person}\ ,\\ \qquad\qquad \texttt{UnionOf}(\textit{dbp:Settlement},\ \textit{dbp:Country}))\end{array}$$

The proof 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 constraint is solved by applying the *least upper bound* rule and *atoms* rule. The least upper bound rule generates two inequalities between atomic types that were declared to be in SC^* above. The above subtype inequality between properties suggests that a property with range *dbp:PopulatedPlace* can also range over resources that are settlements or countries.

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.

$$\vdash \begin{array}{l}\texttt{IntersectionOf}\ (\ \textit{yago:SovietComputerScientists}\ ,\\ \qquad\qquad\qquad \textit{yago:FellowsOfTheBritishComputerSociety}\ )\end{array} \leq \textit{dbp:Person}$$

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

The cut rule. A subtype relation is expected to be a transitive relation. To prove that subtyping is transitive, we include 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. In proof theory, this consistency result is known as *cut elimination*.

Theorem 1 (Cut elimination). *The cut rule can be eliminated from the given subtype system 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 axioms for atoms. 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 atom axioms.

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

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.

$$\left[\!\!\left[\frac{\begin{array}{c}\pi\\ \vdash C \leq D\end{array} \quad \frac{}{\vdash D \leq \mathit{owl{:}Thing}}}{\vdash C \leq \mathit{owl{:}Thing}}\right]\!\!\right] \longrightarrow \frac{}{\vdash C \leq \mathit{owl{:}Thing}}$$

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.

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

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.

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

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

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.

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

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.

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

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.

$$\left[\!\!\left[\dfrac{\dfrac{\begin{array}{c}\pi_0\\ \vdash D_0 \leq C_0\end{array} \quad \begin{array}{c}\pi'_0\\ \vdash D_1 \leq C_1\end{array}}{\vdash \texttt{Property}(C_0, C_1) \leq \texttt{Property}(D_0, D_1)} \quad \dfrac{\begin{array}{c}\pi_1\\ \vdash E_0 \leq D_0\end{array} \quad \begin{array}{c}\pi'_1\\ \vdash D_1 \leq E_1\end{array}}{\vdash \texttt{Property}(D_0, D_1) \leq \texttt{Property}(E_0, E_1)}}{\vdash \texttt{Property}(C_0, C_1) \leq \texttt{Property}(E_0, E_1)} \right]\!\!\right]$$

$$\longrightarrow \quad \dfrac{\left[\!\!\left[\dfrac{\begin{array}{c}\pi_1\\ \vdash E_0 \leq D_0\end{array} \quad \begin{array}{c}\pi_0\\ \vdash D_0 \leq C_0\end{array}}{\vdash E_0 \leq C_0} \right]\!\!\right] \quad \left[\!\!\left[\dfrac{\begin{array}{c}\pi'_0\\ \vdash E_1 \leq D_1\end{array} \quad \begin{array}{c}\pi'_1\\ \vdash D_1 \leq C_1\end{array}}{\vdash E_1 \leq C_1} \right]\!\!\right]}{\vdash \texttt{Property}(C_0, C_1) \leq \texttt{Property}(E_0, E_1)}$$

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 mutiset ordering argument [8] 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. □

Cut elimiation 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 1. *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.*

Theorem 1 and Proposition 1 are sufficient to establish the consistency of the subtype system.

Our subtype system is closely related to the functional programming language with intersection and union types presented by Barbanera et al. [1]. Our subtype system without properties coincides with the subtype system of Barbanera et al. without implication. Furthermore, properties can be encoded using implication, so our system is a restriction of the system presented in [1].

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 [6]; which is based on process calculi presented in [9,12]. 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. 3. Terms in the language are *URIs* which are identifiers for resources, or *variables* of the form `$x`. RDF triples [7] 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.

$$
\begin{array}{lll@{\qquad}lll}
\mathit{term} & ::= & \mathit{variable} \mid \mathit{uri} & \mathit{script} & ::= & \texttt{ok} \\
 & & & & \mid & \texttt{where}\ \mathit{term}\ \mathit{term}\ \mathit{term}\ \mathit{script} \\
\mathit{data} & ::= & \mathit{term}\ \mathit{term}\ \mathit{term} & & \mid & \texttt{from}\ \mathit{term}\ \mathit{script} \\
 & \mid & \mathit{data}\ \mathit{data} & & \mid & \texttt{select}\ \mathit{variable} : \mathit{type}\ \mathit{script}
\end{array}
$$

Fig. 3. The syntax of scripts and data.

The keyword `from` *res:Andrei_Yershov* 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 [6] 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 [12] for the analysis of more expressive query languages based on SPARQL [10,15]).

4.2 An Algorithmic Type System for Scripts and Data

We type scripts for two purposes. Firstly, if the script is correctly typed, then we can check that it is well typed and execute the script without throwing any warnings. However, if the script is not well typed, we can use the type system as the basis of an algorithm for generating warnings. Scripts and data are typed using the system presented in Fig. 4. There are typing rules for each form of term, script and data.

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

$$\frac{\mathrm{Env}, \texttt{\$x}\colon \textit{type} \vdash \textit{script}}{\mathrm{Env} \vdash \texttt{select}\ \texttt{\$x}\colon \textit{type}\ \textit{script}}\;\textit{select} \qquad \frac{\mathrm{Env} \vdash \textit{script}}{\mathrm{Env} \vdash \texttt{from}\ \textit{uri}\ \textit{script}}\;\textit{from}$$

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

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

$$\frac{\mathrm{Env} \vdash \textit{data}_0 \quad \mathrm{Env} \vdash \textit{data}_1}{\mathrm{Env} \vdash \textit{data}_0\ \textit{data}_1}\;\textit{compose} \qquad \frac{\mathrm{Env} \vdash \textit{term}\colon \textit{atom}}{\mathrm{Env} \vdash \textit{term}\ \ \textit{rdf:type}\ \ \textit{atom}}\;\textit{RDF type}$$

$$\frac{\mathrm{Env} \vdash \textit{term}\colon D \quad \vdash C \leq D}{\mathrm{Env} \vdash \textit{term}\colon D}\;\textit{subsumption}$$

Fig. 4. 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 can be a person even though his type is the intersection of several professions, as illustrated in the previous section.

The type rule for triples assumes 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 URI *rdf:type* in the property position are treated differently from other triples. In the *RDF type* 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 *type ascription*. Further special type ascription rules could be added to extract more refined types based on OWL [11]; however the rule for atomic types is sufficient for the examples in this paper.

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 `$x`: C. As standard, 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 [6], 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. The *subsumption* rule relaxes the type of a term at any point in a typing derivation. Since we can apply subsumption repeatedly it could give rise to type derivations of an unbounded size. By showing that subsumption can be eliminated, we establish that the type system without subsumption is an *algorithmic type system.*

Proposition 2 (Algorithmic typing). *For any type assumption that can be derived using the type system, 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. $$\dfrac{\dfrac{\text{Env} \vdash \textit{term}\colon C \quad \vdash C \leq D}{\text{Env} \vdash \textit{term}\colon D} \qquad \vdash D \leq E}{\text{Env} \vdash \textit{term}\colon E} \quad \text{yields} \quad \dfrac{\text{Env} \vdash \textit{term}\colon C \quad \vdash C \leq E}{\text{Env} \vdash \textit{term}\colon E}$$

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

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

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 1. *When we would like to be explicit about the subtype assumptions SC and type assumptions Ty used in a type judgement Env $\vdash$ script and subtype judgement $\vdash C \leq D$, we use the following notation:*

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

We define a refinement relation $(Ty', SC') \leq (Ty, SC)$, 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. The lemmas establish that anything that is well typed remains well typed in a refined type system.

Lemma 1. *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 2. *The following monotonicity properties hold for scripts and data respectively.*

1. *If $\vdash_{SC}^{Ty}$ script and $(Ty', SC') \leq (Ty, SC)$, then $\vdash_{SC'}^{Ty'}$ script.*
2. *If $\vdash_{SC}^{Ty}$ data and $(Ty', SC') \leq (Ty, SC)$, then $\vdash_{SC'}^{Ty'}$ 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 1, $\vdash_{\mathrm{SC}'} D \leq E$. 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 1, 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.

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

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. □

5 An Operational Semantics Aware of Descriptive Types

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.

5.1 The Operational Semantics

The rules of the operational semantics are presented in Fig. 5. 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 such that the composition of data is associative and commutative.

Configurations. A configuration (*script*, *data*, Ty, SC) represents the state that can change during the execution of a program. 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.

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

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

$$\frac{}{\begin{array}{l}(\ \texttt{where}\ \mathit{term}_0\ \mathit{term}_1\ \mathit{term}_2\ \mathit{script},\\ \quad \mathit{term}_0\ \mathit{term}_1\ \mathit{term}_2\ \mathit{data}, \mathrm{Ty}, \mathrm{SC})\end{array} \longrightarrow (\mathit{script}, \mathit{term}_0\ \mathit{term}_1\ \mathit{term}_2\ \mathit{data}, \mathrm{Ty}, \mathrm{SC})}\ \mathit{where}$$

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

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

Fig. 5. The operational semantics for scripts. Note that, in the *from* rule, data_1 is the data obtained at runtime by dereferencing the resource *uri*.

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 rdf:type dbp:PopulatedPlace

– some type assumptions Ty such that:

$$\begin{aligned}\mathrm{Ty}(\mathit{dbp{:}birthPlace}) &= \texttt{Property}(\ \mathit{dbp{:}Person},\\ &\qquad\qquad \mathit{dbp{:}PopulatedPlace}\ )\\ \mathrm{Ty}(\mathit{res{:}Andrey\_Ershov}) &= \mathit{owl{:}Thing}\\ \mathrm{Ty}(\mathit{res{:}SovietUnion}) &= \mathit{dbp{:}PopulatedPlace}\end{aligned}$$

– 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 the resource with URI *dbp:Andrey_Ershov* . This includes triples such as:

> *res:Andrei_Ershov rdf:type 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 *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:

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

The first option above is the default option taken by RDF Schema [5]. 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 choice of option is subjective, so 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:

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

– some data $data_0$ $data_1$ including the new data obtained by dereferencing the resource *dbp:Andrey_Ershov* ;
– a refined type assumption Ty′, such that:

$$\mathrm{Ty}'(\mathit{res{:}Andrey\_Ershov}) = \mathit{yago{:}FellowsOfTheBritishComputerSociety}$$

– the refined subtype assumptions SC′ suggested in the third option above.

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.

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

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

`where` *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.

Example of a bad script. Now consider the example in the motivating section. We begin with the following configuration, where the wrong URI has been used for Ershov:

– the following script:

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

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

free:book.author.works_written rdfs:domain free:book.author
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.

$$\texttt{Property}(\textit{owl:Thing}, \textit{free:book})$$

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.

res:Andrei_Yershov rdf:type 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 *yago:IceHockeyPlayer* and *dbp:Person* as its type.
2. Refine the type of the property *free:book.author.works_written* such that it is of the following type.

```
IntersectionOf( Property( free:book.author,
                          free:book ),
                Property( dbp:IceHockeyPlayer,
                          free:book ) )
```

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.

5.2 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. 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:

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

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.

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

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

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

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

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

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

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

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

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

We use these upper bounds to generate the options that appear in warnings.

Maximise type of property. To generate the first option Ty_1, we maximise the type of properties by ensuring that Ty(*free:book.author.works_written*) is equal to the upper bound on the property. This yields the following type inequality.

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

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 constraint $X \leq \mathit{free{:}book.author}$. From this constraint we derive the most general solution $X \mapsto \mathit{free{:}book.author}$. Thereby we obtain a refined type system such that.

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

The above is exactly what RDF Schema would infer [14].

Maximise type of subject/object. To generate the second option Ty_2, we maximise the type of the subject be setting $\mathrm{Ty}(\mathit{res{:}Andrei\_Yershov})$ to be equal to the upper bound on the resource. This yields the following type inequality.

$$\mathit{dbp{:}IceHockeyPlayer} \leq \mathtt{IntersectionOf}(\mathit{dbp{:}IceHockeyPlayer}, X)$$

As in the previous example, we unfold the rules of the algorithmic type system to derive the constraint $\mathit{dbp{:}IceHockeyPlayer} \leq X$. We then maximise the type of the property with respect to this constraint, yielding the most general solution $X \mapsto \mathit{dbp{:}IceHockeyPlayer}$. Thereby we obtain a refined type system such that.

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

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.

Let $\mathrm{Ty}_1 = \mathrm{Ty}_2$ if and only if $(\mathrm{Ty}_1, \mathrm{SC}) \leq (\mathrm{Ty}_2, \mathrm{SC})$ and $(\mathrm{Ty}_2, \mathrm{SC}) \leq (\mathrm{Ty}_1, \mathrm{SC})$. Now $\mathrm{Ty}_1 = \mathrm{Ty}_2$ if and only if the following equalities hold.

$$\mathit{dbp{:}IceHockeyPlayer} = \mathtt{IntersectionOf}(\mathit{dbp{:}IceHockeyPlayer},\ \mathit{free{:}book.author})$$

$$\begin{array}{c} \mathtt{Property}(\ \mathit{free{:}book.author}\ ,\\ \mathit{free{:}book}\ ) \end{array} = \begin{array}{l} \mathtt{IntersectionOf}(\\ \ \mathtt{Property}(\mathit{free{:}book.author},\ \mathit{free{:}book})\ ,\\ \ \mathtt{Property}(\mathit{dbp{:}IceHockeyPlayer},\ \mathit{free{:}book})\ ) \end{array}$$

By using the cut-free subtype system, we can calculate that the above equalities hold only if the following subtype inequality holds.

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

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, we are led to the constraint.

$$\mathit{owl{:}Thing} \leq \mathit{free{:}book.author}$$

The above inequality cannot be induced by extending the set of subtype assumption, 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. 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 for upper bounds on Ty'.
2. We generate the first option by, for every property, setting the type in Ty to be equal to the upper bound on constraints. We then solve the system of equalities using unification to obtain a suitable unifier. This is used to obtain solution Ty_1.
3. We generate the second option by, for every subject and object, setting the type in Ty to be equal to the upper bound on constraints. Again we solve the system of constraints using unification to obtain Ty_2.
4. We set $\mathrm{Ty}_1 = \mathrm{Ty}_2$ and solve the system of equalities to obtain a set of subtype inequalities over atomic types. If there is a solution, we extend SC with these constraints and fix Ty.

The second point above generates classes for resources as expected by RDF Schema [5]; hence RDF Schema is sound with respect to our descriptive type system. The third and fourth points above provide alternative, more general modes of inference. Thus the above algorithm extends RDF Schema inference.

If there is a solution to the fourth point above with an empty set of subtype inequalities, then the script can be typed without refining the type system. In this case, the constraints could be solved efficiently, using techniques in [17]. Further analysis of the above algorithm is future work.

5.3 Subject Reduction

There are two reasons why a system is well-typed. Either *a priori* the script was well typed before changing the type system, or at some point during the execution the programmer acted to resolve the 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 guarantees after choosing to resolve a warnings, by selecting the optional rule. In particular, after choosing to resolve warnings the script is also well-typed with respect to the refined type system.

Proposition 3. *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 2, $\vdash^{\mathrm{Ty}'}_{\mathrm{SC}'}$ *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 the variable for a resource of that type and preserve typing.

Lemma 3. *Assume that* $\vdash uri\colon C$. *Then the following statements hold:*

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

Proof. Assume that $\vdash uri\colon C$. 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.

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

Hence, by Proposition 2, Env $\vdash uri\colon D$ 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 Env, `$x`: $C \vdash$ `select $y`: $D\,script$ holds. If `$x` = `$y`, then `$x` does not appear free in `select $x`: D, hence Env $\vdash$ `select $x`: D *script* as required. If `$x` $\neq$ `$y`, then, by the induction hypothesis, if Env, `$x`: C, `$y`: $D \vdash script$ holds then Env, `$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.

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

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

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

The cases for other rules follow immediately by induction. □

We also require the following result, the proof of which is straightforward.

Lemma 4. *Assume that* $data_0 \equiv data_1$. *If* $\vdash data_0$ *then* $\vdash data_1$.

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

Theorem 2 (Subject reduction). *If* $\vdash^{Ty}_{SC} script$ *and* $\vdash^{Ty}_{SC} data$, *then if*

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

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

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

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

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

The above holds only if $\mathtt{\$x}\colon C \vdash \mathit{script}$, by the type rule for `select`. By Lemma 3, since $\mathtt{\$x}\colon C \vdash \mathit{script}$ and $\vdash \mathit{uri}\colon C$, it holds that $\vdash \mathit{script}\left\{{}^{\mathit{uri}}/_{x}\right\}$. Therefore the `select` rule preserves types.

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

$$\vdash \mathtt{where}\, \mathit{term}_0\ \mathit{term}_1\ \mathit{term}_2\ \mathit{script} \qquad \vdash \mathit{term}_0\ \mathit{term}_1\ \mathit{term}_2\ \mathit{data}$$

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

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

$$\vdash^{\mathrm{Ty}}_{\mathrm{SC}} \mathtt{from}\, \mathit{uri}\ \mathit{script} \qquad \vdash^{\mathrm{Ty}}_{\mathrm{SC}} \mathit{data}_0 \qquad \vdash^{\mathrm{Ty}'}_{\mathrm{SC}'} \mathit{data}_1 \qquad (\mathrm{Ty}', \mathrm{SC}') \leq (\mathrm{Ty}, \mathrm{SC})$$

The first assumption above holds only if $\vdash^{\mathrm{Ty}}_{\mathrm{SC}} \mathit{script}$ holds, by the type rule for `from`. Since $(\mathrm{Ty}', \mathrm{SC}') \leq (\mathrm{Ty}, \mathrm{SC})$, by Lemma 2, $\vdash^{\mathrm{Ty}'}_{\mathrm{SC}'} \mathit{script}$ holds. By Lemma 2 again, $\vdash^{\mathrm{Ty}'}_{\mathrm{SC}'} \mathit{data}_0$ holds. Hence $\vdash^{\mathrm{Ty}'}_{\mathrm{SC}'} \mathit{data}_0\ \mathit{data}_1$ holds. Therefore the `from` rule preserves types.

Consider the case of the *optional* operational rule. For some initial configuration $(\mathit{script}, \mathit{data}, \mathrm{Ty}, \mathrm{SC})$, we assume that $\vdash^{Ty}_{\mathrm{SC}} \mathit{data}$. The result then follows from Prop. 3. □

6 Conclusion

The descriptive type system introduced in this work formalises the interplay between runtime schema inference and scripting languages that interact with Linked Data. The system formalises how to build RDF Schema inference into scripts at runtime. The system also permits new inference mechanisms that refine the types assigned to properties and extend the subtype relation.

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 2), hence suitable for inference. We specify the runtime behaviour of scripts using an operational semantics, and prove a subject reduction result (Theorem 2) that proves that well typed scripts do not raise unnecessary warnings.

We also provide an algorithm for solving systems of constraints to generate warnings at runtime. This suggests a line of future work to investigate the optimality of the algorithm presented. The descriptive type system can be employed in expressive scripting languages [12], and extract more type information based on RDF Schema and OWL from data. This descriptive type system can coexist with a prescriptive type system for simple data types as presented in [6].

A subjective question is the following. At what point does the programmer stop ignoring the warnings and become suspicious? Many programmers are likely to ignore warnings until the script stops working. At this point, they will inspect the warnings and, based on their subjective human judgement, decide whether suggestions are consistent or conflicting. Most programmers will be happy to let fellows of the British Computer Society be people, but will have second thoughts about letting all ice hockey players be authors. The Web is an open world of subjective knowledge. Our descriptive type system assists subjective decisions that keep data and schema information consistent.

Acknowledgements. We are grateful to the organisers of the Ershov memorial conference, PSI 2014, for inviting this work to be presented as a keynote speech. The work of the first and second authors was supported by a grant of the Romanian National Authority for Scientific Research, project number PN-II-ID-PCE-2011-3-0919.

References

1. Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: Syntax and semantics. *Inf. Comput.*, 119(2):202–230, 1995.
2. Tim Berners-Lee. Linked data. *International Journal on Semantic Web and Information Systems*, 4(2):1, 2006.
3. Christian Bizer et al. 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.
4. 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.
5. Dan Brickley and Ramanathan V. Guha. RDF vocabulary description language 1.0: RDF Schema. Edited recommendation PER-rdf-schema-20140109, W3C, 2014.
6. Gabriel Ciobanu, Ross Horne, and Vladimiro Sassone. Local type checking for linked data consumers. In António Ravara and Josep Silva, editors, *WWV*, volume 123 of *EPTCS*, pages 19–33, 2013.
7. Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 concepts and abstract syntax. Recommendation REC-rdf11-concepts-20140225, W3C, 2014.
8. Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.
9. 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.
10. Steve Harris and Andy Seaborne. SPARQL 1.1 query language. Recommendation REC-sparql11-query-20130321, W3C, MIT, MA, 2013.
11. Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. OWL 2 Web Ontology Language primer (second edition). Recommentation REC-owl2-primer-20121211, W3C, 2012.
12. Ross Horne and Vladimiro Sassone. A verified algebra for read-write Linked Data. *Science of Computer Programming*, 89(A):2–22, 2014.
13. Holbrook Mann MacNeille. Extensions of partially ordered sets. *Proceedings of the National Academy of Sciences of the United States of America*, 22(1):45–50, 1936.
14. 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.
15. Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009.
16. 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.
17. Jerzy Tiuryn. Subtype inequalities. In *Logic in Computer Science, 1992. LICS'92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 308–315. IEEE, 1992.