A Typed Model for Linked Data

ROSS HORNE AND VLADIMIRO SASSONE

Electronics and Computer Science, University of Southampton, United Kingdom
Email: {rjh06r,vs}@ecs.soton.ac.uk

The term Linked Data is used to describe ubiquitous and emerging semi-structured data formats on the Web. URIs in Linked Data allow diverse data sources to link to each other, forming a Web of Data. A calculus which models concurrent queries and updates over Linked Data is presented. The calculus exhibits operations essential for declaring rich atomic actions. The operations recover emergent structure in the loosely structured Web of Data. The calculus is executable due to its operational semantics. A light type system ensures that URIs with a distinguished role are used consistently. The main theorem verifies that the light type system and operational semantics work at the same level of granularity, so are compatible.

Examples show that a range of existing and emerging standards are captured. Data formats include RDF, named graphs and feeds. The primitives of the calculus model SPARQL Query and the Atom Publishing Protocol. The subtype system is based on RDFS, which improves interoperability. Examples focus on the SPARQL Update proposal for which a fine grained operational semantics is developed. Further potential high level languages are outlined for exploiting Linked Data.

Keywords: operational semantics, type systems, semi-structured data

1. INTRODUCTION

The View is one component in the Model-View-Controller architecture, which is widely adopted for application development. Another component, the Model, provides data which forms the subject of the application. The Controller coordinates interactions with the Model to achieve some objective. Having successfully moved the View onto the Web, standards bodies are tackling the problem of moving the Model onto the Web. The claim is that moving the Model onto the Web allows common subject matter to be shared between applications. Evidence of the potential of sharing data on the Web is the ubiquity of feeds, e.g., RSS and Atom [1], which deliver data on demand between news sources and consumers.

Making data available on the Web gives the potential for data to link across traditional boundaries. This is enabled by using the URI as a standardised naming system. By naming a resource with a URI the resource can be referred to from any other location. Efforts to exploit these links between data sources have resulted in several proposed standards. The common aim of proposed standards is often referred to as establishing a Web of Data [2]. Data which exploits the link structure of the Web is distinguished as Linked Data. Linked Data is supported by W3C recommendations and working drafts, which reflect a consensus on the aims of the initiative [3, 4, 5]. This work draws from key standards for Linked Data and presents an executable model in which the standards coexist.

At a low level, Linked Data is delivered as messages in a semi-structured data format. The Resource Description Format (RDF) is the standardised semi-structured data format for Linked Data [3]. At this level, an HTTP request to a URI produces some RDF which describes the resource represented by the URI. No requirements are enforced on how the RDF is produced or what is done with the RDF. Message passing on channels is modelled by many process calculi [6, 7, 8, 9].

At a higher level, Linked Data can be gathered in stores which are accessed using queries. A store responds to queries as prescribed by the SPARQL Query standard [10]. Rich data sources are now published as stores, notably the UK Government Data and DBpedia [11, 12, 13]. These examples gather data, from UK Government Databases and Wikipedia respectively, then prepare the data for queries. No requirements are placed on the method of preparation. SPARQL Query has been modelled as a graph query language and using relational algebra [14, 15].

The executable model presented here is tailored to problems introduced by an update mechanism. Challenges associated with updates are highlighted by Tim Berners-Lee in a note on Read-Write Linked Data. Updates are considered at several levels of granularity. At a coarse granularity of update the contents of a store are replaced periodically. Periodic updates are adequate when data changes infrequently. An intermediate granularity is achieved by dividing a store into regions, where each region is updated independently. This idea is captured by named graphs for RDF [16]. A protocol for updating named graphs is under development [17]. Feeds and standardised protocols for feeds also work at a similar level of granularity [18, 19].

The primary challenge is to model fine grained updates at the level of triples. Triples are the basic components of RDF which resemble simple sentences in natural language of the form subject–verb–object. Fine grained updates account for exactly the triples required to perform an update. Updates which use disjoint triples may occur concurrently. By using minimal resources, an update causes minimal

interruption to a store. This approach avoids regions, which are difficult to design when the long term behaviour cannot be predicted. Fine grained updates are known to present conceptual difficulties, as highlighted by Reynolds in the traditional setting of shared memory [20]. The model is a contribution to the understanding of fined grained updates for Linked Data.

Implementing Read-Write Linked Data is necessary for using Linked Data in modern applications. For instance, in wikis or social media users increasingly write data. In contrast, existing Linked Data applications are limited to reading data. Furthermore, without an update mechanism for the Model, the Model and the Controller in the modern application architecture cannot be decoupled. A Controller instead requires lower level access to the Model to perform updates. This work therefore supports efforts towards a standardised approach to Read–Write Linked Data [21].

The structure of the paper. In Section 2, a core untyped calculus is presented. The core calculus motivates the basic unit of semi-structured data as resembling a simple sentence in natural language. A high level language for queries and updates recovers expressive power. Both the data format and the language are specified using an abstract syntax and operational semantics.

In Section 3, the untyped calculus is extended with key features. The motivation for the extensions are greater interoperability with existing semi-structured data formats and common uses cases such as provenance and syndication. The extra features are specified by extending the syntax and semantics of the core calculus.

In Section 4, light types for URIs and literals are introduced. The role of each type is explained and a subtype system specifies a partial order over types. A distinction between static and dynamic classes is highlighted.

In Section 5, the types and the calculus are combined. The type system specifies a type checker for data, queries and updates. A theorem verifies that the subtype system and the type system are compatible.

In Section 6, the operational semantics and the type system are combined. The typed operational semantics enable a feasible type system, by minimising runtime type checks. The type preservation theorem verifies that the type system and typed operational semantics are compatible.

In Section 7, further contributions enabled by the calculus are outlined. The type system forms a foundation for type inference. The operational semantics forms a foundation for an algebra of updates. Further features suggest high level languages for Linked Data. The demand is a more subtle notion of logic than first envisioned for the Web of Data.

2. THE CORE SYNTAX AND SEMANTICS

This section focuses the core of the key standards for Linked Data. In particular, the core of the semi-structured data format RDF and the core of the SPARQL Update language are captured. The approach is that of structural operational semantics. An abstract syntax is defined, then the operational semantics are defined by relations over the abstract syntax. Operational semantics specifies the runtime behaviour of a language.

The role of the abstract syntax differs from the role of common concrete syntaxes for RDF [16]. A concrete syntax is intended for human readability or message exchange. In contrast, the abstract syntax is for the purpose of compiler engineering. It captures the essence of the language, without redundancies. Connectives are chosen to highlight close connections to connectives in constructive logics. Brief examples of the concrete syntax are provided, then the abstract syntax is fully defined.

2.1. A Syntax for the Resource Description Framework

The Web is based on documents, represented by one URI, which link to other documents, represented by another URI. The link structure of the Web can therefore be represented by pairs of URIs. This link structure has been exploited by organisations such as Google [22]. The source and target URIs are the subject and the object of the link, respectively.

RDF extends the link structure of the Web to the power of simple sentences. In natural languages, a verb indicates how a subject is related to an object. In English for instance the structure of a simple sentence is subject–verb–object e.g. “Kleinberg writes Authoritative Sources in a Hyperlinked Environment.” RDF extends the link structure of the Web to include a predicate. The predicate serves the same role as a verb, by indicating the nature of the connection between a subject and an object.

Like the subject and the object, the predicate is also a URI. A URI is a standardised global identifier for any resource, so need not identify a document. Thus the URI of a predicate is just a global identifier from some vocabulary. Similarly, the URI of the subject and the object need not refer to a document. Instead the URI could provide a global reference to some resource which, in a traditional setting, would normally be a local identifier in a database. The following is an example of two triples.

```
soton:24123 foaf:knows soton:10511 .
soton:doc1 dc:creator soton:10511.
```

Note that soton:, foaf: and dc: represent URI prefixes `http://ecs.soton.ac.uk/rdf/`, `http://xmlns.com/foaf/0.1/` and `http://purl.org/dc/elements/1.1/` respectively. The first is a namespace used by Southampton University. The second and third are namespaces used for terminology in the Friend-of-a-Friend and Dublin Core metadata vocabularies.

Another notion generalised by RDF is the idea that a URI is associated with a document on the Web. RDF allows several pieces of traditional data to be associated with a URI. As with links between resources a predicate indicates how a URI is related to a piece of traditional data. Again this resembles simple sentences in natural language. The following is an example of two triples.

```
soton:doc1 dc:title "Tae a Link" .
soton:doc1 dc:description "Some poem."
```

$$
\begin{array}{rlrrlr}
o & ::= & v & \text{literal} & C & ::= & (a\ a\ o) & \text{triple} \\
 & | & x & \text{variable} & & | & C \parr C & \text{par} \\
 & | & a & \text{name} & & | & \bot & \text{nothing}
\end{array}
$$

FIGURE 1. A syntax for objects and RDF content.

$$C \parr \bot \equiv C \qquad C \parr D \equiv D \parr C$$

$$C \parr (D \parr E) \equiv (C \parr D) \parr E$$

FIGURE 2. The structural congruence for RDF content.

The examples above are written using the Turtle syntax for RDF. Turtle is one of several concrete languages for presenting RDF. Here an abstract syntax captures the essence of these formats, without redundancy.

2.1.1. An Abstract Syntax for RDF.
For the purpose of defining operational semantics, an abstract syntax for RDF is defined. The abstract syntax captures the idea that RDF is a collection of triples of the form subject–predicate–object, as presented in Fig. 1.

The atoms of the syntax are names, variables and literals. Names represent URIs. For simplicity of examples, names are any identifier in italics, e.g. *24123*, *knows*. Names bound by quantifiers represent place holders for URIs. Literals represent traditional data such as a string of characters or an integer, such as `'Authoritative Sources'` or `'7'`. Traditional data is well understood so, the technicalities of literals are left to the XML datatypes specification [23]. Variables are explicit place holders for literals.

A triple consists of a subject, a predicate and an object. The subject and predicate are names. The object is either a name, a literal or a variable. A URI as an object generalises the notion of a link between Web pages. Similarly, the use of a literal as an object generalises the notion of the document associated with a link. Triples are composed in parallel using the par operator. The following demonstrates two triples composed in parallel.

$$(\textit{doc1 creator Burns}) \parr (\textit{doc1 title}\ \texttt{'Links'})$$

Par is associative and commutative, so brackets and the order of triples can be ignored. The unit of par is nothing, which represents the absence of data. Content therefore forms a commutative monoid, as defined by the structural congruence in Fig. 2.

Note that the W3C recommendation describes how to obtain labelled directed graphs from the syntax of RDF [3]. This provides a denotational semantics, which is used by graph query languages [14]. In contrast, this work remains strictly at the level of syntax. Denotational semantics for concurrency are notoriously difficult [24].

2.2. A Syntax and Semantics for Queries and Updates

When data is published openly it is rarely possible to predict how it might be used. It is therefore difficult to decide a suitable format in which to publish the data. Preferably, the application which consumes the data should decide. For this reason RDF is a simple semi-structured data format. Power is regained from this minimal structure by an expressive query language. The query language enables the consumption of emergent structures conveyed in RDF. In this way the power is shifted from the producer to the consumer and lowers barriers to publishing Linked Data.

The language SPARQL Query is the agreed standard for the purpose of querying RDF. The first SPARQL Query standard has been widely deployed [5, 12]. A second draft for SPARQL Query learns from the experiences of the first [25]. A SPARQL end point is used to observe an RDF store. The observer declares the link patterns of interest using SPARQL Query. The query language also determines the format in which results are presented.

For instance an application may be interested the question, "Obtain names for either products related to the show or products related to an exhibitor at the show." The example scenario can be specified as follows in the SPARQL Query concrete syntax, where `eg:show2011` identifies the show and `eg:exhibitor` and `eg:product` identify predicates from some vocabulary.

```
SELECT ?product WHERE {
 {
  eg:show2011 eg:exhibitor ?exhibitor .
  ?exhibitor eg:product ?product
 }
 UNION
 { eg:show2011 eg:product ?product }
}
```

An analogy is that queries support compound sentences although RDF only supports simple sentences. Several simple sentences can verify the truth of a compound sentence. The returned result is witness to the veracity of the compound sentence in the given context. While the truth of RDF is subjective, the truth represented by a successful SPARQL Query is intersubjective [26]. For intersubjective truth there is an subjective agreement between multiple parties. The parties involved are the client that poses the query (compound sentence) and the providers of the triples (simple sentences).

Current SPARQL recommendations have no constructs for maintenance. However, there is a proposal by Hewlett-Packard Labs and a working draft for a language called SPARQL Update [10, 21]. The proposals allow RDF to be inserted and deleted at the level of triples. Update operations reuse the operations of SPARQL Query for powerful updates.

The example above can be extended to specify the update, "For either products related to the show or products related to an exhibitor at the show, insert a link from another show to that product." The extension is to add one clause to the

$$
\begin{array}{rcll}
\phi & ::= & \mathrm{I} & \text{true} \\
& | & 0 & \text{false} \\
& | & \phi \wedge \phi & \text{and} \\
& | & \phi \vee \phi & \text{or} \\
& | & \neg\phi & \text{not} \\
& | & \ldots & \text{etc.}
\end{array}
$$

$$
\begin{array}{rcll}
U & ::= & |C| & \text{ask} \\
& | & C^{\perp} & \text{delete} \\
& | & C & \text{insert} \\
& | & \phi & \text{filter} \\
& | & U \oplus U & \text{choice} \\
& | & U \otimes U & \text{join} \\
& | & *U & \text{iteration} \\
& | & \bigvee a.U & \text{select name} \\
& | & \bigvee x.U & \text{select literal}
\end{array}
$$

FIGURE 3. A syntax for constraints and updates.

query. The clause inserts a triple which relates the show to the discovered product.

A model for SPARQL Update is sufficient to model SPARQL Query. However, a model for SPARQL Update is more subtle than a model for SPARQL Query. Not only is the truth represented by a successful intersubjective interaction, it is also dependent on time.

2.2.1. An abstract syntax for updates.
An updates is a declarative specification of the intention of the programmer. The meaning of an update is independent of a particular implementation. An application expects some behaviour and an implementation provides a behaviour within the bounds of expectation. The common interface between the application and the store is the syntax of the language. An abstract syntax for Updates is provided in Fig. 3.

Basic queries are formed by embedding the syntax of RDF. The embedding '*ask*' is used to demand that a some RDF is matched. This models asking a query, which has no side effects. The following is an example of asking a query which is satisfied by the example RDF in Section 2.1.1.

$$|(\textit{doc1 creator Burns}) \parr (\textit{doc1 title } \texttt{`Links'})|$$

Basic updates are also formed by embedding the syntax of RDF in two ways. The embedding '*delete*' demands that some persistently stored RDF should be removed. The embedding '*insert*' stores some RDF persistently. Unlike queries, both delete and insert have side effects. For instance the above RDF could be inserted into a store then removed from the store.

Update can be formed using two binary operations, '*join*' and '*choose*'. Join is the operation which combines two updates to ensure that they happen in the same atomic commitment. For instance, a query and an insert can be joined to ensure that an insert occurs if and only if the query is satisfied. Choose presents an option where either the left update or the right update occurs. For instance, a choice can be presented between two possible query patterns. The iteration operation indicates that zero, one, two or more copies of an update are simultaneously applied. The constructs choose, join, iterate, true and false form a Kleene algebra.

Syntactic conventions for Kleene algebras are adopted for examples. The operator $\otimes$ binds stronger than $\oplus$ and the operator $\otimes$ can be omitted; hence $(U \otimes V) \oplus W$ is abbreviated $UV \oplus W$. The following example presents a syntax for the update described in the previous section. The update is a extension of the query given in concrete syntax. Here the query is translated into abstract syntax and an insert is joined to the query.

$$
*\bigvee a.\bigvee b.\left(\begin{array}{l}
\left(\begin{array}{l}
|(\textit{show2011 exhibitor } b)| \\
|(b \textit{ product } a)| \\
\oplus \\
|(\textit{show2011 product } a)|
\end{array}\right) \\
(\textit{show2011 product } x)
\end{array}\right)
$$

Updates extend Kleene algebras with quantifiers. The select quantifier binds occurrences of a name not known in advance. For instance, in the example above the name of the product is not known. The name is bound in both the query and the insert, so the name discovered by the query is also the name inserted. Names and literals are disjoint, so a separate quantifier is provided for variables. Quantifiers highlight the logical content of updates.

The syntax of constraints is embedded in the syntax of updates. A constraint imposes a condition on the update taking place. Typically variables which occur as the object of a triple are constrained. For instance a variable may represent a string of characters which satisfies a regular expression, or a numeral within a range of values. Like literals, constraints are well understood, so technicalities are left to the SPARQL Query standards [5, 25]. It is sufficient to note that constraints form a Boolean algebra.

For the purpose of defining operational semantics, a syntax for processes is introduced. Processes allow updates to be considered simultaneously. The commutative monoid using par and nothing defined for RDF is extended to processes, as in Fig. 2. The convention, common to sequent style deductive systems, is that the symbol $\parr$ is abbreviated with a comma in examples.

2.2.2. An operational semantics for atomic updates.
The behaviour of an update at the level of the syntax is captured by operational semantics. A preliminary draft of an operational semantics for SPARQL Update was published in October 2010 [21]. The operational semantics presented elaborates the draft. A fine grained operational semantics for updates are specified using atomic actions.

Atomic actions are specified as a relation over processes called the commitment relation. The process on the left of the relation is exactly the processes used by the action. The process on the right of the relation is exactly the processes

$$C \parr C^{\perp} \rhd \bot \qquad C \rhd C \qquad |C| \parr C \rhd C$$

$$\frac{\vDash \phi}{\phi \rhd \bot} \qquad \frac{P \parr U \rhd P' \quad Q \parr V \rhd Q'}{P \parr Q \parr (U \otimes V) \rhd P' \parr Q'}$$

$$\frac{P \parr U \rhd Q}{P \parr (U \oplus V) \rhd Q} \qquad \frac{P \parr V \rhd Q}{P \parr (U \oplus V) \rhd Q}$$

$$*U \rhd \bot \qquad \frac{P \parr U \rhd Q}{P \parr *U \rhd Q} \qquad \frac{P \parr (*U \otimes *U) \rhd Q}{P \parr *U \rhd Q}$$

$$\frac{P \parr U\left\{{}^{b}/_{a}\right\} \rhd Q}{P \parr \bigvee a.U \rhd Q} \qquad \frac{P \parr U\{{}^{v}/_{x}\} \rhd Q}{P \parr \bigvee x.U \rhd Q}$$

FIGURE 4. The axioms and rules form atomic commitments: delete axiom, insert axiom, query axiom, filter axiom, join rule, choose left rule, choose right rule, weakening axiom, dereliction rule, contraction rule, select name rule, select literal rule.

after the action. Thus an commitment relation describes only the local behaviour of an update. A similar approach is given by commitment relations in the π-calculus [6]. In the π-calculus there is one type of commitment – the passing of a name on a channel. Coordination of Web Services motivates extending the commitments to joins [7]. SPARQL provides a compelling reason to extend commitments to all updates. The commitment relation $\rhd$ is defined in Fig. 4.

The delete axioms. A simple interaction is when an update deletes a triple and the triple is available to delete. For instance, the delete and triple below are expected to interact. The result of the interaction is that the delte and the matching triple are consumed. The following is an instance of the delete axiom.

$$(\mathit{doc1\ creator\ Burns})^{\perp}, (\mathit{doc1\ creator\ Burns}) \rhd \bot$$

Notice that the axioms of Linear Logic and the atomic commitments of CCS are of a similar form [27, 28]. Note the syntactic convention of using a comma for $\parr$.

The insert axiom and query axiom. RDF to be stored after an update appears on the right of a commitment relation. There are two ways in which RDF can appear on the right. The first scenario is that a triple is inserted into a store. This is captured by the insert axiom. The insert axiom states that some RDF intended to be stored is stored by a successful update.

The second scenario is that some stored RDF can be used to answer a query. The stored RDF then returned to the store unaltered. For instance, the following example consists of a stored triple and a query asking for that triple. The query is answered and the triple remains stored.

$$\begin{array}{l}|(\mathit{doc1\ creator\ Burns})|\,,\\ (\mathit{doc1\ creator\ Burns})\end{array} \rhd (\mathit{doc1\ creator\ Burns})$$

The syntax for an insert is the same as the syntax for some stored RDF. Therefore a trivial update which inserts some RDF is used to model stored RDF. Other SPARQL Query results may be modelled similarly to inserts, by indicating the results on the right of the commitment relation. Related calculi investigate updates and queries over inserted data as concurrent constraint satisfaction problems for Web Services [29, 30].

The join rule. The join rule forces two updates to occur in the same commitment. Join meets a requirement of SPARQL Update that a delete and insert can occur atomically. Join avoids the issue of reversing an insert, when a delete fails.

Another requirement met by join is that updates can be dependent on queries. The following example demonstrates an insert joined with a query. The available triple is adequate for the query, so the insert takes place. Both the stored triple and the inserted triple persist, so are composed after the transaction.

$$\begin{array}{l}\left(\begin{array}{l}|(\mathit{doc1\ title}\ \texttt{`Links'})|\\ (\mathit{doc1\ creator\ Burns})\end{array}\right),\\ (\mathit{doc1\ title}\ \texttt{`Links'})\end{array} \rhd \begin{array}{l}(\mathit{doc1\ creator\ Burns}),\\ (\mathit{doc1\ title}\ \texttt{`Links'})\end{array}$$

Join splits a query into two updates which can be resolved in separate locations. For instance, in the above query the two joined parts can be resolved on different machines in a cluster of servers. Thus, join serves the same purpose as join in relational algebra [31, 15]. The join rule also appears as the rule for multiplicative conjunction (times) in linear logic and as atomic commitments in process calculi for Web Services [27, 7].

The choose rules. Choice allows the programmer to specify several possible updates. The example below asks for a triple where the predicate is one of two options. The branch with the query which matches the available data is chosen. This is an external choice dependent on the available data.

$$\begin{array}{l}\left(\begin{array}{l}(\mathit{doc1\ creator\ Burns})^{\perp}\\ \oplus\\ (\mathit{Burns\ author\ doc1})^{\perp}\end{array}\right),\\ (\mathit{Burns\ author\ doc1})\end{array} \rhd \bot$$

If both branches of a choice can be enabled, one is chosen non-deterministically. The choose rules correspond to the rules for additive disjunction (plus) of linear logic and an external choice in process calculi for Web Services [27, 9].

The select rules. Most constructs work at the level of triples. Quantifiers are required to access names within triples. The select name rule works by substituting a name for the quantified name. For instance, in the example below the bound name a is replaced by *person*. This particular substitution allows the query to be answered and determines the name in the triple inserted.

$$\begin{array}{l}\bigvee a.\left(\begin{array}{l}|(\mathit{doc1\ creator\ a})|\\ (\mathit{Hamish\ knows\ a})\end{array}\right),\\ (\mathit{doc1\ creator\ Burns})\end{array} \rhd (\mathit{Hamish\ knows\ Burns})$$

The effect above is that the substituted name is passed from the triple to the update. This is also the effect of the atomic commitments of the π-calculus [6]. The commitments of the π-calculus are decomposed into: a select which inputs the name; join which composes a continuation; and ask which poses a guard. By replacing triples with channel-value pairs and inserts with processes, the π-calculus can be recovered as investigated by Miller [32].

The select literal rule substitutes literals for variables. As above, this captures the passing of literals from triples to updates. Value passing is achieved by atomic commitments in the applied π-calculus [8]. The select rules match the rule for first-order existential quantification (some) in linear logic [27].

The constraint satisfaction relation. In general constraints form a Boolean algebra. True formulae are indicated by $\vDash$ the constraint satisfaction relation. The definition of the constraint satisfaction relation is left to the SPARQL Query standards [5, 25]. For instance, the constraint below is satisfied when x is at least 20 years before the current year. Select substitutes x for `'1987'`, enabling the following commitment.

$$\bigvee a.\bigvee x.\left(\begin{array}{l}|(a\ \mathit{year}\ x)|\\ (\texttt{year-now} - x > 20)\\ (a\ \mathit{copyright\ open})\end{array}\right),\ (\mathit{paper\ year}\ 1987) \ \triangleright\ \begin{array}{l}(\mathit{paper\ year}\ 1987),\\ (\mathit{paper\ copyright\ open})\end{array}$$

An equality comparison over names is another form of constraint. An equality comparison joined to an update, captures match found in common process calculi [6, 8]. The constraints true and false are the top and bottom elements of the Boolean algebra. True always holds, so true is embedded as the multiplicative unit in linear logic. False never holds, so like the additive zero in linear logic, no rule can be applied [27]. The embedding of Boolean algebras in Kleene algebras is elaborated by Kozen [33].

The rules for iteration. Without iteration updates are only applied once. This enables a protocol where the programmer requests an update. The user then observes the commitment relation. If the update was not as the user intended, the update can be revoked. Then the next update is observed until the user is satisfied. Caution is exercised when the exact update is difficult to express or the content to update is not certain. When a user is certain that the update is intended, the update can be applied iteratively. The replace tool in the reader's text editor probably has similar functionality.

To apply an iterated update zero times, the weakening axiom is used. To apply an iterated update once, the dereliction rule is used. To apply an iterated update twice, the contraction rule creates two joined copies of the update. Join ensures that both copies occur in the same commitment. The example below demonstrates two nested iterations. The outermost applies twice, the innermost applies both once and twice.

$$*\bigvee a.\left(\begin{array}{l}|(a\ \mathit{status\ hidden})|,\\ *\bigvee b.(a\ \mathit{knows}\ b)^{\perp}\end{array}\right),\ \begin{array}{l}(\mathit{Alice\ status\ hidden}),\\ (\mathit{Bob\ status\ hidden}),\\ (\mathit{Alice\ knows\ Bob}),\\ (\mathit{Alice\ knows\ Chris}),\\ (\mathit{Bob\ knows\ Chris})\end{array} \ \triangleright\ \begin{array}{l}(\mathit{Alice\ status\ hidden}),\\ (\mathit{Bob\ status\ hidden})\end{array}$$

Iteration is the Kleene star in regular algebra. A classic result is that nested iteration can be represented by a single iteration [34, 33]. However, quantifiers ensure that the example above cannot be expressed without nested iteration. The SPARQL Query recommendation does not have nested iteration, so cannot express the corresponding query [5].

Iteration is not replication in process calculi. Iteration defines a single commitment of an unbounded size; whereas replication persists a process across an unbounded number of commitments [6]. The use of contraction, dereliction and weakening is similar to the exponentials in linear logic, but does not correspond to either. Iteration has been used by Hoare to specify unbounded behaviour [35].

$$\frac{P \triangleright Q}{P \to Q} \qquad \frac{P \to Q \quad Q \to R}{P \to R} \qquad \frac{P \to P' \quad Q \to Q'}{P \otimes Q \to P' \otimes Q'}$$

FIGURE 5. Reduction relations: action, transitivity and mix.

2.2.3. A transition system for SPARQL processes. Atomic commitment relations describe the local behaviour of an update. Only processes consumed and inserted are accounted for in a commitment relation. Another relation between processes, called the reduction relation $\to$, captures the evolution of a store, defined in Fig. 5.

The action rule turns a local commitment relation into a reduction relation. The action rule can be applied repeatedly, to allow multiple updates to happen in parallel. Multiple updates and idled inserts are composed in parallel using the mix rule. The transitive closure of the transition relation permits a sequence of updates on a store. The following demonstrates two separate deletes which may be applied in parallel or in sequence with the same outcome.

$$\begin{array}{l}(\mathit{doc1\ creator\ Burns})^{\perp},\\ (\mathit{doc1\ creator\ Burns}),\\ (\mathit{doc1\ title}\ \texttt{'Links'})^{\perp},\\ (\mathit{doc1\ title}\ \texttt{'Links'})\end{array} \ \to \perp$$

The reduction relation provides a concise model of the evolution of a store, with respect to updates. The commitment relation and the reduction relation cannot be combined, without breaking the atomicity of an update. The separation of the actions (given by $\triangleright$) and the representation of actions on the space on which they act (given by $\to$) is common across dynamic systems [36, 37]. The combination

of mix and transitivity defines a truly concurrent process calculus [38].

3. FEATURES FOR SYNDICATION

The core calculus focusses on fine grained updates, where updates act at the level of individual triples. This section considers a coarser level of granularity. A coarser granularity of data divides a store into regions, each regions contains triples. Each region can be considered separately from other regions. Regions impact the querying of data by allowing queries to be directed at particular regions. Regions also enable a coarser level of update, where entire regions become atomic units.

This section argues that two key Web technologies work at the granularity of regions — feeds and named graphs [1, 16]. At a suitable level of abstraction, feeds and named graphs can be queried and updated in one model. The model demonstrates that several key standards for feeds and named graphs enable a programming language for Linked Data. Furthermore, prominent examples of feeds and named graphs suggest several useful scenarios, including syndication and provenance. Both syndication and provenance have been found to be essential for a Web of Data, where separate authorities contribute separate data.

3.1. Extensions of RDF for Named Graphs

Named graphs are introduced as a minimal extension to RDF such that a large monolithic knowledge base, consisting of a single RDF store can be divided into smaller stores, each individually named [16]. The name of the graph is a URI, which can be liked to like any other URI. The RDF associated with the name of a graph, can describe the nature of the knowledge represented by the graph. Applications including provenance and access control have lead to the widespread acceptance of named graphs. Named graphs are primitive in SPARQL Query and SPARQL Update [25, 21].

The following is an example of two named graphs. The example is expressed in the TriG syntax, an extension of the Turtle syntax [16]. The first graph contains some RDF data. The second graph contains some RDF data about the first graph. This enables the user make decisions based on the source of the RDF. The user may trust RDF with provenance *eg:G1* and use that directly, ignoring data in *eg:G2*. Alternatively, the user may trust RDF with provenance *eg:G2* on the subject of whether to use data in *eg:G1*.

```
eg:G1 {
  _:Monica eg:name "Monica Murphy" .
  _:Monica eg:email <monica@murphy.org> .
}
eg:G2 {
  eg:G1 eg:author eg:Chris .
  eg:G1 eg:date "2003-09-03"^^xsd:date
  eg:G1 eg:disallowedUsage eg:Marketing.
}
```

$$
\begin{array}{rcll}
C & ::= & \bot & \text{nothing} \\
& | & (a\ a\ o) & \text{triple} \\
& | & C \parr C & \text{par} \\
& | & \bigwedge a.C & \text{blank node} \\
\\
G & ::= & \mathcal{G}_a C & \text{named graph} \\
& | & C & \text{default graph} \\
& | & G \parr G & \text{par} \\
& | & \bot & \text{nothing}
\end{array}
$$

FIGURE 6. An extended abstract syntax for RDF Content and Graphs.

The above example also introduces a feature of RDF. Identifiers with prefix `eg:` are URIs in some example namespace. The identifier with prefix `_:` represents a blank node. A blank node is used in place of a URI when a URI is not explicily assigned. A blank node is a local identifier which cannot be linked to directly. Blank nodes reduce the barrier between RDF and other data formats, by allowing common data structures to be encoded in RDF without introducing new URIs. A constraint placed on named RDF graphs is that blank nodes are local to each individual named graph. This preserves the integrity of data structures encoded in named graphs.

3.2. An abstract syntax for named graphs

The abstract syntax of RDF content (from Fig. 1) is extended with blank nodes (in Fig. 6). Blank nodes are indicated by a quantifier for names, similarly to N3 logic [39]. The scope of the quantifier indicates the RDF content in which a blank node is bound. Bound names represent blank nodes, whereas unbound names represent URIs.

The syntax of graphs indicates both named graphs and unnamed RDF content. Named graphs are represented by a prefix with a subscript indicating the name, called the naming operator. RDF content without a naming operator represent the default graph, which allows RDF to be published without the named graph mechanism. The example from the previous section is expressed below in the abstract syntax. The blank node quantifier binds the name *Monica* in the first graph. However, the name *G1* is not bound by the naming operator so can be linked to from the second graph.

$$
\begin{array}{l}
\mathcal{G}_{G1}\left(\begin{array}{l} \bigwedge \mathit{Monica}. \\ \left(\begin{array}{l} (\mathit{Monica\ name}\ \texttt{`Monica Murphy'}), \\ (\mathit{Monica\ email\ monica@murphy.org}) \end{array}\right) \end{array}\right), \\
\mathcal{G}_{G2}\left(\begin{array}{l} (\mathit{G1\ author\ Chris}), \\ (\mathit{G1\ date}\ \text{`2003-09-03'}), \\ (\mathit{G1\ disallowedUsage\ Marketing}) \end{array}\right)
\end{array}
$$

The above example shows that par, abbreviated by comma, is used as before to compose triples and also graphs. The structural congruence over RDF content, in Fig. 2, ensures that par and nothing form a commutative

$$\bigwedge a.\bot \equiv \bot \qquad \bigwedge a.C \parr D \equiv \bigwedge a.(C \parr D) \quad a \notin \mathrm{fn}(C)$$

$$\bigwedge a.\bigwedge b.C \equiv \bigwedge b.\bigwedge a.C \qquad \mathcal{G}_a(C \parr D) \equiv \mathcal{G}_a C \parr \mathcal{G}_a D$$

FIGURE 7. The structural congruence extended for blank node quantifiers and naming operators.

monoid. The structural congruence is extended to blank node quantifiers in Fig. 7. The first rule allows blank nodes to be eliminated if they bind nothing. The second rule allows a blank node to be distributed across some RDF content where the name does not occur. The third rule allows the order of two quantifiers to be swapped. The blank node rules preserve the free URIs in RDF content.

As standard, bound names can be α-converted. This avoids name clashes between blank nodes. Content where only blank nodes differ are equivalent, by α-conversion. This is a syntactic approach to the graph isomorphisms defined in the RDF standards. Like α-conversion, the graph isomorphisms preserve the structure and URIs but allow the blank nodes to change [3, 40].

The structural congruence also extends to naming operators. A named graph can be split into two pieces each with the same name. This is for the purpose of fine grained updates, as only the part of a graph required for an update need be considered. The naming operator and the quantifiers do not commute, so blank nodes remain within their designated graph.

Related work constrains named graphs so that the boundaries of a named graphs are fixed [16]. For instance, knowing the boundaries of a named graph enable the named graph to be completely dropped from a store, as in the drop operation of SPARQL Update [21]. This structural constraint is a perpendicular concern to this work. Similar constraint on global structure are tackled, for instance, in dynamic epistemic logic as structure preserving maps [37]. Stronger preservation of structure is extensively researched in the context of the Web using ontologies [41]. This work focusses on Linked Data without such a global perspective on structure.

3.3. SPARQL Update over Named Graphs

Queries and updates also work in the the named graph setting. The abstract syntax for updates, in Fig. 3, is extended to named graphs, by replacing RDF content (from Fig. 1) with named graphs (from Fig. 6). The same rules for atomic commitments (in Fig. 4) work for named graphs.

The SPARQL Update submission describes an update, where the title of a book is replaced by a new title [10]. This example is captured by the commitment relation below. The delete axiom removes a triple from a graph, the insert axiom inserts the new triple in to the graph and the join rule ensures the delete and the insert happen synchronously. The presence of the naming prefix makes no difference to the

$$\frac{P \rhd P' \quad Q \rhd Q'}{P \parr Q \rhd P' \parr Q'} \qquad \frac{P \parr Q \rhd P' \parr Q'}{\bigwedge a.P \parr Q \rhd \bigwedge a.P' \parr Q'} \quad a \notin \mathrm{fn}(Q) \quad a \notin \mathrm{fn}(Q')$$

$$\frac{\mathcal{G}_b P \parr Q \rhd \mathcal{G}_b P' \parr Q'}{\mathcal{G}_b \bigwedge a.P \parr Q \rhd \mathcal{G}_b \bigwedge a.P' \parr Q'} \; a \notin \mathrm{fn}(Q, Q', b)$$

FIGURE 8. Commitments extended for mix, blank nodes and blank nodes in named graphs.

rules.

$$\left(\begin{array}{l} \mathcal{G}_{store}(book_3\ title\ \texttt{`Designs'})^{\perp} \\ \mathcal{G}_{store}(book_3\ title\ \texttt{`Design'}) \end{array}\right),$$
$$\mathcal{G}_{store}(book_3\ title\ \texttt{`Designs'})$$

$$\rhd \mathcal{G}_{store}(book_3\ title\ \texttt{`Design'})$$

The largest example given in initial SPARQL Update drafts can now be expressed [10]. The update combines a query which finds the date of a book, a filter which checks the date is in a certain range, and an iterated update on books in that range. The iterated update will only trigger if the query and filter are satisfied. The iterated update moves triples about a book across from one graph to another graph, by combining a delete and insert. The example can be expressed in the abstract syntax, as follows.

$$*\bigvee d.\bigvee book.$$
$$\left(\begin{array}{l} (d \leq \text{`01-01-2000'}) \\ |\mathcal{G}_{store1}(book\ date\ d)| \\ *\bigvee a.\left(\begin{array}{l} \mathcal{G}_{store1}(book\ note\ a)^{\perp} \\ \mathcal{G}_{store2}(book\ note\ a) \end{array}\right) \end{array}\right),$$
$$\mathcal{G}_{store1}\left(\begin{array}{l} (Kidnapped\ date\ \texttt{`01-05-1886'}), \\ (Kidnapped\ note\ classic) \end{array}\right)$$

$$\rhd \begin{array}{l} \mathcal{G}_{store1}(Kidnapped\ date\ \texttt{`01-05-1886'}), \\ \mathcal{G}_{store2}(Kidnapped\ note\ classic) \end{array}$$

The above example is compact compared to the example in the draft concrete syntax. The presentation here is enabled by the constructs in the abstract syntax, whereas a single compound construct is used in the concrete syntax.

3.4. Updates for RDF with Blank Nodes

To query and update blank nodes, the commitment relation is extended in Fig. 8. The trick is to replace a quantified name with a temporary free name. This allows the quantifier to be removed and the rules of the calculus to be applied as if there were no quantifiers.

The temporary name is chosen to be fresh in the context. By choosing a fresh name, the name can be tracked before and after the commitments. This ensures that the same name that was quantified before is quantified after, as expressed by the first blank node rule. The blank node rule is similar to universal quantification in Linear Logic and new name quantification in the π-calculus [27, 6].

The second blank node rule ensures that a blank node which originates in an named graph is returned to the same named graph. The extra mix rule allows triples in the scope of a blank node which are not used in a commitment to idle.

The following example demonstrates an update which involves a blank node in a named graph. The name of the blank node in the graph is replaced by a temporary name. The temporary name is discovered by the select quantifier as normal. A new triple with the temporary name is inserted into the same graph as normal. The blank node rule ensures that the temporary is bound after the commitment.

$$\begin{array}{l} \bigvee person.\bigvee x.\bigvee y. \\ \left(\begin{array}{l} \left|\mathcal{G}_{graph1}(person\ nickname\ x)\right| \\ \mathcal{G}_{graph1}(person\ email\ y) \\ (y = \texttt{concat}(x, \texttt{`@soton.ac.uk'})) \end{array}\right), \\ \mathcal{G}_{graph1} \bigwedge a.(a\ nickname\ \texttt{`Rabbie'}) \end{array}$$

$$\triangleright\ \mathcal{G}_{graph1}\ \bigwedge a.\left(\begin{array}{l} (a\ nickname\ \texttt{`Rabbie'}), \\ (a\ email\ \texttt{`Rabbie@soton.ac.uk'}) \end{array}\right)$$

In the example the blank node does not leave the graph. Suppose that instead the update inserts the new triple into a different named graph. In this case the update cannot be applied since the blank node would appear free in another graph. The side condition would be violated.

3.5. Feeds as a ubiquitous syndication format.

RDF is the format standardised by the W3C, however feeds are ubiquitous on the Web. Like RDF, feeds are a semi-structured data format which identifies resources using URIs. The two ubiquitous feed formats are RSS and Atom. RSS was originally created by Netscape and comes in several varieties. Atom has the same purpose as RSS but is standardised [1, 18]. Atom has been adopted by Google for its Google Data protocol, which shares data between applications.

Feeds are particularly suited to syndication. Syndication is the strategy of delivering data to the intended audience on demand. Feeds typically represent the view point of some authority. A BBC News feed on Africa contains data representing the viewpoint of the BBC on the topic of news in Africa. A user who is interested in that viewpoint can obtain that feed on demand. The user can answer questions such as, "According to BBC News on Africa, what are the headlines today?"

The following is an example of the Atom Syndication Format. Notice that the feed and the entry are identified by URIs, which are abbreviated here as *eg:feed_id* and *eg:entry_id*. The tags such as *title* and *updated* are also URIs indicated by the XML namespace.

```
<feed xmlns="http://www.w3.org/2005/Atom">
 <title>Example Feed</title>
 <link href="http://example.org/"/>
 <updated>2003-12-13</updated>
 <author>
  <name>John Doe</name>
 </author>
 <id>eg:feed_id</id>
 <entry>
  <title>Example Entry</title>
  <link href="http://example.org/03"/>
  <id>eg:entry_id</id>
  <updated>2003-12-13</updated>
  <summary>Some text.</summary>
 </entry>
</feed>
```

The above example can be represented using named graphs and blank nodes as follows. The entries are translated into triples and form the content of a named graph. The triples associate with the feed are part of the default graph. The XML style above does not indicate a URI for the author of the feed. Below the implicit author is represented by introducing a blank node.

$$\begin{array}{l} \bigwedge Doe.\left(\begin{array}{l} (Doe\ name\ \texttt{`John Doe'}), \\ (feed\_id\ author\ Doe) \end{array}\right), \\ (feed\_id\ title\ \texttt{`Example Feed'}), \\ (feed\_id\ updated\ \texttt{`2003-12-13'}), \\ (feed\_id\ link\ http://example.org/), \\ \mathcal{G}_{feed\_id}\left(\begin{array}{l} (entry\_id\ title\ \texttt{`Example Entry'}), \\ (entry\_id\ link\ http://example.org/03), \\ (entry\_id\ updated\ \texttt{`2003-12-13'}), \\ (entry\_id\ summary\ \texttt{`Some text.'}) \end{array}\right) \end{array}$$

The above syntax demonstrates one representation of Atom in RDF, however there is no standard representation. Some varieties of RSS encode feeds using triples. However, named graphs are primitive in SPARQL, so are suggested here as a representation of the content of a feed.

3.6. The Atom Publishing Protocol

For RSS an application implements its own update mechanism. In contrast, the Atom Publishing Protocol extends Atom with a standard update mechanism [19]. The publishing protocol allows new resources to be published and existing resources to be edited. The protocol works at the low level of passing messages using an HTTP protocol. However, feeds can still be updated at the high level offered by SPARQL. This section demonstrates a high level update of a feed and outlines the corresponding low level operations which realise the high level update.

The Atom publishing protocol specification allows variations on the basic protocol. The example in this section features a main feed of articles and comment feed linked to each entry of an article. Firstly, a feed is declared such that initially it contains no resources. The data associated with the feed indicates the author of the feed and a title for the feed.

$$\begin{array}{l} (feed\ author\ Hamish), \\ (feed\ title\ \texttt{`Caucasus reported'}), \\ \mathcal{G}_{feed}\bot \end{array}$$

The first update, defined below, creates a new article in the feed and an empty comment feed to go with the article.

The comment feed is linked to the new article. The triple associated with *entry* indicates a title and a modification date.

$$\mathcal{G}_{feed}\left(\begin{array}{l}(\textit{entry title } \texttt{`Invaded'})\\(\textit{entry updated } \text{`01-02-2008'})\\(\textit{entry comments discussion})\end{array}\right)$$
$$(\textit{discussion subject entry})$$
$$\mathcal{G}_{discussion}\bot$$

A second update, defined below, changes the title and the date the feed was updated. The update first discovers the old title and old date using select quantifiers. The update then deletes the old triples and inserts the new triples.

$$\bigvee s,d.\mathcal{G}_{feed}\left(\begin{array}{l}(\textit{entry title } s),\\(\textit{entry updated } d)\end{array}\right)^{\bot}$$
$$\mathcal{G}_{feed}\left(\begin{array}{l}(\textit{entry title } \texttt{`Ossetia invaded'}),\\(\textit{entry updated } \text{`02-04-2008'})\end{array}\right)$$

A third update creates a new comment in the comment feed associate with the entry. A query discovers the relevant comment feed and a new entry is inserted in that comment feed. The new comment is identified by a blank node rather than a URI.

$$\bigvee \textit{discussion}.\left|\mathcal{G}_{feed}(\textit{entry comments discussion})\right|$$
$$\bigwedge \textit{reaction}.$$
$$\mathcal{G}_{discussion}\left(\begin{array}{l}(\textit{reaction content } \texttt{`Why?'}),\\(\textit{reaction author Dmitri}),\\(\textit{reaction updated } \text{`05-04-2008'})\end{array}\right)$$

The updates can be applied to the initial configuration. By applying the operational semantics the results is the configuration bellow.

$$(\textit{feed author Hamish}),$$
$$(\textit{feed title } \texttt{`Caucasus reported'}),$$
$$\mathcal{G}_{feed}\left(\begin{array}{l}(\textit{entry title } \texttt{`Ossetia invaded'}),\\(\textit{entry updated } \text{`02-04-2008'})\\(\textit{entry comments discussion}),\end{array}\right),$$
$$\bigwedge \textit{reaction}.$$
$$\mathcal{G}_{discussion}\left(\begin{array}{l}(\textit{reaction content } \texttt{`Why?'}),\\(\textit{reaction author Dmitri}),\\(\textit{reaction updated } \text{`05-04-2008'})\end{array}\right)$$

This example demonstrates that the same language for updating named graphs can be used to update feeds. The underlying operations of a publishing protocol can realise these updates. Operations of the protocol are described by the verbs post, put and get as found in a REST protocol [42].

The underlying REST operations can be outlined as follows. The first update corresponds to posting an entry to the feed and posting a new feed to the store. The second update corresponds to getting the entry and putting it back in its updated form. The third entry corresponds to getting the entry, evaluating a query and posting a new entry in the comment feed.

This example demonstrates that other semi-structured data formats, such as Atom, are compatible with RDF. It also demonstrates that SPARQL Update can be realised by the operations of a lower level protocol. The details of the low level protocol are hidden from the programmer. Related work demonstrates high level operations encoded using low level operations in the setting of Web Services [7].

4. LIGHT TYPES FOR URIS AND LITERALS

Many structural constraints, often expressed using an ontology [41], are not tackled in this work. The issue is that many invariants on structures which are imposed by an ontology require a global perspective on data. Apparently simple invariants such as, "Resources with a surname have a nick name," are difficult to impose in an open environment. If delete removes one nickname, is there another nickname that maintains the invariant? This cannot be confirmed without knowing the extent of the entire store. Furthermore, a query which checks for a triple indicating a nickname might be unsuccessful. An unsuccessful query does not mean that the triple does not exist, only that the query was unsuccessful.

A compromise between ontologies and pure data exists. A light type system which only deals with URIs and literals in triples, rather then structures across several triples, is proposed. Given one triple it is easier to tell whether the subject and object are of the correct type for a predicate. For instance, a type system may allow the assumption that a predicate *surname* relates a person to a string. Thus for any triple in which *surname* is observed as the predicate, the subject of the triple is a person and the object is a string. No knowledge of other triples is required.

It is still naive to assume that triples can be typed. Given a literal, say '3', it is reasonable to assume that '3' is an integer. However given a URI, say `http://eprints.ecs.soton.ac.uk/15017/`, what is the type of the URI? Is it a person? Is it a predicate? The only sure answer is that it is a URI. Inside knowledge may say that the prefix of the URI indicates a paper, however in general such policies are not available. In general on the Web few type assumptions can be made about URIs.

Intrinsic problems associated with semi-structured data on the Web are well known. In isolating the essential aspects of semi-structured data, Abiteboul highlights prevailing challenges [43]. Abiteboul argues for a light exchange model with an a-posteriori data guide; which addresses challenges including, eclectic types and a blurring of the distinction between schema and data. A light flexible type system for Linked Data addresses issues highlighted by Abiteboul.

For flexibility the type system works at three levels. Firstly, the XML Schema Datatype standard is reused to form a solid basic type system for RDF, where only literals are typed. Secondly, some types for URIs are moved from the data to the type system as propositional types. Thirdly, the standard inference system from the W3C standard RDFS is adapted to form a subtype system [4]. The subtype system offers flexibility and interoperability between the different strengths of typing. The three perspectives offer

a compromise between static typing and dynamic data. In the presence of updates, static types are preserved while dynamic data changes.

4.1. A Standardised Type System for Literals

A type system for literals can detect basic programming errors in queries. Literals can appear in constraints in which only literals of a certain type make sense. For instance a regular expression only makes sense over as string. A comparison between literals only makes sense if the two literals are of the same type. An inequality between literals only makes sense if the two literals are of the same type and there is a natural order over that type of data.

The SPARQL Query recommendation defines the operations which appear in filters [5]. The XML Schema Datatypes recommendation is reused to define the types for literals [23]. Literals are well understood, so it assumed that a type system for literals exists. From these standards a basic type system for updates can be defined. The basic type system annotates variables with data types as follows.

$$\bigvee x : \textsc{Date}. \begin{pmatrix} (\texttt{`01-01-1960'} \leq x) \\ (x < \texttt{`01-01-1970'}) \\ |(\textit{document1 created } x)| \\ (\textit{document1 note candidate}) \end{pmatrix}$$

The data types can be used in a type system to check that the constraints are correctly typed. In the example above, the constraints are inequalities between dates and a variable which is presumed to be a date. A type system accepts this update. If a constraint that checks the variable for a regular expression is also added then a type error is triggered.

Due to the data type standards, this level of typing can always be applied. Furthermore, a type inference algorithm allows the programmer to specify an update without types and still take advantage of the type system. A suitable type system and inference algorithm for literals is assumed [44].

4.2. Light Propositional Types for RDF

By typing predicates, basic errors can be detected in the structure of triples. When the choice of verb does not match the choice of subject and object, no information needs to be known about the context of a sentence to reject a sentence. A simple sentence such as, "The mountain writes the fish," will always be nonsense.

Without type information it is less obvious that the above sentence is nonsense. Naming the mountain 'Ararat' and the fish 'Nemo' might give, "Ararat writes Nemo." By supposing that Ararat is a persons name and Nemo is a story, the nonsense appears to make sense. However, cultural experience suggests that Ararat refers to Mount Ararat, rather than some person. A mountain cannot be the subject of the verb to write, so the sentence remains nonsense.

The type of a URI is harder to establish than in natural language. What is the type of the URI `dbpr:Mt_Ararat`? According to DBpedia the type is another URI `dbpo:Place`. The relationship between the URI and its type can be represented by the following triple.

$$\begin{array}{rrr} \textsc{Data} := & \textsc{String} & \text{string type} \\ | & \textsc{Date} & \text{date type} \\ | & \ldots & \text{etc.} \end{array}$$

$$\begin{array}{rrr} \tau ::= & \textsc{a} & \text{atomic proposition} \\ | & \mathrm{p}(\tau, \tau) & \text{predicate type} \\ | & \mathrm{p}(\tau, \textsc{Data}) & \text{data predicate type} \\ | & \top & \text{top type} \\ | & \textsc{class} & \text{dynamic class type} \\ | & \tau \cup \tau & \text{union type} \\ | & \#\tau & \text{container type} \end{array}$$

FIGURE 9. The syntax of types and type environments.

```
dbpr:Mt_Ararat rdf:type dbpo:Place .
```

Note the namespace prefixes `dbpr:` and `dbpo:` abbreviate `http://dbpedia.org/resource/` and `http://dbpedia.org/ontology/`, which are used by DBpedia for resources and terminology respectively [12]. The namespace prefix `rdf:` corresponds to the `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, which is used for standardised terminology for RDF [3]. The classification of the subject is indicated by the object of the predicate `rdf:type` in a triple, as above. The URIs used for classification are referred to as classes.

The type system can treat classes in two different ways. In the first interpretation, classes are treated like any other URI. A class treated as a URI can be used in data, so can be linked to and updated as normal. In the second interpretation, a class is distinguished as an atomic type. A class distinguished as an atomic type indicates static properties of URIs. The type system is designed so that both approaches may be used. This allows an interplay between applications which use classes in data and use classes as static types. This design decision is a blurring of the distinction between schema and data, highlighted by Abiteboul [43].

4.2.1. The syntax of propositional types.

The syntax of types which defines propositions which can be assigned to URIs, is presented in Fig. 9. The definition of types uses atomic propositions, which depend on the application.

Atomic propositional types. For a type system, a number of atomic propositional types are fixed. Atomic types are indicated by small capitals, such as ARTICLE, PERSON. Atomic types are application specific. They indicate static assumptions about a URI. For instance, an application which plots resources on a map deals with URIs typed by proposition PLACE. A different application which maintains a calendar of concerts may use the proposition EVENT.

Unlike datatypes, which restrict the structure of literals, atomic propositional types do not impose structure on a URI. Atomic propositional types are just syntax which guides how

a URI is used.

Predicates between URIs. Atomic types can be used to construct predicates types. A predicate type indicates the type of a subject and object. For instance, the predicate *writes* may relate a person to a document. This is indicated by the predicate type $\mathtt{p}(\textsc{person}, \textsc{document})$. The predicate *knows* may relate a person to a person, indicated by type $\mathtt{p}(\textsc{person}, \textsc{person})$.

Predicate types can be used to catch basic errors in triples, where the subject or the object does not match the expected type. For instance, a subject of type LOCATION and object of type ANIMAL are not valid for the predicate *writes*, under the assumptions above.

Datatype predicates. Predicates which allow literals as objects, are indicated using datatype predicates. For instance, the predicate *created* relates a document to a date literal, indicated by type $\mathtt{p}(\textsc{document}, \textsc{Date})$. This allows both literals and variables of type DATE to be used as the object of *created*.

In the example in Sec. 4.1, the variable appears as the object of a triple with predicate *created*. The type assumption for *created* can be used to check that the type of the variable, matches how the variable is used in the triple. In the example, the variable also appears in a constraint. A type error occurs if the type of the variable in the triple and the type of the same variable in the constraint do not match.

The top type. A URIs can be assigned the top type $\top$, which represents anything. The top type is used to indicate URIs where no static type information is known. This allows a light type system, by allowing any URI as the subject or object of a predicate. For instance, the atomic proposition DOCUMENT may be too stringent for the predicates *writes* and *created*. Instead, the types $\mathtt{p}(\textsc{person}, \top)$ and $\mathtt{p}(\top, \textsc{Date})$ can be used where any resource can be written and any resource can be created.

The top type can only be applied to URIs. Literals have their own top type defined in the datatype standard [23]. By keeping these two top types distinct the type system for URIs and for literals do not interfere with each other.

The union type. The union type offers a compromise between atomic propositions and top. For instance, the predicate *writes* may apply to two atomic propositions ARTICLE and BOOK. The object of the predicate becomes the union type of the two atoms, as follows $\mathtt{p}(\textsc{person}, \textsc{article} \cup \textsc{book})$.

The class type. For classes which appear in data (dynamic classes) the class type is introduced. The class type is just another propositional type. Just as the propositional type PERSON is used to type URIs of type person, the propositional type CLASS is used to type URIs which are used as classes. The predicate *rdf:type* may be assigned the type $\mathtt{p}(\top, \textsc{class})$.

A common misunderstanding here arises from two approaches to using using classes, illustrated as follows. A URI *person* can be assigned the type CLASS. This represents a dynamic class in data which may be linked to and updated like any other URI. However, the URI *Hamish* may be assigned the type PERSON. In this case the class PERSON has been lifted to the type system, so can no longer be treated like any other URI.

The mistake at this point is to link the dynamic class *person* and the static type PERSON. By making this mistake the static type would be of type CLASS and the dynamic URI would be a type. The distinction between data and types is lost. The RDFS standard unfortunately makes this mistake by not distinguishing between data and types. The result is that serious paradoxes are breached, which may be partially resolved by a higher-order type system [45]. Higher-order type systems are technically complicated and add almost nothing to this application domain. By treating CLASS as a simple proposition these problems disappear.

Types for feeds and named graphs. To type named graphs and feeds the container type is introduced. Typically, the subject of a triple is what is described by the triple. By analogy, the subject of a simple sentence is what is described by the sentence. A feed can therefore indicate what type of resources its triples describe. For instance, a feed of articles contains triples with subjects of type article. A feed of articles is indicated by the type $\#\textsc{article}$.

Container types are well suited to feeds. For instance, BBC News delivers feeds of articles, Flickr delivers feeds of photos and Google Calendar delivers feeds of events. However, named graphs are intended to contain diverse triples. The most general container type $\#\top$ indicates a named graph with no restrictions on content.

Container types allow novel features. For instance, the *seeAlso* predicate indicates where to find more information about a resource. A suitable type would be $\mathtt{p}(\top, \#\top)$, which suggests that more information about the subject can be found in a named graph indicated by the object. Another novel type is $\#(\mathtt{p}(\top, \top) \cup \textsc{class})$ which suggests a feed of predicates and dynamic classes. This type of container is useful for presenting meta vocabularies using feeds.

Note that types for atomic propositions, predicates and union types are implicit in the RDFS standard. However, there is no standard type for named graphs, since named graphs and RDFS were introduced independently [4, 16]. The named graph type suggests one light approach to typing named graphs.

4.2.2. A subtype system based on RDFS.

Subtypes are essential for enabling some basic scenarios. The subtype system, presented in Fig. 10, defines a preorder over types. The subtype system enables interoperability by enabling different strengths of type system to coexist. For instance, data which is heavily typed can still be used if very little type information is required. This light approach to interoperability avoids typical data integration problems, such as the integration of schema with conflicting constraints [46].

$$\frac{\tau_0 \leq \tau_1}{\tau_0 \leq \tau_1 \cup \tau_2} \qquad \frac{\tau_0 \leq \tau_2}{\tau_0 \leq \tau_1 \cup \tau_2} \qquad \frac{\tau_0 \leq \tau_2 \quad \tau_1 \leq \tau_2}{\tau_0 \cup \tau_1 \leq \tau_2}$$

$$\tau \leq \top \qquad \frac{\tau_0' \leq \tau_0 \quad \tau_1' \leq \tau_1}{\mathrm{p}(\tau_0, \tau_1) \leq \mathrm{p}\left(\tau_0', \tau_1'\right)} \qquad \frac{\tau_0 \leq \tau_1 \quad \mathrm{D}_0 \leq \mathrm{D}_1}{\mathrm{p}(\tau_1, \mathrm{D}_1) \leq \mathrm{p}(\tau_0, \mathrm{D}_0)}$$

$$\frac{\tau_0 \leq \tau_1}{\#\tau_1 \leq \#\tau_0} \qquad \frac{\tau_0 \leq \tau_1 \quad \tau_1 \leq \tau_2}{\tau_0 \leq \tau_2} \qquad \tau \leq \tau$$

FIGURE 10. Axioms and rules of the subtype system: left injection, right injection, least upper bound, top, predicate, data predicate, feed, transitivity and reflexivity.

The subtype axioms. The subtype relation $\tau_0 \leq \tau_1$ can be read τ_0 is stronger than or equivalent to τ_1. So, if something is of type τ_0, then it is also of type τ_1. The basic axiom of the subtype system, reflexivity, states that every type is at least as strong as itself.

Interoperability of systems can be further enabled by application specific axioms. For instance an application may define types IMAGE and MEDIA. To indicate that an image can also be treated as media, the axiom IMAGE $\leq$ MEDIA can be included. Application specific subtype axioms always relate an atomic proposition to another atomic proposition. They correspond to the subClass predicate in RDFS, lifted to the type system [4].

Subtypes for union types. A union type indicates that a term is of one of two types. For instance, the type ARTICLE is a subtype of ARTICLE $\cup$ IMAGE. This is the left injection of a type into a union type. Similarly, the type IMAGE is a subtype of ARTICLE $\cup$ IMAGE by the symmetric right injection rule.

A union type is the least upper bound of two types. If ARTICLE $\leq$ MEDIA and IMAGE $\leq$ MEDIA are subtype axioms, then ARTICLE $\cup$ IMAGE is also bound above by MEDIA. A URI of type ARTICLE $\cup$ IMAGE means that the URI may vary between identifying either an article or an image.

Subtypes for top types. Every type is bound above by the top type. Thus both PERSON and p(PERSON, PERSON) are bound above by top. If a URI can be anything then a person or a predicate are both suitable. This eases the restriction in ontologies that URIs for resources and predicates are disjoint [41]. Similarly, the type CLASS is also bound above by the top type, so dynamic classes in the data are just URIs, like resources and predicates.

Subtypes for predicates. Predicate types are contravariant in both the subject and object. Contravariance switches the direction of the subtype relation. For instance, a predicate of type p(PERSON, $\top$) is also of type p(PERSON, ARTICLE). Contravariance allows the top type to be strengthened to the article type. So a predicate which allows anything as the object can certainly have an article as the object.

Data predicates are also contravariant in both arguments. For instance, a predicate of type $\mathrm{p}(\top, \text{STRING})$ can be used as a predicate of type p(PERSON, STRING). The subtype relation for datatypes is defined in the XML Datatypes standard [23]. For instance, a 'normalised string' from the standard can be used in place of a string. Because datatypes and types for URIs are separate the subtype systems do not interfere.

Subtypes for named graphs and feeds. With subtypes, the type of feeds containing more than one type of resource can be expressed. For instance, #(ARTICLE $\cup$ PHOTO) is a feed of articles and photos collectively have their union type. The feed constructor is contravariant meaning that the type of the content of the feed may be strengthened. For instance, a feed containing resources which are either articles or photos can be treated as a feed containing only articles by ignoring the photos. This is captured by the subtype relation #(ARTICLE $\cup$ PHOTO) $\leq$ #ARTICLE.

4.2.3. *Cut Elimination for the Subtype System.*

Cut elimination allows transitivity to be eliminated from subtype proofs. Cut elimination exhibits the categorical content of the subtype system. Arrows are normalised proof trees and objects are types. The theorem is justification for the rules of the subtype system.

THEOREM 4.1 (Cut elimination). *Given a proof of a subtype relation $\tau_0 \leq \tau_1$, there exists a normalised proof with the same conclusion which does not use cut (i.e., the transitivity rule in Fig. 10). Furthermore, proofs form a category such that union types are coproducts and top is terminal.*

The routine proof appears in the forthcoming thesis of the first author. It relies on using the completion of the subtype axioms. For instance, consider the atomic types, REPORTER, JOURNALIST, PERSON, and the subtype axioms REPORTER $\leq$ JOURNALIST and JOURNALIST $\leq$ PERSON. Cut is used to determine the subtype relation REPORTER $\leq$ PERSON, which must also be a primitive subtype axiom.

The Dedekind-MacNeille completion can be used to determine all the subtype axioms for atomic types [47]. The completion associates each atomic type with a set of atomic types which are less than all types greater than the type. This set is the principle ideal of the atomic type. The ideals form a complete join semi-lattice from which the subtype axioms can be efficiently recovered.

4.2.4. *Interoperability of Subtype Systems.*

A store may include Linked Data from more than one source with static type information. The subtype system enables interoperability between different subtype type systems. For instance, suppose there exist three stores, for distinct applications. Suppose that one store uses atomic types MUSICIAN and VENUE, while another store uses atomic types PERSON and LOCATION. A third store uses content from both servers, so must handle all four atomic types. Furthermore, the third server is given the subtype assumptions MUSICIAN $\leq$ PERSON and VENUE $\leq$ LOCATION, which improves interoperability of content from

both servers.

In the example involving three stores, the subtype systems of the first two stores can be extended to the subtype system of the third store. A subtype system δ_0 is defined to extend to a subtype system δ_1 if and only if the completion of subtype axioms in δ_0 is contained in the completion of subtype axioms in δ_1. Thus if δ_0 extends to δ_1, then all subtype assumptions with respect to δ_0 are subtype assumptions with respect to δ_1. Valid extensions can be checked efficiently using the Dedekind-MacNeille completion [47].

Subtypes ease restrictions imposed by types. Linked Data can involve data from stores with different subtype systems. By extending the subtype systems lightweight interoperability across diverse Linked Data systems can be achieved.

5. THE TYPED SYNDICATION CALCULUS

This section introduces the typed calculus. The typed calculus builds on the calculus described in previous sections. The extra type information assigns types to URIs and literals. The type system investigates the feasibility of lifting a small amount of the data to a type system. In particular, the typed calculus provides a model to evaluate the effectiveness of using RDF classes in a conventional type system for Linked Data [4].

A type system allows data and updates to be statically type checked. This type system ensures that URIs assigned a distinguished role are always used consistently. Notice that the data formats and query system exists in major deployments [12]. Also, a preliminary update system is under development, so is considered a requirement for Linked Data. The type system for literals has certain benefits for catching basic programming errors. However, the type system for URIs is a design decision, rather than a requirement. It depends on the application whether a type system for URIs should be used.

Although, the type system requires a design decision, the barriers imposed by the type system are less than those imposed by traditional database schema. For instance, an application may decide that a URI refers to an article, but the data associated with that article may change. An entirely new vocabulary might be used to replace the data about an article. However the URI of type article is remains a URI of type article. As long as the new vocabulary allows articles to be described, then the article can still be described.

As expected from an application dependant type system, care should be taken with what data is part of the type system. For instance, a URI may be of type person. It is reasonable to assume that a person will not morph into a bat, so person is a good choice of static type. However in an application, if a person is a banker, that person may become a bar tender. In this case role banker is too strong to be a static type, so should remain part of the data. For flexibility, the RDFS standard can also be considered at the level of data [48].

$$
\begin{array}{rll}
\alpha ::= & a : \tau & \text{name assignment} \\
| & x : \textsc{Data} & \text{variable assignment} \\
| & \epsilon & \text{empty environment} \\
| & \alpha, \alpha & \text{environment composition}
\end{array}
$$

$$\alpha, \epsilon \equiv \alpha \qquad \alpha_0, (\alpha_1, \alpha_2) \equiv (\alpha_0, \alpha_1), \alpha_2$$

$$\alpha_0, \alpha_1 \equiv \alpha_1, \alpha_0 \qquad \alpha, \alpha \equiv \alpha$$

FIGURE 11. A syntax for type environments and structural rules over type environments: unit, associativity, exchange, contraction and weakening.

5.1. Type Rules for Linked Data and Updates

RDF content and updates are typed to ensure that URIs used in RDF content are consistent with type assumptions. This section presents type rules for both RDF content and updates. The type rules for updates ensure that an update is only well typed if it updates well typed RDF content. A type rule for each construct of content is provided in Fig. 12. In a type judgement, the turnstile $\vdash$ separates the context on the left, represented by a type environment, from the well typed term on the right.

5.1.1. Type Environments for names and literals.
Type environments are finite partial functions from names to types. Syntactically, a type assignment is a name–type pair. If the pair *Alice* : person occurs, the URI *Alice* is said to be assigned type person. Similarly, type assignments allow variables to be assigned datatypes. The type environment is built from comma separated type assignments. Type environment composition is associative, with the empty environment as a unit, as indicated by the congruence over type environments in Fig. 11.

The type system uses the standard structural rules exchange and contraction. Exchange allows the order of type assignments to be changed. Contraction allows two identical assumptions can be reduced to a single assmption. For instance *Alice* : person, *Alice* : person is equivalent to *Alice* : person. Exchange and contraction are captured by the congruence over type environments in Fig. 11. These structural rules are standard for type systems. The congruence can always be applied to the environment on the left of the turnstile in a type judgement.

To ensure that environments are partial functions from URIs and variables to types and datatypes respectively, type environments must satisfy the following condition. If a URI or variable occurs in two type assignments within one type environment, then in each case the URI must be assigned the same type. Two type environments are compatible if and only if their composition still satisfies this constraint. For instance, the type environment *Alice* : person and the type environment *Alice* : book are incompatible. It is useful to denote the domain of a type environment α, by $\mathrm{dom}(\alpha)$.

$$a\colon \tau \vdash a\colon \tau \qquad x\colon \textsc{Data} \vdash x\colon \textsc{Data}$$

$$\frac{\alpha \vdash a\colon \tau_0 \quad \alpha \vdash p\colon \mathtt{p}(\tau_0,\tau_1) \quad \alpha \vdash b\colon \tau_1}{\alpha \vdash (a\ p\ b)\colon \tau_0}$$

$$\frac{\alpha \vdash a\colon \tau \quad \alpha \vdash p\colon \mathtt{p}(\tau,\textsc{Data}) \quad \alpha \vdash e\colon \textsc{Data}}{\alpha \vdash (a\ p\ e)\colon \tau}$$

$$\frac{\alpha \vdash a\colon \#\tau \quad \alpha \vdash C\colon \tau}{\alpha \vdash \mathcal{G}_a C\colon \#\tau} \qquad \frac{\alpha, a\colon \tau_0 \vdash P\colon \tau_1}{\alpha \vdash \bigwedge a\colon \tau_0.P\colon \tau_1}$$

$$\vdash \bot\colon \tau \qquad \frac{\alpha \vdash P\colon \tau \quad \alpha \vdash P\colon \tau}{\alpha \vdash P \parr P\colon \tau}$$

$$\frac{\alpha_0 \vdash P\colon \tau}{\alpha_0, \alpha_1 \vdash P\colon \tau} \qquad \frac{\alpha \vdash P\colon \tau_0 \quad \tau_0 \leq \tau_1}{\alpha \vdash P\colon \tau_1}$$

FIGURE 12. Type rules for RDF content and named graphs: name assignment, variable assignment, type triple, type triple with literal object, type named graph, type blank node, type nothing, type par, weakening and subsumption.

5.1.2. Axioms, weakening, subsumption and literals.
The type system uses the standard axioms, which states that, assuming that a URI is of a particular type, the URI is of the given type. Thus if *Ossetia* is assumed to be an article, then *Ossetia* is an article. The same shape of axiom applies to variables, but types and datatypes do not overlap so the axioms are separate.

The weaken environment rule allows unused type assignments to be added to the context. So weakening can be applied to an axiom to give, if *Ossetia* is an article and *Exchange* is an article, then *Ossetia* is an article. The subsumption rule allows the subtype system to be applied at any point. So, if *Ossetia* is an article, then *Ossetia* is anything, by the top axiom from the subtype system. Weakening and subsumption enable the intuitive presentation of the type system in Fig. 12.

Data literals are defined independently from the calculus. For the purpose of examples, intuitive type judgements are assumed to hold, such as $\vdash$ '09-09-2008': DATE or $\vdash$ `'Hamish'`: STRING. Technical details are left to the standards [23].

5.1.3. Type rules for triples and simple RDF content.
Predicate types indicate the subject and object of predicate. The type rules for triples ensure that the subject and object are of the correct type. For instance, suppose that *author* is of type $\mathtt{p}(\textsc{article}, \textsc{person})$. If *Ossetia* is of type ARTICLE and *Hamish* is type PERSON then the triple (*Ossetia author Hamish*) is well typed.

Data predicates are typed using a similar rule. The data predicate type indicates the type of the subject and the datatype of the object. For instance, suppose that the predicate *name* is of type $\mathtt{p}(\textsc{person}, \textsc{String})$. Given that the name *Hamish* is of type PERSON and the literal `'Hamish'` is of data type STRING, then the triple (*Hamish name* `'Hamish'`) is well typed.

In both cases a well typed triple takes on the type of the subject. So the first triple above is of type ARTICLE and the second triple is of type PERSON. Only the type of the subject of the triple is indicated. This allows collections of triples with the same type of subject to be identified. For instance, some content may consist of triples with subjects which are articles. As noted in Sec. 4, typing triples according to the type of the subject is an application specific choice. The top type can be used to indicate that the type of the subject is irrelevant.

Triples and processes are composed using par. The type rule for par allows two triples of the same type to be composed. For instance, the two triples in this section can be composed. Subtyping is applied to weaken the types of both triples to the appropriate union type.

$$\begin{array}{l} \textit{Hamish}\colon \textsc{person}, \textit{Ossetia}\colon \textsc{article}, \\ \textit{author}\colon \mathtt{p}(\textsc{article}, \textsc{person}), \textit{name}\colon \mathtt{p}(\textsc{person}, \textsc{String}) \end{array}$$
$$\vdash \begin{array}{l} (\textit{Hamish give\_name}\ \texttt{`Hamish'}), \\ (\textit{Ossetia author Hamish}) \end{array} \colon \textsc{person} \cup \textsc{article}$$

The type judgement indicates that the locality contains triples which describe either people or articles.

5.1.4. Type rules for blank nodes.
The blank node quantifier binds names which represent blank nodes. In the typed calculus, bound names are annotated with a type information. The rule for blank nodes first types some RDF assuming that the blank node is a normal URI. The rule then internalises the type information as a quantifier.

The following example internalises three type assumptions, which represent blank nodes. Two blank nodes indicate that they are two separate events. The third blank nodes is of the top type. The scope of the quantifier indicates that the same resource judged both events but no information is known about that resource.

$$\textit{judge}\colon \mathtt{p}(\top,\top), \textit{date}\colon \mathtt{p}(\top, \textsc{Date})$$
$$\vdash \bigwedge a\colon \top. \left(\begin{array}{l} \bigwedge \textit{event1}\colon \textsc{event}. \\ \left(\begin{array}{l} (\textit{event1 judge } a), \\ (\textit{event1 date } \text{`13-01-2011'}) \end{array} \right), \\ \bigwedge \textit{event2}\colon \textsc{event}. \\ \left( \begin{array}{l} (\textit{event2 judge } a), \\ (\textit{event2 date } \text{`14-01-2011'}) \end{array} \right) \end{array} \right) \colon \textsc{event}$$

Subjects bound by typed blank nodes help determine the type of the content. In the above example, since the subject of all triples are events, the whole resource is of type event.

5.1.5. Type rules for named graphs.
The type of a named graph indicates the type of content that may be contained in the named graph. For instance, a named graph of type #ARTICLE has content of type ARTICLE. The following named graph models a feed named *Caucuses*,

$$\frac{\alpha \vdash G : \tau}{\alpha \vdash |G| : \tau} \qquad \frac{\alpha \vdash G : \tau}{\alpha \vdash G^{\perp} : \tau} \qquad \frac{\alpha \vdash \phi}{\alpha \vdash \phi : \tau}$$

$$\frac{\alpha \vdash S : \tau \quad \alpha \vdash T : \tau}{\alpha \vdash S \oplus T : \tau} \qquad \frac{\alpha \vdash S : \tau \quad \alpha \vdash T : \tau}{\alpha \vdash S \otimes T : \tau}$$

$$\frac{\alpha, a : \tau \vdash S : \tau}{\alpha \vdash \bigvee a : \tau . S : \tau} \qquad \frac{\alpha, x : \mathrm{D} \vdash S : \tau}{\alpha \vdash \bigvee x : \mathrm{D} . S : \tau} \qquad \frac{\alpha \vdash S : \tau}{\alpha \vdash * S : \tau}$$

FIGURE 13. Type rules for updates: type ask, type delete, type filter, type choice, type join, type select name, type select literal, type exponential.

appearing below. The four triples in the named graph have subjects which are articles, so the feed is well typed.

$$\begin{array}{l} \mathit{title} : \mathrm{p}(\textsc{article}, \textsc{String}), \mathit{published} : \mathrm{p}(\textsc{article}, \textsc{Date}), \\ \mathit{editor} : \mathrm{p}(\#\textsc{article}, \textsc{person}), \mathit{Caucuses} : \#\textsc{article}, \\ \mathit{Hamish} : \textsc{person}, \mathit{Ossetia} : \textsc{article}, \mathit{exchange} : \textsc{article} \end{array}$$

$$\vdash \; (\mathit{Caucuses\ editor\ Hamish}), \quad \mathcal{G}_{\mathit{Caucuses}} \left(\begin{array}{l} (\mathit{Ossetia\ title}\ \texttt{`Ossetia invaded'}), \\ (\mathit{Ossetia\ published}\ \text{`09-09-2008'}), \\ (\mathit{exchange\ title}\ \texttt{`Stock collapse'}), \\ (\mathit{exchange\ published}\ \text{`08-10-2008'}) \end{array} \right) : \top$$

A URIs for a named graph are be treated like any other URI. Thus triples can be assigned to named graphs, such as the triple which indicates the editor of the named graph in the example above.

5.1.6. Type rules for updates and queries.
A delete, insert or a query have the same type as the RDF content that they act on. This ensures that only RDF content which makes sense can be updated or queried. For instance, the following type judgement holds, which indicates a resource and an update which intends to replace *in* with *out*.

$$\begin{array}{l} \mathit{Dmitri} : \textsc{person}, \\ \mathit{status} : \mathrm{p}(\top, \top), \\ \mathit{in} : \top, \mathit{out} : \top \end{array} \quad \vdash \quad \begin{array}{l} (\mathit{Dmitri\ status\ in}), \\ (\mathit{Dmitri\ status\ in})^{\perp} \\ (\mathit{Dmitri\ status\ out}) \end{array} \quad : \textsc{person}$$

In the example above deleted and inserted data has a subject of type person, so the update maintains the type of the context. A type checker can detect malformed triples in a delete or insert before the update is applied.

5.1.7. Type rules for select quantifiers.
Select quantifiers consist of a type assignment and an update. The type environment constrains the type of name to select. The example below selects a URI of type person. The type information permits the assumption that a selected URI will be of type person. The object of the triple in both the query and the insert are expected to be of type person.

$$\begin{array}{l} \mathit{article} : \textsc{article}, \mathit{editor} : \mathrm{p}(\textsc{article}, \textsc{person}), \\ \mathit{club} : \top, \mathit{member} : \mathrm{p}(\top, \textsc{person}) \end{array} \quad \vdash \quad \begin{array}{l} \bigvee p : \textsc{person} . \left(\begin{array}{l} |(\mathit{article\ editor\ p})| \\ (\mathit{club\ member\ p}) \end{array} \right), \\ \bigwedge \mathit{Hamish} : \textsc{person} . (\mathit{article\ editor\ Hamish}) \end{array} \quad : \top$$

The above update is in the presence of some data where a blank node appears. The type assignment for the blank node is the same as the type assignment for the select quantifier.

5.1.8. Type rules for literals in filters and selects.
A constraint which contains variables or names can be typed. For instance, under the assumption that $x : \textsc{Date}$, constraint $x \leq$ '01-01-1950' is well typed. A date literal substituted for x results in a constraint such as '01-05-1886' $\leq$ '01-01-1950', which is also well typed. In the example below, the select quantifier introduces the assumption that x is a date. This type assumption allows the filter and triple in the query to be typed. The update satisfies the following type judgement.

$$\begin{array}{l} \mathit{Kidnapped} : \textsc{book}, \mathit{published} : \mathrm{p}(\textsc{book}, \textsc{Date}), \\ \mathit{note} : \mathrm{p}(\top, \top), \mathit{classic} : \top \end{array} \quad \vdash \quad \begin{array}{l} \bigvee x : \textsc{Date} . \bigvee \mathit{book} : \textsc{book} . \\ \left(\begin{array}{l} (x \leq \text{'01-01-1950'}) \\ |(\mathit{book\ published\ x})| \\ (\mathit{book\ note\ classic}) \end{array} \right), \\ (\mathit{Kidnapped\ published}\ \text{'01-05-1886'}) \end{array} \quad : \textsc{book}$$

Typing literals is the minimal type system for updates. Literals can still be typed without application specific type information for URIs.

5.1.9. Type rules for joins, choice and iteration.
Joins ensure that two well typed updates are applied atomically. A choice between two well typed updates is presented. A join or choice assumes a type that both components can assume. Iteration does not affect the type of a process. In the following example all components are of type person so the whole update is of type person.

$$\begin{array}{l} \mathit{guard} : \textsc{class}, \mathit{attendant} : \textsc{class}, \mathit{porter} : \textsc{class}, \\ \mathit{type} : \mathrm{p}(\top, \textsc{class}) \end{array} \quad \vdash \quad \begin{array}{l} * \bigvee a : \textsc{person} . \left(\begin{array}{l} \left(\begin{array}{l} (a\ \mathit{type\ attendant})^{\perp} \\ \oplus \\ (a\ \mathit{type\ guard})^{\perp} \end{array} \right) \\ (a\ \mathit{type\ porter}) \end{array} \right), \\ \bigvee b : \textsc{person} . (b\ \mathit{type\ attendant}), \\ \bigvee c : \textsc{person} . (c\ \mathit{type\ guard}) \end{array} \quad : \textsc{person}$$

The above example demonstrates a mix of static classes as types and dynamic classes as data. The people are always people, but their role changes.

5.2. Algorithmic Typing for the Calculus

In the type system in the previous section, the subsumption and weakening rules can be applied at any point. An

$$\frac{\tau_0 \leq \tau_1}{a\colon \tau_0 \Vdash a\colon \tau_1} \qquad \frac{\alpha + a\colon \tau_1 \Vdash P\colon \tau_0}{\alpha \Vdash \bigwedge a\colon \tau_1.P\colon \tau_0}\ a \in \mathrm{fn}(P)$$

$$\frac{\alpha \Vdash P\colon \tau_0}{\alpha \Vdash \bigwedge a\colon \tau_1.P\colon \tau_0}\ a \notin \mathrm{fn}(P) \qquad \frac{\alpha_0 \Vdash U\colon \tau \quad \alpha_1 \Vdash V\colon \tau}{\alpha_0, \alpha_1 \Vdash U \otimes V\colon \tau}$$

$$\frac{\alpha_0 \Vdash a\colon \tau_0 \quad \alpha_1 \Vdash p\colon \mathrm{p}(\tau_0, \tau_1) \quad \alpha_2 \Vdash b\colon \tau_1 \quad \tau_0 \leq \tau_2}{\alpha_0, \alpha_1, \alpha_2 \Vdash (a\ p\ b)\ :\ \tau_2}$$

FIGURE 14. Variations in rules for the algorithmic type system.

algorithmic type system controls the use of subsumption and weakening. Subsumption can instead be applied as early as possible. Weakening can be applied as late as possible. The algorithmic type system can be less intuitive but is syntax directed, so easier to work with for proofs and type inference algorithms [49].

Key differences between the rules of the type system and the algorithmic type system are presented in Fig. 14. The first variation is that the axioms immediately weaken the type to the correct type required. The second variations is that, when two terms are composed, the type environments are merged whenever they are compatible. This is characterised by the type rule for join. Merging environments avoids weakening both environments before updates are combined. The third variation is that the blank node and select rules, which internalise the type environment permit weakening of the environment. This permitted weakening is expressed using the congruence over environments. Exchange and contraction still apply and + indicates disjoint environments.

The soundness and completeness of the algorithmic type system with respect to the intuitive type system ensures that results carry from one system to the other. The proof begins with a technical lemma. The lemma demonstrates that, for the algorithmic type system, the environment on the left of the turnstile covers exactly the URIs that occur free in the term.

Lemma 5.1. *If* $\alpha \Vdash P\colon \tau$ *then* $\mathrm{fn}(P) = \mathrm{dom}(\alpha)$.

Soundness of the algorithmic type system is established by a straightforward rewrite from an algorithmic type tree to a normal type tree. The effect of the typing is preserved by the rewrite.

Theorem 5.1 (Soundness of algorithmic typing). *If* $\alpha \Vdash P\colon \tau$ *then* $\alpha \vdash P\colon \tau$.

Proof. Soundness is established by a straight forward translation of proof trees. Each algorithmic type rule which involves subtypes can be replaced by their equivalent type rule followed by a subsumption link.

For blank nodes, if $a \notin \mathrm{fn}(P)$ then the algorithmic type rule is transformed into the type rule, preceded by application of weakening, as follows.

$$\frac{\overset{\pi}{\alpha \Vdash P\colon \tau_0}}{\alpha \Vdash \bigwedge a\colon \tau_1.P\colon \tau_0} \quad \text{yields} \quad \frac{\overset{\pi}{\alpha \vdash P\colon \tau_0}}{\dfrac{\alpha_0, a\colon \tau_1 \vdash P\colon \tau_0}{\alpha_0 \vdash \bigwedge a\colon \tau_1.P\colon \tau_0}}$$

Hence, each algorithmic type tree corresponds to a type tree with the same conclusion. □

The proof of completeness of the algorithmic type system is a transformation of proof trees which pushes subsumption towards the leaves and weakening towards the root of a type tree.

Theorem 5.2 (Completeness of algorithmic subtyping). *If* $\alpha \vdash P\colon \tau$*, then there exists some* α_0, α_1 *such that* $\alpha_0, \alpha_1 \equiv \alpha$ *and* $\alpha_0 \Vdash P\colon \tau$.

Proof. The transformation $[\![\cdot]\!]$ pushes subsumption rules as deep as possible into the proof tree and suspends weakening.

There are two special cases. If two subsumption links appear consecutively, then they can be performed in a single subsumption link using cut in the subtype system. Also, weakening rules can be deleted, since weakening is controlled by the induction hypothesis.

For axioms, subsumption is absorbed by the algorithmic rule.

$$\left[\!\!\left[\frac{a\colon \tau_0 \vdash a\colon \tau_0 \qquad \overset{\pi}{\vdash \tau_0 \leq \tau_1}}{a\colon \tau_0 \vdash a\colon \tau_1} \right]\!\!\right] \text{yields} \ \frac{\overset{\pi}{\vdash \tau_0 \leq \tau_1}}{a\colon \tau_0 \Vdash a\colon \tau_1}$$

For join, the subsumption link is pushed up each branch. By structural induction, α_0, α_1 are type environments, such that $\alpha \equiv \alpha_0, \alpha_0'$ and $\alpha \equiv \alpha_1, \alpha_1'$ and also each forms the premises of the conclusions of the respective branches of the resulting tree.

$$\left[\!\!\left[\frac{\dfrac{\overset{\pi_0}{\alpha \vdash P\colon \tau_0} \qquad \overset{\pi_1}{\alpha \vdash Q\colon \tau_0}}{\alpha \vdash P \otimes Q\colon \tau_0} \qquad \overset{\pi_2}{\vdash \tau_0 \leq \tau_1}}{\alpha \vdash P \otimes Q\colon \tau_1} \right]\!\!\right]$$

yields

$$\frac{\alpha_0 \Vdash P\colon \tau_1 \quad \alpha_1 \Vdash Q\colon \tau_1}{\alpha_0, \alpha_1 \Vdash P \otimes Q\colon \tau_1}$$

where

$$\alpha_0 \Vdash P\colon \tau_1 = \left[\!\!\left[\frac{\overset{\pi_0}{\alpha \vdash P\colon \tau_0} \qquad \overset{\pi_2}{\vdash \tau_0 \leq \tau_1}}{\alpha \vdash P\colon \tau_1} \right]\!\!\right]$$

and

$$\alpha_1 \Vdash Q\colon \tau_1 = \left[\!\!\left[\frac{\overset{\pi_1}{\alpha \vdash Q\colon \tau_0} \qquad \overset{\pi_2}{\vdash \tau_0 \leq \tau_1}}{\alpha \vdash Q\colon \tau_1} \right]\!\!\right]$$

The triple rule absorbs a subsumption link. By induction, there exists type environments $\alpha_0, \alpha_1, \alpha_2$ such that $\alpha \equiv$

$\alpha_0, \alpha_1, \alpha_2, \alpha'$ and the following transformation holds.

$$\left[\!\!\left[\dfrac{\dfrac{\pi_0 \quad \pi_1 \quad \pi_2}{\alpha \vdash a\colon \tau_0 \quad \alpha \vdash p\colon \mathrm{p}(\tau_0,\tau_1) \quad \alpha \vdash b\colon \tau_1}}{\alpha \vdash (a\ p\ b) : \tau_0} \quad \dfrac{\pi_3}{\vdash \tau_0 \leq \tau_2}}{\alpha \vdash (a\ p\ b) : \tau_2} \right]\!\!\right]$$

yields

$$\dfrac{\alpha_0 \Vdash a\colon \tau_0 \quad \alpha_1 \Vdash p\colon \mathrm{p}(\tau_0,\tau_1) \quad \alpha_2 \Vdash b\colon \tau_1 \quad \dfrac{\pi_3}{\vdash \tau_0 \leq \tau_2}}{\alpha_0, \alpha_1, \alpha_2 \Vdash (a\ p\ b) : \tau_2}$$

For blank nodes, subsumption is pushed straight up the tree. Consider the following proof tree.

$$\left[\!\!\left[\dfrac{\dfrac{\pi_0}{\alpha, a\colon \tau_2 \vdash P\colon \tau_0}}{\alpha \vdash \bigwedge a\colon \tau_2.P\colon \tau_0} \quad \dfrac{\pi_1}{\vdash \tau_0 \leq \tau_1}}{\alpha \vdash \bigwedge a\colon \tau_2.P\colon \tau_1} \right]\!\!\right]$$

By induction, there is some α_1 and α' such that the following transformation holds, where $\alpha_1, \alpha' \equiv \alpha, a\colon \tau_2$.

$$\left[\!\!\left[\dfrac{\dfrac{\pi_0 \quad \pi_1}{\alpha, a\colon \tau_2 \vdash P\colon \tau_0 \quad \vdash \tau_0 \leq \tau_1}}{\alpha, a\colon \tau_2 \vdash P\colon \tau_1} \right]\!\!\right] = \quad \dfrac{\pi_3}{\alpha_1 \Vdash P\colon \tau_1}$$

If $a \notin \mathrm{fn}(P)$, then the following proof tree holds.

$$\dfrac{\dfrac{\pi_3}{\alpha_1 \Vdash P\colon \tau_1}}{\alpha_1 \Vdash \bigwedge a\colon \tau_2.P\colon \tau_1}$$

If $a \in \mathrm{fn}(P)$ then, by Lemma 5.1 there exists α_1' such that $\alpha_1 \equiv \alpha_1' + a\colon \tau_2$, and the following holds.

$$\dfrac{\dfrac{\pi_3}{\alpha_1' + a\colon \tau_2 \Vdash P\colon \tau_1}}{\alpha_1' \Vdash \bigwedge a\colon \tau_2.P\colon \tau_1}$$

Further cases are similar to the above. Thus by induction over the proof trees a transformation from any type tree to an algorithmic type tree exists. □

The algorithmic type system demonstrates that type checking is syntax directed. Even for a light type system, type checking a store at each operational step is costly. A feasible approach is to type check updates.

6. THE TYPED OPERATIONAL SEMANTICS

Given a well typed update, the expectation is that Linked Data need only be typed once. Well typed updates applied to well typed Linked Data should result in well typed Linked Data, without the need to recheck the Linked Data. The light type system works locally, in the sense that the correctness of one triple is not affected by other triples. Similarly, the commitment relation over updates describes the local behaviour of updates, since unused triples are ignored. The type system and commitment relations therefore work at the same level of granularity, so are compatible.

$$P \parr \bot \equiv P \qquad P \parr (Q \parr R) \equiv (P \parr Q) \parr R$$

$$P \parr Q \equiv Q \parr P \qquad \mathcal{G}_a(C \parr D) \equiv \mathcal{G}_a C \parr \mathcal{G}_a D$$

$$\bigwedge a\colon \tau.(P \parr Q) \equiv \bigwedge a\colon \tau.P \parr Q \quad a \notin \mathrm{fn}(Q)$$

$$\bigwedge a\colon \tau.\bot \equiv \bot \qquad \begin{array}{l} \bigwedge a\colon \tau_0.\bigwedge b\colon \tau_1.P \\ \quad \equiv \bigwedge b\colon \tau_1.\bigwedge a\colon \tau_0.P \end{array} \quad a \neq b \text{ or } \tau_0 = \tau_1$$

FIGURE 15. The structural congruence over content and processes: unit, associativity, commutativity, split named graph, distribute blank node, eliminate blank node and commute blank node.

This section demonstrates that the specification of atomic commitments can be extended to ensure that the type system and the commitment relation are compatible. The typed commitment rules introduce minimal assumptions about the context. The assumptions about the context are that names selected in an update are of the correct type. This amounts to a minimal dynamic type check on selected names. Under minimal assumptions about the context, type judgements are preserved by the dynamically typed commitment relation, as verified by Theorem 6.1.

The typed operational semantics are defined by combining the following components. α-conversion of bound names, the structural congruence in Fig. 15 and a typed commitment relation in Fig. 16. Examples throughout this section illustrate the operational behaviour of typed updates.

6.1. The Structural Congruence for Typed Linked Data

The structural congruence for typed content is gathered in Fig. 15. The structural congruence captures the commutative monoid formed by par and nothing, which is used for both RDF content and processes. The structural congruence allows blank node quantifiers to distribute over tensor and be eliminated in the presence of nothing. Compatible blank node quantifiers may be swapped. The side condition for swapping type assignments ensures that the composition of the assignments form a partial function. Type environments must be partial functions. The split named graph rule allows named graphs to be decomposed for fine grained updates.

The first type preservation result verifies that the structural congruence preserves types. Thus given a well typed process, processes structurally congruent to the process are well typed. Lemma 6.1 verifies this compatibility between the structural congruence and the type system. The proof makes use of algorithmic typing to simplify proofs.

Lemma 6.1 (Structural congruence preserves types). *Assuming that $P \equiv Q$, $\alpha \vdash P\colon \tau$ if and only if $\alpha \vdash Q\colon \tau$.*

Proof. For each algorithmic type tree and rule of structural

congruence, an algorithmic type tree of the equivalent process can be constructed, by Theorems 5.1 and 5.2.

For distributivity of blank nodes over par, assume that $a \notin \mathrm{fn}(Q)$. Also assume that $a \in \mathrm{fn}(P)$ and the following proof tree holds.

$$\dfrac{\dfrac{\alpha_0 + a\colon \tau_0 \Vdash P\colon \tau_1}{\alpha_0 \Vdash \bigwedge a\colon \tau_0.P\colon \tau_1} \quad \alpha_1 \Vdash Q\colon \tau_1}{\alpha_0, \alpha_1 \Vdash \bigwedge a\colon \tau_0.P \parr Q\colon \tau_1}$$

Now $a \notin \mathrm{dom}(\alpha_1)$, by Lemma 5.1, and $a \notin \mathrm{dom}(\alpha_0)$ thus $a \notin \mathrm{dom}(\alpha_0, \alpha_1)$, so the following proof tree holds. The converse is immediate.

$$\dfrac{\dfrac{\alpha_0 + a\colon \tau_0 \Vdash P\colon \tau_1 \quad \alpha_1 \Vdash Q\colon \tau_1}{(\alpha_0, \alpha_1) + a\colon \tau_0 \Vdash P \parr Q\colon \tau_1}}{\alpha_0, \alpha_1 \Vdash \bigwedge a\colon \tau_0.(P \parr Q)\colon \tau_1}$$

Now, assume that $a \notin \mathrm{fn}(P)$ and the following proof tree holds. Clearly $a \notin \mathrm{fn}(P \parr Q)$ so the following proof trees can be interchanged.

$$\dfrac{\dfrac{\alpha_0 \Vdash P\colon \tau_1}{\alpha_0 \Vdash \bigwedge a\colon \tau_0.P\colon \tau_1} \quad \alpha_1 \Vdash Q\colon \tau_1}{\alpha_0, \alpha_1 \Vdash \bigwedge a\colon \tau_0.P \parr Q\colon \tau_1}$$

iff

$$\dfrac{\dfrac{\alpha_0 \Vdash P\colon \tau_1 \quad \alpha_1 \Vdash Q\colon \tau_1}{\alpha_0, \alpha_1 \Vdash P \parr Q\colon \tau_1}}{\alpha_0, \alpha_1 \Vdash \bigwedge a\colon \tau_0.(P \parr Q)\colon \tau_1}$$

Remaining cases are straight forward. The result follows by induction over the derivation of an equivalence. □

The structural congruence covers the reorganisation of content and processes. The structural congruence is always reversible. In contrast, the effect of updates are generally irreversible, so are captured by a commitment relation.

6.2. Typed Atomic Commitments

Atomic commitments were introduced in Sec. 2 to specify an operational semantics for queries and updates over Linked Data. In Sec. 3, atomic commitments were extended to cover key features for syndication. In this section, atomic commitments are extended with a type environment, called the context. Otherwise, the role of atomic commitments remains the same. The process on the left indicates exactly the processes consumed. The processes on the right indicates the exact processes which replace the processes consumed.

The context for typed atomic commitments indicates a minimal dynamic type check required by a commitment. By minimising dynamic type checks, a feasible type system is enabled. The context represents these minimal type checks as a type environment. Any processes in the vicinity of the commitment must agree on the the assignments of names to types in the context.

For instance, a context for a commitment relation may indicate that the name *Burns* is of type WRITER. However, if it is assumed elsewhere that *Burns* is a PERSON, then the commitment cannot be applied, since the required context indicates a stronger type. The rules for typed commitments are presented in Fig. 16. The higher-order π-calculus similarly constrains the context of a transition using type environments [50].

6.2.1. *Fully type safe commitments*

Assuming that a process is well typed, a commitment which only uses axioms requires no dynamic type checks. When there are no type checks most other rules behave like the untyped calculus. For instance, in the example below the deletes and inserts have an empty context, so their join has an empty context.

$$\vdash \left(\begin{array}{l} (\mathit{studio\ status\ closed})^{\perp} \\ (\mathit{studio\ status\ open}) \\ \mathcal{G}_{\mathit{studio}}(\mathit{Dmitri\ status\ out})^{\perp} \\ \mathcal{G}_{\mathit{studio}}(\mathit{Dmitri\ status\ in}) \end{array}\right), \quad (\mathit{studio\ status\ closed}), \ \mathcal{G}_{\mathit{studio}}(\mathit{Dmitri\ status\ out}) \rhd (\mathit{studio\ status\ open}), \ \mathcal{G}_{\mathit{studio}}(\mathit{Dmitri\ status\ in})$$

Because the context above is empty, any environment which types the process before the commitment also types the process after the commitment.

6.2.2. *The dynamically typed select quantifier.*

The select quantifier introduces the need for dynamic type checks. A select quantifier annotates a name with a type. The type annotation imposes an upper bound on the type of the selected name. For instance, the select quantifier below requires that the selected name is of type PERSON. The commitment selects the name *Dmitri* according to the triple to be deleted. However, the given process does not indicate that *Dmitri* is of type PERSON. The missing assumption is indicated by the context in front of the commitment.

$$\mathit{Dmitri}\colon \textsc{person} \vdash \begin{array}{l} \bigvee a\colon \textsc{person}. \\ (\mathit{Hamish\ knows\ } a)^{\perp}, \\ (\mathit{Hamish\ knows\ Dmitri}) \end{array} \quad \rhd \perp$$

The context above indicates that the commitment can only be applied safely when *Dmitri* is of type PERSON. Further information required to type the process, such as *knows* is of type $\mathsf{p}(\textsc{person}, \textsc{person})$ and *Hamish* is of type PERSON, is not required for the commitment.

6.2.3. *Joined commitments with non-empty context.*

Join is used to synchronise updates. If two updates each require a context, then the join of the updates composes the contexts. For instance, the following example consists of two commitments where each requires a URI to be of type person. The commitments are joined so the context indicates

$$\vdash C \parr C^{\perp} \rhd \perp \qquad \vdash C \rhd C \qquad \vdash C \parr |C| \rhd C$$

$$\frac{\alpha \vdash P \parr U \rhd Q}{\alpha \vdash P \parr (U \oplus V) \rhd Q} \qquad \frac{\alpha \vdash P \parr V \rhd Q}{\alpha \vdash P \parr (U \oplus V) \rhd Q}$$

$$\frac{\alpha_0 \vdash P \parr U \rhd P' \quad \alpha_1 \vdash Q \parr V \rhd Q'}{\alpha_0 + \alpha_1 \vdash P \parr Q \parr (U \otimes V) \rhd P' \parr Q'} \qquad \frac{\vDash \phi}{\vdash \phi \rhd \perp}$$

$$\frac{\alpha \vdash P \parr U \rhd Q}{\alpha \vdash P \parr *U \rhd Q} \qquad \frac{\alpha \vdash P \parr (*U \otimes *U) \rhd Q}{\alpha \vdash P \parr *U \rhd Q}$$

$$\vdash *U \rhd \perp \qquad \frac{\alpha \vdash P \parr U\{{}^{b}/_{a}\} \rhd Q \quad \vdash \tau_0 \leq \tau_1}{\alpha + b\colon \tau_0 \vdash P \parr \bigvee a\colon \tau_1 . U \rhd Q}$$

$$\frac{\alpha \vdash P \parr U\{{}^{v}/_{x}\} \rhd Q \quad \vdash v\colon \mathrm{D}}{\alpha \vdash P \parr \bigvee x\colon \mathrm{D} . U \rhd Q}$$

$$\frac{\alpha_0 \vdash P \rhd P' \quad \alpha_1 \vdash Q \rhd Q'}{\alpha_0, \alpha_1 \vdash P \parr Q \rhd P' \parr Q'}$$

$$\frac{\alpha + a\colon \tau \vdash P \parr Q \rhd P' \parr Q'}{\alpha \vdash \bigwedge a\colon \tau . P \parr Q \rhd \bigwedge a\colon \tau . P' \parr Q'} \; a \notin \mathrm{fn}(Q, Q')$$

$$\frac{\alpha + a\colon \tau \vdash \mathcal{G}_b P \parr Q \rhd \mathcal{G}_b P' \parr Q'}{\alpha \vdash \mathcal{G}_b \bigwedge a\colon \tau . P \parr Q \rhd \mathcal{G}_b \bigwedge a\colon \tau . P' \parr Q'} \; a \notin \mathrm{fn}(Q, Q', b)$$

FIGURE 16. The axioms and rules form atomic commitments: delete axiom, insert axiom, query axiom, choose left rule, choose right rule, join rule, filter axiom, dereliction rule, contraction rule, weakening axiom, select name rule, select literal rule, mix rule, blank node rule, named graph rule.

that both URIs are of type person.

$$\begin{array}{l} \mathit{user1}\colon \textsc{person}, \mathit{user2}\colon \textsc{person} \\ \quad (\mathit{user1\ status\ busy}), \\ \quad (\mathit{user2\ status\ ready}), \\ \vdash \left(\begin{array}{l} \bigvee a\colon \textsc{person}. \\ \left(\begin{array}{l} (a\ \mathit{status\ busy})^{\perp} \\ (a\ \mathit{status\ ready}) \end{array} \right) \\ \bigvee b\colon \textsc{person}. \\ \left(\begin{array}{l} (b\ \mathit{status\ ready})^{\perp} \\ (b\ \mathit{status\ busy}) \end{array} \right) \end{array} \right) \rhd \begin{array}{l} (\mathit{user1\ status\ ready}), \\ (\mathit{user2\ status\ busy}) \end{array} \end{array}$$

In the above example, the names in the context are distinct. The join rule forces combined contexts to be disjoint. By forcing disjoint contexts, two select quantifiers cannot discover the same name. Consequently, the more controlled 'select distinct' quantifier is modelled from SPARQL Query [5].

In contrast, the join rule in Sec. 2 models the normal select quantifier in SPARQL Query. The normal select allows different selects to discover the same name. The normal quantifier can be achieved here by removing the constraint that joined contexts are disjoint. Removing the constraint allows the same name to appear in the combined environment, hence contraction may be applied. Contraction allows two different select quantifiers to share the same resource. The two variations on the select quantifier may coexist by extending the type environment in the calculus. A more subtle type environment can control the use of contraction, as investigated in the logic of bunched implications [51].

6.2.4. Dynamic type checks for selected literals.
The select literal quantifier annotates a variable with a data type. The annotation constrains the type of a literal discovered using the select literal rule. To enforce the constraint, the select literal rule dynamically type checks the selected literal. In the example below, the literal input by the select quantifier is successfully checked to be a date. The syntax of the literal is enough information to check the type.

$$\begin{array}{l} \mathit{Kidnapped}\colon \textsc{book} \\ \quad \bigvee x\colon \textsc{Date}. \bigvee \mathit{book}\colon \textsc{book}. \\ \vdash \left(\begin{array}{l} (x \leq \text{'01-01-1950'}) \\ |(\mathit{book\ published}\ x)| \\ (\mathit{book\ status\ classic}) \end{array} \right), \\ \quad (\mathit{Kidnapped\ published}\ \text{'01-05-1886'}) \\ \qquad \rhd\ (\mathit{Kidnapped\ published}\ \text{'01-05-1886'}), \\ \qquad\quad (\mathit{Kidnapped\ status\ classic}), \end{array}$$

The select literal performs the dynamic type check immediately. No further information about the literal is required from the environment. In contrast, there is not enough information to check the name *Kidnapped* is a book. This minimum requirement placed on the context is indicated by the type environment.

6.2.5. Typed Commitments involving Choice
The branches in a choice may depend on different contexts. In the update below a person and a string are always selected. The string is immediately type checked and the check for the person is indicated by the context. The update features a third select which demands a name of type place, but alternatively offers the choice of the unit update. In the commitment below the unit branch is chosen, so the third select does not contribute to the context.

$$\begin{array}{l} \mathit{Burns}\colon \textsc{person} \\ \vdash \bigvee x\colon \textsc{String}. \bigvee a\colon \textsc{person}. \\ \quad \left(\begin{array}{l} |(a\ \mathit{email}\ x)| \\ \mathcal{G}_{\mathit{poets}}(a\ \mathit{email}\ x) \\ \left(\begin{array}{l} \bigvee h\colon \textsc{place}. \\ \left(\begin{array}{l} |(a\ \mathit{home}\ h)| \\ \mathcal{G}_{\mathit{poets}}(a\ \mathit{home}\ h) \end{array} \right) \oplus \mathrm{I} \end{array} \right) \end{array} \right), \\ \quad (a\ \mathit{email}\ \texttt{'Rabbie@soton.ac.uk'}) \\ \qquad \rhd\ (\mathit{Burns\ address}\ \texttt{'Rabbie@soton.ac.uk'}), \\ \qquad\quad \mathcal{G}_{\mathit{poets}}(\mathit{Burns\ address}\ \texttt{'Rabbie@soton.ac.uk'}) \end{array}$$

The choice between an update and the unit update models the operator 'opt' in SPARQL Query [5, 14]. Since the unit update is always enabled, the other branch may always be

ignored so is optional. This demonstrates that, firstly, opt is not primitive and, secondly, opt works for updates. In related work, opt is borrowed from relational algebra for modelling queries [15].

6.2.6. Iterated updates and dynamic types.
Iteration allows multiple copies of an update to be applied. For instance, the following iteration joins two copies of the inner update. Due to the use of join in the contraction rule, selected names are forced to be disjoint. In the following example, the context demands three disjoint names of type person.

$$\begin{array}{l} \mathit{Alice}\colon \textsc{person}, \mathit{Bob}\colon \textsc{person}, \mathit{Chris}\colon \textsc{person} \\ \quad \bigvee a\colon \textsc{person}. \\ \vdash \left(\begin{array}{l} |(a \textit{ type journalist})| \\ *\bigvee b\colon \textsc{person}. \\ \left(\begin{array}{l} |(b \textit{ type photographer})| \\ (a \textit{ knows } b) \end{array} \right) \end{array} \right), \\ \quad (\mathit{Alice} \textit{ type journalist}), \\ \quad (\mathit{Bob} \textit{ type photographer}), \\ \quad (\mathit{Chris} \textit{ type photographer}) \\ \quad \triangleright\ (\mathit{Alice} \textit{ type journalist}), (\mathit{Alice} \textit{ knows } \mathit{Bob}), \\ \qquad (\mathit{Alice} \textit{ knows } \mathit{Chris}), (\mathit{Bob} \textit{ type photographer}), \\ \qquad (\mathit{Chris} \textit{ type photographer}) \end{array}$$

Now suppose that the whole of the above update is also iterated. Due to the disjunction of environments forced by join, each journalist is assigned distinct photographers. To allow names to be shared the join rule can be relaxed, as discussed above.

6.2.7. Commitments for typed blank nodes.
The blank node rule allows a blank node to be used in place of a URI. The typed blank node also indicates a lower bound on the type of URI the blank node can represent. For instance, the example below involves a blank node quantifier annotated with type person. The query demands a name of any type, so the assumption that the blank node is of type person is strong enough for the following commitment.

$$\begin{array}{l} \quad \bigvee b\colon \top. \\ \vdash \left(\begin{array}{l} |(b \textit{ name } \texttt{`Burns'})| \\ (\mathit{Dmitri} \textit{ knows } b) \end{array} \right), \\ \quad \bigwedge a\colon \textsc{person}. \\ \quad (a \textit{ name } \texttt{`Burns'}) \end{array} \quad \triangleright \quad \left(\begin{array}{l} \bigwedge a\colon \textsc{person}. \\ (a \textit{ name } \texttt{`Burns'}), \\ (\mathit{Dmitri} \textit{ knows } a) \end{array} \right)$$

The context is used to ensure that type of the select quantifier and the blank node quantifier match. The select quantifier introduces to the context an assignment of a name to type person. The blank node rule eliminates that assignment from the context. In the above example, this leaves an empty context so no dynamic checks are required.

6.3. Type Preservation for Commitments

Type preservation verifies that given a well typed process the resulting process after a commitment is well typed with respect to the same environment. This means that the use of a URI after an update is consistent with the use of a URI before the update. For a commitment relation with a non-empty context, the context must agree with the type environment used to type the process. The following substitution lemma is required for selected names and literals.

Lemma 6.2 (Substitution preserves types). *For names,*

$$\textit{if } \alpha, a\colon \tau_1 \vdash U\colon \tau \textit{ and } \tau_0 \leq \tau_1, \textit{ then } \alpha, b\colon \tau_0 \vdash U\left\{{}^{b}/_{a}\right\}\colon \tau.$$

Similarly for literals,

$$\textit{if } \alpha, x\colon \mathrm{D} \vdash U\colon \tau \textit{ and } \vdash v\colon \mathrm{D}, \textit{ then } \alpha \vdash U\{{}^{v}/_{x}\}\colon \tau.$$

The proof of the lemma follows by structural induction. The type preservation theorem also uses type preservation of the structural congruence, Lemma 6.1. The soundness and completeness of the algorithmic type system eliminate the need to consider subsumption and weakening rules, Theorems 5.1 and 5.2.

Theorem 6.1 (Commitments preserve types). *If $\alpha_0 \vdash P \triangleright Q$, then $\alpha_0, \alpha_1 \vdash P\colon \tau$ yields that $\alpha_0, \alpha_1 \vdash Q\colon \tau$.*

Proof. The axioms are immediate. The structural induction proof for choose, join, select and blank nodes are demonstrated.

Consider the choose rule and assume that the following type tree holds.

$$\dfrac{\alpha_0 \vdash P\colon \tau \qquad \dfrac{\alpha_1 \vdash U\colon \tau \quad \alpha_2 \vdash V\colon \tau}{\alpha_1, \alpha_2 \vdash U \oplus V\colon \tau}}{\alpha_0, \alpha_1, \alpha_2 \vdash P \parr (U \oplus V) : \tau}$$

Therefore the following type tree holds.

$$\dfrac{\alpha_0 \vdash P\colon \tau \quad \alpha_1 \vdash U\colon \tau}{\alpha_0, \alpha_1 \vdash P \parr U\colon \tau}$$

Now assume that the choose left rule is used to resolve a commitment, where $\alpha, \alpha' \equiv \alpha_0, \alpha_1, \alpha_2$.

$$\dfrac{\alpha \vdash P \parr U \triangleright Q}{\alpha \vdash P \parr (U \oplus V) \triangleright Q}$$

By induction, $\alpha \vdash P \parr U \triangleright Q$ and $\alpha_0, \alpha_1 \vdash P \parr U\colon \tau$ yields the following type judgement, as required.

$$\alpha_0, \alpha_1, \alpha_2 \vdash Q\colon \tau$$

Consider the join rule and suppose that the following type judgement holds.

$$\dfrac{\dfrac{\alpha_0 \vdash P\colon \tau \quad \alpha_1 \vdash Q\colon \tau}{\alpha_0, \alpha_1 \vdash P \parr Q\colon \tau} \qquad \dfrac{\alpha_2 \vdash U\colon \tau \quad \alpha_3 \vdash V\colon \tau}{\alpha_2, \alpha_3 \vdash U \otimes V\colon \tau}}{\alpha_0, \alpha_1, \alpha_2, \alpha_3 \vdash P \parr Q \parr (U \otimes V) : \tau}$$

Hence the following two judgements hold.

$$\alpha_0, \alpha_2 \vdash P \parr U\colon \tau \quad \text{and} \quad \alpha_1, \alpha_3 \vdash Q \parr V\colon \tau$$

Now, assume that the following commitment holds, where $\alpha_0, \alpha_2 \equiv \beta_0, \beta_0'$ and $\alpha_1, \alpha_3 \equiv \beta_1, \beta_1'$.

$$\frac{\beta_0 \vdash P \parr U \rhd P' \quad \beta_1 \vdash Q \parr V \rhd Q'}{\beta_0 + \beta_1 \vdash P \parr Q \parr (U \otimes V) \rhd P' \parr Q'}$$

Hence by induction, the following type judgement holds, as required.

$$\frac{\alpha_0, \alpha_2 \vdash P' : \tau \quad \alpha_1, \alpha_3 \vdash Q' : \tau}{\alpha_0, \alpha_1, \alpha_2, \alpha_3 \vdash P' \parr Q' : \tau}$$

Consider the select rule and suppose that the following type tree holds.

$$\frac{\alpha_0 \Vdash P : \tau \quad \dfrac{\alpha_1 + a : \tau_1 \Vdash U : \tau}{\alpha_1 \Vdash \bigvee a : \tau_1 . U : \tau}}{\alpha_0, \alpha_1 \Vdash P \parr \bigvee a : \tau_1 . U : \tau}$$

Assuming that $\tau_0 \leq \tau_1$, by the substitution lemma, $\alpha_1 + a : \tau_1 \vdash U : \tau$ yields $\alpha_1, b : \tau_0 \vdash U\left\{{}^{b}/_{a}\right\} : \tau$, so the following proof tree can be constructed.

$$\frac{\alpha_0 \vdash P : \tau \quad \alpha_1, b : \tau_0 \vdash U\left\{{}^{b}/_{a}\right\} : \tau}{\alpha_0, \alpha_1, b : \tau_0 \vdash P \parr U\left\{{}^{b}/_{a}\right\} : \tau}$$

Also, assume that the following commitment holds, where $\alpha_0, \alpha_1 \equiv \alpha, \alpha'$, for some α'.

$$\frac{\alpha \vdash P \parr U\left\{{}^{b}/_{a}\right\} \rhd Q}{\alpha + b : \tau_0 \vdash P \parr \bigvee a : \tau_1 . U \rhd Q}$$

By the induction hypothesis, $\alpha \vdash P \parr U\left\{{}^{b}/_{a}\right\} \rhd Q$ and $\alpha_0, \alpha_1, b : \tau_0 \vdash P \parr U\left\{{}^{b}/_{a}\right\} : \tau$ yield the following, as required.

$$\alpha_0, \alpha_1, b : \tau_0 \vdash Q : \tau$$

Consider the blank node rule and assume that the following type tree holds.

$$\frac{\alpha_0 \vdash \bigwedge a : \tau_0 . P : \tau \quad \alpha_1 \vdash Q : \tau}{\alpha_0, \alpha_1 \vdash \bigwedge a : \tau_0 . P \parr Q : \tau}$$

Hence, assuming that $a \notin \mathrm{fn}(Q)$, the following type tree holds.

$$\frac{\alpha_0 + a : \tau_0 \vdash P' : \tau \quad \alpha_1 \vdash Q : \tau}{\alpha_0, \alpha_1 + a : \tau_0 \vdash P \parr Q : \tau}$$

Hence by induction, there exists α_0', α_1' such that $\alpha_0, \alpha_1 \equiv \alpha_0', \alpha' 1$ and the following type tree holds.

$$\frac{\alpha_0' + a : \tau_0 \vdash P' : \tau \quad \alpha_1' \vdash Q' : \tau}{\alpha_0', \alpha_1' + a : \tau_0 \vdash P' \parr Q' : \tau}$$

Therefore, assuming that $a \notin \mathrm{fn}(Q')$ the following proof tree holds, as required.

$$\frac{\dfrac{\alpha_0' + a : \tau_0 \vdash P' : \tau}{\alpha_0' \vdash \bigwedge a : \tau_0 . P' : \tau} \quad \alpha_1' \vdash Q' : \tau}{\alpha_0', \alpha_1' \vdash P' \parr Q' : \tau}$$

The remaining cases follow a similar pattern. Therefore, by induction on the structure of a commitment derivation, types are preserved by atomic commitments. □

6.3.1. Monotonicity of contexts.
The examples in the previous section indicate the weakest context for a transition. However, the example involving the blank node quantifier uses subtyping in the select name rule to select a stronger name. Similarly, in all examples subtyping allows a stronger type to be used in the environment.

For instance, a context which requires that *Burns* is of type PERSON, is satisfied by context which instead assigns the subtype WRITER to the same name. In general, Proposition 6.1 verifies that a stronger context can be used in place of a weaker context without breaking a commitment. The preorder extends subtyping point-wise to environments.

PROPOSITION 6.1 (Monotonicity). *If $\alpha_0 \leq \alpha_1$, then $\alpha_1 \vdash P \rhd Q$ yields that $\alpha_0 \vdash P \rhd Q$.*

The proof pushes the strengthening of the context towards the select quantifiers, where it is eliminated. A similar proof shows the monotonicity of typing. Monotonicity facilitates integration by allowing processes to be moved to a stronger environment without further type checks.

6.3.2. Recovering the untyped calculus.
The relationship between the typed and untyped calculus is acknowledged through erasure. Erasure removes all type annotations whilst retaining operational behaviour. In particular, the type annotations which appear in select and blank node quantifiers are removed, as defined by the transformation `erase`. As verified in Proposition 6.2, all transitions possible in the typed calculus are possible in the untyped calculus. For an exact match, the join rule is relaxed to remove requirement that joined context are disjoint.

PROPOSITION 6.2. *If $\alpha \vdash P \rhd Q$, then $\texttt{erase}\, P \rhd \texttt{erase}\, Q$.*

However, as expected, the converse does not hold. There exist transitions in the untyped calculus that are impossible in the typed calculus. For instance in the following example the blank node can only be selected after erasure, since $\top$ is not a subtype of DOCUMENT.

$$\texttt{erase}\left(\begin{array}{l} \bigwedge a : \top . (a \ \textit{status official}) , \\ \bigvee b : \textsc{document} . (b \ \textit{status official})^{\perp} \end{array} \right) \rhd \perp$$

The main differences between the typed commitments in Fig 16, and the untyped commitments in Figs. 4 and 8, are summarised are as follows. The select name quantifier inputs a name of a given type, rather than any name. The select literal rule checks the datatype of the literal, rather than accepting any literal. Rules propagate resulting constraints on the context, whereas the untyped calculus does not constrain the context. The blank node quantifier rule simulates URIs of a given type, rather than any URI.

The combination of the subtype system, literal only typing, monotonicity and erasure allow different strengths of type system to be used in different applications.

7. FURTHER WORK

The calculus provides an abstract syntax and operational semantics which model a language where Linked Data is primitive. The focus of this work is to investigate the capabilities of a of light type system for Linked Data programming languages. The rules of the type system provide a model, which can be used to implement type inference algorithms for Linked Data. The operational semantics can also be used to derive an algebra for optimisations of queries and updates. A range of extensions can enrich the calculus for particular applications. In particular, extensions for data interoperability and further higher level programming languages are outlined.

7.1. Type inference algorithms

Type inference reduces constraints imposed by the type system by inferring types from partial type information. An update can be provided untyped by a programmer. An algorithm then automatically infers the missing type annotations. Type inference makes programming easier and improves interoperability with Linked Data systems with different degrees of static type information.

For instance, in the example below the types of *Hamish* and *Dmitri* are unknown, so are assigned fresh type variables X and Y. The constraints $\mathrm{X} \leq \textsc{person}$ and $\mathrm{Y} \leq \textsc{person}$ are obtained by unfolding the tree of the following algorithmic type judgement.

$$\begin{array}{l} \mathit{knows} : \mathtt{p}(\textsc{person}, \textsc{person}), \\ \mathit{Hamish} : \mathrm{X}, \mathit{Dmitri} : \mathrm{Y} \end{array} \Vdash (\mathit{Hamish\ knows\ Dmitri}) : \top$$

A type inference algorithm discovers the minimal unifier. The minimal unifier above is $\mathrm{X} \mapsto \textsc{person}, \mathrm{Y} \mapsto \textsc{person}$. This unifier is a substitution, which gives an valid type judgement when applied to the above tree.

The general inference algorithm proceeds as follows. Firstly, apply algorithmic subtyping to obtain a proof tree indicating a set of subtype constraints over types involving fresh atomic types, as in the example above. Secondly, apply unification to the constraints, until either the constraints are rejected or the algorithm terminates successfully. If the constraints are rejected, there is no unifier. If the constraints are accepted, then the minimal unifier is generated [52].

Type inference appears implicitly in the RDFS standard. The rules of RDFS state that given a predicate, the type of the domain of the predicate bounds the type of any subject of the predicate. Similarly, the range of a predicate bounds the type of any object [4]. As demonstrated above, the same effect is achieved by type inference in this work. Type inference is performed at compile time, so incurs no cost to queries or updates.

7.2. An algebra for optimising updates

The calculus can be used as a basis for optimisation of queries. In particular, an algebra over updates can be derived. A suitable algebra allows updates to be rewritten whilst preserving the operational semantics of an update. An update before the algebra is applied should have the same operational behaviour as an update after the algebra is applied. For instance, the following two updates are expected to have the same operational behaviour.

$$\bigvee a. \left(\begin{array}{l} \bigvee b.(b\ \mathit{knows}\ a)^{\perp} \\ \left(\begin{array}{l} |(a\ \mathit{type\ spy})| \\ \oplus \\ |(a\ \mathit{type\ hermit})| \end{array} \right) \end{array} \right) \sim \bigvee a. \bigvee b. \left(\begin{array}{l} \left(\begin{array}{l} (b\ \mathit{knows}\ a)^{\perp} \\ |(a\ \mathit{type\ spy})| \end{array} \right) \\ \oplus \\ \left(\begin{array}{l} (b\ \mathit{knows}\ a)^{\perp} \\ |(a\ \mathit{type\ hermit})| \end{array} \right) \end{array} \right)$$

The algebra can be applied to rewrite queries to normal forms. For instance, a suitable normal form can be used to optimise an update for distribution over a cluster of machines. The details of the algebra and proof that it preserves operational behaviour is worthy of detailed attention, so is the focus of a paper in preparation.

7.3. Extensions for interoperability of data

Interoperability problems similar to those for static types, tackled in Sec. 4, also apply to dynamic Linked Data. The intrinsic interoperability problem for URIs is known as co-reference [53]. Linked Data from different authorities may use different URIs to identify the same resource. For instance, many different URIs identify Robert Burns.

The problem results in an equivalence, or more generally a preorder, over URIs. The preorder, say $\sqsubseteq$, can be used to improve queries and updates, by permitting discrepancies between URIs. This light URI level reasoning can built into the axioms. The following instance of an extended delete axiom allows a triple to be deleted by a weaker triple.

$$\frac{(\mathit{Burns'\ author\ tribute}) \sqsubseteq (\mathit{Burns\ creator\ tribute})}{(\mathit{Burns'\ author\ tribute}) \otimes (\mathit{Burns\ creator\ tribute})^{\perp} \rhd \perp}$$

The above example works under the assumptions that *Burns = Burns'* and *author* $\sqsubseteq$ *creator*. This demonstrates one use of subProperty from the RDFS standard and sameAs from the OWL standard [41].

7.4. Extensions for future high level languages.

Extensions can specify long term behaviour. A continuation process (indicated by `then` below) describes update which follows the current update. Recursion (indicated by ν below) allows a precess to be unfolded. For instance, the following example unfolds a process twice to navigate a chain of knows predicates until it reaches a leader.

$$\begin{array}{l} \nu Y(\mathit{Ali}). \\ \left(\begin{array}{l} \bigvee b. \\ |(\mathit{Ali\ knows}\ b)|\ \mathtt{then}\ Y[b] \\ \oplus \\ |(\mathit{Ali\ type\ leader})|\ \mathtt{then}\ P \end{array} \right), \\ (\mathit{Ali\ knows\ Bob}), \\ (\mathit{Bob\ type\ leader}) \end{array} \rightarrow \begin{array}{l} (\mathit{Ali\ knows\ Bob}), \\ (\mathit{Bob\ type\ leader}), \\ P\left\{{}^{Bob}/_{Ali}\right\} \end{array}$$

The above example, Upon reaching the leader performs some process P using the URI of the leader. Recursion is

useful for describing long term behaviour of an automata. An automata can control navigation in a site dependant on Linked Data.

In another paradigm the right hand side of a commitment relation gathers a monoid of query results. Results are then manipulated in a functional language. The suggestion is that the standards are approaching a high level language for engaging with Linked Data.

8. CONCLUSIONS

The calculus has a distinguished role in the emerging Web of Linked Data. The calculus provides a level of abstraction where atomic actions can be executed using only local observations about data. The level of abstraction is above concerns such as the messages sent between machines, but below concerns such as the global structure of a store.

The calculus models substantial queries and updates which span diverse data sources. It is assumed that mechanisms can aggregate Linked Data in one environment — typically a store. The contribution is an operational semantics for standards which operate at this level of abstraction. In particular, an operational semantics for SPARQL Query and SPARQL Update are specified. Examples focus on SPARQL Update which is crucial for Read–Write Linked Data.

The calculus works with a light type system, which can be checked locally. This contrasts to stronger schema, which require a global perspective on data. The calculus demonstrates that appropriate data can be lifted to a feasible type system. Literals in a query or update can always be type checked. By providing type assumptions, URIs can also be type checked. The contribution is an investigation into the RDFS standard as a type system.

The calculus simultaneously models key Linked Data standards. The challenge is that collectively the standards do not form an established logical system. Classical logic is found in constraints, linear logic is found in atomic actions and a truly concurrent process algebra combines actions, Sec. 2. A partial order over URIs and types enables interoperability, Sec. 4. The type system introduces two intuitionistic logics, Sec. 5, 6. Kleene algebras found optimisations, Sec. 7. Related work includes models based on graphs, epistemic logics and bunched implications [14, 37, 51]. The suggestion is that any model for Linked Data accommodates several notions of truth.

REFERENCES

[1] Sayre, R. (2005) Atom: the standard in syndication. *IEEE Internet computing*, **9**, 71–78.

[2] Bizer, C. (2009) The emerging Web of Linked Data. *IEEE Intelligent Systems*, **24**, 87–92.

[3] REC-rdf-concepts-20040210 (2004) *Resource Description Framework: Concepts and Abstract Syntax*. W3C. MIT, Cambridge, MA.

[4] REC-rdf-schema-20040210 (2004) *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C. MIT, Cambridge, MA.

[5] REC-rdf-sparql-query-20080115 (2008) *SPARQL Query Language for RDF*. W3C. MIT, Cambridge, MA.

[6] Milner, R. (1993) The polyadic π-calculus: A tutorial. In Bauer, F., Brauer, W., and Schwichttenburg, H. (eds.), *Logic and Algebra in Specification*. Springer, New York.

[7] Bocchi, L. and Lucchi, R. (2006) Atomic commit and negotiation in service oriented computing. In Ciancarini, P. and Wiklicky, H. (eds.), *Coordination Models and Languages*. Springer, Berlin/Heidelberg.

[8] Abadi, M. and Fournet, C. (2001) Mobile values, new names, and secure communication. *Proceedings of the 28th ACM SIGPLAN-SIGACT*, London, January 17-19, pp. 104–115. ACM, NY.

[9] Carbone, M., Honda, K., and Yoshida, N. (2007) Structured communication-centred programming for Web Services. In De Nicola, R. (ed.), *Programming Languages and Systems*, Lecture Notes in Computer Science, **4421**, pp. 2–17. Springer, Berlin/Heidelberg.

[10] HPL-2007-102 (2007) *SPARQL/Update: A language for updating RDF graphs*. Hewlit Packard Labs. Bristol.

[11] Omitola, T. et al. (2010) Put in your postcode, out comes the data: A case study. In Aroyo, L. et al. (eds.), *The Semantic Web: Research and Applications. 7th Extended Semantic Web Conference*, Heraklion, Crete, May 30 - June 3, pp. 318–332. Springer Berlin/Heidelberg.

[12] Bizer, C. et al. (2009) DBpedia: A crystallization point for the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, **7**, 154–165.

[13] Kobilarov, G. et al. (2009) Media meets Semantic Web: How the BBC uses DBpedia and Linked Data to make connections. In Aroyo, L. (ed.), *The Semantic Web: Research and Applications. 6th European Semantic Web Conference*, Heraklion, Greece, May 31 - June 4, pp. 723–737. Springer, Berlin/Heidelberg.

[14] Pérez, J., Arenas, M., and Gutierrez, C. (2009) Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, **34**, 1–45.

[15] HPL-2005-170 (2005) *A relational algebra for SPARQL*. Hewlit Packard Labs. Bristol.

[16] Carroll, J. J., Bizer, C., Hayes, P., and Stickler, P. (2005) Named graphs. *Web Semantics: Science, Services and Agents on the World Wide Web*, **3**, 247–267.

[17] WD-sparql11-http-rdf-update-20101014 (2010) *SPARQL 1.1 Uniform HTTP Protocol for Managing RDF Graphs*. W3C. MIT, Cambridge, MA.

[18] rfc4287 (2005) *The Atom Syndication Format*. Internet Engineering Task Force. Fremont, CA.

[19] rfc5023 (2007) *The Atom Publishing Protocol*. Internet Engineering Task Force. Fremont, CA.

[20] Reynolds, J. (2005) Toward a grainless semantics for shared-variable concurrency. In Lodaya, K. and Mahajan, M. (eds.), *Foundations of Software Technology and Theoretical Computer Science*, pp. 11–38. Springer, Berlin/Heidelberg.

[21] WD-sparql11-update-20091022 (2010) *SPARQL 1.1 Update*. W3C. MIT, Cambridge, MA.

[22] Kleinberg, J. M. (1999) Authoritative sources in a hyperlinked environment. *Journal of the ACM*, **46**, 604–632.

[23] REC-xmlschema-2-20041028 (2004) *XML Schema part 2: Datatypes Second Edition*. W3C. MIT, Cambridge, MA.

[24] Abramsky, S. (2006) What are the fundamental structures of concurrency?: We still don't know! *Proceedings of the*

Workshop Essays on Algebraic Process Calculi, September 29, pp. 37–41. Elsevier.

[25] WD-sparql11-query-20101014 (2010) *SPARQL 1.1 Query Language*. W3C. MIT, Cambridge, MA.

[26] Girard, J.-Y. (2007) Truth, modality and intersubjectivity. *Mathematical Structures in Computer Science*, **17**, 1153–1167.

[27] Girard, J.-Y. (1987) Linear logic. *Theoretical Computer Science*, **50**, 1–112.

[28] Milner, R. (1980) *A calculus of communicating systems*. Springer, NJ.

[29] Buscemi, M. and Montanari, U. (2007) CC-Pi: A constraint-based language for specifying service level agreements. In De Nicola, R. (ed.), *Programming Languages and Systems*, Lecture Notes in Computer Science, **4421**, pp. 18–32. Springer, Berlin/Heidelberg.

[30] Saraswat, V. A., Rinard, M., and Panangaden, P. (1991) The semantic foundations of concurrent constraint programming. *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Orlando, Florida, January 21-23, pp. 333–352. ACM, NY.

[31] Hartig, O. et al. (2009) Executing SPARQL Queries over the Web of Linked Data. In Bernstain, A. et al. (eds.), *The Semantic Web – ISWC 2009*, Chantilly, VA, October 25-29, pp. 293–309. Springer Berlin/Heidelberg.

[32] Hodas, J. S. and Miller, D. (1994) Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, **110**, 327–365.

[33] Kozen, D. (1997) Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, **19**, 427–443.

[34] Conway, J. H. (1971) *Regular Algebra and Finite Machines*. Chapman and Hall, London.

[35] Hoare, C., Möller, B., Struth, G., and Wehrman, I. (2009) Concurrent Kleene algebra. In Bravetti, M. and Zavattaro, G. (eds.), *CONCUR 2009*, Bologna, Italy, September 1-14, pp. 399–414. Springer Berlin/Heidelberg.

[36] Abramsky, S. and Vickers, S. (1993) Quantales, observational logic and process semantics. *Mathematical Structures in Computer Science*, **3**, 161–227.

[37] Baltag, A., Coecke, B., and Sadrzadeh, M. (2007) Epistemic Actions as Resources. *Journal of Logic and Computation*, **17**, 555–585.

[38] Abramsky, S., Gay, S., and Nagarajan, R. (1995) Interaction categories and the foundations of typed concurrent programming. In Broy, M. (ed.), *Proceedings of the NATO Advanced Study Institute on Deductive Program Design*, Marktoberdorf, Germany, July 26 - August 7, pp. 35–114. Springer, Berlin/Heidelberg.

[39] Berners-Lee, T., Connolly, D., Kagal, L., Scharf, Y., and Hendler, J. (2008) N3logic: A logical framework for the world wide web. *Theory and Practice of Logic Programming*, **8**, 249–269.

[40] Baget, J.-F. (2005) RDF entailment as a graph homomorphism. In Gil, Y., Motta, E., Benjamins, V., and Musen, M. (eds.), *The Semantic Web – ISWC 2005*, Galway, Ireland, November 2005, pp. 82–96. Springer Berlin/Heidelberg.

[41] Horrocks, I. and Patel-Schneider, P. (2004) Reducing OWL entailment to description logic satisfiability. *Web Semantics: Science, Services and Agents on the World Wide Web*, **1**, 345–357.

[42] Fielding, R. T. and Taylor, R. N. (2002) Principled design of the modern Web architecture. *ACM Transactions on Internet Technology*, **2**, 115–150.

[43] Abiteboul, S. (1997) Querying semi-structured data. In Afrati, F. and Kolaitis, P. (eds.), *Database Theory – International Conference on Database Theory*, Delphi, Greece, January 8-10, pp. 1–18. Springer Berlin/Heidelberg.

[44] Brown, A., Fuchs, M., Robie, J., and Wadler, P. (2002) MSL: A model for W3C XML Schema. *Computer Networks*, **39**, 507–521.

[45] Coquand, T. and Huet, G. (1988) The calculus of constructions. *Inf. Comput.*, **76**, 95–120.

[46] Fagin, R., Kolaitis, P., Miller, R., and Popa, L. (2003) Data exchange: Semantics and query answering. In Calvanese, D., Lenzerini, M., and Motwani, R. (eds.), *Database Theory – International Conference on Database Theory*, Siena, Italy, January 8-10, pp. 207–224. Springer Berlin/Heidelberg.

[47] MacNeille, H. M. (1936) Extensions of partially ordered sets. *Proceedings of the National Academy of Sciences of the United States of America*, **22**, 45–50.

[48] Muñoz, S., Pérez, J., and Gutierrez, C. (2007) Minimal deductive systems for RDF. *The Semantic Web: Research and Applications, 4th European Semantic Web Conference*, Innsbruck, Austria, June 3-7, pp. 53–67. Springer.

[49] Mitchell, J. C. (1991) Type inference with simple subtypes. *Journal of Functional Programming*, **1**, 245–285.

[50] Jeffrey, A. and Rathke, J. (2005) Contextual equivalence for higher-order π-calculus revisited. *Logical Methods in Computer Science*, **1**, paper 4.

[51] O'Hearn, P. W. and Pym, D. J. (1999) The logic of bunched implications. *Bulletin of Symbolic Logic*, **5**, 215–244.

[52] Martelli, A. and Montanari, U. (1982) An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, **4**, 258–282.

[53] Alani, H. et al. (2002) Managing reference: Ensuring referential integrity of ontologies for the Semantic Web. In Gómez-Pérez, A. and Benjamins, V. (eds.), *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web*, pp. 235–246. Springer, Berlin/Heidelberg.