Operational Semantics for SPARQL Update

Ross Horne, Vladimiro Sassone, and Nicholas Gibbins

Electronics and Computer Science
University of Southampton, UK
{rjh06r,vs,nmg}@ecs.soton.ac.uk

Abstract. Concurrent fine grained updates are essential for using RDF stores in dynamic modern Web applications, where users increasingly contribute content as often as they read content. SPARQL Update is a language proposed by the W3C for fine grained updates for RDF stores. In this work we propose an operational semantics for an update language for RDF, which models core features of SPARQL Update. Firstly, an abstract syntax for RDF and updates is presented. Secondly, the operational semantics is defined using relations over the abstract syntax. The operational semantics specifies all possible operational behaviours of updates in the presence of an RDF store. The specification is useful as a common reference for compiler engineers and as a foundation for the static analysis of updates.

1 Introduction

An open problem is to provide an operational semantics for the W3C SPARQL Update working draft [9]. SPARQL Update is a development of an earlier proposal from Hewlett-Packard Labs [18]. The language is introduced to extend the SPARQL Query language [17] to enable fine grained updates over an RDF store.

The recommended semantics for SPARQL Query are influenced by the work of Pérez et al. [15], which provides a set-based denotational semantics for queries. In contrast, the semantics presented here for updates are operational in nature. The difference between a denotational semantics and an operational semantics is that the former builds an external model (typically a static set in which behaviours may exist), whereas the later is defined directly over an abstract syntax for the language.

There are several advantages of operational semantics. An operational semantics works like an interpreter, so is at an appropriate level for compiler engineering. Operational semantics is also suited to ad-hoc features which appear in real programming languages, which SPARQL Update intends to be. Furthermore, operational semantics are suited to specifying the complex long term behaviour of systems, including concurrency as required by servers. Denotational semantics for both application driven ad-hoc features and long term behaviour are notoriously difficult [1]. Thus, operational semantics can easily and insightfully be adapted to queries [12]; but denotational semantics do not extend easily to updates, since non-standard mathematics would be required.

Consider an analogy. All readers are familiar with the concept of a regular expression or use tools which involve regular expressions. For instance, the replace tool in your text editor is appropriate for every day updates in text documents. This update language extends the power of regular expressions generalised appropriately to RDF triples (instead of characters), with quantifiers to access URIs and literals in triples. For the sake of clarity, here a core update language, rather than full SPARQL Update, is presented where only the default RDF graph is updated. The model can be extended to handle named graphs [6]. Also, the model can accommodate updates with respect to entailments, such as those defined in RDFS [5].

In Section 2 further motivation is provided to emphasise the importance of an operational semantics for SPARQL Update. In Section 3 a syntax for RDF Data and Updates is established. In Section 4 an equivalence is provided over RDF Data. This equivalence defines when two pieces of RDF Data have the same meaning. In Section 5 the behaviour of Updates is specified using a deductive system which derives relations over the syntax. The relations indicate how some RDF Data is transformed by a Update into some other RDF Data. Examples of each feature of Updates are provided along with the rules of the operational semantics.

2 Background and Motivation

Before defining the operational semantics, some motivation is provided in this section. The tradition of using operational semantics to specify programming languages is discussed, highlighting benefits offered to Web standards. A short sketch of the relationship between this work and the draft specification is also provided, including a preview of the abstract syntax.

2.1 A Case for an Operational Semantics for SPARQL Update

A structural approach to operational semantics was first introduced in a seminal note by Plotkin [16]. At the time, the behaviour of languages tended to be specified using a reference compiler. The correctness of an implementation of a language would be verified by checking that its behaviour matched the behaviour of the reference compiler. Plotkin's work introduced a methodology for precisely specifying the operational behaviour of languages directly over an abstract syntax.

Web standards, such as SPARQL Update, require clear specifications of their behaviour. Structural operational semantics are designed precisely for this scenario. A clear operational semantics can be concisely communicated in a document, which can be used as a reference to ensure that all implementations of a standard have a common operational behaviour. For this reason, an operational semantics for SPARQL Update is a valuable contribution to the current standardisation process at the W3C [9].

An operational semantics for a language has further advantages. It allows clarity and methodology when design decisions are considered. Furthermore, operational semantics can be used as the basis for powerful techniques and tools for the language, such as a static type checker or algebra for composition and optimisation of updates [12,11]. Such tools cannot be confidently developed without an operational semantics. Furthermore, the distinction between static and dynamic types is not evident until updates are considered.

2.2 A Comparison of SPARQL Update to This Work

The intention of the update language introduced is to model the core of SPARQL Update. A basic comparison between this update language and SPARQL Update is provided here.

The following example is adapted from the current working draft [9]. The update deletes zero or more RDF triples where the literal `"Bill"` appears and inserts a triple where `"William"` appears. The update can only occur if the subject of the triple is of type person.

Concrete Update:

```
DELETE { ?person foaf:givenName "Bill" }
INSERT { ?person foaf:givenName "William" }
WHERE { ?person rdf:type foaf:Person }
```

The above update can be expressed in an abstract syntax as follows.

Abstract Update:

```
DO SELECT :person {
  DELETE { :person foaf:givenName "Bill" }
  INSERT { :person foaf:givenName "William" }
  WHERE { :person rdf:type foaf:Person }
}
```

The abstract syntax above is more explicit than the concrete syntax. The select quantifier explicitly indicates the scope of the bound variable. Also, the abstract syntax of the update language explicitly indicates that the update may be applied zero or more times. This makes the definition of the operational semantics cleaner.

3 A Concise Abstract Syntax

This section presents an abstract syntax used to define an operational semantics for Updates. The abstract syntax is intended for the purpose of compiler engineering (as opposed to defining an exchange format). Three generators are sufficient to specify this abstract syntax: one for RDF Data; one for constraints; and a third for Updates.

3.1 Syntactic Conventions

The following namespace prefix bindings are used throughout this document.

dc:	`http://purl.org/dc/terms/`
dc11:	`http://purl.org/dc/elements/1.1/`
foaf:	`http://xmlns.com/foaf/0.1/`
eg:	`http://example.org/`

Prefixes abbreviate URIs, for readability in examples. Curly brackets are used to resolve ambiguity in examples.

3.2 A Syntax for RDF Data

The following grammar presents an abstract syntax for RDF. Several concrete syntaxes have been proposed for RDF, such as Turtle and N3 for the purpose of tersely presenting RDF to humans [2,4]. In contrast, the following abstract syntax for RDF Data is designed for compiler engineering.

object ::=	`"literal"`	a literal
	| *URI*	a URI
	| *?variable*	a variable

Data ::=	{}	nothing
	| { *URI URI object* }	a triple
	| *Data* , *Data*	par
	| `BNODE` *URI Data*	blank node

Two forms of triple represent RDF triples, with either a URI or a literal as the object. A variable indicates an unknown literal (for pattern matching in queries). Nothing represents the absence of any RDF triples. The operator 'par' composes RDF Data, thus for instance two triples can be composed using par. The Blank node quantifier binds a name which can only be refered to locally (as opposed to a URI which is a global name). For elegance, the same syntax is used to name URIs and blank nodes; they are distinguished only by quantifiers, which give local names scope. Many examples of RDF Data are presented throughout this work.

3.3 A Syntax for Constraints

Constraints are defined fully in the SPARQL Query recommendation [17], hence only an outline grammar for constraints is provided here. The following is enough to suggest that constraints form a Boolean algebra with built in primitives. Constraints may contain variables and URIs.

Constraint ::=	`true`	true
	| `false`	false
	| *Constraint* && *Constraint*	and
	| *Constraint* `||` *Constraint*	or
	| `!`*Constraint*	not
	| `regex`(*?variable*, *RegularExpression*)	regular expression
	| ...	etc.

A constraint is satisfied if and only if it evaluates to true. The evaluation of constraints is detailed in the SPARQL Query recommendation [17]. Examples of constraints include regular expressions parametrised on a variable and inequality tests on numbers.

3.4 A Syntax for Updates

The following grammar proposes an abstract syntax for updates. The language is inspired by core features of SPARQL Update. A successful update results in an atomic change to an RDF store. This abstract syntax allows constructs to be nested. Nesting is useful for expressiveness and optimisation purposes.

Update ::=	`DELETE` *Data*	delete a term
	| `INSERT` *Data*	insert a term
	| `FILTER` *Constraint*	impose a constraint
	| *Update* `CHOOSE` *Update*	choose a branch
	| *Update* `JOIN` *Update*	synchronise updates
	| `SELECT` *URI Update*	select a URI
	| `SELECT` *?variable Update*	select a literal
	| `DO` *Update*	iteratively apply an update

Delete removes the indicated RDF Data from the store. Insert introduces some RDF Data to the store. Filter imposes a constraint on an update. Choose offers the choice of either a left or right update. Join ensures that two updates happen in the same atomic update. Select parametrises an update on either a URI or a literal which is not known in advance. (Note that in this abstract syntax, URIs and literals are distinguished in Selects for clarity.) Iteration (DO) performs an update zero, one, two or more times, in the same atomic update. Without iteration an update is applied once.

Examples of each construct are provided along with the operational semantics for the construct in Section 5.

3.5 Abbreviations for Common Updates

A number of common updates can be defined using the basic updates above. The use of abbreviations avoids redundancy in the operational semantics. Identifying redundant operators is useful for compiler engineering, since the number of operators to implement directly is reduced.

An optional update gives the choice of performing an update or not performing an update. The optional update can be defined by a choice between an update and the true constraint which always holds, as follows. (This avoids a left outer join operator, which is used to provide the semantics of 'optional' in SPARQL Query [15].)

$$\texttt{OPTIONAL}\, \mathit{Update} \triangleq \mathit{Update}\ \texttt{CHOOSE}\ \texttt{FILTER}\,\texttt{true} \qquad \text{optional update}$$

Successive select queries are can be combined. The combined variables are listed in a single select quantifier, as follows.

$$\begin{array}{l} \texttt{SELECT}\ ?\mathit{variable}_0\ ?\mathit{variable}_1 \\ \quad \mathit{Update} \end{array} \triangleq \begin{array}{l} \texttt{SELECT}\ ?\mathit{variable}_0 \\ \quad \texttt{SELECT}\ ?\mathit{variable}_1 \\ \quad\quad \mathit{Update} \end{array} \qquad \text{multiple selects}$$

In this paper queries are encoded naïvely, using the keyword `WHERE`. The effect of a query can be achieved by joined insert and delete, as follows.

$$\texttt{WHERE}\, \mathit{Term} \triangleq \texttt{DELETE}\, \mathit{Term}\ \texttt{JOIN}\ \texttt{INSERT}\, \mathit{Term} \qquad \text{queries}$$

The joined delete and insert has the effect of a querying for a term: The term deleted must exist for the delete to be applied, but the insert immediately replaces the deleted term in the same atomic step. Queries could alternatively be defined as primitives of the language, as in [12].

4 An Equivalence over RDF Terms

This section identifies equivalent syntax. A syntactic equivalence imposes less constraints on RDF than any requirement that collections of triples are sets. Instead, obviously equivalent syntax is considered to serve the same purpose, as defined by a structural congruence.

4.1 A Structural Congruence

A structural congruence, written = below, is a relation between RDF Terms. A congruence is an equivalence relation (reflexive, symmetric and transitive) which holds in all contexts. The structural congruence satisfies the following equations — associativity, unit and commutativity.

$$\text{Associativity:}\quad Data_0 \texttt{ , } \{Data_1 \texttt{ , } Data_2\} = \{Data_0 \texttt{ , } Data_1\} \texttt{ , } Data_2$$

$$\text{Unit:}\quad Data \texttt{ , } \{\} = Data \qquad\qquad \text{Commutativity:}\quad Data_0 \texttt{ , } Data_1 = Data_1 \texttt{ , } Data_0$$

The structural congruence can be applied at any point, when evaluating the operational semantics in Section 5.

Example of Applying the Structural Congruence. The following RDF Data can be used interchangeably. If the RDF on the left appears in a rule in the next section, then it can be replaced by the RDF on the right.

```
{eg:book1 eg:price "£10"} ,                    {
{} ,                                             {eg:book1 dc:title "Web of Data"} ,
{                                                {eg:book1 eg:price "£10"}
  {eg:book2 dc:title "Linked Data"} ,          } ,
  {eg:book1 dc:title "Web of Data"}            {eg:book2 dc:title "Linked Data"}
}
```

Brackets are used similarly for Group Graph Patterns in SPARQL Query [17]. Associativity of par allows most brackets to be omitted for readability.

Alpha Conversion. The standard notion of alpha conversion can be applied to blank node quantifiers. Alpha conversion allows a bound name to be replaced by a fresh name, to avoid name clashes. For instance the following RDF Data is equivalent.

```
BNODE :a {                                    BNODE :b {
  {:a foaf:familyName "Carrol"} ,               {:b foaf:familyName "Carrol"} ,
  {:a foaf:knows eg:Klyne}                      {:b foaf:knows eg:Klyne}
}                                             }
```

Alpha conversion captures the isomorphisms in the RDF specification [13].

5 A Commitment Relation for Updates

The commitment relation specifies atomic changes which can be made to an RDF store. Atomicity focuses on the local effect of an update. The RDF Data which is required to perform an atomic update is accounted for. An advantage of this approach is that the data indicated by a commitment relation can be locked to ensure that an update occurs atomically.

A commitment relation consists of the RDF Data before an update, an Update and the resulting RDF Data after the update. Thus commitment relations are relations of the following form.

$$\text{Before: } \mathit{Data} \qquad \text{Update: } \mathit{Update} \qquad \text{After: } \mathit{Data}$$

Commitment relations can also be derived from rules. The premises of a rule are one or more commitment relations and the conclusion is a single commitment relation. The conclusion holds only if all the premises hold. The axioms and rules which specify operational semantics for Updates are defined throughout this section.

5.1 The Delete Axiom

The Delete Axiom removes some RDF Data from the store. The committed RDF Data, *Data*, and committed delete update, `DELETE`*Data*, interact. After the interaction both the Data is removed from the store. This results in the empty process.

$$\text{Before: } \mathit{Data} \qquad \text{Update: } \texttt{DELETE}\,\mathit{Data} \qquad \text{After: } \{\}$$

Example of the Delete Axiom. The following triple can be removed by the following update due to the following commitment relation. This commitment relation is an instance of the Delete Axiom.

$$\begin{aligned} \text{Before: } & \{\, \mathit{eg{:}book1}\ \mathit{dc{:}title}\ \texttt{"The Semantic Web"} \,\} \\ \text{Update: } & \texttt{DELETE}\{\, \mathit{eg{:}book1}\ \mathit{dc{:}title}\ \texttt{"The Semantic Web"} \,\} \\ \text{After: } & \{\} \end{aligned}$$

5.2 The Insert Axiom

The Insert Axiom adds some designated RDF Data to the store. The designated RDF Data is indicated by the INSERT keyword. The result of this update is to make the designated RDF Data available after the commitment.

$$\text{Before: } \{\} \qquad \text{Update: } \texttt{INSERT}\,\mathit{Data} \qquad \text{After: } \mathit{Data}$$

Example of the Insert Axiom. The two triples below can be inserted into anything (since nothing is required), due to the following commitment relation. This commitment relation is an instance of the Insert Axiom.

$$\begin{aligned} \text{Before: } & \{\} \\ \text{Update: } & \texttt{INSERT}\{\, \mathit{eg{:}book1}\ \mathit{dc{:}title}\ \texttt{"The Web of Linked Data"} \,\} \\ \text{After: } & \{\, \mathit{eg{:}book1}\ \mathit{dc{:}title}\ \texttt{"The Web of Linked Data"} \,\} \end{aligned}$$

5.3 The Join Rule

The Delete Axiom and the Insert Axiom allow basic updates to take place where either the exact RDF Data to be deleted is known, or the exact RDF Data to be inserted is known, respectively. For more substantial updates, rules are required to build commitment relations. The first of these rules is the Join Rule.

The Join Rule ensures that two updates occur atomically, in the same commitment relation. If one update has one effect and another update has another effect, then the join of the updates is their combined effect. The rule ensures that both updates act simultaneously on separate RDF Data. Suppose that the following commitment relation holds.

$$\text{Before: } \mathit{Data}_0 \qquad \text{Update: } \mathit{Update}_0 \qquad \text{After: } \mathit{Data}_2$$

Also, suppose that the following commitment relation holds.

$$\text{Before: } \mathit{Data}_1 \qquad \text{Update: } \mathit{Update}_1 \qquad \text{After: } \mathit{Data}_3$$

The two commitment relations above can be combined to produce the following commitment relation.

$$\text{Before: } \mathit{Data}_0 \texttt{ , } \mathit{Data}_1 \quad \text{Update: } \mathit{Update}_0 \texttt{ JOIN } \mathit{Update}_1 \quad \text{After: } \mathit{Data}_2 \texttt{ , } \mathit{Data}_3$$

Example of Joined Updates. The update below demonstrates two joined updates. The update combines the examples of Sec. 5.1 and Sec. 5.2 using the join rule. Thus the combined update removes a triple and adds a new triple atomically.

```
Before: { eg:book3 dc:title "The Semantic Web" }
Update: DELETE{ eg:book3 dc:title "The Semantic Web" }
        JOIN
        INSERT{ eg:book3 dc:title "The Web of Linked Data" }
 After: { eg:book3 dc:title "The Web of Linked Data" }
```

Notice that the rules ensure that join is commutative (this is not sequential composition). Also, from here onwards, the join keyword is omitted from examples. This makes examples more readable and closer to the SPARQL Update syntax.

5.4 The Select Literal Rule and Select URI Rule

The Select Literal Rule is parametrised on a variable. The variable is bound to the update indicated (so cannot be refered to from outside the select). The Select Rule allows any literal which enables a commitment to be substituted for the variable. The result of the commitment with the variable substituted for a literal, becomes the result of the commitment with the variable bound by a Select. Note that substitution is indicated by square brackets where the literal on the left replaces the variable on the right. Suppose that the following commitment relation holds.

$$\text{Before: } \mathit{Data}_0 \qquad \text{Update: } \mathit{Update}[\texttt{"literal"}/\mathit{?variable}] \qquad \text{After: } \mathit{Data}_1$$

Given the commitment relation above the following commitment relation holds.

Before: $Data_0$ Update: `SELECT` *?variable Update*   After: $Data_1$

The Select URI Rule has the same shape. In the case of URIs, a correct URI to input is substituted for the temporary URI which is bound in the Select expression. Thus two URIs replace both the variable and literal in the Select Literal Rule.

Example of the Select Literal Rule. The following example demonstrates how Select can be used to delete some RDF Data which involves a literal not known in advance. The update deletes a triple in which the variable *?title* appears. The variable can be instantiated with the literal `"SPARQL Tutorial"`. Thus the delete matches the committed triple. Therefore the following commitment is valid.

```
Before: { eg:book4 dc:title "SPARQL Tutorial" }
Update: SELECT ?title {
          DELETE { eg:book4 dc:title ?title }
        }
 After: {}
```

5.5 The Choose Left Rule and Choose Right Rule

The Choose Rules allow one of two updates to be committed. The choose rule has a left and right form, where respectively the left or right update is applied. The result of a choice is the result of the update chosen. Consider the Choose Left Rule and suppose that the following commitment relation holds.

Before: $Data_0$ Update: $Update_0$ After: $Data_1$

Given the above commitment relation, the following commitment relation holds.

Before: $Data_0$ Update: $Update_0$ `CHOOSE` $Update_1$ After: $Data_1$

The rule above chooses the left update. The Choose Right Rule is the symmetric rule which chooses the right branch instead.

Example of a Choice of Updates. The following demonstrates an update where either the first delete or second delete may be triggered. The two branches use different versions of the Dublin Core metadata vocabulary. In this case, the committed RDF Data matches the right branch. The result is that the committed triple is deleted.

```
Before: { eg:book5 dc11:title "SPARQL Protocol Tutorial" }
Update: SELECT ?title {
          DELETE { eg:book5 dc:title ?title }
          CHOOSE
          DELETE { eg:book5 dc11:title ?title }
        }
 After: {}
```

Note that if both branches are satisfied then one branch is chosen non-deterministically. If the update is iterated then two copies of the update can be posed where each copy chooses a different branch. This is different from forcing both branches to be performed simultaneously, which would be expressed as a join of updates rather than as a choice between updates.

5.6 The Filter Axiom

The Filter Axiom imposes a constraint on an update. The constraint is disposed only if the constraint evaluates to true. If the constraint does not evaluate to true then the update is blocked. The procedure for deciding whether a constraint holds is specified in the SPARQL Query Recommendation [17]. Given that the constraint evaluates to true the following commitment relation holds.

Before: {} Update: `FILTER` *Constraint* After: {}

An Example of a Filtered Update. The following commitment relation holds. The update deletes the title of a book, where the title and the book are discovered using Select. The filter imposes the constraint that the title must also satisfy a regular expression. The literal in the committed triple does match the regular expression. The triple is deleted.

```
Before: { eg:book4 dc:title "SPARQL Tutorial" }
Update: SELECT :a, ?title {
          DELETE { :a dc:title ?title }
          FILTER regex(?title, "^SPARQL")
        }
 After: {}
```

5.7 The Rules for Iterated Updates

All updates above are applied exactly once. Often the update should be applied wherever possible in an RDF store. This is achieved by iteration. The rules for iteration are similar to those for a Kleene star in a regular expression. Regular expressions are commonly used to update text files. This work is a generalisation of this common technique to RDF stores. Generalisations of regular expression date back to the commutative regular algebras of J. H. Conway [8], and remain a prominent area of research [14,10].

Updates can be applied any number of times. Iteration is used when the number of times to apply an update is not known. The Weekening Axiom allows an interated update to be applied zero times, if there is no term which matches the update. The Weakening Axiom terminates an iterated update with no effect.

Before: {} Update: `DO` *Update* After: {}

The Dereliction Rule allows an iterated update to be applied once. Assume that an update can be committed in the presence of some term resulting in some process. Dereliction allows the same update but iterated to be committed in the presence of the same term with the same resulting process. Suppose that the following commitment relation holds.

Before: *Data*$_0$ Update: *Update* After: *Data*$_1$

Given the above commitment relation, the following commitment relation holds.

Before: *Data*$_0$ Update: `DO` *Update* After: *Data*$_1$

The Contraction Rule allows two copies of an iterated update to be simultaneously committed. Contraction can be applied repeatedly, along with the Join Rule and Dereliction Rule, to simultaneously commit any number of copies of an iterated update. Suppose that the following commitment relation holds.

Before: *Data*$_0$ Update: `DO` *Update* `JOIN` `DO` *Update* After: *Data*$_1$

Given the commitment relation above, the following commitment relation holds.

Before: *Data*$_0$ Update: `DO` *Update* After: *Data*$_1$

The combination of the Weakening, Dereliction and Contraction rules allow zero, one, two, or more copies of an iterated update to be atomically committed. The use of Join in the Contraction Rule ensures that disjoint RDF Data is used for each copy of the update.

An Example of an Iterated Update. The following demonstrates an iterated update. The update replaces occurrence of the predicate *dc11:title* with the predicate *dc:title*. The iteration of this update means that the update can be applied twice. The result is that two triples are committed and replaced by two new triples.

```
 Before: { eg:book5 dc11:title "Query Tutorial" } ,
         { eg:book6 dc11:title "Update Tutorial" }
 Update: DO SELECT :a ?x {
           DELETE { :a dc11:title ?x }
           INSERT { :a dc:title ?x }
         }
  After: { eg:book5 dc:title "Query Tutorial" } ,
         { eg:book6 dc:title "Update Tutorial" }
```

5.8 The Context Rule for Unused Data

The Context Rule allows some data to not be used in an update. This is important when considering updates in the context of an RDF store. The rule composes the same unchanged data before and after the update. Assume that the following commitment holds.

Before: *Process*$_0$ Update: *Update* After: *Process*$_1$

Given the above commitment, the following commitmnet holds.

Before: $Process_2$, $Process_0$ Update: *Update* After: $Process_2$, $Process_1$

The context rule is demonstrated in the example in the next section, where one of the two triples is not updated.

5.9 The Blank Node Rule for Updating Local Names

The Blank Node Rule is used for updates which involve blank nodes. The trick is to treat the blank node as a temporary URI in the premise of the rule. The temporary URI must not appear free in the conclusion of the rule, thus an extra side-condition is required. Suppose that the following commitment holds.

Before: $Data_0$, $Data_1$ Update: *Update* After: $Data_2$, $Data_3$

Given that the above commitment holds, such that *:a* is not free in $Data_0$ or $Data_2$, then the following commitment holds.

Before: $Data_0$, `BNODE` *:a* $Data_1$ Update: *Update* After: $Data_2$, `BNODE` *:b* $Data_3$

An Example of the Blank Node Rule. The following example demonstrates a blank node updated. A temporary URI can represent *:a* in the premise of the Blank Node Rule. This allows the update to be considered as if *:a* is not bound. One triple is deleted by a commitment relation, which discovers the temporary URI. However, the conclusion of the Blank Node Rule still binds *:a*. This has the effect of discovering the blank node and using it in an update.

Before:

```
 Before: BNODE :a {
           { :a foaf:name "Alice" } ,
           { :a foaf:mbox mailto:alice@example.org }
         }
 Update: SELECT :b {
           DELETE { :b foaf:mbox mailto:alice@example.org }
           INSERT { :b foaf:mbox mailto:alice@new.org }
         }
  After: BNODE :a {
           { :a foaf:name "Alice" } ,
           { :a foaf:mbox mailto:alice@new.org }
         }
```

5.10 An Example of a Nested Update

This example, firstly, demonstrates most of the constructs combined to answer a larger update. Secondly, it demonstrates a common scenario which is enabled by nested selects and nested explicit iteration, which is impossible to express as an atomic update in initial proposals for SPARQL Update [18,9]. Consider the following commitment, which removes all *foaf:knows* links to people younger than 18.

```
 Before: { eg:youth0 eg:dob '01-01-2010' } ,
         { eg:youth1 eg:dob '01-02-2010' } ,
         { eg:person foaf:knows eg:youth0 } ,
         { eg:person foaf:knows eg:youth1 } ,
         { eg:youth0 foaf:knows eg:youth1 }
 Update: DO SELECT :a ?dob {
           WHERE { :a eg:dob ?dob }
           FILTER(current-year – year(?dob) < 18)
           DO SELECT :b {
            DELETE { :b foaf:knows :a }
           }
         }
  After: { eg:youth0 eg:dob '01-01-2010' } ,
         { eg:youth1 eg:dob '01-02-2010' }
```

Without the nested iteration and selects, the effect of the above update could only be achieved using two updates. This means that the update would not be atomic. The above update is correct and atomic. This example highlights a common problem which also appears in the first SPARQL Query recommendation [17]. This illustrates an improvement made by this work to the expressiveness of the language.

6 Related Work

Updates for RDF have been considered before using the RUL language [7]. The work on RUL focusses on the effect of updates in the presence of RDFS entailment. The effect of RDFS entailments and the operational semantics of the core of SPARQL Update are perpendicular issues. Thus, this update language can be extended to accommodate RDFS entailment as considered in RUL.

This language only claims to model a core of SPARQL Update. A feature missing is the handling of named graphs [6]. Named graphs can be achieved naïvely by allowing quads as well as triples, in the data format. However, further subtleties may arise depending on design decisions of the working group.

An operational semantics for updates enables further formal investigations. This operational semantics can be used to derive equivalences over updates, where two equivalent updates are operationally indistinguishable. This equivalence can be used to verify an algebra for rewriting updates without changing their operational behaviour. An algebra over updates is useful for the optimisation of updates. This line of work has been investigated for queries by the authors [12]. The same results also hold for updates; for instance iteration, join, choice, true and false form a commutative Kleene algebra with tests and quantifiers [14]. Kleene algebras are a common and useful algebra in computer science, a prominent example being the equations of regular expressions.

An operational semantics also allows a type system for updates to be specified. A simple static type system checks that literals are of the correct type to evaluate any constraints in which they appear. The operational semantics are then essential to verify that the static properties guaranteed by the type system are preserved by the behaviour of updates. A type system can be realised through type reconstruction, meaning that a programmer does not need to program using types for the type system to be useful. More powerful type systems inspired by RDFS can also ensure that URIs are used consistently, as investigated by the first author [11].

7 Conclusion

This work introduces a language with an operational semantics for fine grained updates over RDF. A fine grained update language is important for enabling a Read–Write Web of Data [3], where contributing content is as important as consuming content. SPARQL Update is currently being developed by the W3C as a standardised fine grained update language for RDF, for which an operational semantics is beneficial [9]. This work provides the first such operational semantics in the established tradition of Plotkin [16].

A specification with an operational semantics has many benefits. This operational semantics can be used as the basis of tools for compiler verification, type checking and optimisation [12,11]. It can also be used to evaluate the proposed language to tackle issues including: redundant operations, ambiguous definitions, rigid syntax, inadequate expressiveness, and incomplete specifications, as described below.

The operational semantics exposes redundant operations, which could instead be expressed using a combination of operators. Redundancies reduce the number of operators which must be directly implemented. For instance, it is shown in Sec. 3.5 that it is a myth that operator `OPTIONAL` is primitive.

Ambiguous definitions are clarified by the operational semantics. For instance, it is not clear in the original proposal [18] whether queries occur sequentially before or in parallel to an update. In this operational semantics queries occur in parallel, which allows more concise updates and avoids concurrency issues.

The abstract syntax (Sec. 3.4) is more explicit than the concrete syntax. For instance, the quantifiers of the abstract syntax clearly delimit the scope of a variable; whereas in the original proposal it is not obvious which variables share the same scope. Furthermore the abstract syntax is compositional, meaning that updates can be written in a modular fashion then combined, enabling interesting programming techniques and optimisations.

Some modest improvements to the expressive power of the language can be suggested. For instance, the `CHOOSE` operator (Sec. 5.5) can be extended to the entire language, instead of only queries. Similarly, Example 5.10 cannot be expressed without explicit iteration. Such improvements enable some common scenarios to be expressed.

The operational semantics presented here, goes considerably further than the formal model sketched in the current working draft. For instance, this operational semantics precisely specifies how blank nodes are updated (Sec. 5.9). Importantly, the model is purely operational, unlike the sketched model which combines operational and denotational approaches.

This work has been presented without using any meta-syntax to express the operational semantics. This can be achieved since an operational semantics is defined directly over the syntax of a language. The intention is that this style of presentation of operational semantics is accessible to a diverse audience.

References

1. Abramsky, S.: What are the fundamental structures of concurrency? We still don't know! Electronic Notes in Theoretical Computer Science 162, 37–41 (2006);Proceedings of the Workshop Essays on Algebraic Process Calculi
2. Beckett, D., Berners-Lee, T.: Turtle - Terse RDF Triple Language. Team submission, W3C (2008)
3. Berners-Lee, T.: Read-Write Linked Data, personal view (December 2010), `http://www.w3.org/DesignIssues/ReadWriteLinkedData.html`
4. Berners-Lee, T., Connolly, D., Kagal, L., Scharf, Y., Hendler, J.: N3logic: A logical framework for the World Wide Web. Theory and Practice of Logic Programming 8(3), 249–269 (2008)
5. Brickley, D., Guha, R.: RDF vocabulary description language 1.0: RDF Schema. Recommendation REC-rdf-schema-20040210, W3C (2004)
6. Carroll, J.J., Bizer, C., Hayes, P., Stickler, P.: Named graphs. Journal of Web Semantics 3(4), 247–267 (2005)
7. Magiridou, M., Sahtouris, S., Christophides, V., Koubarakis, M.: RUL: A Declarative Update Language for RDF. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, pp. 506–521. Springer, Heidelberg (2005)
8. Conway, J.H.: Regular Algebra and Finite Machines. Chapman and Hall (1971)
9. Gearon, P., Passant, A., Polleres, A.: SPARQL 1.1 Update. Working draft WD-sparql11-update-20110512, W3C (May 2011)
10. Hoare, C., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene Algebra. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 399–414. Springer, Heidelberg (2009)
11. Horne, R.: Programming Languages and Principles for Read–Write Linked Data. Ph.D. thesis, University of Southampton (2011)
12. Horne, R., Sassone, V.: A verified algebra for Linked Data. In: FOCLASA 2011, Aachen, August 10. Electronic Proceedings in Theoretical Computer Science, vol. 58, pp. 20–33 (2011)
13. Klyne, G., Carroll, J.: Resource Description Framework: Concepts and abstract syntax. Recommendation REC-rdf-concepts-20040210, W3C (2004)
14. Kozen, D.: Kleene algebra with tests. ACM Transactions on Programing Languages and Systems 19, 427–443 (1997)
15. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Transactions on Database Systems 34(3), 1–45 (2009)
16. Plotkin, G.D.: A structural approach to operational semantics. internal notes. Aarhus University (1981)
17. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. Recommendation REC-rdf-sparql-query-20080115, W3C (2008)
18. Seaborne, A., Manjunath, G.: SPARQL/Update: A language for updating RDF graphs. External HPL-2007-102, HP Labs Bristol (2007)

Appendix: Summary of the Operation Semantics

For a concise summary of the operational semantics, some meta-syntax is introduced. Commitments are written in the body of the paper as follows.

$$\text{Before: } \mathit{Data}_0 \qquad \text{Update: } \mathit{Update} \qquad \text{After: } \mathit{Data}_1$$

In this summary commitments are instead written as follows.

$$\mathit{Data}_0 \;,\; \mathit{Update} \longrightarrow \mathit{Data}_1$$

Rules which are explained in English in the body of this work are presented in the style of natural deduction, as conventional for operational semantics [16]. The horizontal line separate the premises on the top from the conclusion on the bottom. Some rules have side conditions. The axioms and rules of the core update language are therefore summarised as follows.

$$\text{Delete: } \mathit{Data} \;,\; \texttt{DELETE}\,\mathit{Data} \longrightarrow \{\} \qquad \text{Insert: } \{\} \;,\; \texttt{INSERT}\,\mathit{Data} \longrightarrow \mathit{Data}$$

$$\text{Filter: } \{\} \;,\; \texttt{FILTER}\,\mathit{Constraint} \longrightarrow \{\} \;\text{ only if } \mathit{Constraint} = \texttt{true}$$

$$\text{Join: } \frac{\mathit{Data}_0 \;,\; \mathit{Update}_0 \longrightarrow \mathit{Data}_2 \quad \mathit{Data}_1 \;,\; \mathit{Update}_1 \longrightarrow \mathit{Data}_3}{\mathit{Data}_0 \;,\; \mathit{Data}_1 \;,\; \mathit{Update}_0 \;\texttt{JOIN}\; \mathit{Update}_1 \longrightarrow \mathit{Data}_2 \;,\; \mathit{Data}_3}$$

$$\text{Select literal: } \frac{\mathit{Data}_0 \;,\; \mathit{Update}[\texttt{"literal"}/?\mathit{variable}] \longrightarrow \mathit{Data}_1}{\mathit{Data}_0 \;,\; \texttt{SELECT}\; ?\mathit{variable}\;\mathit{Update} \longrightarrow \mathit{Data}_1}$$

$$\text{Select URI: } \frac{\mathit{Data}_0 \;,\; \mathit{Update}[:b/:a] \longrightarrow \mathit{Data}_1}{\mathit{Data}_0 \;,\; \texttt{SELECT}\; :a\;\mathit{Update} \longrightarrow \mathit{Data}_1}$$

$$\text{Choose left: } \frac{\mathit{Data}_0 \;,\; \mathit{Update}_0 \longrightarrow \mathit{Data}_1}{\mathit{Data}_0 \;,\; \mathit{Update}_0 \;\texttt{CHOOSE}\; \mathit{Update}_1 \longrightarrow \mathit{Data}_1}$$

$$\text{Choose right: } \frac{\mathit{Data}_0 \;,\; \mathit{Update}_1 \longrightarrow \mathit{Data}_1}{\mathit{Data}_0 \;,\; \mathit{Update}_0 \;\texttt{CHOOSE}\; \mathit{Update}_1 \longrightarrow \mathit{Data}_1}$$

$$\text{Weakening: } \{\} \;,\; \texttt{DO}\,\mathit{Update} \longrightarrow \{\} \qquad \text{Dereliction: } \frac{\mathit{Data}_0 \;,\; \mathit{Update} \longrightarrow \mathit{Data}_1}{\mathit{Data}_0 \;,\; \texttt{DO}\,\mathit{Update} \longrightarrow \mathit{Data}_1}$$

$$\text{Contraction: } \frac{\mathit{Data}_0 \;,\; \texttt{DO}\,\mathit{Update} \;\texttt{JOIN}\; \texttt{DO}\,\mathit{Update} \longrightarrow \mathit{Data}_1}{\mathit{Data}_0 \;,\; \texttt{DO}\,\mathit{Update} \longrightarrow \mathit{Data}_1}$$

$$\text{Structure: } \frac{\mathit{Data}_0 = \mathit{Data}_2 \quad \mathit{Data}_0 \;,\; \mathit{Update} \longrightarrow \mathit{Data}_1 \quad \mathit{Data}_1 = \mathit{Data}_3}{\mathit{Data}_2 \;,\; \mathit{Update} \longrightarrow \mathit{Data}_3}$$

$$\text{Context: } \frac{\mathit{Data}_0 \;,\; \mathit{Update} \longrightarrow \mathit{Data}_1}{\mathit{Data}_0 \;,\; \mathit{Data}_2 \;,\; \mathit{Update} \longrightarrow \mathit{Data}_1 \;,\; \mathit{Data}_2}$$

$$\text{Blank node: } \frac{\mathit{Data}_0 \;,\; \mathit{Data}_1 \;,\; \mathit{Update} \longrightarrow \mathit{Data}_2 \;,\; \mathit{Data}_3}{\mathit{Data}_0 \;,\; \texttt{BNODE}\,:a\,\mathit{Data}_1 \;,\; \mathit{Update} \longrightarrow \mathit{Data}_2 \;,\; \texttt{BNODE}\,:a\,\mathit{Data}_3}$$
$$\text{only if } :a \text{ does not appear free in } \mathit{Data}_0 \text{ or } \mathit{Data}_2$$

Any commitment which can be derived using the above axioms and rules is a valid commitment. Valid commitments include all examples in the body of this work.