An RDF Design Pattern for the Structural Representation and Querying of Expressions

Sébastien Ferré

An RDF Design Pattern for the Structural Representation and Querying of Expressions

Sébastien Ferré

IRISA, Université de Rennes 1
Campus de Beaulieu, 35042 Rennes cedex, France
Email: ferre@irisa.fr

Abstract. Expressions, such as mathematical formulae, logical axioms, or structured queries, account for a large part of human knowledge. It is therefore desirable to allow for their representation and querying with Semantic Web technologies. We propose an RDF design pattern that fulfills three objectives. The first objective is the structural representation of expressions in standard RDF, so that expressive structural search is made possible. We propose simple Turtle and SPARQL abbreviations for the concise notation of such RDF expressions. The second objective is the automated generation of expression labels that are close to usual notations. The third objective is the compatibility with existing practice and legacy data in the Semantic Web (e.g., SPIN, OWL/RDF). We show the benefits for RDF tools to support this design pattern with the extension of SEWELIS, a tool for guided exploration and edition, and its application to mathematical search.

1 Introduction

Complex expressions account for a large part of human knowledge. Common examples are mathematical equations, logical formulae, regular expressions, or parse trees of natural language sentences. In the Semantic Web (SW) [4], they can be OWL axioms, SWRL rules, or SPARQL queries. It is therefore desirable to allow for their representation in RDF so that they can be mixed with other kinds of knowledge. For example, it should be possible to describe a theorem by its author, its discovery date, its informal description as a text, and its formal description as a mathematical expression, all in RDF. An expected advantage of the formal representation of expressions is the ability to search those expressions by their content, which is a problem that has been studied in *mathematical search* [3,6,16,12,1]. For example, we may wish to retrieve all expressions that are an integral in some variable x and whose body contains x^2 as a sub-expression. Correct answers are $\int x^2 + 1\,dx$ and $\int y^2 - y\,dy$. This example exhibits two difficulties in expression search. The first difficulty is to take into account the nested structure of expressions, e.g., the fact that the sub-expression x^2 must be in the scope of the integral. The second difficulty is to abstract over the name of bound variables, e.g., the variable x bound by the

integral $\int\ dx$ can be renamed as y without changing the meaning of the expression. Another difficulty that we do not consider here is the need for logical and mathematical reasoning to recognize, for instance, that $x(x+1)$ or $\Sigma_{i=1}^{x} i$ implicitly contains x^2. The two difficulties above also apply to other kinds of expressions, and this paper addresses the more general problem of the structural representation and querying of expressions, only using mathematical expressions as a representative illustration.

The need for representing expressions in RDF has already been recognized, but to our knowledge, no generic solution has been proposed. In the domain of mathematical search, the survey by Lange [7] shows that few complete RDF representations have been proposed [10,14], and that none of them have been implemented and adopted because of awkward representations, and lack of support by RDF tools. More conservative approaches are easier to adopt but miss the objective of a tight integration of mathematical knowledge to the Semantic Web. For example, XML literals can be used to represent mathematical objects in RDF graphs but the content of literals is mostly opaque to RDF tools. In the Semantic Web, a well-known use case is the representation of complex classes in OWL ontologies. To this purpose, the OWL vocabulary defines a number of *structural* classes (e.g., `owl:Restriction`) and properties (e.g., `owl:onProperty`), which have no ontological meaning in themselves. Another example of a vocabulary that defines structural classes and properties is SPIN SPARQL Syntax [15], for the representation of SPARQL queries. OWL/RDF and SPIN use similar RDF patterns in their representations, and therefore offer a good basis for generalization.

In this paper, we propose an RDF design pattern for expressions that fulfills three objectives. The first objective (Section 2) is the structural representation of expressions in standard RDF, so that expressive structural querying (e.g., mathematical search) is made possible. We propose to re-use the standard RDF containers, and we introduce a small extension of Turtle and SPARQL notations for a more concise notation of descriptions and queries. The second objective (Section 3) is the generation of expression labels that are close to usual notations (e.g., using infix operators, symbols). This is important for RDF tools because expressions are generally represented by blank nodes, and because it would be tedious to manually attach a label to every expression and sub-expression. The third objective (Section 4) is the backward compatibility of systems using our expression design pattern with existing practice, and legacy data, in the Semantic Web (e.g., OWL/RDF, SPIN). This implies that legacy data need not be changed in order to benefit from the advantages of our design pattern, in particular the generation of labels. We claim that RDF tools can be adapted to support our design pattern, and that such support can enable users to describe and query complex expressions of any kind. We illustrate this (Section 5) by adapting SEWELIS [2], a tool for the guided exploration and edition of RDF graphs, and by applying it to mathematical search. We show (Section 6) that this approach is competitive with state-of-the-art in mathematical search w.r.t. expressiveness and readability of queries.

2 Representation of Expressions in RDF

In order to represent expressions in RDF, we rely on the fact that every expression can be represented as a syntax tree, and hence as a graph. A side-advantage of graphs compared to syntax trees is the possibility to share sub-expressions. Syntax tree leaves, i.e. *atomic expressions*, can be symbols, values, and variables; and syntax tree nodes, i.e. *compound expressions*, are labeled by symbols (e.g., operators, functions, quantifiers). Symbols are constants with a universal scope, and therefore, they are naturally represented by URIs. Note that those URIs could be derived from existing vocabularies, such as OpenMath[1]. A benefit of URIs is that, when a symbol (e.g., e) has different meanings (e.g., the base of the natural logarithm or the elementary charge), different URIs can be used to avoid ambiguity. A symbol can be linked to a URI as a label to specify its surface form, and nothing forbids different URIs to have the same label. Values such as numbers and strings are naturally represented by RDF literals. To account for the three facts that a variable can have several occurences, that distinct variables may have the same name, and that variables are not accessible out of their scope, we choose to represent variables as blank nodes, and to represent variable names as labels of those blank nodes. The same choice has been made in SPIN.

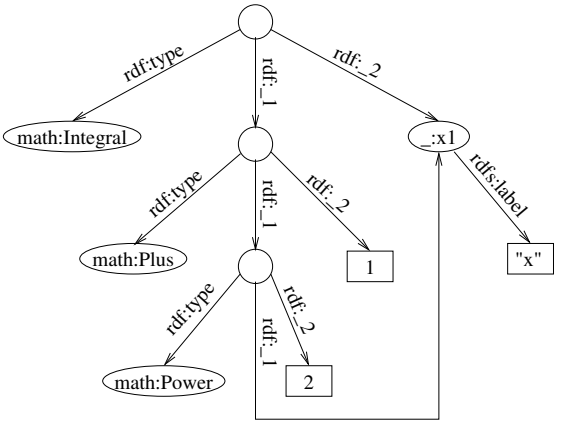
It remains to define the representation of compound expressions, i.e. tree nodes. A compound expression is completely defined by the node label, which we here call *constructor*, and the sequence of sub-expressions, which we here call *arguments*. We propose to represent a compound expression by RDF containers. An RDF container is generally a blank node (it can also be a URI), has a type (e.g., `rdf:Seq` for sequences), and is linked to its elements through the predefined predicates `rdf:_1` (1st element), `rdf:_2` (2nd element), etc. The idea is here to use the constructors of compound expressions (i.e., node labels) as container types, i.e. as subclasses of `rdfs:Container`. In each compound expression, the arity of the constructor determines the number of arguments. That representation is close to what is done in OWL/RDF or SPIN, except that membership properties `rdf:`$\_n$ are used for the arguments instead of *ad-hoc* properties (e.g., `owl:onProperty`). Section 4 explains how to reconcile the two approaches in practice.

The advantages of our RDF representation are its simplicity, and its conformance to existing standard. Indeed, no special vocabulary needs to be defined apart from the URIs that are used for symbols. It is true that there is little support for containers in existing tools. However, we show in this paper that notations and tools can be extended at a moderate cost, and it will be arguably easier to convince their developers to do so for a standard RDF notion, containers, than for a domain-specific vocabulary. Moreover, support for containers could be useful in other contexts than the management of expressions.

Figure 1 shows the graphical and Turtle forms of the RDF representation of the expression $\int x^2 + 1\, dx$. This expression contains mathematical operators `math:Integral`[2] (integral), `math:Plus` (addition) and `math:Power` (power), which

[1] `http://www.openmath.org/cdindex.html`

[2] We here assume a namespace `math:` for mathematical constructors.




```
[ a math:Integral ;
  rdf:_1
     [ a math:Plus ;
       rdf:_1
          [ a math:Power ;
            rdf:_1 _:x1 ;
            rdf:_2 2 ] ;
       rdf:_2 1 ] ;
   rdf:_2 _:x1 ] .
_:x1 rdfs:label "x" .
```

Fig. 1. RDF representation of the expression $\int x^2 + 1\, dx$: graphical (a), Turtle (b).

are all used as binary constructors. In the case of `math:Integral`, the second argument plays the role of the binding variable of the integral. The expression also contains the integer literals `1` and `2`, as well as a blank node `_:x1`, labelled `"x"`, to represent the variable x. The standard property `rdf:type` is used to link compound expressions to their constructor. This implies that a constructor is also an RDF class that contains all compound expressions based on that constructor. The standard properties `rdf:_`n are used to link compound expressions to their arguments. The standard property `rdfs:label` is used to link variable identifiers to variable names. In the Turtle notation, every compound expression appears as delimited by square brackets, and the standard property `rdf:type` is abbreviated by `a` (meaning "is a").

This representation correctly handles the full content of the expression while abstracting over possible variations in the presentation (e.g., brackets, various notations of integrals). This representation also makes it possible to distinguish different variables that have the same name (e.g., in $x + \int x\, dx$) by using distinct blank nodes. Invariance to the renaming of bound variables is addressed by separating the identity of the variable (as a blank node) and its concrete name (as a label). Indeed, invariance to renaming also applies to blank nodes: `_:x1` can be replaced by `_:x2` without changing the meaning of the RDF graph.

About RDF collections. An alternative representation of compound expressions would be RDF collections (aka. RDF lists) whose elements would be the constructor followed by arguments. However it would require twice as many triples, and would make querying more costly. Its notation in Turtle and SPARQL would be more concise for expression $\int x^2 + 1\, dx$: `(math:Integral (math:Plus (math:Power _:x1 2) 1) _:x1)`. However, it is only because abbreviations have been defined for RDF collections and not yet for RDF containers. We precisely propose such abbreviations for containers in the following.

Table 1. Expansion rules for new Turtle/SPARQL syntactic abbreviations.

$$C(E_1,\ldots,E_n) \longrightarrow \texttt{[a } C\texttt{; rdf:\_1 } E_1\texttt{; ...; rdf:\_}n\ E_n\ \texttt{]}$$
$$\ldots E\ldots \longrightarrow \texttt{[rdfs:member* } E\ \texttt{]}$$
$$S\ \texttt{is [} P_1\ O_1\texttt{; ...; } P_n\ O_n\ \texttt{]} \longrightarrow S\ P_1\ O_1\texttt{; ...; } P_n\ O_n$$

Turtle Abbreviations for Expressions As the above Turtle notation is rather verbose, we propose a small extension of the syntax of Turtle and SPARQL with a few abbreviations in order to allow for more concise descriptions and queries (see Table 1). Those abbreviations work exactly the same as the Turtle/SPARQL notation for RDF collections (a.k.a. lists), where the form `(`E_1 ... E_n`)` is expanded into the form `[rdf:first` E_1`; rdf:rest` ... `[rdf:first` E_n`; rdf:rest rdf:nil]` ... `]`. The first abbreviation is a functional notation for RDF containers (hence for expressions), where container types (hence constructors) play the role of functions, and container elements play the role of arguments. The second and third abbreviations are ellipsis notations to reach sub-expressions in queries, and rely on transitive closures of the property `rdfs:member`, which is a super-property of the properties `rdf:`$\_n$. Those notations can be used everywhere blank nodes can be used, including as a whole statement. The last abbreviation allows blank nodes, and hence the functional and ellipsis notations, to be used as a predicate-object list by prefixing it with the keyword `is`. With this extension, which we name Turtle+/SPARQL+ in the scope of this paper, the example in Figure 1 can be written as follows, using the first abbreviation.

```
math:Integral(math:Plus(math:Power(_:x1,2),1),_:x1) .
_:x1 rdfs:label "x" .
```

Other abbreviations are illustrated below in queries. Note that it is easy to extend Turtle and SPARQL parsers to accept those abbreviations, and that no change at all is required on the RDF data model or the SPARQL query language.

SPARQL Expression Patterns In order to validate the adequacy of our representation for structural search in SPARQL, we discuss the formulation of SPARQL queries for a few representative examples. SPARQL graph patterns are here used to constrain the shape of searched expressions. SPARQL variables are here used to match arbitrary sub-expressions, and to state equality constraints between several sub-expressions. This provides a way to solve the difficulty related to the renaming of bound variables: it suffices to introduce a SPARQL variable for each bound variable. For example, if we look for expressions like $\int x^2\,dx$ or $\int y^2\,dy$, we can use the following SPARQL+ query, where SPARQL variables stand for expressions (not values!).

```
SELECT ?e WHERE { ?e is math:Integral(math:Power(?x,2),?x) . }
```

The query in the introduction that retrieves the integrals in x whose body contains the term x^2 can then be expressed as follows, using the notation $\ldots E\ldots$ (see Table 1) to express the relation from an expression to its sub-expressions.

```
SELECT ?e WHERE { ?e is math:Integral(...math:Power(?x,2)..., ?x).}
```

This query returns the expressions $\int x^2\,dx$, $\int x^2 + 1\,dx$, $\int y^2 - y\,dy$, but not the expressions $\int x^2 + y\,dy$ and $\int 2x\,dx = x^2 + c$. Now, starting from the same example, assume that we want to retrieve the bodies of the integrals instead of the integrals themselves. After a reformulation to introduce a variable for the body of the integral, we obtain:

```
SELECT ?e WHERE { ?e is ...math:Power(?x,2)... .
                  math:Integral(?e,?x) . }
```

This query looks for an expression that contains x^2 as a sub-expression, and that appears as the body (1st argument) of an integral whose binding variable is x. As a conclusion, SPARQL provides enough expressivity to cover the needs of the structural querying of expressions. A comparison with existing query languages for mathematical search is given in Section 6.2.

3 Labelling of Expressions in RDF Tools

The main reason why RDF structural representations have not been widely adopted is the lack of support by RDF tools. This lack of support concerns in particular n-ary structures such as RDF containers and RDF collections (chained lists), which are necessary for the representation of expressions [7]. More essentially, this lack of support concerns blank nodes, which are often associated to n-ary structures. The problem with blank nodes is that they are notoriously difficult to present in query results, and in RDF tools in general [8]. Indeed, blank nodes have no names (anonymous resources), and their identifiers are contingent, and only relevant for internal use.

Our proposal is therefore to display blank nodes by the RDF structure they represent, rather than by their internal identifier as this is generally done. Turtle+ can be used to render expression contents. For example, the expression $\int x^2 + 1\,dx$ can be rendered by the string "`math:Integral(math:Plus(math:Power([rdfs:label "x"],2),1),[rdfs:label "x"])`". This is more concise than using standard Turtle, but this is still far from the usual mathematical notation.

We here propose to express annotations on constructors that can be used by RDF tools to generate usual representations of expressions. A similar approach has been followed in XML for the rendering of mathematical expressions from OpenMath and MathML-Content to LaTeX or MathML-Presentation [5]. The principle is that, when an expression has not been annotated explicitly with a label, a label will automatically be generated for it as an aggregation of the labels of its parts. The generated label need not be added to the store, but may simply be generated dynamically by the tool, as needed. By default, the functional notation is used, like in Turtle+, but replacing the constructor and the arguments by their label. Of course, the label of an argument can itself be generated in the case it is a compound expression. Because those labels are only for display, all

Table 2. Notation specifications (template and priorities) for a few constructors.

constructor	template	priorities
math:Plus	`_ + _`	plus(plus,plus)
math:Minus	`_ - _`	plus(plus,times)
math:Times	`_ _`	times(times,times)
math:Power	`_^_`	power(atom,power)
math:Sin	`sin _`	plus(times)
math:Fact	`_!`	power(atom)
math:Integral	∫ `_ d_`	atom(plus,atom)
math:Set	{`_, _`}	atom(plus)

Unicode characters can be used, including mathematical symbols (e.g., ∫ for `math:Integral`, + for `math:Plus`, ^ for `math:Power`). Applying this to the above example generates the label "∫`(+(^(x,2),1),x)`" for the expression $\int x^2 + 1\, dx$.

Many mathematical operators use different notations than the functional notation: e.g., infix notation in $x+1$, prefix notation in $sin\ x$, postfix notations in $n!$, and mixfix notations in $\int x^2\, dx$. With appropriate annotations of constructors, we could generate the label "∫ `x^2 + 1 dx`" for the above example, which is very close to the mathematical notation. However, those notations could lead to ambiguities, and brackets must be inserted according to the priority level of operators. For example, in the expression `math:Div(math:Plus(1,math:Power(_:x,2)),_:x)`, brackets are necessary around the addition, but not around the power, according to usual priorities. Therefore, the minimal bracketing of the expression leads to the label "`(1 + x^2)/x`". Without the brackets, the expression would be misinterpreted by human users, and adding superfluous brackets would make the expression label less readable.

We propose to use classical pretty-printing techniques to generate expression labels, based on the fixity and priority of operators. In order to allow RDF tools to perform this pretty-printing in a generic way, it is necessary to annotate constructors with all the necessary information. The necessary information comprises the *template* and the *priority signature* of the constructor. Table 2 gives the template and priority signature for a few constructors. A template is a string where the markers `_1`, ..., `_n` are placeholders for the (generated) labels of arguments. For example, a template for the addition is "`_1 + _2`". When arguments are placed in order, the generic marker `_` can be used instead: e.g., "`_ + _`". Alternately to strings, templates could be defined as XML literals, e.g., using the MathML presentation language, which would allow for a nice rendering in a browser [5]. A priority signature uses the functional notation with priority levels in place of the constructor and each argument. Priority levels are here named after their most representative operator, and are ordered in the usual way (e.g., 'times' represents higher priority than 'plus')[3]. For example, the priority sig-

[3] The precise representation of priority levels is not detailed here. URIs could be used, possibly from a standard vocabulary, and compared with a standard transitive property, e.g., prio:isHigherThan.

nature of math:Minus, `plus(plus,times)`, says that substraction has the same priority level as addition, and is left associative. More precisely, it says that additive expressions can be used without brackets as left argument, but must be used with brackets as right argument. The general rule is that an argument expression must be bracketed if its priority level is lower than the priority level of the argument. By default, round brackets are used, but a template with one place holder (e.g., "{_}" for curly brackets) can be associated to each priority level that uses a different bracketing. Finally, some constructors expect a variable number of arguments. For example, the set $\{1, 2, 3\}$ can be represented by `math:Set(1,2,3)`, where the constructor `math:Set` has an arbitrary arity. In this case, we use two placeholders in the template (here "{_, _}") in order to specify the separator (here the comma) to be used between elements; and the priority signature uses only one argument, assuming that all arguments have the same priority level. Examples of generated labels from definitions in Table 2 are: "$\int$ `x^2 + 1 dx`", "`(a + b)^2 = a^2 + 2 a b + b^2`", "`{1, x, x^2, x^3}`".

4 Compatibility with Legacy RDF Structures

Blank nodes and structural classes/properties have been used in a number of circumstances for representing structures in RDF. For example, in OWL/RDF an existential restriction $\exists r.C$ is represented by a combination of the class `owl:Restriction`, and the properties `owl:onProperty` and `owl:someValuesFrom`, i.e. by the blank node `[ a owl:Restriction ; owl:onProperty` r `; owl:someValuesFrom` C `]`. . The same representation principles are used for RDF collections with class `rdf:List` and properties `rdf:first` and `rdf:rest`, for SPIN SPARQL syntax, and in other circumstances [8].

OWL restrictions could equally well be represented as expressions, using our approach. Assuming the constructor `owl:Some` in the OWL namespace, an existential restriction $\exists r.C$ would be represented as `owl:Some(`r`,`C`)`. That representation is close to OWL functional syntax[4], and are valid notations in Turtle+. A first advantage of our approach is that each construct is defined by a single constructor URI instead of a combination of classes and properties. A second advantage is a better separation between ontological classes and properties (e.g., `owl:equivalentClass`) and structural classes and properties (e.g., `owl:onProperty`). In our approach, the latter are only the constructors and the container membership properties `rdf:_`n. A third advantage is that natural labels for expressions can be generated in a more systematic way, as explained in Section 3. Indeed, a system only needs to read the annotations of constructors, and needs not be hard-coded w.r.t. an *ad-hoc* vocabulary. An advantage of OWL/RDF and similar approaches is that arguments have an explicit name rather than a position.

In order to reconcile legacy data and the naming of arguments with functional notations and the systematic labelling of expressions, we introduce *implicit constructors*. An implicit constructor does not occur explicitly in the RDF graph of

[4] `http://www.w3.org/TR/owl2-syntax/#Functional-Style_Syntax`

Table 3. Definitions of a few implicit constructors for OWL/RDF and SPIN.

implicit constructor	constructor class	argument properties
owl:Some	owl:Restriction	(owl:onProperty,owl:someValuesFrom)
owl:All	owl:Restriction	(owl:onProperty,owl:allValuesFrom)
owl:And	owl:Class	(owl:intersectionOf)
sp:Select	sp:Select	(sp:resultVariables,sp:where)
sp:TriplePattern	sp:TriplePattern	(sp:subject,sp:predicate,sp:object)
sp:Filter	sp:Filter	(sp:expression)
sp:lt	sp:lt	(sp:arg1,sp:arg2)

Table 4. Different notations of a complex OWL class: Manchester (1), Turtle (2), Turtle+ (3), generated label (4).

1	*Pizza and hasTopping some MeatTopping and hasTopping some FishTopping*

2	`[ a owl:Class ;`
	`  owl:intersectionOf`
	`  ( ex:Pizza`
	`  [ a owl:Restriction ;`
	`  owl:onProperty ex:hasTopping ;`
	`  owl:someValuesFrom ex:MeatTopping ]`
	`  [ a owl:Restriction ;`
	`  owl:onProperty ex:hasTopping ;`
	`  owl:someValuesFrom ex:FishTopping ] ) ]`

3	`owl:And( ( ex:Pizza`
	`  owl:Some(ex:hasTopping,ex:MeatTopping)`
	`  owl:Some(ex:hasTopping,ex:FishTopping) ) )`

4	`"Pizza and has topping some Meat and has topping some Fish"`

expressions, but it is mapped to a combination of structural classes and properties. It serves to translate from functional notation to standard RDF, and as a handle for the annotations about the generation of labels. Table 3 defines a few implicit constructors for OWL/RDF and SPIN expressions, annotating each implicit constructor *Cons* by a class C, and a sequence of properties $(P_1, \ldots, P_n)$. Given such annotations, a Turtle+ expression $Cons(E_1, \ldots, E_n)$ is translated to the blank node `[ a` C `;` P_1 E_1 `;` $\ldots$ `;` P_n E_n `]`. For example, `owl:Some` is defined as an implicit constructor for existential restrictions in OWL/RDF. From there, it is easy to get any of the three main syntaxes for OWL restrictions as generated labels, e.g. for restriction $\exists child.Doctor$. For the functional syntax, it is enough to define the label "`some`" on constructor `owl:Some` to produce label "`some(child,Doctor)`". For the Manchester syntax, `owl:Some` has to be defined as a right-associative infix operator, like the power operator, but with template "_ `some` _": this produces label "`child some Doctor`". For the DL syntax, the constructor has instead to be defined as a mixfix operator with template "∃_._", producing label "∃`child.Doctor`".

Table 5. Different notations of a complex SPARQL query: SPARQL (1), Turtle (2), Turtle+ (3), generated label (4).

1	`SELECT ?x WHERE { ?x ex:age ?age . FILTER (?age < 18) }`

2	`[ a sp:Select ;`
	`  sp:resultVariables (_:x) ;`
	`  sp:where ([ a sp:TriplePattern ;`
	`  sp:subject _:x ;`
	`  sp:predicate ex:age ;`
	`  sp:object _:age ]`
	`  [ a sp:Filter ;`
	`  sp:expression [ a sp:lt ;`
	`  sp:arg1 _:age ;`
	`  sp:arg2 18 ] ]) ] .`
	`_:x a sp:Variable ; sp:varName "x" .`
	`_:age a sp:Variable ; sp:varName "age" .`

3	`sp:Select((_:x),`
	`  (sp:TriplePattern(_:x,ex:age,_:age)`
	`  sp:Filter(sp:lt(_:age,18)))) .`
	`_:x is sp:Variable("x") .`
	`_:age is sp:Variable("age") .`

4	`"SELECT ?x WHERE { ?x has age ?age . FILTER (?age < 18) }"`

Tables 4, and 5 compare different notations of two complex expressions in two different languages: OWL, and SPARQL. In each table, the first line is the native syntax of the language (Manchester syntax for OWL). The second line is the Turtle notation of the RDF representation (OWL/RDF for OWL, SPIN for SPARQL). The third line is the Turtle+ functional notation based on implicit constructors. The fourth line is the generated label assuming that implicit constructors have been defined, and that appropriate labelling notations have been associated to them. Note how the generated label can be made very similar to the native syntax. For the OWL Manchester syntax, `owl:And` has a collection argument whose separator is " `and` ", and `owl:Some` is defined as a right associative infix operator. `ow:Some` is given higher priority than `owl:And`. For SPIN SPARQL queries, `sp:Select` has two collection arguments whose separators are the space character, `sp:TriplePattern` simply uses the template "_ _ _ .", `sp:lt` is defined as an infix operator, and `sp:Variable` is defined as a prefix operator with template "`?`_". The priority level for atomic graph patterns uses the template "{ _ }" as brackets instead of the default round brackets.

5 Implementation in SEWELIS and Application to Mathematical Search

We have evaluated our RDF design pattern by implementing it in an RDF tool, and by applying it to mathematical search. We chose to add expression support

Table 6. Different notations of an ingredient description: English (1), Turtle (2), Turtle+ (3), generated label (4).

1	*1 lb of green mango*

2	`[ a ex:Ingredient ;`
	`  ex:ingredient ex:GreenMango ;`
	`  ex:amount [ a ex:Measure ;`
	`  ex:value 1 ;`
	`  ex:unit ex:lb ] ]`

3	`ex:Ingredient(ex:GreenMango,ex:Measure(1,ex:lb))`
4	`"1 lb of green mango"`

to SEWELIS, an RDF tool for the exploration and authoring of RDF graphs [2]. SEWELIS enables users to interactively build complex queries without the need to write anything, and hence without the risk of syntax errors. At each interaction step, query elements are computed from the dataset, and suggested to users for insertion in the query under construction. Therefore, the query and the results can be pretty-printed without the risk to introduce ambiguities. The same can be done in principle with other syntax-based editors, but SEWELIS has the additional advantage to be also semantic-based in that it provides only suggestions that match actual data, and it prevents users to fall on empty results. It also has the benefits to support exploratory search [9], when users do not have a precise idea of what they are looking for.

We have extended SEWELIS with the RDF representation of expressions, and the automated generation of labels, as presented in previous sections. The pretty-printing of queries is based on the same principles as for the generation of labels (see Section 3), extended to expression patterns. Variables are noted like in SPARQL (e.g., `?X`), and a bare question mark `?` is an anonymous variable. The following list shows how the queries from Section 2 are displayed in SEWELIS, along with their meaning for recall. The underlined part represents the *focus* that indicates which part of the query answers are to be displayed.

- $\underline{\int\ \texttt{?X\^{}2 d?X}}$: the integrals in x of x^2;
- $\underline{\int\ \texttt{...?X\^{}2...}\ \texttt{d?X}}$: the integrals in x whose body contains x^2;
- $\int\ \underline{\texttt{...?X\^{}2...}}\ \texttt{d?X}$: the bodies of the integrals in x that contain x^2.

We have then applied SEWELIS to the exploration of a collection of 70 formulas taken from an official document[5] used for the French high-school final exam. This is a small dataset but our purpose is to demonstrate the applicability and benefits of our RDF design pattern, and not the scalability of SEWELIS and RDF representations. Note that our RDF expressions are made of standard RDF for which efficient stores and query engines exist. Figure 2 shows the subset of those formulas that contain a square, as displayed in SEWELIS. This list has been obtained as the answers to the query `...?^2...`, which can be reached in

[5] `http://www.lyc-monod-clamart.ac-versailles.fr/IMG/pdf/FormulaireBac2003.pdf`

(u / v) ' = (u ' v - u v ') / v^2
(1 / u) ' = - u ' / u^2
a^3 - b^3 = (a - b) (a^2 + a b + b^2)
a^3 + b^3 = (a + b) (a^2 - a b + b^2)
(a - b)^2 = a^2 - 2 a b + b^2
(a - b)^3 = a^3 - 3 a^2 b + 2 a b^2 - b^3
(a + b)^2 = a^2 + 2 a b + b^2
(a + b)^3 = a^3 + 3 a^2 b + 2 a b^2 + b^3

Fig. 2. The list of formulas containing a square, as displayed in SEWELIS.

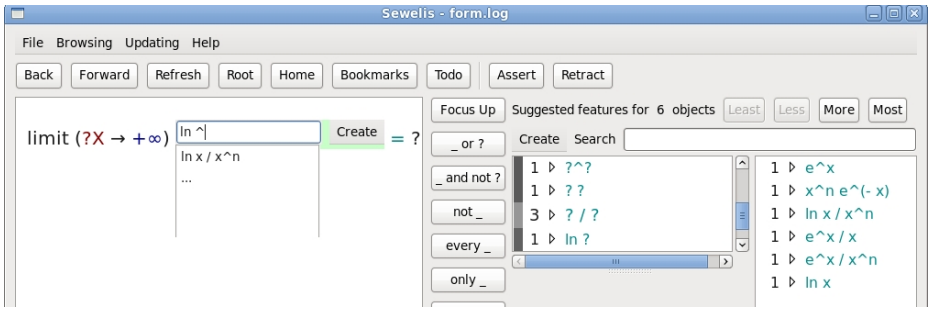


Fig. 3. A screenshot of SEWELIS where the current query retrieves the expressions whose limit at positive infinity is equal to something.

three navigation steps from the empty query: `...?...` (*contains...*), `...?^?...` (*a power...*), `...?^2...` (*of 2*). Figure 3 shows a complete screenshot of SEWELIS during the construction of a query. The query is at the left, and states that *the limit at positive infinity of something is equal to something*. The textfield marks the position of the focus, here on the body of the `limit` constructor. The middle column suggests the possible constructors at the focus, and the right column lists the possible expressions at the focus. The latter therefore contains the answers to the current query with respect to the current focus. Note that no blank node identifier is visible thanks to generated labels, even though all expressions *are* blank nodes. Suggestions can also be found and selected by auto-completion from the focus textfield. The next step for the user could be to select one of those expressions, and then to move the focus after the equal sign in order to discover the value of the limit for the chosen expression.

6 Related Work

We compare our approach first with other languages for representing expressions, and second with query languages for structural search among expressions. The latter are used for instance in proof assistants, such as Coq [3].

6.1 Representation Languages

The reference language for the representation of mathematical expressions is MathML [11], an XML dialect. In fact, MathML defines two languages: a *presentation* language, and a *content* language. Only the latter interests us because it represents the logical structure of expressions, and avoids ambiguity problems (e.g., the letter e that denotes either the Neperian constant or a variable) as well as synonymy problems (e.g., x/y and $\frac{x}{y}$ for division). The LaTeX language plays the same role as the presentation language of MathML, and therefore exhibits the same problems. However, presentation languages can be used for the labelling templates of expression constructors.

The (strict) content language of MathML is based on a small number of XML tags that encapsulate different types of expressions: `<cn>` (numbers), `<ci>` (identifiers), `<csymbol>` (symbols), `<cs>` (strings), `<apply>` (applications of functions and operators), `<bind>` and `<bvar>` (bindings). For example, the expression $\int x^2\,dx$ has the following MathML representation:

```
<bind><csymbol>integral</csymbol>
  <bvar><ci>x</ci></bvar>
  <apply><csymbol>power</csymbol> <ci>x</ci> <cn>2</cn> </apply>
</bind>
```

RDF expressions are expressive enough to represent all MathML contents. Numbers and strings are naturally mapped to RDF literals of different datatypes (e.g., `xsd:integer` for integers, `xsd:string` for strings). Symbols (e.g., functions, operators, constants) are naturally mapped to URIs, ideally defined in standard vocabularies. Identifiers are mapped to variables, i.e. blank nodes. Applications (of a function to arguments) are mapped to RDF expressions, where the constructor represents the applied function, and elements represent passed arguments. Finally, bindings are also mapped to expressions, where the binder (e.g., $\exists$, $\forall$, $\int$) is the constructor, and the bound variable is a distinguished argument that can only be filled with a variable. The Turtle+ representation of the above example is therefore `math:Integral(math:Power(_:x,2),_:x)`, where by convention, the second argument of `math:Integral` is the bound variable. An advantage of the RDF representation of expressions is its interoperability with a general-purpose knowledge representation language, RDF. Compared to MathML, this makes it possible to freely mix mathematical knowledge and non-mathematical knowledge by allowing RDF annotations on expressions and sub-expressions.

6.2 Query Languages

The approaches that consist in applying textual search methods by linearizing expressions [12] cannot correctly account for the nested structure of expressions, and for the issue of bound variables [16]. When looking for integrals in x whose body contains x^2, a textual search would have false positives such as $\int 2x\,dx = x^2 + c$ (x^2 is not in the scope of $\int$), and would have false negatives such as $\int y^2 - y\,dy$ (y instead of x). On the contrary, the approaches based on

a structured query language [1,3,6,13] correctly account for nested structures and bound variables by reasoning directly on the structure of expressions, and by using *wild cards* as place-holders for variables and sub-expressions. MathWebSearch [6] defines an XML query language that extends MathML. Its XML syntax makes it difficult to use, and its expressiveness is limited. For example, it cannot express the relation between an expression and its sub-expressions (e.g., `...x^2...` in SPARQL+). The query language QMT by Rabe [13] has much in common with RDF and SPARQL, despite having a different style. However, it is applied at the theory level of mathematical knowledge, and provides no special support for expression patterns even though it inherits expression representations from OpenMath objects. For example, the above query can be expressed in QMT as follows (XQuery-style): `for` e_1 `in obj(integral) and` e_2 `in subobj(`$\mathtt{arg}_1(e_1)$`,power) where` $\mathtt{arg}_1(e_2)=\mathtt{var}_1(e_1)$ `and` $\mathtt{arg}_2(e_2)=2$ `return` e_1. The query language that is most similar to our approach is from Altamimi and Youssef [1]. They use an ASCII notation that is similar to the LaTeX notation, and a set of 6 wild cards that can be used in place of: one or several characters, one or several atoms, one or several expressions. The above example is expressed by the query `\int ...$1^2... d$1`. Our approach has a higher expressiveness, and a number of additional advantages for users. SPARQL and SEWELIS have a higher expressiveness by offering disjunction, negation, and the possibility to search for sub-expressions appearing in some context. For example, it is possible to look for the bodies of integrals in x that contain y^2 or y^3, where y is not x: $\int$`...(not ?X)^(2 or 3)... d?X`, as displayed in SEWELIS. The 6 kinds of wild cards are covered by the combination of: the empty pattern `?` (a universal wild card), SPARQL variables for co-occurences of a same sub-expression, ellipsis `...` for reaching sub-expressions, and classical queries for constraining the name and type of the atoms of expressions. An additional advantage when using SEWELIS is that users do not need to master the syntax of the query language because they are guided step after step in the construction of queries. This comes with the guarantee of non-empty results. Another advantage is that pretty-printing (UTF-8 characters, mixfix notations, etc.) can be used for expressions and queries because query elements are selected, not written.

7 Conclusion

We have proposed an RDF design pattern for the representation of expressions as containers that is compatible with existing practice such as in OWL/RDF and SPIN, and that allows for the expressive structural querying of expressions based on their contents and context of occurence. With a simple syntactic extension of Turtle and SPARQL, those expressions can be noted in a concise and familiar way, i.e. using the functional notation. With labelling annotations on expression constructors, human-readable labels can be automatically generated for each expression. We have adapted an existing RDF tool, SEWELIS, to support the presentation and manipulation of such expressions. As SEWELIS is a

rather complex system, we are confident that similar adaptations can be applied to other RDF tools. We have also shown the benefits of using SEWELIS on the important task of mathematical search. We believe that the scope of our design pattern goes beyond mathematical expressions, as we have shown in this paper with OWL axioms and SPARQL queries, and that it is relevant to the representation of all kinds of structures and symbolic data in RDF.

References

1. Altamimi, M.E., Youssef, A.S.: A math query language with an expanded set of wildcards. Mathematics in Computer Science 2(2), 305–331 (2008)
2. Ferré, S., Hermann, A.: Reconciling faceted search and query languages for the Semantic Web. Int. J. Metadata, Semantics and Ontologies 7(1), 37–54 (2012)
3. Guidi, F., Schena, I.: A query language for a metadata framework about mathematical resources. In: Asperti, A., et al (eds.) Int. Conf. Mathematical Knowledge Management (MKM). pp. 105–118. LNCS 2594, Springer (2003)
4. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. Chapman & Hall/CRC (2009)
5. Kohlhase, M., Müller, C., Rabe, F.: Notations for living mathematical knowledge. In: Autexier, S., Campbell, J. (eds.) Intelligent Computer Mathematics (CICM). pp. 504–519. LNAI 5144, Springer (2008)
6. Kohlhase, M., Sucan, I.: A search engine for mathematical formulae. In: Calmet, J., Ida, T., Wang, D. (eds.) Int. Conf. Artificial Intelligence and Symbolic Computation. pp. 241–253. LNCS 4120, Springer (2006)
7. Lange, C.: Ontologies and languages for representing mathematical knowledge on the semantic web. Semantic Web 4(2), 119–158 (2013)
8. Mallea, A., Arenas, M., Hogan, A., Polleres, A.: On blank nodes. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N.F., Blomqvist, E. (eds.) Int. Semantic Web Conf. pp. 421–437. lncs 7031, Springer (2011)
9. Marchionini, G.: Exploratory search: from finding to understanding. Communications of the ACM 49(4), 41–46 (2006)
10. Marchiori, M.: The mathematical semantic web. In: Asperti, A., Buchberger, B., Davenport, J.H. (eds.) Mathematical Knowledge Management. pp. 216–224. LNCS 2594, Springer (2003)
11. MathML: Mathematical markup language 3.0 (2010), http://www.w3.org/TR/MathML3/, W3C Recommendation
12. Miner, R., Munavalli, R.: An approach to mathematical search through query formulation and data normalization. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) Calculemus/Mathematical Knowledge Management (MKM). pp. 342–355. LNCS 4573, Springer (2007)
13. Rabe, F.: A query language for formal mathematical libraries. In: Jeuring, J., Campbell, J.A. (eds.) Int. Conf. Intelligent Computer Mathematics. pp. 143–158. LNCS 7362, Springer (2012)
14. Robbins, A.: Semantic MathML (2009), http://straymindcough.blogspot.fr/2009/06/
15. SPIN - SPARQL syntax (2011), http://www.w3.org/Submission/2011/SUBM-spin-sparql-20110222/, W3C Member Submission
16. Youssef, A.: Roles of math search in mathematics. In: Borwein, J.M., Farmer, W.M. (eds.) Int. Conf. Mathematical Knowledge Management (MKM). pp. 2–16. LNCS 4108, Springer (2006)