An RDF Vocabulary for the Representation and Exploration of Expressions with an Illustration on Mathematical Search

Sébastien Ferré

An RDF Vocabulary for the Representation and Exploration of Expressions with an Illustration on Mathematical Search

Sébastien Ferré

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

Abstract. Complex expressions, as used in mathematics and logics, account for a large part of human knowledge. It is therefore desirable to allow for their representation in RDF and for their exploration through semantic search. We propose an RDF vocabulary that fulfills three objectives. The first objective is the accurate representation of expressions in standard RDF, so that expressive mathematical search is made possible. We here propose a syntactic extension of Turtle and SPARQL for the concise notation of such expressions. The second objective is the automated generation of expression labels that are close to usual mathematical notations (e.g., infix operators, symbols). The third objective is the compatibility with existing practice and legacy data in the Semantic Web community for the representation of expressions and structures (e.g., OWL/RDF, SPIN). We illustrate the use of this vocabulary on mathematical search using SEWELIS, a tool for the guided exploration and edition of RDF graphs, and discuss the benefits compared to state-of-the-art in mathematical search.

1 Introduction

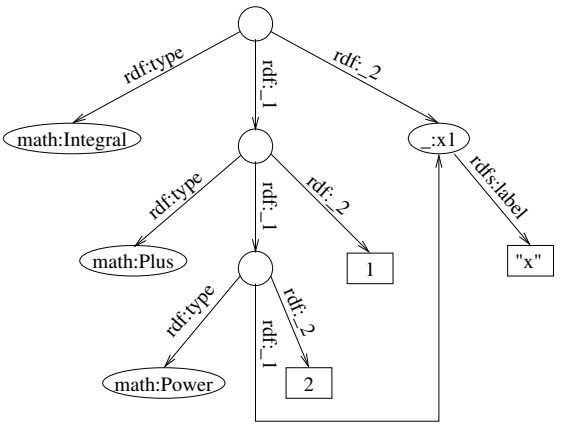
Complex expressions account for a large part of human knowledge. Common instances of expressions are mathematical equations, logical formulae, regular expressions, or parse trees of natural language sentences. In the domain of the Semantic Web [5], they can be OWL axioms [11], SWRL rules [16], or SPARQL queries [14]. It is therefore desirable to allow for their representation in RDF [12] 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 and logical 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 known as *mathematical search* [3,7,17,10,1]. For example, we may wish to retrieve all expressions that are an integral in some variable x and whose body contains the sub-expression x^2. Correct answers are $\int x^2 + 1\,dx$ and $\int y^2 - y\,dy$. This example exhibits two difficulties in mathematical 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 fact that the above variable x that is bound by the integral $\int\ dx$ can be renamed as y without changing the meaning of the expression.

The approaches that consist in applying textual search methods by linearizing expressions [10] cannot correctly account for the above difficulties [17]. In the above example, 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 [1,7,3] correctly account for the above difficulties by reasoning directly on the structure of expressions, and by using *jokers* as place-holders for variables and sub-expressions. In the language defined in [1], the query `\int ...$1^2... d$1` returns the correct answers to the above question. However, those query languages are limited to mathematical expressions, and are not interoperable with Semantic Web languages.

The need for representing expressions in RDF has already been felt in various situations, but to our knowledge, no generic solution has been proposed. A well-known example is the representation of complex classes in OWL ontologies [11]. To this purpose, the OWL vocabulary defines a number of "syntactic" classes (e.g., `owl:Restriction`) and properties (e.g., `owl:onProperty`). The fact that they are mixed with "semantic" classes (e.g., `owl:Class`) and properties (e.g., `owl:equivalentClass`) is a frequent cause for confusion among beginners, which indicates that it would be beneficial to separate the representation of complex expressions from assertions. Another example of a vocabulary that defines "syntactic" classes and properties is SPIN SPARQL Syntax [15], defined for the representation of SPARQL queries. OWL/RDF and SPIN follow the same principles, and therefore offer a good basis for generalization.

In this paper, we propose an RDF vocabulary for expressions that fulfills three objectives. The first objective (Section 2) is the accurate representation of expressions in standard RDF, so that expressive structured search (e.g., mathematical search) is made possible. We also propose a syntactic extension of Turtle and SPARQL syntaxes 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 mathematical notations (e.g., infix operators, symbols). This is important for Semantic Web 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 vocabulary with existing practice, and legacy data, in the Semantic Web community (e.g., OWL/RDF, SPIN). This implies that legacy data need not be changed in order to benefit from the advantages of our vocabulary, in particular the generation of labels. We illustrate (Section 5) those advantages on mathematical search using SEWELIS [2], a tool for the guided exploration and edition of RDF graphs, and show (Section 6) that this approach is competitive w.r.t. expressiveness, displays more natural representations of expressions, and offers the guided construction of queries.




```
[ 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).

2 Representation of Expressions in RDF

An RDF dataset is a graph whose nodes are resources (URIs, blank nodes, and literals), and whose edges are triples (s, p, o). In a triple (s, p, o), p is an RDF property that plays the role of predicate, and states a relationship from the subject s to the object o. In order to represent mathematical expressions in RDF, we rely on the fact that every expression can be represented as an abstract 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, quantifiers). Symbols naturally map to URIs, and values map to RDF literals. To account for the fact 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, whose label is the variable name. The same choice has been made in SPIN. It remains to define the representation of compound expressions. A compound expression is completely defined by the node label, which we call *constructor*, and the sequence of sub-expressions, which we call *arguments*. We propose to represent a compound expression as an RDF container with the constructor as a class in place of `rdf:Seq`, `rdf:Bag`, or `rdf:Alt`. In this way, each constructor defines a subclass of `rdfs:Container`. Assuming the namespace `expr:` for our expression vocabulary, we introduce `expr:Constructor` as the class of constructors, i.e. the class of container types. This 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. Section 4 explains how to reconcile the two approaches in practice.

Figure 1 shows the graphical and Turtle forms of the RDF representation of the expression $\int x^2 + 1\, dx$. This expression includes mathematical operators

notation	definition
C(`(`)E_1,...,E_n(`)`)	`[ a` C`; rdf:_1` E_1`; ...; rdf:_`n E_n `]`
...E...	`[ rdfs:member*` E `]`
..E..	`[ rdfs:member+` E `]`
`is [` PO_1`; ...;` PO_n `]`	PO_1`; ...;` PO_n

Table 1. New notations for Turtle and/or SPARQL, and their definition.

`math:Integral`[1] (integral), `math:Plus` (addition) and `math:Power` (power), which 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 includes the integer literals `1` and `2`, as well as a blank node `_:x1`, labelled `"x"`, to represent the variable x. This representation correctly handles all the contents of the expression while abstracting over possible variations in the presentation (e.g., adding brackets, varying the notation for the integral). 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`.

As this notation is rather verbose, we propose to extend the syntax of Turtle and SPARQL with a few notations in order to allow for more concise descriptions and queries (see Table 1). The first one is a functional notation for containers, and hence for expressions, where the container type plays the role of the function, and container elements play the role of arguments. The second and third one 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 (and collections) can be used. The last notation allows blank nodes, and hence the functional notation and the ellipsis notation, 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 rewritten as follows.

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

In order to test the validity of our representation, we discuss the formulation of SPARQL 1.1 queries [14] for a few representative examples of semantic search. SPARQL variables are 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 variable: 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.

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

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

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 (the same) x. As a conclusion, SPARQL 1.1 provides enough expressivity to cover the needs of mathematical search, which is not surprising considering that expressions are standard RDF graphs. A comparison with existing query languages for mathematical search is given in Section 6.2.

3 Generation of Labels for Expressions

As the previous section shows, expressions make a heavy use of blank nodes. Conversely, most blank nodes in existing RDF graphs can be seen as expressions: e.g., complex classes in OWL/RDF, SPARQL syntax in SPIN, structures for representing coordinates, intervals, measures. A difficulty with blank nodes is that they are notoriously difficult to present in query results, and in Semantic Web tools in general [8]. Literals have a direct representation, and URIs are generally given a label. In fact, we see blank nodes as compound literals, whose contents is defined by the tree of triples that start from the blank node, and ends at URIs and literals. Like literals, blank nodes have no identity, which means that two structurally equivalent blank nodes are interchangeable. In this view, it is correct to represent a blank node by its contents, so that blank node identifiers are completely avoided, except for circular structures, which have been shown to be rare in practice [8]. Turtle provides a generic notation for blank node contents (using square brackets `[...]`), and a custom notation for collections (using round brackets `(...)`). Turtle+ additionally provides a concise notation for containers and expressions, using the functional notation (see Table 1).

While the Turtle+ and SPARQL+ notations are much more concise and readable than sets of triples full of blank node identifiers, they are still far from the usual mathematical notations for expressions. For recall, the expression $\int x^2 + 1\,dx$ is represented in Turtle+ as

`math:Integral(math:Plus(math:Power(_:x1,2),1),_:x1)`. Note the necessary blank node identifier for the variable because of a cycle in the RDF graph. In case expressions have to be parsed from files or user input, formal representations are necessary to allow machine-understanding, which is one of the main objectives of Semantic Web languages. However, in the case where expressions are only displayed to users, or can be manipulated through a point-and-click user interface, less formal representations become possible.

We here propose a vocabulary to express annotations on constructors that can be used by Semantic Web tools to generate natural representations of expressions. 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 can simply be generated dynamically by the tool, on the need. 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. Because those labels are only for display, all Unicode characters can be used, including mathematical symbols (e.g., $\int$ 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 such as $x + 1$, prefix notation such as $sin\ x$, postfix notations such as $n!$, and mixfix notations such as $\int x^2\, dx$. However, those notations can lead to ambiguities, and brackets must be inserted according to priority levels 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 humans, and adding superfluous brackets would make the expression label less readable for humans.

We propose to describe constructors with all the necessary information to generate natural labels in a generic way, i.e. for all kinds of notations. The necessary information comprises the *template*, the *priority level* of the constructor, and for each argument, the *expected priority level* plus additional information depending on the *kind* of the argument (see below). 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"`. If arguments are placed in order, the generic marker `_` can be used instead: e.g., `"_ + _"`. Alternately, the templace could be defined as an XML literal instead of a string, e.g., using the MathML [9] presentation language for rendering in a browser. Priority levels are often defined as integers. However, for interoperability and readability, we propose to define them as URIs that belong to the class `expr:PriorityLevel`. We introduce two predefined priority levels in our vocabulary: `expr:atomLevel` as the highest priority level, and `expr:topLevel` as the lowest priority level. Priority levels can be ordered through the property `expr:hasPriorityOver`: e.g., the triple

`expr:atomLevel expr:hasPriorityOver expr:topLevel` holds. For common mathematics, we assume the following priority levels, ordered by increasing priority (partial): `expr:topLevel`, `math:equalLevel`, `math:plusLevel`, `math:timesLevel`, `math:powerLevel`, `expr:atomLevel`. Each constructor argument is described by an expected priority level. For example, because the addition is a left associative operator with priority level `math:plusLevel`, its left argument has `math:plusLevel` as expected priority, and its right argument has `math:timesLevel` as expected priority. An expression inherits the priority level of its constructor. Then, when the priority level of an argument is not higher than the expected priority level (e.g., `math:plusLevel` instead of `math:timesLevel`), brackets are added around the label of the argument before insertion into the template of the constructor. By default, round brackets are used, but a template with one place holder (e.g., `"{_}"` for curly brackets) can be associated to a priority level through the property `expr:brackets`. Finally, some constructors expect a variable number of sub-expressions. For example, the standard RDF constructors (e.g., `rdf:Seq(1,2,3)`) can have an arbitrary number of arguments; and we can imagine a constructor `math:Average` that takes an RDF collection as an argument (e.g., `math:Average((3 7 5))`). In those cases, we need to specify a separator between sub-expressions, and the expected priority level applies to collection elements.

We once more demonstrate the reflexive capabilities of Semantic Web languages by using RDF expressions to represent the description of notations for constructors! Indeed, a notation description is the association of several pieces of information that do not make sense in isolation. Instead of a tedious definition of the constructors involved in those notation definitions, we resort to a number of self-explanatory examples (comments below).

```
math:Plus expr:notation expr:Notation("_ + _",math:plusLevel,
      rdf:Seq(expr:One(math:plusLevel),expr:One(math:timesLevel))) .
math:Times expr:notation expr:Notation("_ _",math:timesLevel,
      rdf:Seq(expr:One(math:timesLevel),expr:One(math:powerLevel))) .
math:Power expr:notation expr:Notation("_^_",math:powerLevel,
       rdf:Seq(expr:One(expr:atomLevel),expr:One(math:powerLevel))) .
math:Sin expr:notation expr:Notation("sin _",math:plusLevel,
       rdf:Seq(expr:One(math:timesLevel))) .
math:Fact expr:notation expr:Notation("_!",math:powerLevel,
       rdf:Seq(expr:One(expr:atomLevel))) .
math:Integral expr:notation expr:Notation("∫ _ d_",expr:atomLevel,
       rdf:Seq(expr:One(math:plusLevel),expr:One(expr:atomLevel)) .
math:Average expr:notation expr:Notation("avg(_)",expr:atomLevel,
       rdf:Seq(expr:Collection(",",math:plusLevel))) .
rdf:Bag expr:notation expr:Notation("{_}",expr:atomLevel,
       rdf:Seq(expr:Many(", ",math:plusLevel))) .
```

Constructors are linked to notations through the property `expr:notation`. This allows for the definition of several notations for a same constructor, differing the choice to tools and users. The constructor `expr:Notation` takes as arguments the template, the priority level, and a sequence of argument descriptions. There

are three kinds of arguments: one expression, many expressions, a collection of expressions. Examples of generated labels from those definitions are: "∫ x^2 + 1 dx", "(a + b)^2 = a^2 + 2 a b + b^2", "avg(3,7,5)", "{1, 3, 3, 8}".

4 Compatibility with Legacy RDF Structures

Blank nodes and "syntactic" classes and 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. the blank node [a owl:Restriction ; owl:onProperty r ; owl:someValuesFrom C]. A universal restriction $\forall r.C$ is represented similarly, using the property owl:allValuesFrom instead of owl:someValuesFrom. 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.

OWL restrictions could equally well be represented as expressions, using our approach. Assuming the two constructors owl:Some and owl:All in the OWL namespace, an existential restriction $\exists r.C$ would be represented as owl:Some(r,C), and a universal restriction $\forall r.C$ would be represented as owl:All(r,C). Those representations are close to the OWL functional syntax[2], and are valid notations in Turtle+. A first advantage of expressions 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 "semantic" properties (e.g., owl:equivalentClass) and "syntactic" properties (e.g., owl:onProperty). In our expressions, the latter are only 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 a name instead of a position.

In order to fully reconcile legacy data and the naming of arguments with the systematic generation of natural labels for expressions, we introduce *implicit constructors*. An implicit constructor does not occur explicitly in the RDF representation of expressions and structures, but it is mapped to a combination of syntactic classes and properties, and it serves as a handle for the annotations about the generation of labels. We introduce the class expr:ImplicitConstructor, a subclass of expr:Constructor, to identify the set of implicit constructors. Then, assuming an implicit constructor *Cons*, the expression $Cons(E_1, \ldots, E_n)$ is mapped to the blank node [a C ; P_1 E_1 ; ... ; P_n E_n], where C is the *implicit class* of *Cons* (optional), and $(P_1, \ldots, P_n)$ is the sequence of *implicit properties* of *Cons*. We introduce the property expr:implicitClass to link an implicit constructor to its implicit class; and we introduce the property expr:implicitProperties to link an implicit constructor to the RDF sequence of implicit properties. For example,

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

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:Not(owl:Some(ex:hasTopping,ex:MeatTopping))`
	`  owl:Not(owl:Some(ex:hasTopping,ex:FishTopping)) ) )`

4	`"Pizza and not has topping some Meat and not has topping some Fish"`

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

`owl:Some` can be defined as an implicit constructor with the following Turtle+ statement.

```
owl:Some a expr:ImplicitConstructor ;
         expr:implicitClass owl:Restriction ;
         expr:implicitProperties
             rdf:Seq(owl:onProperty,owl:someValuesFrom) .
```

From there, it is easy to get any of the three main syntaxes for OWL restrictions as generated labels. For the functional syntax, it is enough to define a label on constructor `owl:Some`: e.g., `"some(child,Doctor)"`. For the Manchester syntax[3], `owl:Some` has to be defined as a right-associative infix operator, like the power operator: e.g., `"child some Doctor"`. For the DL syntax, it has instead to be defined as a mixfix operator with template `"∃_._"`: e.g., `"∃child.Doctor"`.

Tables 2, 3, and 4 compare different notations of three complex expressions in three different languages: OWL, SPARQL, and ingredient descriptions (the latter example is adapted from [5], p. 42). In each table, the first line is the native syntax of the language (Manchester syntax for OWL, English for ingredient descriptions). 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 assuming that the "syntactic" classes are used as constructors, like for `owl:Some` above. The fourth line is the generated label assuming that implicit constructors have been defined, and that appropriate 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 "`, `owl:Not` is defined as a prefix operator, and `owl:Some` is defined as a right associative infix operator. Those constructors

[3] `http://www.co-ode.org/resources/reference/manchester_syntax/`

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) }"`

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

are given increasing priority. For SPIN SPARQL queries, `sp:Select` has two collection arguments whose separators are the space, `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. For the ingredient description, `ex:Ingredient` uses the template `"_2 of _1"` to reverse the order of arguments, and `ex:Measure` simply concatenates the value and the unit.

5 Application to Mathematical Search in SEWELIS

We illustrate the use of RDF expressions and their generated labels on mathematical search. In Section 2, we have shown our representation of expressions allows for the expressive search of expressions and sub-expressions, using SPARQL as the query language. However, writing queries has a number of difficulties for end-users: (1) syntax errors (*what is the grammar of the query language?*), (2) vocabulary errors (*which are the available URIs, classes and properties?*), and (3) lack of control on the amount of results (too few or too many). Another difficulty is that it is difficult to conciliate a non-ambiguous syntax, and natural notations. Our extension of Turtle and SPARQL with the functional notation for expressions is an improvement over explicit blank nodes, but it is still far from usual mathematical notations. Ideally, we would like the syntax of generated

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"`

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

labels for expressions, as presented in Section 3, but this is incompatible with the hand-writing and machine-parsing of queries.

SEWELIS[4] is a Semantic Web tool for the exploration and edition of RDF graphs that reconciles the expressivity of query languages, and the guided exploration of faceted search [4,13], a.k.a. Query-based Faceted Search (QFS) [2]. This means that users can reach complex queries (involving property paths and cycles, disjunctions, and negations) without writing anything. Starting with the empty query, users iteratively refine it by selecting classes, properties, resources, and literals among suggestions given by SEWELIS. Unlike syntactic editors (e.g., SemanticCrystal [6], the SCRIBO Graphical Editor[5]), at each step of the interactive construction, a valid query is formed, and SEWELIS returns its results. A key notion is the *focus* that determines the part of the query to be refined, and the query variable to be used for results. Only and all relevant suggestions are given so that the exploration is proven both *safe* (no dead-end, no empty result) and *complete* (every safe query is reachable). Because of those strong properties, writing queries becomes completely unnecesary, and it then becomes possible to use any notation is prefered for queries.

We have integrated into SEWELIS the expression vocabulary for the representation of expressions in descriptions and queries. The pretty-printing of queries is based on the same principles as for the generation of labels, extended to expression/graph patterns. 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, and indicates which part of the query answers are to be displayed.

- $\int$ `?X^2 d?X`: the integrals in x of x^2;
- $\int$ `...?X^2... d?X`: the integrals in x whose body contains x^2 as a sub-expression;
- $\int$ `...?X^2...` `d?X`: the bodies of the integrals in x that contains x^2 as a sub-expression.

[4] Visit SEWELIS' page at `http://www.irisa.fr/LIS/softwares/sewelis/`.
[5] `http://www.scribo.ws/xwiki/bin/view/Blog/SparqlGraphicalEditor`

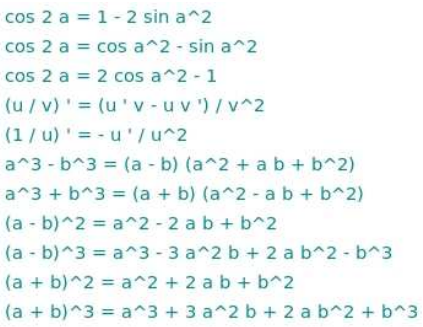


Fig. 2. The list of formulas containing a square, as displayed in SEWELIS.

We have applied SEWELIS on the exploration of a collection of 70 formulas taken from an official document[6] used for the French baccalaureate. 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 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 of the current query with respect to the focus. Note that no blank node identifier is displayed 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 mathematical expressions, and second with query languages for searching mathematical expressions. The later are used for instance in proof assistants, such as Coq [3]. To the best of our knowledge, no existing approach allows for the guided exploration of such expressions.

6.1 Representation Languages

The reference language for the representation of mathematical expressions is MathML [9], an XML dialect. In fact, MathML defines two languages: a *pre-*

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

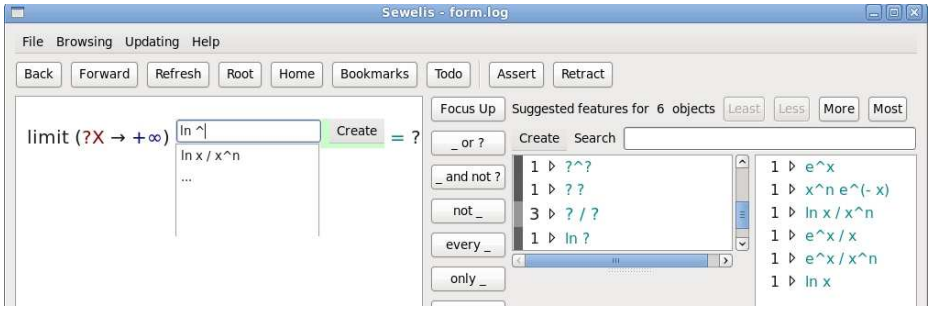


Fig. 3. A screenshot of SEWELIS where the current query retrieves the expressions whose limit at positive infinity is equal to something.

sentation 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.

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 generalist 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

Letting aside the approaches based on textual search, whose limits have already been exposed in the introduction, we find query languages that directly operate on the logical structure of expressions. MathWebSearch [7] defines an XML query language that extends MathML. Its XML syntax makes it very 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 SEWELIS).

The most advanced query language is from Altamimi and Youssef [1]. They use an ASCII notation that is similar to LaTeXnotation, and a set of 6 jokers that can be used in place of: one or several characters, one or several atoms, one or several expressions. Our approach has a higher expressiveness, and a number of additional advantages for users. SPARQL has 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 jokers are covered by the combination of: the empty pattern `?` (a universal joker), 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 representation of expressions as containers that is compatible with existing practice such as in OWL/RDF and SPIN, and that allows for the expressive search 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. in the functional notation. By annotating expression constructors with notation descriptions, which are expressions themselves, human-readable labels can be automatically generated for each expression. We have illustrated the representation and exploration of expressions on mathematical search, and we have shown the benefits of our approach compared to existing mathematical query languages. We hope that the expression vocabulary that we have sketched in this paper will be improved by the community, and adopted by Semantic Web tools. We believe that its scope goes beyond mathematical expressions, as we have shown in this paper, and that it is relevant to the representation of all kinds of structures and symbolic data.

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.: Semantic search: Reconciling expressive querying and exploratory search. In: Aroyo, L., Welty, C. (eds.) Int. Semantic Web Conf. pp. 177–192. LNCS 7031, Springer (2011)
3. Guidi, F., Schena, I.: A query language for a metadata framework about mathematical resources. In: Asperti, A., Buchberger, B., Davenport, J.H. (eds.) Int. Conf. Mathematical Knowledge Management (MKM). pp. 105–118. LNCS 2594, Springer (2003)
4. Hearst, M., Elliott, A., English, J., Sinha, R., Swearingen, K., Yee, K.P.: Finding the flow in web site search. Communications of the ACM 45(9), 42–49 (2002)
5. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. Chapman & Hall/CRC (2009)
6. Kaufmann, E., Bernstein, A.: Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases. J. Web Semantics 8(4), 377–393 (2010)
7. 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)
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. MathML: Mathematical markup language - version 3.0 (2010), http://www.w3.org/TR/MathML3/, `http://www.w3.org/TR/MathML3/`, W3C Recommendation
10. 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)
11. OWL 2 web ontology language (2009), http://www.w3.org/TR/owl2-overview/, `http://www.w3.org/TR/owl2-overview/`, W3C Recommendation
12. RDF primer (2004), http://www.w3.org/TR/2004/REC-rdf-primer-20040210/, `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`, W3C Recommendation
13. Sacco, G.M., Tzitzikas, Y. (eds.): Dynamic taxonomies and faceted search. The information retrieval series, Springer (2009)
14. SPARQL 1.1 query language (2012), http://www.w3.org/TR/sparql11-query/, `http://www.w3.org/TR/sparql11-query/`, W3C Proposed Recommendation
15. SPIN - SPARQL syntax (2011), http://www.w3.org/Submission/2011/SUBM-spin-sparql-20110222/, `http://www.w3.org/Submission/2011/SUBM-spin-sparql-20110222/`, W3C Member Submission
16. SWRL: A semantic web rule language - combining OWL and RuleML (2004), http://www.w3.org/Submission/SWRL/, `http://www.w3.org/Submission/SWRL/`, W3C Member Submission
17. 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)