Bridging the Gap Between Formal Languages and Natural Languages with Zippers

Sébastien Ferré[✉]

IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes, France
`ferre@irisa.fr`

Abstract. The Semantic Web is founded on a number of Formal Languages (FL) whose benefits are precision, lack of ambiguity, and ability to automate reasoning tasks such as inference or query answering. This however poses the challenge of mediation between machines and users because the latter generally prefer Natural Languages (NL) for accessing and authoring knowledge. In this paper, we introduce the `N<A>F` design pattern based on Abstract Syntax Trees (AST), Huet's zippers and Montague grammars to zip together a natural language and a formal language. Unlike question answering, translation does not go from NL to FL, but as symbol `N<A>F` suggests, from ASTs (`A`) of an intermediate language to both NL (`N<A`) and FL (`A>F`). ASTs are built interactively and incrementally through a user-machine dialog where the user only sees NL, and the machine only sees FL.

1 Introduction

The Semantic Web is founded on a number of formal languages to represent and reason on facts (RDF), logical axioms (OWL), or queries (SPARQL). Those languages make data processable by machines but they also constitute a language barrier to the production and consumption of semantic data by end users. An important issue in the Semantic Web is to bridge the gap between formal languages (FL) designed for the machines, and natural languages (NL) understood by humans. A crucial aspect of this issue is the *adequacy* between the expressivity of FL and NL. Users should be guided to NL expressions that have a FL counterpart to avoid the *habitability* problem [11]; and all or most of the *expressivity* of the FL should be expressible through NL if we are to avoid two-class citizenship among users.

We propose the `N<A>F` design pattern to bridge the gap between NL and FL, along with techniques to ease its implementations. As a design pattern, it is not a finished design but a reusable template. We have successfully used it to deal with the NL-FL gap in different tasks: querying SPARQL endpoints [5], authoring RDF descriptions [8], and completing OWL ontologies [7]. However, in those previous works, the design pattern was mixed with other aspects, and only presented through its concrete applications. We here give a detailed and

This research is supported by ANR project IDFRAud (ANR-14-CE28-0012-02).

independent account of the `N<A>F` design pattern in order to facilitate its reuse in other contexts.

The `N<A>F` design pattern drastically differs from Question Answering (QA) approaches [13] in two ways. First, translation does not go from NL to FL *through* intermediate representations, but as symbol `N<A>F` suggests, *from* an intermediate language (`A`) to *both* NL (`N<A`) *and* FL (`A>F`). Second, questions are not written but *interactively and incrementally built* through a user-machine dialogue. That approach provides a number of added values compared to existing approaches (see Sect. 6 for details). Compared to QA, it ensures a strong reliability because it has no habitability problem, and it allows for higher expressivity. Compared to Controlled Natural Languages (CNL) with auto-completion [10], it offers similar expressivity with more flexibility in query construction, and more semantic and more fine-grained guidance. Compared to graphical query builders [11], it hides FL behind NL, and also offers more semantic and more fine-grained guidance. In truth, those added values do not guarantee user preference, a *subjective* value, but they are *objective* values along which different approaches can be characterized and compared precisely. Still, user preference is our ultimate goal, and this is why we develop and maintain a number of tools, one of which (Sparklis [5]) has already attracted a large number of users[1]. The design pattern is computationally lightweight, and can adapt to all sorts of FL and tasks. Its main limitation is that it cannot be used to process NL expressions that were not edited through it, typically existing texts.

The paper is structured as follows. Section 2 presents the theoretical foundations: Abstract Syntax Trees (AST), Huet's zippers, and Montague grammars. Section 3 presents an overview of our design pattern, and Sect. 4 illustrates it in detail to SPARQL-based querying. Section 5 demonstrates the generality of our design pattern by reporting on three different applications. Finally, Sect. 6 discusses related work, and Sect. 7 concludes.

2 Theoretical Foundations

2.1 Abstract Syntax Trees

Abstract syntax is a way to describe the structure of the sentences of a language, while abstracting over their concrete representation. Abstract syntax is generally not represented as a text but as a tree data structure called Abstract Syntax Tree (AST). In the compilation of programming languages, ASTs are commonly used as an intermediate representation between the source code (a text) and the target code (a binary). They here play the same role between NL and FL.

ASTs are best defined with algebraic datatypes that allow to recursively compose base types[2]. For example, assuming base type *term* for SPARQL variables and RDF nodes, the following datatypes define a simplified subset of SPARQL

[1] http://www.semantic-web-journal.net/system/files/swj1236.pdf.
[2] In OO programming, algebraic datatypes are modelled with the Composite pattern.

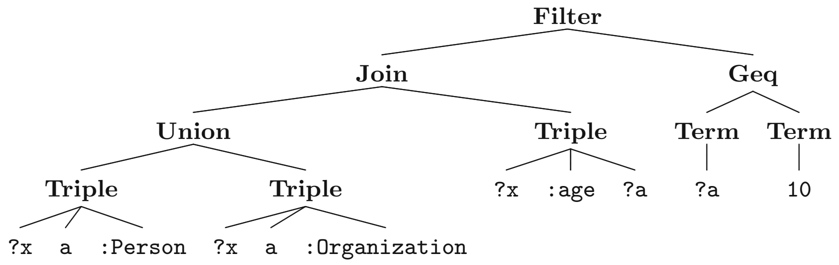

Fig. 1. AST graphical representation of the SPARQL graph pattern `{ ?x a :Person } UNION { ?x a :Organization } ?x :age ?a FILTER (?a >= 10)`

graph patterns (gp), and expressions ($expr$):

$$\begin{aligned} gp \;&::= \mathbf{Triple}(term, term, term) \mid \mathbf{Filter}(gp, expr) \\ &\mid\; \mathbf{Join}(gp, gp) \mid \mathbf{Union}(gp, gp) \mid \mathbf{Optional}(gp, gp) \\ expr \;&::= \mathbf{Term}(term) \mid \mathbf{Geq}(expr, expr) \mid \mathbf{Regex}(expr, expr) \end{aligned}$$

Each italic name is a datatype, and each boldface name is a *constructor* (AST node label). Every datatype may have any number of alternative constructors, and every constructor may have any number of arguments (including 0 for constant constructors). Figure 1 shows a graphical representation of the AST of a SPARQL graph pattern.

2.2 Huet's Zippers

In his "Functional Pearl" [9], Huet introduced the *zipper* as a technique for traversing and updating a data structure in a purely functional way, and yet in an efficient way. Purely functional programming completely avoids modification in place of data structures, and makes it much easier to reason on program behaviour, and hence to ensure their correctness [17]. We here use zippers for the incremental construction of ASTs. A simple and illustrative example is on simply chained lists, their traversal, and the insertion of elements. Given a base type *elt* for list elements, the *list* datatype is defined with two constructors: one for the empty list, one for adding an element at the head of another list.

$$list ::= \mathbf{Nil} \mid \mathbf{Cons}(elt, list)$$

The AST of list $[1, 2, 3]$ is $\mathbf{Cons}(1, \mathbf{Cons}(2, \mathbf{Cons}(3, \mathbf{Nil})))$. The zipper idea is to keep a location in the list such that it is easy and efficient to insert an additional element at that location, and also to move that location to the left or to the right. A location (e.g. at element 2 in the above list) splits the data structure in two parts: the *sub-structure* at the location ($[2, 3]$), and the surrounding *context* ($[1, \_]$). It has been shown that the context datatype corresponds to a data structure with one hole, and can be seen as the *derivative* of the structure

$$\begin{array}{lll r}
s & \to & np\ vp & (np\ vp) \\
vp & \to & vt\ np & \lambda x.(np\ \lambda y.(vt\ x\ y)) \\
np & \to & e & \lambda d.(d\ e) \\
 & \mid & det\ nn & \lambda d.(det\ nn\ d) \\
det & \to & \mathbf{a(n)} & \lambda d_1.\lambda d_2.(\exists X.((d_1\ X) \wedge (d_2\ X))) \\
 & \mid & \mathbf{every} & \lambda d_1.\lambda d_2.(\forall X.((d_1\ X) \Rightarrow (d_2\ X))) \\
e & \to & \mathbf{John} \mid \ldots \mid \mathbf{Mary} & \mathbf{Mary} \\
nn & \to & \mathbf{man} \mid \ldots \mid \mathbf{woman} & \lambda x.(\mathbf{woman}(x)) \\
vt & \to & \ldots \mid \mathbf{loves} & \lambda x.\lambda y.(\mathbf{loves}(x, y))
\end{array}$$

Fig. 2. Montague grammar of a small fragment of English

datatype [1]. We therefore name *list′* the context datatype for lists, and define it as follows.

$$list' ::= \mathbf{Root} \mid \mathbf{Cons}'(elt, list')$$

That definition says that a list occurs either as a root list or as the right-argument of constructor **Cons**, which has in turn its own context. For example, the context at location 3 of list $[1, 2, 3]$ is $\mathbf{Cons}'(2, \mathbf{Cons}'(1, \mathbf{Root}))$. In fact, a list context is the reverse list of elements before the location. Finally, a zipper data structure combines a structure and a context: *zipper* ::=$\mathbf{List}(list, list')$. A zipper contains all the information of a data structure plus a location in that structure. That location is also called “focus”.

A zipper makes it easy to move the location to neighbour locations, and to apply local transformations such as insertions or deletions. For example, to insert an element x in a list zipper $\mathbf{List}(l, l')$ is as simple as returning the zipper $\mathbf{List}(\mathbf{Cons}(x, l), l')$. Given a zipper $\mathbf{List}(l, \mathbf{Cons}'(e, l'))$, the location can be moved to the left by returning the zipper $\mathbf{List}(\mathbf{Cons}(e, l), l')$.

2.3 Montague Grammars

Montague grammars [4] are an approach to translation from NL to FL that is based on context-free grammars and simply typed λ-calculus. In a Montague grammar, each rule is decorated by a λ-expression that denotes the FL semantics of the syntactic construct defined by the rule. We can use them for translation from ASTs to FL because our ASTs have a syntactic structure close to NL.

Figure 2 shows an example of Montague grammar for a small fragment of English, and its semantics in first-order logic. That fragment covers sentences like `'John loves Mary'`, and `'every man loves a woman'`. The semantics is defined in a fully compositional style, i.e., the semantics of a construct is always a composition of the semantics of sub-constructs. The first rule in Fig. 2 says that a sentence (s) whose syntax is made of a noun phrase (np) followed by a verb phrase (vp) has its semantics defined by the λ-application ($np\ vp$) of the semantics of the noun phrase to the semantics of the verb phrase. The semantics of a sentence is a logical formula, e.g. $\forall X.(\mathbf{man}(X) \Rightarrow \exists Y.(\mathbf{woman}(Y) \wedge \mathbf{loves}(X, Y)))$ for the sentence `'every man loves a woman'`[3]. The semantics of a verb phrase is a formula λ-abstracted over an entity x ($\lambda x.$) because a verb phrase misses an entity e to form a complete sentence. If we call such an abstraction a *description*, then the semantics of a noun phrase is a formula λ-abstracted over a description d because a noun phrase misses a verb phrase vp to form a complete sentence. A noun nn has the same semantic type as a verb phrase, i.e. a formula missing an entity. The semantics of a transitive verb vt is a formula missing two entities, a subject x and an object y. The semantics of a determiner det is a quantifier introducing a fresh logical variable X, i.e. a formula abstracted over two descriptions.

Given a sentence, its semantics is obtained by the bottom-up composition of λ-terms from the grammar through the parse tree. For example, from the sentence `'every man loves a woman'` whose parse tree is '$[_s[_{np}[_{det}\texttt{every}]\ [_{nn}\texttt{man}]]\ [_{vp}[_{vt}\texttt{loves}]\ [_{np}[_{det}\texttt{a}]\ [_{nn}\texttt{woman}]]]]$', we obtain the λ-term

$$\begin{array}{l}((\lambda d_1.\lambda d_2.(\forall X.((d_1\ X) \Rightarrow (d_2\ X)))\ \lambda x.(\mathbf{man}(x)))\\ \quad \lambda x.((\lambda d_1.\lambda d_2.(\exists Y.((d_1\ Y) \wedge (d_2\ Y)))\ \lambda x.(\mathbf{woman}(x)))\\ \qquad \lambda y.(\lambda x.\lambda y.(\mathbf{loves}(x,y))\ x\ y)))\end{array}$$

which simplifies by β-reduction to the expected formula $\forall X.(\mathbf{man}(X) \Rightarrow \exists Y.(\mathbf{woman}(Y) \wedge \mathbf{loves}(X,Y)))$.

Although initially designed for translating NL to formal logic, Montague grammars as a technique can be used with other kinds of FLs as a target: e.g., SPARQL [6]. Their main benefits are to bridge the gap from natural to formal languages, and to offer a fully-compositional semantics. Fully-compositional semantics, like purely-functional data structures, makes it easier to ensure the correctness of the translation because there are no side-effects.

3 Overview of the Design Pattern

The principles of the `N<A>F` design pattern are schematized as a "suspended bridge over the NL-FL gap" in Fig. 3. The central pillar is made of *AST zippers*, i.e. AST tree structures with a focus on one AST node. The nature of the intermediate language abstracted by the ASTs depends on the application: e.g., queries, descriptions, logical axioms. The AST zipper is initialized by the system, and modified by users applying structural *transformations*, never by direct textual input. For that reason, it is important to design a *complete* set of transformations, so that every AST is reachable through a finite sequence of transformations. Conversely, only *safe* transformations should be suggested to users, so as to avoid syntactic and semantic errors. In the case of querying, a semantic error could be applying a transformation that leads to an empty result. In the case of ontology construction, it could be constructing an axiom that leads to inconsistency.

[3] We use capital letters for logical variables to distinguish them from λ-variables.

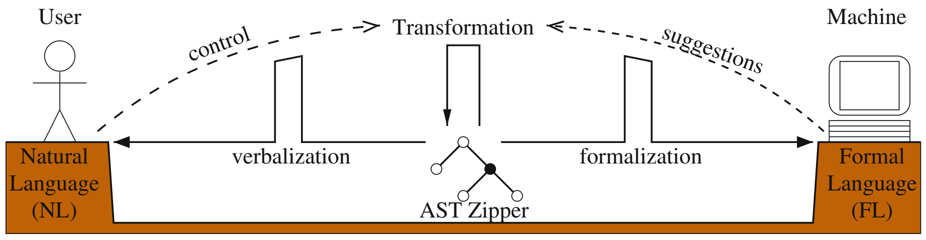

Fig. 3. Principle of zipper-based edition for bridging the gap between NL and FL

In order for the AST structure to be understood by both the user and the machine, *verbalization* translates ASTs to NL, and *formalization* translates ASTs to FL. In addition to those translations, verbalization supports user control by showing suggested transformations in NL, and formalization supports the computation of suggestions by taking into account the semantics of the AST. A key issue in the design of ASTs is to make the two translations semantically transparent, and simple enough. First, the AST structure should reproduce the syntactic structure of NL (e.g., sentences, noun phrases, verb phrases), while abstracting over as many details as possible. Indeed, starting with a flat representation like SPARQL, it is possible to produce a NL version [15], but it is difficult to make it stable across transformations. Second, the AST structure should semantically align with the target FL. Indeed, every AST that can be obtained by a sequence of transformations must have a semantics that is expressible in FL.

4 Application to a Core RDF Query Language

In this section, we explain in detail the application of the `N<A>F` design pattern to a core RDF query language (CRQL for short)[4]. CRQL covers a significant fragment of SPARQL 1.0 SELECT queries (i.e., triple patterns, UNION, simple filters) but restricted to tree patterns, and extended with negation (NOT EXISTS). Section 5 discusses variations around CRQL to deal with more expressive querying and other tasks.

4.1 AST Zippers

The following datatype definitions describe the structure of CRQL ASTs. Given base types for RDF nodes (*node*), RDFS classes (*class*), RDFS properties (*prop*), and literals (*lit*), we define abstract sentences (*s*), abstract noun phrases (*np*), and abstract verb phrases (*vp*).

$$\begin{aligned}
s &::= \textbf{Select}(np)\\
np &::= \textbf{Something} \mid \textbf{Some}(class) \mid \textbf{Node}(node) \mid \textbf{That}(np, vp)\\
vp &::= \textbf{IsA}(class) \mid \textbf{Has}(prop, np) \mid \textbf{IsOf}(prop, np) \mid \textbf{Geq}(lit)\\
&\quad\mid \textbf{True} \mid \textbf{And}(vp, vp) \mid \textbf{Or}(vp, vp) \mid \textbf{Not}(vp)
\end{aligned}$$

[4] We also provide online the source code in two programming languages: Java and OCaml. Visit http://www.irisa.fr/LIS/ferre/pub/CRQL/.

Node np_{ex} in Fig. 4 points to the tree representation of an example CRQL AST. It has type *np*, and specifies *"any film that has director Steven Spielberg and that has release date something that is after January 1st, 2010"*. The AST definitions reflect NL syntax with noun phrases and verb phrases, but is indeed abstract because all sorts of syntactic distinctions are ignored. The two constructors **Has** and **IsOf** account for the traversal direction of a property, but not whether the property is rendered by a verb, a noun, or a transitive adjective. Similarly, constructor **Geq** has different NL renderings depending on the datatype of the literal.

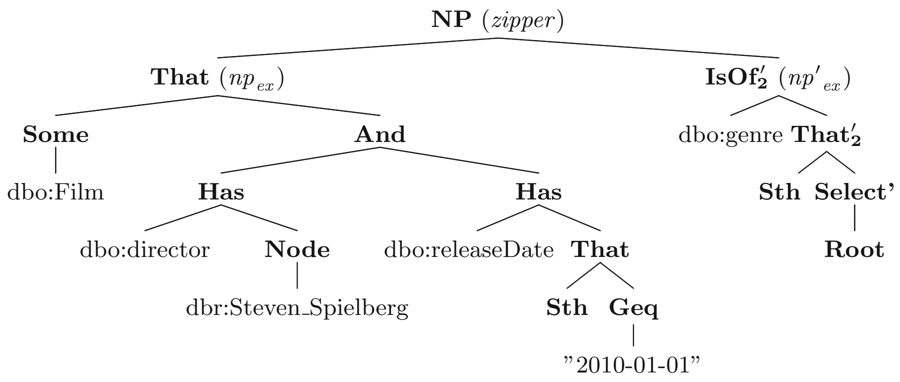

Fig. 4. Example CRQL zipper made of a sub-structure (np_{ex}), and a context (np'_{ex})

The following datatype definitions describe the structure of CRQL contexts. They are automatically obtained as the derivatives of the AST datatypes (see Sect. 2.2). When a constructor has several arguments (e.g., **And**), the derived constructors are indexed by the position of the focus (e.g., $\mathbf{And}'_2$ for a focus on the second argument).

$$\begin{aligned} s' &::= \mathbf{Root} \\ np' &::= \mathbf{Select}'(s') \mid \mathbf{That}'_1(np', vp) \mid \mathbf{Has}'_2(prop, vp') \mid \mathbf{IsOf}'_2(prop, vp') \\ vp' &::= \mathbf{That}'_2(np, np') \mid \mathbf{Not}'(vp') \\ &\mid\ \mathbf{And}'_1(vp', vp) \mid \mathbf{And}'_2(vp, vp') \mid \mathbf{Or}'_1(vp', vp) \mid \mathbf{Or}'_2(vp, vp') \end{aligned}$$

Node np'_{ex} in Fig. 4 points to the tree representation of an example CRQL context. It has type np', and specifies the one-hole AST *"select something that is the genre of _"*, where the underscore (hole) gives the location of the zipper substructure. Finally, the following datatype definition describes a CRQL zipper, combining an AST datatype and its derivative.

$$zipper ::= \mathbf{S}(s, s') \mid \mathbf{NP}(np, np') \mid \mathbf{VP}(vp, vp')$$

Therefore, when editing a CRQL query, the focus can be put on the whole sentence, on any noun phrase (i.e. on any entity involved in the query), or on any particular verb phrase (i.e. on any description of any entity in the query). Figure 4 displays the tree representation of an example CRQL zipper that represents the NL question *"Give me the genre of films directed by Steven Spielberg and whose release date is after January 1st, 2010"*, where the focus is on films.

4.2 A Complete Set of Zipper Transformations

Because ASTs are only built by the successive and interactive application of transformations, it is important to define an initial zipper and a set of zipper transformations that makes the building process complete.

Definition 1 (Completeness). *An initial zipper and a set of zipper transformations is* complete *w.r.t. AST datatypes iff every AST zipper can be reached by applying a finite sequence of transformations starting with the initial zipper.*

We start by defining an initial AST x_0 for each AST datatype x: $s_0 := \mathbf{Select}(np_0)$, $np_0 := \mathbf{Something}$, $vp_0 := \mathbf{True}$. The initial zipper is then defined as $zipper_0 := \mathbf{S}(s_0, \mathbf{Root})$. It corresponds to the totally unconstrained query that returns the list of everything.

We continue by defining a number of zipper transformations. A zipper transformation is formally defined as a partial mapping from zipper to zipper. Transformations are denoted by all-uppercase names, and are defined by unions of mappings from a zipper pattern to a zipper expression. For example, transformations that introduce constructors **That**, **Not**, **And**, **Or** are defined as follows. Transformation *NOT* toggles the application of negation; other transformations coordinate a sub-structure x with an initial AST x_0, and move the focus to x_0.

$$\begin{array}{lll}
THAT & := & \mathbf{NP}(np, np') \to \mathbf{VP}(vp_0, \mathbf{That}'_2(np, np')) \\
NOT & := & \mathbf{VP}(\mathbf{Not}(vp), vp') \to \mathbf{VP}(vp, vp') \\
 & \mid & \mathbf{VP}(vp, vp') \to \mathbf{VP}(\mathbf{Not}(vp), vp') \\
AND & := & \mathbf{VP}(vp, vp') \to \mathbf{VP}(vp_0, \mathbf{And}'_2(vp, vp')) \\
OR & := & \mathbf{VP}(vp, vp') \to \mathbf{VP}(vp_0, \mathbf{Or}'_2(vp, vp'))
\end{array}$$

In order to allow transformations at an arbitrary focus, it is important to allow moving the focus location through the AST. To this purpose, we define four transformations *UP*, *DOWN*, *LEFT*, *RIGHT* to move the focus respectively up to the parent AST node, down to the leftmost child, to the left sibling, and to the right sibling. We only provide the definitions for constructor **That** as other constructors work in a similar way.

$$\begin{array}{lll}
DOWN & := & \mathbf{NP}(\mathbf{That}(np, vp), np') \to \mathbf{NP}(np, \mathbf{That}'_1(np', vp)) \\
UP & := & \mathbf{NP}(np, \mathbf{That}'_1(np', vp)) \to \mathbf{NP}(\mathbf{That}(np, vp), np') \\
 & \mid & \mathbf{VP}(vp, \mathbf{That}'_2(np, np')) \to \mathbf{NP}(\mathbf{That}(np, vp), np') \\
RIGHT & := & \mathbf{NP}(np, \mathbf{That}'_1(np', vp)) \to \mathbf{VP}(vp, \mathbf{That}'_2(np, np')) \\
LEFT & := & \mathbf{VP}(vp, \mathbf{That}'_2(np, np')) \to \mathbf{NP}(np, \mathbf{That}'_1(np', vp))
\end{array}$$

Most transformations are performed by inserting or deleting a sub-structure at the focus. We define a generic transformation *INSERT* that works on any zipper with focus on an initial AST, $\mathbf{X}(x_0, x')$, where x stands for any AST datatype.

$$\mathit{INSERT}(x) := \mathbf{X}(x_0, x') \to \mathbf{X}(x, x')$$

Note that *INSERT* takes an AST of type x as an argument, the sub-structure to insert. Because ASTs cannot be textually edited, the insertable sub-structures must be picked from a finite collection, and have to be suggested by the system. A generic *DELETE* transformation can also be defined to undo insertions.

The above set of transformations can be proved complete provided that a proper collection of insertable elements is defined.

Theorem 1 (Completeness). *Assume the following* insertable elements *for each datatype, where 'node', 'class', and 'prop' hold for any value present in the RDF dataset, and where'lit' holds for any filtering value input by users:*

np : $\mathbf{Some}(\mathit{class})$, $\mathbf{Node}(\mathit{node})$,
vp : $\mathbf{IsA}(\mathit{class})$, $\mathbf{Has}(\mathit{prop}, np_0)$, $\mathbf{IsOf}(\mathit{prop}, np_0)$, *and* $\mathbf{Geq}(\mathit{lit})$.

Then, initial zipper$_0$ *and the set of transformations defined above (INSERT*(x) *applying to insertable elements) is* complete.

$T(\mathbf{Select}(np))$	$:= \mathit{DOWN}; T(np); \mathit{UP}$
$T(\mathbf{Something})$	$:= \mathit{ID}$
$T(\mathbf{Some}(\mathit{class}))$	$:= \mathit{INSERT}(\mathbf{Some}(\mathit{class}))$
$T(\mathbf{Node}(\mathit{node}))$	$:= \mathit{INSERT}(\mathbf{Node}(\mathit{node}))$
$T(\mathbf{That}(np, vp))$	$:= T(np); \mathit{THAT}; T(vp); \mathit{UP}$
$T(\mathbf{IsA}(\mathit{class}))$	$:= \mathit{INSERT}(\mathbf{IsA}(\mathit{class}))$
$T(\mathbf{Has}(\mathit{prop}, np))$	$:= \mathit{INSERT}(\mathbf{Has}(\mathit{prop}, np_0)); \mathit{DOWN}; T(np); \mathit{UP}$
$T(\mathbf{IsOf}(\mathit{prop}, np))$	$:= \mathit{INSERT}(\mathbf{IsOf}(\mathit{prop}, np_0)); \mathit{DOWN}; T(np); \mathit{UP}$
$T(\mathbf{Geq}(\mathit{lit}))$	$:= \mathit{INSERT}(\mathbf{Geq}(\mathit{lit}))$
$T(\mathbf{True})$	$:= \mathit{ID}$
$T(\mathbf{And}(vp_1, vp_2)$	$:= T(vp_1); \mathit{AND}; T(vp_2); \mathit{UP}$
$T(\mathbf{Or}(vp_1, vp_2)$	$:= T(vp_1); \mathit{OR}; T(vp_2); \mathit{UP}$
$T(\mathbf{Not}(vp))$	$:= T(vp); \mathit{NOT}$

Fig. 5. Recursive definition of the transformation sequence $T(x)$ from x_0 to AST x

Proof. First, it is easy to show that the moving transformations give access to all zippers of an AST. Therefore, to prove completeness, it is enough to prove that every zipper in the form $\mathbf{S}(s, \mathbf{Root})$ is reachable. Figure 5 defines a recursive function $T(x)$ that returns a transformation sequence from an initial AST x_0 to any AST x, where x stands for any AST datatype. For each case, it can be proved that the transformation sequence $T(x)$ indeed leads from x_0 to x, and that every recursive call is well defined (sub-structure x_0 at focus). ■

The above proof provides an algorithm T for computing the transformation sequence leading to any AST x. The example zipper given in Fig. 4 can be reached with the following sequence:

$$\begin{array}{l} \mathit{DOWN};\ \mathit{THAT};\ \mathit{INSERT}(\mathbf{IsOf}(\texttt{dbo:genre}, np_0));\ \mathit{DOWN}; \\ \ \mathit{INSERT}(\mathbf{Some}(\texttt{dbo:Film})); \\ \ \mathit{THAT};\ \mathit{INSERT}(\mathbf{Has}(\texttt{dbo:director}, np_0));\ \mathit{DOWN}; \\ \ \mathit{INSERT}(\mathbf{Node}(\texttt{dbr:Steven\_Spielberg}));\ \mathit{UP}; \\ \ \mathit{AND};\ \mathit{INSERT}(\mathbf{Has}(\texttt{dbo:releaseDate}, np_0));\ \mathit{DOWN}; \\ \ \mathit{THAT};\ \mathit{INSERT}(\mathbf{Geq}(\texttt{"2010-01-01"}));\ \mathit{UP};\ \mathit{UP};\ \mathit{UP};\ \mathit{UP}. \end{array}$$

In practice, it appears useful to tune transformations so as to minimize the number of interaction steps for users. For example, moving down after inserting a property can be made automatic given its frequency.

4.3 Formalization to SPARQL

AST zippers are given a semantics by translating them to a formal language. Here, we translate CRQL to SPARQL. Translating queries to SPARQL makes it easy to evaluate them with SPARQL engines and through SPARQL endpoints. Before the translation itself, we apply to zippers a transformation *NORM* to normalize them into sentence zippers $\mathbf{S}(s, \mathbf{Root})$. This has the advantage to simplify the translation by restricting it to ASTs, and making it unnecessary for zipper contexts.

$$\begin{array}{rl} \mathit{NORM} := & \mathbf{S}(s, \mathbf{Root}) \to \mathbf{S}(s, \mathbf{Root}) \\ \mid & \mathbf{VP}(vp_1, \mathbf{Or'_1}(vp', vp_2)) \to \mathit{NORM}(\mathbf{VP}(vp_1, vp')) \\ \mid & \mathbf{VP}(vp_2, \mathbf{Or'_2}(vp_1, vp')) \to \mathit{NORM}(\mathbf{VP}(vp_2, vp')) \\ \mid & \mathbf{VP}(vp, \mathbf{Not'}(vp')) \to \mathit{NORM}(\mathbf{VP}(vp, vp')) \\ \mid & \mathbf{X}(x, x') \to \mathit{NORM}(\mathit{UP}(\mathbf{X}(x, x'))) \end{array}$$

By default, normalizing a zipper is simply moving the focus up repeatedly (last line), until the sentence level is reached (first line). However, constructors **Or** (union), and **Not** (negation) are removed when the focus is in their scope (middle lines). The reason is to make sure that the query results are defined and relevant to the focus. For example, if the focus is in one branch of an union, then the other branch can be temporarily ignored in the semantics. In SPARQL, a variable that is introduced inside a negation cannot be returned in results. Moving the focus in the scope of a negation is then a way to access the values of such a variable, by temporarily ignoring negation.

Once a sentence AST s is obtained by normalization, its translation to a SPARQL SELECT query can be defined by the Montague grammar in Fig. 6. SPARQL strings are delimited with quotes, and concatenated with +. One SPARQL variable '?x$_i$' is introduced for each indefinite head (constructors **Something** and **Some**). For example, the translation of the zipper given in Fig. 4 produces the following SPARQL query. The three variables correspond respectively to the film's genre, the film, and the film's release date.

```
SELECT ?x1 ?x2 ?x3 WHERE
```

$$\begin{array}{rcll}
s & ::= & \mathbf{Select}(np) & \texttt{'SELECT ?x}_1\texttt{ ... ?x}_n\texttt{ WHERE \{'} + (np\ \lambda x.(\texttt{''})) + \texttt{'\}'} \\
np & ::= & \mathbf{Something}_i & \lambda d.((d\ \texttt{'?x}_i\texttt{'})) \\
 & | & \mathbf{Some}(\mathit{class})_i & \lambda d.(\texttt{'?x}_i\texttt{ a'} + \mathit{class} + \texttt{'.'} + (d\ \texttt{'?x}_i\texttt{'})) \\
 & | & \mathbf{Node}(\mathit{node}) & \lambda d.((d\ \mathit{node})) \\
 & | & \mathbf{That}(np, vp) & \lambda d.(np\ \lambda x.((d\ x) + \texttt{'.'} + (vp\ x))) \\
vp & ::= & \mathbf{IsA}(\mathit{class}) & \lambda x.(x + \texttt{'a'} + \mathit{class}) \\
 & | & \mathbf{Has}(\mathit{prop}, np) & \lambda x.((np\ \lambda y.(x + \mathit{prop} + y))) \\
 & | & \mathbf{IsOf}(\mathit{prop}, np) & \lambda x.((np\ \lambda y.(y + \mathit{prop} + x))) \\
 & | & \mathbf{Geq}(\mathit{lit}) & \lambda x.(\texttt{'FILTER('} + x + \texttt{'>='} + \mathit{lit} + \texttt{')'}) \\
 & | & \mathbf{True} & \lambda x.(\texttt{''}) \\
 & | & \mathbf{And}(vp_1, vp_2) & \lambda x.((vp_1\ x) + \texttt{'.'} + (vp_2\ x)) \\
 & | & \mathbf{Or}(vp_1, vp_2) & \lambda x.(\texttt{'\{'} + (vp_1\ x) + \texttt{'\} UNION \{'} + (vp_2\ x) + \texttt{'\}'}) \\
 & | & \mathbf{Not}(vp) & \lambda x.(\texttt{'FILTER NOT EXISTS \{'} + (vp\ x) + \texttt{'\}'})
\end{array}$$

Fig. 6. Formalization to SPARQL of CRQL ASTs

```
{ ?x2 a dbo:Film . ?x2 dbo:genre ?x1 .
  ?x2 dbo:director dbr:Steven_Spielberg .
  ?x2 dbo:releaseDate ?x3 . FILTER (?x3 >= "2010-01-01") }
```

4.4 Verbalization to English

ASTs are given a concrete and user-friendly syntax by verbalizing them to NL. It is simpler than direct verbalizations of FL expressions, e.g. SPARQL queries [15], because ASTs follow the phrase structure of NL. A good quality verbalization requires linguistic resources about lexical and syntactic aspects (e.g., Lemon lexicons [14], WordNet, Grammatical Framework [16]). However, because humans are more robust than machines at language understanding, a naive verbalization can be good enough in practice for simple languages like CRQL. We here sketch such a verbalization, showing that the technique of Montague grammars is also useful for AST to NL translation.

$$\begin{array}{rcll}
s & ::= & \mathbf{Select}(np) & \texttt{'Give me'} + np \\
np & ::= & \mathbf{That}(np, vp) & np + (vp\ 0) \\
vp & ::= & \mathbf{IsA}(\mathit{class}) & \lambda n.(\texttt{'that'} + (\mathit{is}\ n) + \texttt{'a(n)'} + \mathit{class}) \\
 & | & \mathbf{Has}(\mathit{prop}, np) & \lambda n.(\texttt{'whose'} + \mathit{prop} + (\mathit{is}\ n) + np) \\
 & | & \mathbf{And}(vp_1, vp_2) & \lambda n.((vp_1\ n) + (\mathit{and}\ n) + (vp_2\ n)) \\
 & | & \mathbf{Not}(vp) & \lambda n.(vp\ \overline{n})
\end{array}$$

Fig. 7. Verbalization to English of CRQL ASTs (partial)

Before verbalization, an AST zipper is normalized like for SPARQL translation (Sect. 4.3), except that constructors **Or**/**Not** in the context are not removed but marked for dimmed display. Similarly, the zipper sub-structure is highlighted to show the focus. For a naive verbalization, we assume that nodes, classes and properties are verbalized to their labels, which we assume to be common nouns: e.g., `'film'`, `'director'`, `'release date'`. The Montague grammar translates each AST to a string, and translations of verb phrases are relative clauses abstracted by a Boolean flag, n, in order to propagate negation as a modifier to the relevant verbs. Figure 7 shows a subset of the translation rules. Function *is* verbalizes the copula depending on negation: $(is\ 0) :=$ `'is'`, $(is\ 1) :=$ `'is not'`. Function *and* encodes De Morgan laws: $(and\ 0) :=$ `'and'`, $(and\ 1) :=$ `'or'`. The translation of an AST can be post-processed to further simplify it. For example, the sequence `'something that is` *NP*`'` can be replaced by *NP*, and the sequence `'is something after'` by `'is after'`. Then, the verbalization of the example in Fig. 4 results in `'Give me the genre of a film whose director is Steven Spielberg and whose release date is after January 1st, 2010'`.

5 Validation of the Design Pattern

As the purpose of a design pattern is to provide a same solution to different problems, we validate the `N<A>F` design pattern by showing how it has been used to address the NL-FL gap in three quite different tasks related to the Semantic Web. Each task brought its own contribution relative to the task, and was properly evaluated w.r.t. expressivity, usability, and/or performance. They constitute a demonstration of the effectiveness and genericity of our design pattern. In this section, we shortly describe each tool, and how it can be seen as a variation of querying with CRQL (detailed in Sect. 4).

5.1 Sparklis: Querying SPARQL Endpoints

Sparklis [5] is a tool for querying SPARQL endpoints. Its target FL is hence SPARQL. The covered subset of SPARQL includes CRQL, extended with arbitrary basic graph patterns (including cycles), additional filters, `OPTIONAL`, aggregation and grouping, and solution modifiers (ordering, `HAVING`). Cyclic graph patterns are verbalized with anaphoras: e.g., `'a film that is starring the director of the film'`. Aggregations are verbalized with head modifiers: e.g., `'Give me the number of film whose director is Tim Burton'`. SPARQL endpoints are used not only to retrieve query answers, but also to compute the insertable elements that do not lead to empty answers, i.e. that provide information relevant to the current answers. For example, considering flights arriving in Heraklion, only showing departure cities, not all cities.

Sparklis is available online[5], had about 1000 unique users over 18 months, and was used on more than 100 different endpoints. It scales to datasets as large as DBpedia (several billions triples). In an experiment on QALD-3 DBpedia questions, the median query construction time was 30 s, the maximum time was 109 s, and only one question led to a timeout.

[5] Sparklis online at http://www.irisa.fr/LIS/ferre/sparklis/.

5.2 Utilis: Authoring RDF Descriptions

Utilis [8] is a functionality integrated to Sewelis[6] for authoring RDF descriptions. Its target FL is hence RDF. A subset of CRQL can be used to cover all of RDF. Constructor **Select**(*np*) is replaced by constructor **Describe**(*node*, *vp*), where *node* acts as the RDF node being described, and *vp* acts as its description. Constructors **Or**, **Not**, and filters become irrelevant. Constructors **Something** and **Some** correspond to the introduction of blank nodes. The key difference with Sparklis lies in the semantics of AST zippers. Their semantics is a ranking of the RDF nodes of a dataset, according to their similarity with the entity at the zipper focus. Given the example zipper of Fig. 4, one gets first *films by Spielberg since 2010*, then *films by Spielberg until 2010*, then *films by other directors*, etc. The system uses the description of most similar nodes to suggest insertable elements.

A user study comparing Utilis to Protégé has shown that users prefered the fine-grained suggestions of Utilis to the static entity lists of Protégé. We have also observed that those suggestions improve consistency across RDF descriptions without the rigidity of a prescriptive schema.

5.3 PEW: Completing OWL Ontologies

PEW[7] [7] is a tool for completing OWL ontologies. Its target FL is hence OWL. The covered subset of OWL is made of expression classes with existential restrictions, nominals, conjunctions, disjunctions, atomic negations, and inverse roles. Like for Utilis, a subset of CRQL can be used, excluding filters, and restricting **Not** to atomic verb phrases. Constructor **Select**(*np*) is replaced by **Sat**(*np*) to express the fact that the semantics of an AST is the satisfiability of the corresponding class expression. The suggested insertable elements are those that preserve satisfiability. Negated elements are also suggested, and are the only way to introduce negation in the AST. Because only satisfiable class expressions can be built, PEW allows the exploration of the "possible worlds" of an ontology (its models). When undesirable "possible worlds" are found, the tool allows to automatically produce an OWL axiom so as to exclude them.

We have found the tool useful, and even playful, for completing an ontology with negative axioms (e.g., class disjointness), which are often missing. An experiment has been performed on the well-known Pizza ontology, and has revealed many missing negative axioms through undesirable situations: e.g., *"a country that is also some food"* (missing disjointness), *"a country of origin that is not a country"* (missing range), and even *"a vegetarian pizza that can have meat as ingredient"* (the definition uses `hasTopping` instead of `hasIngredient`).

6 Related Work

The FL-NL gap is not a new issue, and many solutions have been proposed to reduce it. Question Answering (QA) consists in translating NL expressions to

[6] Download and screencasts at http://www.irisa.fr/LIS/softwares/sewelis/.
[7] Download and screencast at http://www.irisa.fr/LIS/softwares/pew/.

FL, generally going through one or several intermediate representations (e.g., parse trees). In the Semantic Web, many systems translate English questions to SPARQL queries (see [13] for a survey), and the QALD[8] challenge is devoted to that task. NL interfaces are attractive for their ease-of-use, and definetely have a role to play, but they suffer from a weak adequacy: (habitability) spontaneous NL expressions often have no FL counterpart or are ambiguous, (expressivity) only a small FL fragment is covered in general. This makes it difficult to interpret an empty answer: Does it reflect actual data? an out-of-scope question? a lack of expressivity or simply a misundertanding by the system? Habitability can be addressed in part by suggesting reformulations of user questions, e.g. based on templates [2], but the number of suggestions tends to grow exponentially with expressivity (query size and number of features). An alternative to spontaneous NL is Controlled NL (CNL) [12]. A CNL typically defines a NL fragment that is adequate to the target FL, and that eliminates or reduces ambiguity. However, while much easier to read than FL, a CNL remains difficult to write because of the constrained syntax. That is why CNL-based editors often come with auto-completion to suggest the next possible words during edition: e.g., ACE Wiki for facts and rules [10], Ginseng for queries [11], Atomate it! for reactive rules [18], Halo for problem solving [3]. Auto-completion offers a limited flexibility because suggestions are only at the end of a partial sentence, and because translation to FL, and hence semantics, is only available when the sentence is complete. The latter limitation also implies that suggestions are not based on semantics, but on syntax and schema only. Note that, by *semantics*, we here mean not only the FL expression, but also any interpretation that may come with it (e.g., query results, satisfiability checks of OWL class expressions).

A noticeable approach that is not based on NL is structural edition, e.g. *query builders* like SemanticCrystal [11], where a point-and-click interface is used to build FL expressions incrementally. They avoid the habitability problem but the presentation of the FL expression is generally based either on its syntax tree, and hence very close to the FL, or on a graphical view that is more intuitive but often limits expressivity. Grammatical Framework (GF) [16] improves structural edition by distinguishing abstract syntax and concrete syntax, only showing the later to users. In fact, the tools and linguistic resources of GF can be used to implement the verbalization part of our `N<A>F` design pattern. However, the GF equivalent of AST transformations are not customizable to the target task and semantics; and ASTs are not guaranteed to have a fully-defined semantics at every edition step. This is because GF is about syntax, not about task-specific semantics.

7 Conclusion

The gap between formal languages and natural languages is deep, and we do not pretend to fill it. However, we believe that our design pattern based on AST zippers provides a powerful strategy to build bridges over the gap (see Fig. 3).

[8] http://greententacle.techfak.uni-bielefeld.de/~cunger/qald/.

For people on the natural language side, those bridges offer a safe and large access to the benefits of formal languages. Once they engage on such a bridge, they cannot fall in the gap (no *habitability* problem), and their access to the formal language side is open to a large area (high *expressivity*). We think that it provides an interesting alternative to question answering by avoiding the major difficulties of NL understanding thanks to NL-based interaction. We have shown the versatility of our design pattern with applications to three different formal languages and tasks. The perspectives consists in creating new applications, and improving them on three axes: formal language coverage (*expressivity*), quality of the verbalization (*readability*), and intelligence of system suggestions (*guidance*).

References

1. Abbott, M., Altenkirch, T., McBride, C., Ghani, N.: ∂ for data: differentiating data structures. Fundamenta Informaticae **65**(1), 1–28 (2005)
2. Chaudhri, V.K., Clark, P.E., Overholtzer, A., Spaulding, A.: Question generation from a knowledge base. In: Janowicz, K., Schlobach, S., Lambrix, P., Hyvönen, E. (eds.) EKAW 2014. LNCS, vol. 8876, pp. 54–65. Springer, Heidelberg (2014)
3. Clark, P., Chaw, S.Y., Barker, K., Chaudhri, V., Harrison, P., Fan, J., John, B., Porter, B., Spaulding, A., Thompson, J., Yeh, P.: Capturing and answering questions posed to a knowledge-based system. In: International Conference Knowledge Capture, pp. 63–70. ACM (2007)
4. Dowty, D.R., Wall, R.E., Peters, S.: Introduction to Montague Semantics. Reidel Publishing Company, Dordrecht (1981)
5. Ferré, S.: Expressive and scalable query-based faceted search over SPARQL endpoints. In: Mika, P., Tudorache, T., Bernstein, A., Welty, C., Knoblock, C., Vrandečić, D., Groth, P., Noy, N., Janowicz, K., Goble, C. (eds.) ISWC 2014, Part II. LNCS, vol. 8797, pp. 438–453. Springer, Heidelberg (2014)
6. Ferré, S.: SQUALL: The expressiveness of SPARQL 1.1 made available as a controlled natural language. Data Knowl. Eng. **94**, 163–188 (2014)
7. Ferré, S., Rudolph, S.: Advocatus diaboli – exploratory enrichment of ontologies with negative constraints. In: ten Teije, A., Völker, J., Handschuh, S., Stuckenschmidt, H., d'Acquin, M., Nikolov, A., Aussenac-Gilles, N., Hernandez, N. (eds.) EKAW 2012. LNCS, vol. 7603, pp. 42–56. Springer, Heidelberg (2012)
8. Hermann, A., Ferré, S., Ducassé, M.: An interactive guidance process supporting consistent updates of RDFS graphs. In: ten Teije, A., Völker, J., Handschuh, S., Stuckenschmidt, H., d'Acquin, M., Nikolov, A., Aussenac-Gilles, N., Hernandez, N. (eds.) EKAW 2012. LNCS, vol. 7603, pp. 185–199. Springer, Heidelberg (2012)
9. Huet, G.: Functional pearl - the zipper. J. Funct. Program. **7**(5), 549–554 (1997)
10. Kaljurand, K., Kuhn, T.: A multilingual semantic wiki based on attempto controlled english and grammatical framework. In: Presutti, V., Hollink, L., Rudolph, S., Cimiano, P., Corcho, O. (eds.) ESWC 2013. LNCS, vol. 7882, pp. 427–441. Springer, Heidelberg (2013)
11. Kaufmann, E., Bernstein, A.: Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases. J. Web Semant. **8**(4), 377–393 (2010)
12. Kuhn, T.: A survey and classification of controlled natural languages. Comput. Linguist. **40**(1), 121–170 (2013)

13. Lopez, V., Uren, V.S., Sabou, M., Motta, E.: Is question answering fit for the semantic web? a survey. Seman. Web **2**(2), 125–155 (2011)
14. McCrae, J., Spohr, D., Cimiano, P.: Linking lexical resources and ontologies on the semantic web with lemon. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) ESWC 2011, Part I. LNCS, vol. 6643, pp. 245–259. Springer, Heidelberg (2011)
15. Ngomo, A.C.N., Bühmann, L., Unger, C., Lehmann, J., Gerber, D.: Sorry, I don't speak SPARQL: translating SPARQL queries into natural language. In: WWW, pp. 977–988 (2013)
16. Ranta, A.: Grammatical framework. J. Funct. Program. **14**(02), 145–189 (2004)
17. Turner, D.: Functional programming and proofs of program correctness. In: Néel, D. (ed.) Tools and Notions for Program Construction: An Advanced Course, pp. 187–209. Cambridge University Press, Cambridge (1982)
18. Van Kleek, M., Moore, B., Karger, D., André, P., Schraefel, M.: Atomate it! end-user context-sensitive automation using heterogeneous information sources on the web. In: International Conference World Wide Web, pp. 951–960. ACM (2010)