Responsive and Flexible Controlled Natural Language Authoring with Zipper-based Transformations

Sébastien Ferré

Responsive and Flexible Controlled Natural Language Authoring with Zipper-based Transformations

Sébastien FERRÉ [1]
Univ Rennes, CNRS, IRISA
Campus de Beaulieu, 35042 Rennes cedex, France
ferre@irisa.fr

Abstract Controlled natural languages (CNL) have the benefits to combine the readability of natural languages, and the accuracy of formal languages. They have been used to help users express facts, rules or queries. While generally easy to read, CNLs remain difficult to write because of the constrained syntax. A common solution is a grammar-based auto-completion mechanism to suggest the next possible words in a sentence. However, this solution has two limitations: (a) partial sentences may have no semantics, which prevents giving intermediate results or feedback, and (b) the suggestion is often limited to adding words at the end of the sentence. We propose a more responsive and flexible CNL authoring by designing it as a sequence of sentence transformations. Responsiveness is obtained by having a complete, and hence interpretable, sentence at each time. Flexibility is obtained by allowing insertion and deletion on any part of the sentence. Technically, this is realized by working directly on the abstract syntax, rather than on the concrete syntax, and by using Huet's zippers to manage the focus on a query part, the equivalent of the text cursor of a word processor.

Keywords. Controlled Natural Languages, Authoring, User Interaction, Abstract Syntax, Huet's Zippers, Focus

1. Introduction

An important issue in the Semantic Web [8], and knowledge-based systems in general, is to fill the gap between the natural language (NL) of users, and the formal languages (FL) of those systems (e.g., OWL, SPARQL, Answer Set Programming). Formal languages make data processable by machines but they also constitute a language barrier to the production and consumption by end users. Controlled Natural Languages (CNL) [12] offer an interesting solution in that they combine the readability of natural languages, and the accuracy of formal languages. An input sentence respecting the syntactic and semantic constraints of the CNL can be parsed non-ambiguously into a formal expression (e.g. a query), and hence can be automatically interpreted (e.g. computing query results). CNLs also offer a good *adequacy* between what can be expressed respectively by the

[1]This research is supported by ANR project PEGASE (ANR-16-CE23-0011-08).

natural language and the formal language. In contrast, the spontaneous natural language approach generally adopted in Question Answering (QA) [13] suffers from problems of ambiguity and adequacy.

While CNLs are much easier to read compared to FL, they remain difficult to write because of the constrained syntax. Syntactic and semantic errors can be frequent and frustrating, especially for users who are new to the CNL and/or the domain vocabulary. The latter is related to the well-known *habitability* problem [11], which occurs when users do not known exactly what falls in the scope of the knowledge-based system, and what is out of scope. A common solution in CNL editors is to use an *auto-completion* mechanism (e.g., ACE [10], Ginseng [11]) that suggests the next possible words in a sentence. Those suggestions are derived from the CNL grammar and the domain vocabulary, and possibly predicted from usage statistics [6,15]. The major advantages of auto-completion are to prevent syntactic errors, and to alleviate the habitability problem. However, we here identify two limitations of auto-completion for CNL authoring. First, as auto-completion is designed to complete a sentence word after word, it results that at most steps the current sentence is not complete, and hence cannot be translated as such to the formal language, and therefore cannot be interpreted. For example, in the case of a query, results can be computed and returned only once the sentence expressing it is complete. When the query happens to have no result, the user will only detect it at the end even if the constraints expressed in the first half of the query were already sufficient to make the results empty. Second, auto-completion is limited to adding words at the end of the partial sentence. In comparison, a text editor allows for insertions and deletions at any position, and has a cursor to let the user control that position. For a user, expressing her knowledge or information needs is generally not a linear process, and goes through rectifications, insertions, substitutions, etc. There is a need to combine suggestions with a flexible authoring process.

We propose an alternative solution to guide users in the authoring of CNL sentences, by following the principles of the `N<A>F` design pattern [3]. In our proposal, the authoring process is based on sequences of transformations at the abstract syntax level, rather than on the addition of words at the concrete syntax level. As a consequence, the concrete form is obtained by NL generation from the abstract form, rather than the reverse by syntactic parsing as usually done in CNL and QA. The formal expression is obtained as usual by translation from the abstract form, generally using compositional semantic techniques such as Montague grammars [2]. We claim that our solution improves auto-completion in terms of *responsiveness* and *flexibility*. It is responsive because transformations are designed to take a complete sentence as input, and to return a complete sentence as output. Therefore, at each step, the current sentence is complete, and can therefore be interpreted to give the user intermediate results and feedback. Our solution is also flexible because it allows for insertions and deletions on any part of the sentence, where a part is defined as a node of the abstract syntax tree. We use Huet's zippers [9] as a technique to represent that part, called *focus*, which plays a role similar to the text cursor of a word processor. We illustrate our approach on a small yet expressive query language whose target FL is SPARQL [17].

The paper is structured as follows. Section 2 shortly recalls the principles of the `N<A>F` design pattern, and Huet's zippers. Section 3 describes our approach in detail through a concrete use case. Section 4 reports on the implementation and application of our approach to several tasks.

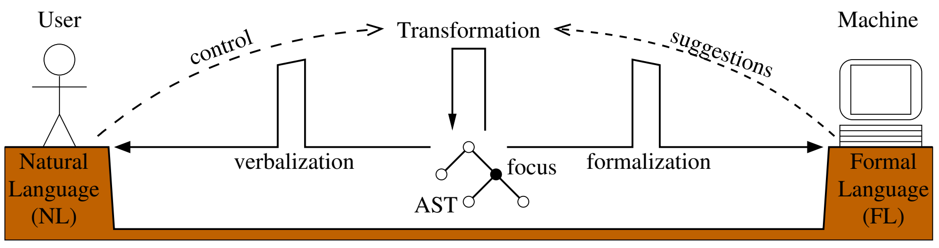


Figure 1. Principle of transformation-based authoring for bridging the gap between NL and FL

2. Preliminaries

2.1. The `N<A>F` *Design Pattern*

The principle of the `N<A>F` design pattern [3] is schematized as a "suspended bridge over the NL-FL gap" in Figure 1. The central pillar is made of Abstract Syntax Trees (AST) 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 is initialized by the system, and modified by users applying structural *transformations*, not 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.

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 as many details as possible. Indeed, starting with a flat representation like SPARQL, it is possible to produce a NL version [14], 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. The design pattern supports multi-lingualism because only verbalization depends on the chosen NL. In particular, the NL can be changed at any time in the course of a user session. Moreover, NL generation is known to be easier than NL understanding so that it is easier to support more languages.

In this paper, we concentrate on the representation of the abstract syntax, the focus, and the transformations. We assume classical techniques for the verbalization to NL (e.g., Grammatical Framework [16]), and for the translation to FL (e.g., Montague grammars [2]).

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. Pure 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 [18]. 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.

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

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

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

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 the elements before the location. Finally, a zipper data structure combines a structure and a context: $\mathit{zipper} ::= \mathbf{List}(\mathit{list}, \mathit{list}')$. 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 $\mathbf{List}(\mathbf{Cons}(e, l), l')$.

3. Detailed Use Case: SPARQL-based Querying

In this section, we describe in detail our approach taking SPARQL-based querying as a use case[2]. Our abstract intermediate query language covers the basic graph patterns, and their composition with union (UNION) and negation (NOT EXISTS).

3.1. *AST Zippers*

The following datatype definitions describe the abstract syntax of our query language. Given base types for RDF nodes (*node*), RDFS classes (*class*), and RDFS properties

[2]The complete source code of an extension of this use case is available online in the OCaml programming languages. Visit `http://www.irisa.fr/LIS/ferre/pub/CNL2018.ml`

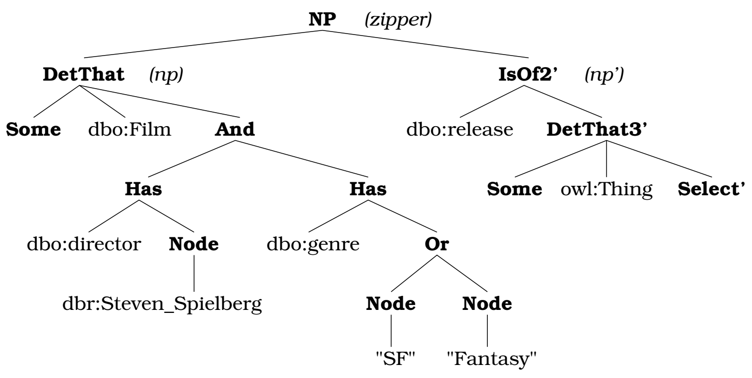


Figure 2. Example zipper made of a sub-structure (np), and a context (np')

($prop$), we define abstract sentences (s), noun phrases (np), determiners (det), and verb phrases (vp).

s ::= **Select**(np)
np ::= **Node**($node$) | **DetThat**($det, class, vp$)
| **And**(np, np) | **Or**(np, np) | **Not**(np)
det ::= **Some** | **Every** | **No**
vp ::= **IsA**($class$) | **Has**($prop, np$) | **IsOf**($prop, np$)
| **True** | **And**(vp, vp) | **Or**(vp, vp) | **Not**(vp)

Note that this is very similar to the definition of abstract syntax in Grammatical Framework [16], with *categories* here in italic, and *functions* here in bold. Node np in Figure 2 points to the tree representation of an example AST. It has type np, and specifies *"any film directed by Steven Spielberg and whose genre is SF or Fantasy"*. 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 type vp is used to represent relative clauses (occurence of vp in **DetThat**) because the two have the same semantics. The two constructors **Has** and **IsOf** account for the traversal direction of a property, but not whether the property is verbalized with a verb, a noun, or a transitive adjective. Note that Boolean connectors are defined on both NPs and VPs.

The following datatype definitions describe the structure of AST contexts. They are automatically obtained as the derivatives of the AST datatypes (see Section 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).

np' ::= $\mathbf{Select}'$ | $\mathbf{Has}'_2(prop, vp')$ | $\mathbf{IsOf}'_2(prop, vp')$
| $\mathbf{And}'_1(np', np)$ | $\mathbf{And}'_2(np, np')$ | $\mathbf{Or}'_1(np', np)$ | $\mathbf{Or}'_2(np, np')$ | $\mathbf{Not}'(np')$
vp' ::= $\mathbf{DetThat}'_3(det, class, np')$
| $\mathbf{And}'_1(vp', vp)$ | $\mathbf{And}'_2(vp, vp')$ | $\mathbf{Or}'_1(vp', vp)$ | $\mathbf{Or}'_2(vp, vp')$ | $\mathbf{Not}'(vp')$

Node np' in Figure 2 points to the tree representation of an example AST context. It has type np', and specifies the one-hole AST *"select the release date of _"*, where the underscore (hole) gives the location of the zipper sub-structure. Note that the path in the context AST from the root to $\mathbf{Select}'$ is the reverse of the path in the one-hole AST from the root to the hole. Finally, the following datatype definition describes a AST zipper, combining an AST and an AST context.

$zipper$ ::= **NP**(np, np') | **VP**(vp, vp')

Therefore, when editing a query, the focus can be put 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 2 displays the tree representation of an example zipper that represents the NL question *"Give me the release date of films directed by Steven Spielberg and whose genre is SF or Fantasy"*, where the focus is on films. Its formalization in SPARQL is: `SELECT ?x1 ?x2 WHERE {?x2 dbo:releaseDate ?x1. ?x2 rdf:type dbo:Film. ?x2 dbo:director dbr:Steven_Spielberg. {?x2 dbo:genre "SF"} UNION {?x2 dbo:genre "Fantasy"}}`.

3.2. A Complete Set of Zipper Transformations

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

Definition 1 (completeness)**.** *A set of initial zippers and a set of zipper transformations are* complete *w.r.t. AST datatypes iff every AST zipper can be reached by applying a finite sequence of transformations starting with an initial zipper.*

We start by defining an initial AST x_0 for each AST datatype x: $s_0 := \textbf{Select}(np_0)$, $np_0 := \textbf{DetThat}(\textbf{Some}, \texttt{owl:Thing}, \textbf{True})$, $vp_0 := \textbf{True}$. An initial zipper can then defined as $zipper_0 := \textbf{NP}(np_0, \textbf{Select}')$. It corresponds to the totally unconstrained query that returns the list of everything. A menu of application-specific initial zippers can be added according to frequent types of queries, and so that the users have less transformations to perform.

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. We strive to define our transformations in terms of the domain vocabulary rather than in terms of the AST constructs, so as to avoid exposing users to the latter. Therefore, our first transformations insert the primitive elements (i.e. nodes, determiners, classes, and properties) depending on the current focus.

$$\begin{array}{lll}
NODE(n) & := & \textbf{NP}(np, np') \to \textbf{NP}(\textbf{Node}(n), np') \\
DET(d) & := & \textbf{NP}(\textbf{DetThat}(det, class, vp), np') \to \textbf{NP}(\textbf{DetThat}(d, class, vp), np') \\
CLASS(c) & := & \textbf{NP}(\textbf{DetThat}(det, class, vp), np') \to \textbf{NP}(\textbf{DetThat}(det, c, vp), np') \\
 & \mid & \textbf{VP}(\textbf{True}, vp') \to \textbf{VP}(\textbf{IsA}(c), vp') \\
PROP(p) & := & \textbf{NP}(\textbf{DetThat}(d, c, \textbf{True}), np') \to \textbf{VP}(\textbf{Has}(p, np_0), \textbf{DetThat}'_3(d, c, np')) \\
 & \mid & \textbf{VP}(\textbf{True}, vp') \to \textbf{VP}(\textbf{Has}(p, np_0), vp') \\
PROP^-(p) & := & \text{(same as } PROP\text{, replacing } \textbf{Has} \text{ by } \textbf{IsOf}\text{)}
\end{array}$$

Other transformations allow the introduction of the Boolean connectors between noun phrases and verb phrases. They are defined in a generic way below, where $\textbf{X}/x$ stands for both $\textbf{NP}/np$ and $\textbf{VP}/vp$. Transformations AND, OR coordinate a substructure x with an initial AST x_0, and move the focus to x_0. Transformation NOT toggles the application of negation.

$$\begin{array}{lll}
AND & := & \textbf{X}(x, x') \to \textbf{X}(x_0, \textbf{And}'_2(x, x')) \\
OR & := & \textbf{X}(x, x') \to \textbf{X}(x_0, \textbf{Or}'_2(x, x')) \\
NOT & := & \textbf{X}(\textbf{Not}(x), x') \to \textbf{X}(x, x') \\
 & \mid & \textbf{X}(x, x') \to \textbf{X}(\textbf{Not}(x), x')
\end{array}$$

$T(\mathbf{Node}(n))$	$:= NODE(n)$
$T(\mathbf{DetThat}(d,c,vp))$	$:= DET(d); CLASS(c); DOWN; T(vp); UP$
$T(\mathbf{IsA}(c))$	$:= CLASS(c)$
$T(\mathbf{Has}(p,np))$	$:= PROP(p); DOWN; T(np); UP$
$T(\mathbf{IsOf}(p,np))$	$:= PROP^{-}(p); DOWN; T(np); UP$
$T(\mathbf{True})$	$:= ID$
$T(\mathbf{And}(x_1,x_2)$	$:= T(x_1); AND; T(x_2); UP$
$T(\mathbf{Or}(x_1,x_2)$	$:= T(x_1); OR; T(x_2); UP$
$T(\mathbf{Not}(x))$	$:= T(x); NOT$

Figure 3. Recursive definition of the transformation sequence $T(x)$ from x_0 to AST x

A generic *DELETE* transformation can also be defined to undo insertions.
$DELETE := \mathbf{X}(x,x') \rightarrow \mathbf{X}(x_0,x)$

In order to allow transformations at an arbitrary focus, it is important to allow moving the focus 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 $\mathbf{And}(np,np)$ as other constructors work in a similar way.

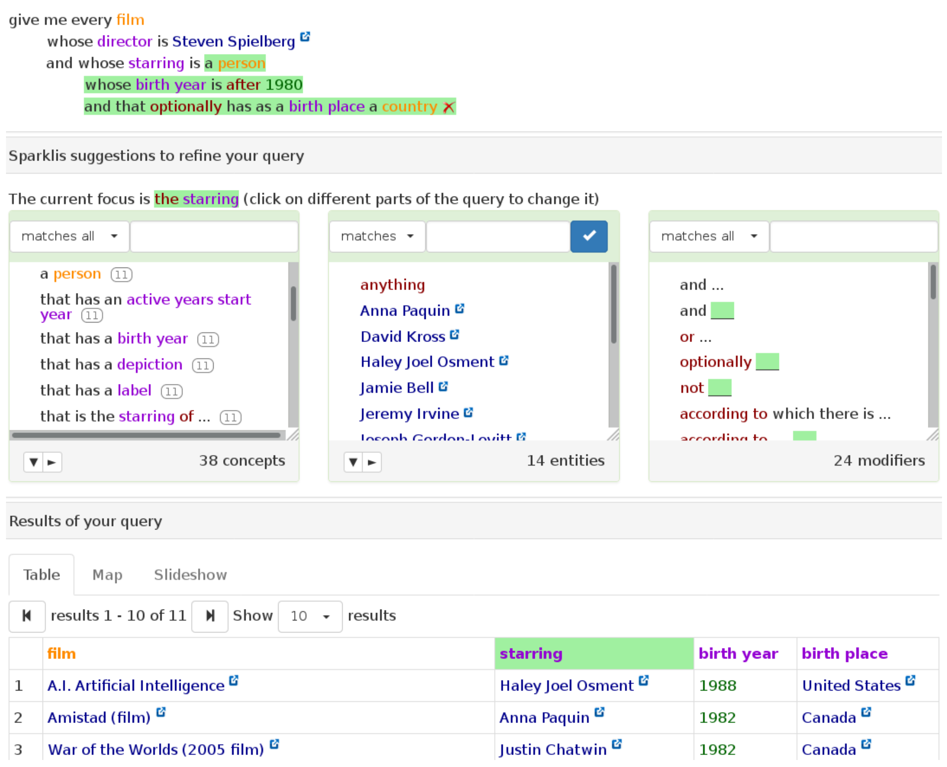
$$\begin{aligned}
DOWN &:= \mathbf{NP}(\mathbf{And}(np_1,np_2),np') \rightarrow \mathbf{NP}(np_1,\mathbf{And}'_1(np',np_2)) \\
UP &:= \mathbf{NP}(np_1,\mathbf{And}'_1(np',np_2)) \rightarrow \mathbf{NP}(\mathbf{And}(np_1,np_2),np') \\
&\mid\ \mathbf{NP}(np_2,\mathbf{And}'_2(np_1,np')) \rightarrow \mathbf{NP}(\mathbf{And}(np_1,np_2),np') \\
RIGHT &:= \mathbf{NP}(np_1,\mathbf{And}'_1(np',np_2)) \rightarrow \mathbf{NP}(np_2,\mathbf{And}'_2(np_1,np')) \\
LEFT &:= \mathbf{NP}(np_2,\mathbf{And}'_2(np_1,np')) \rightarrow \mathbf{NP}(np_1,\mathbf{And}'_1(np',np_2))
\end{aligned}$$

Theorem 1 (completeness)**.** *The initial zipper $zipper_0 = \mathbf{NP}(np_0,\mathbf{Select}')$ and the above set of transformations is* complete*, assuming transformation $NODE(n)$ is suggested for all nodes of the target RDF graph, and similarly for transformations $DET(d)$, $CLASS(c)$, $PROP(p)$, and $PROP^{-}(p)$.*

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 $\mathbf{NP}(np,\mathbf{Select}')$ is reachable. Figure 3 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). *ID* is the identity transformation. □

The above proof provides an algorithm T for computing the transformation sequence leading to any AST x. The example zipper given in Figure 2 can be reached with the following sequence:

DET(**Some**); *CLASS*(`owl:Thing`); *DOWN*;
PROP^{-}(`dbo:release`); *DOWN*; *DET*(**Some**); *CLASS*(`dbo:Film`); *DOWN*;
PROP(`dbo:director`); *DOWN*; *NODE*(`dbr:Steven_Spielberg`); *UP*;
AND; *PROP*(`dbo:genre`); *DOWN*; *NODE*(`"SF"`); *OR*; *NODE*(`"Fantasy"`); *UP*; *UP*....


give me every film
whose director is Steven Spielberg
and whose starring is a person
whose birth year is after 1980
and that optionally has as a birth place a country ✗

Sparklis suggestions to refine your query

The current focus is the starring (click on different parts of the query to change it)

matches all	matches	matches all
a person (11)	anything	and ...
that has an active years start year (11)	Anna Paquin	and 🟩
that has a birth year (11)	David Kross	or ...
that has a depiction (11)	Haley Joel Osment	optionally 🟩
that has a label (11)	Jamie Bell	not 🟩
that is the starring of ... (11)	Jeremy Irvine	according to which there is ...
	Joseph Gordon-Levitt	according to 🟩
38 concepts	14 entities	24 modifiers

Results of your query

Table Map Slideshow

results 1 - 10 of 11 Show 10 results

	film	starring	birth year	birth place
1	A.I. Artificial Intelligence	Haley Joel Osment	1988	United States
2	Amistad (film)	Anna Paquin	1982	Canada
3	War of the Worlds (2005 film)	Justin Chatwin	1982	Canada

Figure 4. Screenshot of SPARKLIS

In practice, it appears useful to tune transformations so as to minimize the number of interaction steps for users: e.g., moving down after inserting a property, moving up after inserting a node, moving down when inserting a property at a noun phrase, avoiding the insertion of default elements (e.g., determiner **Some**, class `owl:Thing`). Applying those rules reduces the number of steps in the example from 19 to 9:

PROP$^-$(`dbo:genre`); *CLASS*(`dbo:Film`);
PROP(`dbo:director`); *NODE*(`dbr:Steven_Spielberg`);
AND; *PROP*(`dbo:genre`); *NODE*(`"SF"`); *OR*; *NODE*(`"Fantasy"`).

4. Implementation, Applications, and Evaluation

Our approach of authoring based on zippers and transformations is implemented in a functional programming language (OCaml), and has already been used in three applications: SPARKLIS[3] [5], UTILIS [7], and PEW [4]. Those applications target different tasks and formal languages (FL). SPARKLIS targets semantic search with SPARQL; UTILIS targets descriptions of RDF nodes; and PEW targets ontology design and completion with OWL class expressions. All include the conjunctive subset of the intermediate language defined in previous section, and extends it with task-specific constructs. In

[3]Online version at `http://www.irisa.fr/LIS/ferre/sparklis/` with examples and screencasts.

each application, the user interface has three parts (see Figure 4 for a screenshot): the current sentence and focus (top), the suggested transformations (center), and the semantic interpretation of the sentence (bottom). The latter is made of query answers for SPARKLIS, similar nodes for UTILIS, and class instances for PEW. The suggested transformations are selected so as to avoid semantically misformed sentences: e.g., empty results in SPARKLIS or inconsistent ontology in PEW. The suggested transformations are generally grouped in three categories: (a) classes and properties, (b) entities and values, (c) Boolean connectors and other transformations. The focus is highlighted in the sentence, and can be moved freely by clicking the relevant parts of the sentence.

User studies have been conducted in all three applications, and are reported in the related papers. SPARKLIS is online since April 2014, and more than 100,000 navigation steps have been performed by more than 1000 unique users. It has recently been officially adopted as a SPARQL query builder by two French institutions: Persée[4] and INIST[5]. 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. Another user study comparing PEW to Protégé has demonstrated promising results in terms of quantity, precision, and recall of the produced axioms, and in terms of usability. A notable result is the increase in recall, from 24% with Protégé to 56% with PEW, where 100% would mean a complete OWL formalization of the domain knowledge for the selected OWL fragment. Overall, our approach requires less background knowledge, and is more productive and safe compared to the direct use of formal languages or to the use of Semantic Web tools such as Protégé. The main difficulty appears to be in the understanding of the focus, and its impact on the suggested transformations. Some training is necessary for new users but most of them progress rapidly.

5. Conclusion and Perspectives

We have presented an alternative solution to the problem of CNL authoring. Like the auto-completion solution, it guides the user in the authoring process, and prevents the construction of misformed sentences. The novelty lies in the fact that the system suggestions are abstract syntax transformations rather than concrete words. As a consequence, the translation between the concrete and the abstract syntaxes goes backward, i.e. from the abstract to the concrete through a verbalization process that also applies to the suggested transformations. There are two benefits: (1) constructed sentences are complete at any step of the authoring process, allowing for intermediate results and feedback (*responsiveness*), and (2) editions can apply at any *focus*, i.e. any position or part of the sentence (*flexibility*). We have shown how to use Huet's zippers on the abstract syntax in order to represent the current *focus* and efficiently apply transformations. An interesting perspective is to implement this approach in Grammatical Framework (GF), in order to offer an alternative authoring process in applications. GF already offers the definition of abstract syntax, and verbalization (called *linearization* in GF). The definitions of contexts and zippers can be automatically derived from the abstract syntax, and zipper transformations can be naturally defined as functions on zippers.

[4]`http://data.persee.fr/`

[5]`https://www.loterre.fr/category/explorer/`

Acknowledgements. We thank the anonymous reviewers for their valuable feedback, and insightful suggestions, as they helped to improve this paper.

References

[1] ABBOTT, M., ALTENKIRCH, T., MCBRIDE, C., AND GHANI, N. ∂ for data: differentiating data structures. *Fundamenta Informaticae 65*, 1 (2005), 1–28.

[2] DOWTY, D. R., WALL, R. E., AND PETERS, S. *Introduction to Montague Semantics*. D. Reidel Publishing Company, 1981.

[3] FERRÉ, S. Bridging the gap between formal languages and natural languages with zippers. In *Extended Semantic Web Conf. (ESWC)* (2016), H. Sack et al., Eds., Springer, pp. 269–284.

[4] FERRÉ, S. Semantic authoring of ontologies by exploration and elimination of possible worlds. In *Int. Conf. Knowledge Engineering and Knowledge Management* (2016), LNAI 10024, Springer.

[5] FERRÉ, S. Sparklis: An expressive query builder for SPARQL endpoints with guidance in natural language. *Semantic Web: Interoperability, Usability, Applicability 8*, 3 (2017), 405–418.

[6] GUY, S., AND SCHWITTER, R. Architecture of a web-based predictive editor for controlled natural language processing. In *Controlled Natural Language* (2014), B. Davis, K. Kaljurand, and T. Kuhn, Eds., Springer, pp. 167–178.

[7] HERMANN, A., FERRÉ, S., AND DUCASSÉ, M. An interactive guidance process supporting consistent updates of RDFS graphs. In *Int. Conf. Knowledge Engineering and Knowledge Management (EKAW)* (2012), A. ten Teije et al., Eds., LNAI 7603, Springer, pp. 185–199.

[8] HITZLER, P., KRÖTZSCH, M., AND RUDOLPH, S. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, 2009.

[9] HUET, G. Functional pearl – the zipper. *J. Functional Programming 7*, 5 (1997), 549–554.

[10] KALJURAND, K., AND KUHN, T. A multilingual semantic wiki based on attempto controlled english and grammatical framework. In *The Semantic Web: Semantics and Big Data*. Springer, 2013, pp. 427–441.

[11] KAUFMANN, E., AND BERNSTEIN, A. Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases. *J. Web Semantics 8*, 4 (2010), 377–393.

[12] KUHN, T. A survey and classification of controlled natural languages. *Computational Linguistics* (2013).

[13] LOPEZ, V., UREN, V. S., SABOU, M., AND MOTTA, E. Is question answering fit for the semantic web?: A survey. *Semantic Web 2*, 2 (2011), 125–155.

[14] NGOMO, A.-C. N., BÜHMANN, L., UNGER, C., LEHMANN, J., AND GERBER, D. Sorry, I don't speak SPARQL: translating SPARQL queries into natural language. In *WWW* (2013), pp. 977–988.

[15] PALMAZ, S., CUADROS, M., AND ETCHEGOYHEN, T. Statistically-guided controlled language authoring. In *Controlled Natural Language* (2016), B. Davis, G. J. Pace, and A. Wyner, Eds., Springer, pp. 37–47.

[16] RANTA, A. Grammatical framework. *Journal of Functional Programming 14*, 02 (2004), 145–189.

[17] SPARQL 1.1 query language, 2012. W3C Recommendation.

[18] TURNER, D. Functional programming and proofs of program correctness. In *Tools and Notions for Program Construction: An Advanced Course*, D. Néel, Ed. Cambridge University Press, 1982, pp. 187–209.