Interactive Multilingual Web Applications with Grammatical Framework

Moisés Salvador Meza Moreno and Björn Bringert

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
meza@student.chalmers.se, bringert@chalmers.se

Abstract. We present an approach to multilingual web content based on multilingual grammars and syntax editing for a controlled language. Content can be edited in any supported language and it is automatically kept within a controlled language fragment. We have implemented a web-based syntax editor for Grammatical Framework (GF) grammars which allows both direct abstract syntax tree manipulation and text input in any of the languages supported by the grammar. With this syntax editor and the GF JavaScript API, GF grammars can be used to build multilingual web applications. As a demonstration, we have implemented an example application in which users can add, edit and review restaurants in English, Spanish and Swedish.

1 Introduction

Current multilingual web applications store a separate version of their content for each language. It is difficult to keep the information consistent and, in some cases, content available in one language is not provided in another. Adding a new language to the application requires translation of the available content from one of the existing languages to the new language.

We suggest a different approach to multilingual web applications, where the content is defined by a multilingual grammar and is created through syntax editing or parsing. Content created by a user who uses one language is automatically available in all the other languages supported by the grammar, and the content is consistent at all times. When the grammar is extended to cover a new language, all existing content is automatically available in that language.

To demonstrate this approach to multilinguality we implemented "The Restaurant Review Wiki", a web-based multilingual application in which users can add, edit and review restaurants in English, Spanish and Swedish. It uses GF grammars and the GF JavaScript API to provide multilinguality.

2 Grammatical Framework

Grammatical Framework (GF) [1] is a type-theoretical grammar formalism. GF grammars can describe both formal and natural languages and consist of an abstract syntax and at least one concrete syntax. The abstract syntax defines the

scope of the grammar, i.e. all the expressions that can be built from it. The concrete syntax defines how the constructs in the abstract syntax are represented in a particular language. GF grammars can be multilingual, each language in the grammar having a separate concrete syntax. For any given grammar, GF provides parsing (going from a concrete to the abstract syntax) and linearization (going from the abstract to a concrete syntax). GF supports dependently typed and higher-order abstract syntax. These features are used, for example, to express conditions of semantic well-formedness. However, they are not used in this article since they are not supported in the implementations described.

GF includes a Resource Grammar Library [2] which defines the basic grammar of (currently) eleven languages. For each language, the Resource Grammar Library provides the complete morphology, a lexicon of approximately one hundred of the most important structural words, a test lexicon of approximately 300 content words, a list of irregular verbs and a substantial fragment of the syntax. The Resource Grammar Library has an API (Application Programming Interface) which allows the user to implement grammars for these languages easily. The API also provides tools to extend the resource grammars, for example, new words can be added to the lexicon. GF is freely available[1] and is distributed under the GNU General Public License (GPL).

2.1 An Example Grammar

To better explain GF grammars, consider a very small grammar that describes simple restaurant reviews. The abstract syntax defines what can be said in the grammar in terms of categories (**cat**) and functions (**fun**). In the example grammar, the abstract syntax (Figure 1) has four categories: Phrase (the start category), Item, Demonym and Quality. It also has some functions that construct terms in these categories. For example, the function *itemIs* takes an Item and a Quality as arguments and produces a Phrase, and an Item can be either *restaurant* or *food*. Examples of abstract terms produced by this abstract syntax are *itemIs* (*qualItem mexican food*) (*very good*) and *itemIs restaurant expensive*.

```
abstract Restaurant = {
  flags startcat = Phrase;
  cat Phrase; Item; Demonym; Quality;
  fun itemIs            : Item → Quality → Phrase;
      restaurant, food  : Item;
      qualItem          : Demonym → Item → Item;
      italian, mexican  : Demonym;
      very              : Quality → Quality;
      good, bad, cheap, expensive : Quality;
}
```

Fig. 1. Abstract syntax for the example grammar.

[1] `http://digitalgrammars.com/gf/`

The concrete syntax specifies how the different abstract syntax terms are expressed in a particular language. There is a linearization type (**lincat**) for every category in the abstract syntax. The linearization type is the type of the concrete syntax terms produced for the abstract syntax terms in a category. Similarly, there is a linearization definition (**lin**) for every function in the abstract syntax. A linearization definition is a function from the linearizations of the arguments of an abstract syntax function to a concrete syntax term.

Figure 2 shows the English concrete syntax for the example grammar. The linearization type for all categories is $\{s : \mathsf{Str}\}$, that is, a record with a single field s of type Str (string). The linearization of the function *restaurant* is the concrete syntax term $\{s = \text{“restaurant”}\}$. The linearization of *itemIs* makes use of the linearizations of its argument terms of type Item and Quality. The linearization of the abstract syntax term *itemIs restaurant expensive* is the string “the restaurant is expensive”.

```
concrete RestaurantEng of Restaurant = {
  lincat Phrase, Item, Demonym, Quality = {s : Str};
  lin itemIs i q    = {s = "the" ++ i.s ++ "is" ++ q.s};
      restaurant    = {s = "restaurant"};
      food          = {s = "food"};
      qualItem d i  = {s = d.s ++ i.s};
      italian       = {s = "Italian"};
      mexican       = {s = "Mexican"};
      very q        = {s = "very" ++ q.s};
      good          = {s = "good"};
      bad           = {s = "bad"};
      cheap         = {s = "cheap"};
      expensive     = {s = "expensive"};
}
```

Fig. 2. English concrete syntax for the example grammar.

Figure 3 shows the Spanish concrete syntax for the example grammar. This concrete syntax is more complex because Spanish nouns have an inherent gender (masculine or feminine). Adjectives are inflected according to the gender of the noun they modify and the form of the definite article depends on the gender of the noun it modifies. Thus the category Item has a linearization type $\{s : \mathsf{Str}; g : \mathsf{Gender}\}$. In addition to the string field s, the record has a field g of type Gender, either Masc or Fem. The categories Demonym and Quality have a linearization type $\{s : \mathsf{Gender} \Rightarrow \mathsf{Str}\}$. The field s is here a function from Gender to Str. Some helper functions (**oper**) are also defined. For example, the function *adjective* takes a Str and returns a record of type $\{s : \mathsf{Gender} \Rightarrow \mathsf{Str}\}$. The abstract syntax term *itemIs* (*qualItem mexican food*) (*very good*) is linearized to “la comida mexicana es muy buena”. If we replace the feminine noun *food* with the masculine noun *restaurant* the linearization changes to “el restaurante mexicano es muy bueno”.

```
concrete RestaurantSpa of Restaurant = {
  lincat Phrase              = {s : Str};
         Item                = {s : Str; g : Gender};
         Demonym, Quality    = {s : Gender ⇒ Str};
  lin itemIs i q     = {s = defArt ! i.g ++ i.s ++ "es" ++ q.s ! i.g};
      restaurant     = {s = "restaurante"; g = Masc};
      food           = {s = "comida"; g = Fem};
      qualItem d i   = {s = i.s ++ d.s ! i.g; g = i.g};
      italian        = adjective "italiano";
      mexican        = adjective "mexicano";
      very qual      = {s = \\g ⇒ "muy" ++ qual.s ! g};
      good           = adjective "bueno";
      bad            = adjective "malo";
      cheap          = adjective "barato";
      expensive      = adjective "caro";
  param Gender = Masc | Fem;
  oper defArt : Gender ⇒ Str = table {Masc ⇒ "el"; Fem ⇒ "la"};
       adjective : Str → {s : Gender ⇒ Str} =
         λx → {s = table {Masc ⇒ x; Fem ⇒ Predef.tk 1 x + "a"}};
}
```

Fig. 3. Spanish concrete syntax for the example grammar.

To write the Spanish concrete syntax, the grammar writer had to take into account the morphological and syntactic features of the Spanish language. Even in this simple example, gender had to be considered; imagine a grammar in which number plus case is also involved, or polarity, or verb conjugation, or all of them at once. The larger the scope of the grammar, the harder it gets to properly handle the features of a language. That is why GF's Resource Grammar Library was implemented: to define the low-level morphological and syntactic rules of languages and allow grammar writers to focus on the domain-specific semantic and stylistic aspects. The idea is that if a grammar uses the Resource Grammar Library in a type correct way, it will produce grammatically correct output. The grammar writer still has to know the target language and the application domain in order to get the semantics and pragmatics right, since the grammar library only handles syntax and morphology. Figure 4 shows a Spanish concrete syntax for the example grammar which uses the Resource Grammar Library. The categories Phrase, Item, Demonym and Quality have the linearization types Phr (phrase), CN (common noun), A (one-place adjective) and AP (adjectival phrase), respectively. All linearizations use functions from the resource grammar, such as $mkN : \mathsf{Str} \to \mathsf{N}$, $mkA : \mathsf{Str} \to \mathsf{A}$ and $mkNP : \mathsf{Det} \to \mathsf{N} \to \mathsf{NP}$.

3 Syntax Editing

A *syntax editor* (also known as *syntax-directed editor*, *language-based editor*, or *structure editor*) lets the user edit documents by manipulating their underlying

```
concrete RestaurantSpaRes of Restaurant = open SyntaxSpa, ParadigmsSpa in {
  lincat Phrase = Phr; Item = CN; Demonym = A; Quality = AP;
  lin itemIs i q    = mkPhr (mkCl (mkNP defSgDet i) q);
      restaurant    = mkCN (mkN "restaurante");
      food          = mkCN (mkN "comida");
      qualItem d i  = mkCN d i;
      italian       = mkA "italiano";
      mexican       = mkA "mexicano";
      very qual     = mkAP very_AdA qual;
      good          = mkAP (mkA "bueno");
      bad           = mkAP (mkA "malo");
      cheap         = mkAP (mkA "barato");
      expensive     = mkAP (mkA "caro");
}
```

Fig. 4. Spanish concrete syntax using the resource grammar library.

structure. Such editors can be constructed for any type of structured document, for example computer programs [3], or structured text documents [4].

In the context of GF, a syntax editor lets the user manipulate abstract syntax terms for a particular grammar, while displaying its linearization(s). Syntax editing with GF grammars is described in more detail by Khegai et al. [5]. To explain GF syntax editing we will make use of the grammar described in Section 2.1. There are two kinds of abstract syntax terms: complete terms, e.g. *itemIs restaurant good* and incomplete terms, e.g. *itemIs food* ?. A question mark in an incomplete term is a *metavariable*, i.e. a non-instantiated term. The metavariable in the incomplete term *itemIs food* ? is of type Quality. Syntax editing starts with a single metavariable and it is refined step-by-step until the desired complete term is constructed.

4 GF JavaScript Syntax Editor

This is a syntax editor written in JavaScript that can be used in any JavaScript enabled web browser. This allows the syntax editor to be embedded into web applications. It can also be used as a complete application by itself, for example, to explore, debug or test GF grammars interactively.

4.1 User Interface

The editor interface contains six panels (Figure 5):

Abstract syntax tree panel Shows a tree representation of the abstract syntax term being edited. Selecting a node will highlight both the node in this panel and its corresponding linearization(s) in the linearization panel.

Linearization panel Shows the linearizations of the current abstract syntax term in all the available concrete syntaxes. A string representation of the

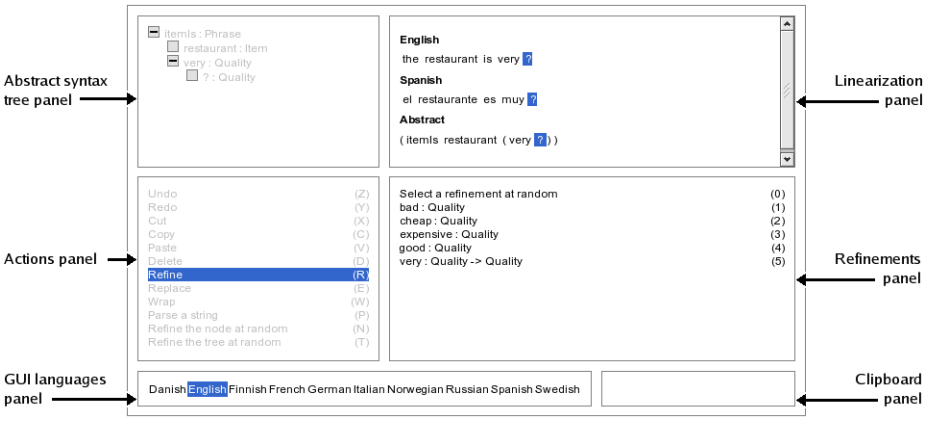

Fig. 5. GF JavaScript syntax editor.

abstract syntax term is also shown. Clicking on a word in a linearization will select the corresponding node in the tree shown in the abstract syntax tree panel. Metavariables are linearized as question marks.

Actions panel Used to show the actions available for the selected node (see Section 4.2). Actions not available for the selected node are grayed out.

Refinements panel Used to show the available refinements or wrappers for the selected node whenever the "Refine" or "Wrap" action is selected.

GUI languages panel Used to show and select the different languages available for the GUI (Graphical User Interface). Currently, three languages are supported: English, Spanish and Swedish. The goal is to support all the languages in GF's Resource Grammar Library. This interface localization is implemented using the approach described in Section 5.2.

Clipboard panel Used to show the name and type of the term currently stored in the clipboard. The clipboard only holds one term at any given time.

4.2 Syntax Editing Actions

There are a number of actions that can be performed on abstract syntax terms. Some of the actions require no further explanation, among those we find: *Undo*, *Redo*, *Cut*, *Copy* and *Paste*. Some of the actions can be easily explained: *Delete* replaces an instantiated term with a metavariable, *Replace* is equivalent to *Delete* followed by *Refine*, except that it is treated as a single action in the edit history and *Refine the node at random* and *Refine the tree at random* respectively instantiate every metavariable in the subtree rooted at the selected node and the entire abstract syntax tree with type-correct objects selected at random. Finally, the following actions deserve a more in depth description:

Refine Replaces a metavariable with a function of the appropriate type. The arguments of the function will all be metavariables. To refine a metavariable

of type Phrase (Figure 6(a)) we need to choose one function from those that have the return type Phrase. Only the function *itemIs* : Item → Quality → Phrase fits this requirement. This refinement will yield a term of the form *itemIs* ?? where the metavariables are of type Item and Quality (Figure 6(b)).

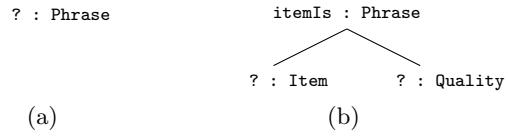

Fig. 6. Refining a metavariable of type Phrase.

Wrap Replaces an instantiated term of type T with a function which has at least one argument of type T and a return type T. The original term is used as the child corresponding to the first argument of type T; the remaining children will be metavariables. In the example grammar, any term of type Quality can be wrapped with the function *very* : Quality → Quality. Wrapping the term *good* of type Quality, shown in Figure 7(a), with the function *very* (Figure 7(b)) results in the term *very good* of type Quality (Figure 7(c)). There is one exception: the top level node can be wrapped by any function which has at least one argument of type T regardless of its return type.

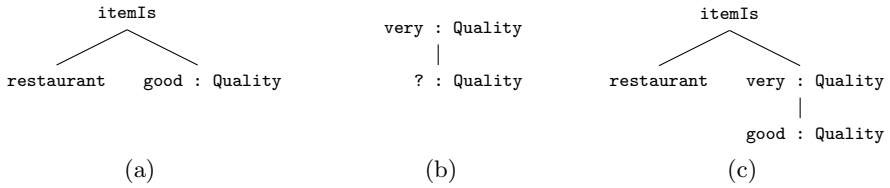

Fig. 7. Wrapping the abstract term *good*.

Parse a string Prompts the user for a string and tries to generate a type-correct subtree by parsing it. On success, the node is instantiated with the resulting subtree. GF grammars can be ambiguous, i.e. two abstract terms can have the same linearization. When parsing an ambiguous string, GF returns a list of abstract terms. In the syntax editor, the different trees produced when parsing an ambiguous string are displayed in the refinements panel so that the user can select one.

4.3 Implementation

We have implemented a GF JavaScript API that allows parsing, linearization, type-annotation of meta-variables, and abstract syntax tree serialization and

deserialization to be done in JavaScript applications. This code is based on the existing GF JavaScript linearization implementation, which was originally used for output generation in GF-generated VoiceXML applications [6]. We have extended it with parsing functionality, by using the active MCFG parsing algorithm described by Burden and Ljunglöf [7].

The GF JavaScript API is now essentially an interpreter for PGF (Portable Grammar Format) [8]. PGF is a low-level format for type-theoretical grammars, and the main target of the GF grammar compiler. The GF grammar compiler has been extended to translate the PGF grammars it produces into a JavaScript representation, which is used by the GF JavaScript API. The JavaScript representation, which is isomorphic to the subset of PGF needed for type-checking, parsing and linearization, is used instead of the standard PGF form in order to avoid the extra computation needed to read PGF files directly in JavaScript.

On top of this API, the syntax editor implements the syntax editing actions, and facilities for supporting the editor user interface. One interesting addition is the support for associating parts of the linearization output with the abstract syntax sub-terms which generated them. Each node in the abstract syntax tree is given an identifier which encodes the path from the root of the tree to the given node. The linearization algorithm has been modified to tag each token that results from linearizing a node with that node's identifier. As a consequence, each token in the sequence of tokens produced by linearizing an abstract syntax tree will be tagged with the identifier of the node that produced it, and the identifiers of all its parent nodes. When the user selects a node in the tree, all tokens tagged with that node's identifier are highlighted. When a token is selected, the deepest node (i.e. longest identifier) which it is tagged with is highlighted.

5 Example Application: The Restaurant Review Wiki

The GF JavaScript API and the syntax editor described in Section 4 can be used together to build a multilingual web application. This section describes the Restaurant Review Wiki, a small demo application developed using these tools.

5.1 Description

The Restaurant Review Wiki is a restaurant database that allows users to add restaurants and reviews and view and edit the information in three languages (English, Swedish and Spanish). It is available online[2].

Users can add new restaurants and edit the information about existing restaurants. For each restaurant there is some basic information, such as address and cuisine, entered using standard HTML forms, and reviews which are created and edited by using the syntax editor as shown in Figure 8. The restaurant review grammar used in this application is an extended version of the grammar described in Section 2.1.

[2] `http://csmisc14.cs.chalmers.se/~meza/restWiki/wiki.cgi/`

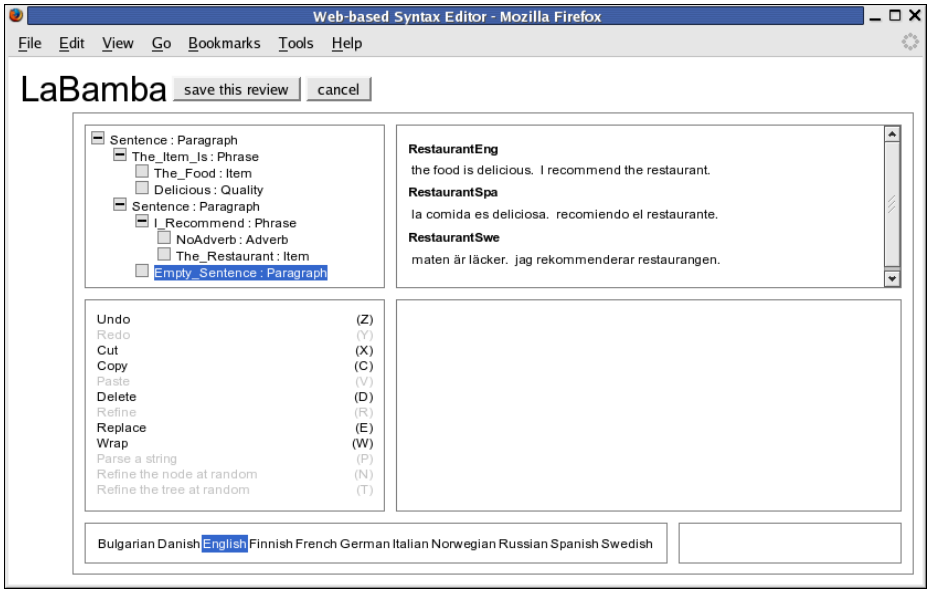

Fig. 8. Review editing page.

When adding a new review, the abstract syntax term in the syntax editor is initially a single metavariable of type Paragraph. The user edits a review by stepwise refining the tree, by parsing a string or by some combination of these. For example, the user may parse a simple sentence such as "the food is delicious", and then use syntax editing commands to elaborate parts of it.

5.2 Implementation

Instead of storing the text in any language, the abstract syntax representation of the information is stored on the server and it is linearized by the client's browser upon request. The algorithms to linearize abstract syntax trees are efficient and with today's computing power the user should not be affected by delays caused by the linearization of the different multilingual elements of a page. Whenever a page is loaded, a linearizing function is called for every multilingual element in the page. This function takes the HTML element to linearize, a reference to the currently selected language and a grammar as arguments. It extracts the string representation of the abstract syntax term from the element, converts it into an abstract syntax tree, linearizes the tree using the concrete syntax for the currently selected language and stores the linearization in the element.

Two GF grammars are used by this application, one that describes the elements of web pages such as headers, field names, country names, cuisines, etc., and another that describes restaurant reviews.

5.3 Discussion

Advantages Since the multilingual information is stored as its abstract syntax representation, all new content created by users is available for all languages immediately, and it is thereby consistent in all languages. In existing multilingual applications such as Wikipedia, multilingual content is created in parallel. This means that there is a different version of the information for each language and there is no guarantee that the information available for a particular language will be available in another nor that they will be consistent.

Having all the information in an abstract representation of a controlled language makes it possible to perform operations such as querying precisely and efficiently. For example, it should be easy to implement functionality that would let the user search for "cheap Thai restaurants close to the university".

Adding a language to the application means adding a concrete syntax for that language to the grammar. Once the concrete syntax is added, all existing information is automatically available in the new language. There is no need to translate the existing information by hand.

Disadvantages The content that can be created using this approach is limited by the coverage of the grammar. This may be too restrictive and it may prevent users from effectively conveying their ideas through the content they create.

In this version of the application, new content is created by using the syntax editor, either by stepwise refining the abstract syntax tree or by parsing a string. The syntax editor has the advantage of generating content within the coverage of the grammar. The problem is that the editor is not very intuitive and it could be hard to use without training, a situation that could discourage potential users. Creating content by parsing is simple, but, if the user is not familiar with the grammar, producing valid content through parsing might be a difficult task unless the grammar has a very wide coverage.

Multilingual processing is done in the client rather than on the server. A JavaScript GF grammar may be larger than 1 MB, which could be a problem for devices with limited bandwidth or memory, such as PDAs or mobile phones. Also, devices with limited processing power may experience delays caused by the linearization of the multilingual elements in pages. Since the current version does linearization in the client even when viewing existing content, search engines may not be able to index the page using the linearized content.

If an abstract syntax used in an application is changed and the new version is not backwards compatible, it may no longer be possible to linearize the stored abstract syntax terms. If the coverage of the new grammar is a superlanguage of the old one, this problem can be solved by linearizing each stored term with the old grammar and parsing it with the new one.

Doing natural language processing client-side tends to stress the web browser implementations. The current state of web standards compatibility in browsers may lead to inconsistent behavior or performance in some web browsers.

6 Related Work

The Grammatical Framework (GF) provided, up until this point, two different syntax editors. The first provides the full functionality of GF but can only be used in machines that have the full GF system installed [1]. One use of this editor is as an integral component of the KeY formal program verification system [9]. The second, Gramlets [10], provides no parsing and no support for dependent types or higher-order functions but can be run on any machine that has a Java Virtual Machine (JVM) installed or in web browsers which have a JVM plug-in. Our syntax editor is more portable than the previous GF syntax editors, can be more easily integrated into web applications, and compared to Gramlets, it offers more functionality, most notably parsing. The syntax editor does not support the full GF language yet, as it only allows grammars which have no dependent types and no higher-order abstract syntax.

WYSIWYM [11] is a structure editor which displays natural language representations during editing. It now also has a JavaScript implementation[3]. Our editor is driven by a declarative specification of the language structure and generation rules. In WYSIWYM these components are built into the editor, which appears to make it more difficult to use the editor for new applications.

7 Future Work

Dependently Typed and Higher-order Abstract Syntax For the syntax editor to support more advanced grammars, the GF JavaScript API should be extended to implement parsing, type-checking and linearization for grammars with dependently typed and higher-order abstract syntax.

Syntax Editor User Interface New content is created using the syntax editor and, as mentioned before, this is too restrictive and could make users lose interest in the application. There is a need for a more intuitive interface which still guarantees that the content is within the domain of the grammar. One way to make the interface more easy to use is to add *completion*. The idea is to make the editor display a list of possible ways to complete the input that the user is typing, as is done in the GF-based WebALT exercise editor for multilingual mathematical exercises [12].

Server-side Processing Instead of doing the multilingual processing in the client, it could be done on the server. This would be beneficial for devices with limited processing power, memory or bandwidth. Especially linearization of existing content should be off-loaded to the server, as this will also help search engines index the content.

8 Conclusions

We have implemented a syntax editor which provides the basic functionality of the Grammatical Framework (GF) in web browsers. It allows the user to stepwise

[3] `http://www.itri.brighton.ac.uk/projects/WYSIWYM/javademo.html`

create the abstract syntax trees described by a GF grammar through the use of special purpose editing actions, while showing linearizations of the trees in multiple languages. It can be used to test and debug GF grammars, or as a component in multilingual web-based applications.

To demonstrate how the syntax editor can be used to implement multilingual web applications, we also implemented "The Restaurant Review Wiki". It is a multilingual restaurant database in which users can add, edit and review restaurants in three different languages. The approach to multilinguality that we suggest makes all information available simultaneously and consistently for all the supported languages, and adding a new language is only a matter of adding a concrete syntax for that language to the application grammar. Additional work is required to make syntax editing more usable for untrained users, and to ensure that the technique works well in resource-constrained computing devices.

References

1. Ranta, A.: Grammatical Framework: A Type-Theoretical Grammar Formalism. Journal of Functional Programming **14**(2) (March 2004) 145–189
2. Ranta, A.: Grammars as software libraries. In Bertot, Y., Huet, G., Lévy, J.J., Plotkin, G., eds.: From semantics to computer science: essays in honor of Gilles Kahn. Cambridge University Press (2008)
3. Teitelbaum, T., Reps, T.: The Cornell program synthesizer: a syntax-directed programming environment. Commun. ACM **24**(9) (September 1981) 563–573
4. Furuta, R., Quint, V., Andre, J.: Interactively Editing Structured Documents. Electronic Publishing **1**(1) (1988) 19–44
5. Khegai, J., Nordström, B., Ranta, A.: Multilingual Syntax Editing in GF. In Gelbukh, A., ed.: Computational Linguistics and Intelligent Text Processing. Volume 2588 of Lecture Notes in Computer Science. (2003) 199–204
6. Bringert, B.: Rapid Development of Dialogue Systems by Grammar Compilation. In Keizer, S., Bunt, H., Paek, T., eds.: Proceedings of the 8th SIGdial Workshop on Discourse and Dialogue, Antwerp, Belgium (September 2007) 223–226
7. Burden, H., Ljunglöf, P.: Parsing Linear Context-Free Rewriting Systems. In: Proceedings of the Ninth International Workshop on Parsing Technology, Vancouver, British Columbia, Association for Computational Linguistics (2005) 11–17
8. Angelov, K., Bringert, B., Ranta, A.: PGF: A Portable Run-Time Format for Type-Theoretical Grammars. Manuscript, `http://www.cs.chalmers.se/~bringert/publ/pgf/pgf.pdf` (2008)
9. Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software: The KeY Approach. Volume 4334 of LNCS. Springer-Verlag (2007)
10. Johannisson, K., Khegai, J., Forsberg, M., Ranta, A.: From Grammars to Gramlets. In: The Joint Winter Meeting of Computing Science and Computer Engineering, Chalmers University of Technology (2003)
11. Power, R., Scott, D., Evans, R.: What You See Is What You Meant: direct knowledge editings with natural language feedback. In: 13th European Conference on Artificial Intelligence (ECAI 1998). (1998) 677–681
12. Cohen, A., Cuypers, H., Poels, K., Spanbroek, M., Verrijzer, R.: WExEd - WebALT Exercise Editor for Multilingual Mathematical Exercises. In Seppälä, M., Xambo, S., Caprotti, O., eds.: WebALT 2006, First WebALT Conference and Exhibition, Eindhoven, The Netherlands. (January 2006) 141–145