View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace at Tartu University Library

Editing Syntax Trees on the Surface

Peter Ljunglöf
Department of Computer Science and Engineering
University of Gothenburg and Chalmers University of Technology
Gothenburg, Sweden
peter.ljunglof@gu.se

Abstract

We describe a system for interactive modification of syntax trees by intuitive editing operations on the surface string. The system has a graphical interface, where the user can move, replace, add, and in other ways modify, words or phrases. During editing, the sentence is kept grammatical, by automatically rearranging words and changing inflection, if necessary. This is accomplished by combining constraints on syntax trees with a distance measure between trees.

1 Introduction

In this paper we describe the underlying theory of a grammatical editing system, where the actions of the user are interpreted as constraints on the syntax tree. The different editing operations that the user performs on the surface string, are interpreted as constraints on the underlying syntax tree which is never shown to the user. The system then searches for the closest matching tree, in terms of a suitable tree distance measure.

We believe that our editing system is more intuitive and easy to use than a system where the syntax is shown explicitly. Then it can be a useful pedagogical tool for supporting language learning and training, for children with communicative disabilities, and for people learning a second language. We also hope that the ideas can be useful in touch screen devices, as an additional editing layer in text-based applications such as translation, email and chat.

The current system is a pedagogical tool for language learning, and is still work in progress. As of April 2011, there is a functioning demo system which needs more work to be useful for the intended audience.

2 System overview

2.1 No free text input

The editing interaction is purely graphical, which means that the user is not allowed to enter words, phrases or sentences from the keyboard. There are several reasons for this, but the main reason is to avoid problems with words and grammatical constructions that the system doesn't know anything about. Systems that are supposed to handle free text input sooner or later run into problems with unknown words or phrases (Heift, 2001).

Another reason for disallowing free text input is to make the system accessible to people with communicative and/or physical disabilities, or for alternative input methods such as mobile phone touch screens.

2.2 Interacting with the system

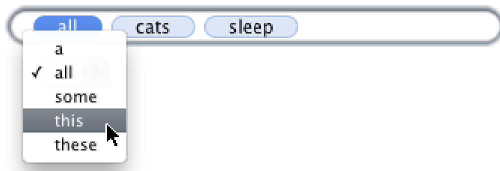
The words that the user is editing are icon-like objects that can be selected, inserted, moved around and deleted. A word is selected by clicking, and the selection can be increased to multi-word phrases. The selected word or phrase has an associated context-menu consisting of similar words or phrases, such as different inflection forms, or synonyms, homonyms, etc. When an item is selected from the context-menu, it replaces the old word or phrase, and if necessary, the nearby words are also modified and rearranged to keep the sentence grammatical.

The user can move the selection to another position in the sentence, and the system will automatically keep the sentence grammatical by rearrangingthe words and change inflection, if necessary. Phrases can be deleted from the sentence by dragging them away. The user can also add or replace words by dragging new words into the sentence. All the time, the sentence will adapt by rearranging and inflecting.

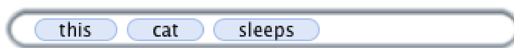
Bolette Sandford Pedersen, Gunta Nešpore and Inguna Skadiņa (Eds.)
NODALIDA 2011 Conference Proceedings, pp. 138–145

2.2.1 Example: Modifying a phrase

Assume that the system starts with the sentence "*all cats sleep*", and the user wants to see the possible alternatives to the determiner "*all*":



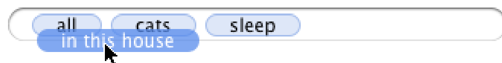
The user now changes "*all*" to the word "*this*" instead, thereby changing the number from plural to singular. Then the system automatically change the inflection of the other words in the sentence, so that it is kept grammatical:



2.2.2 Example: Inserting a phrase

Another alternative is to insert a new phrase into the sentence, by dragging it from a heap of possible phrases:

The system knows where the new phrase can be inserted and shows it by making room for it. On the other hand, the system does not react if the user tries to insert the phrase in an ungrammatical position:



2.3 Implementation

The system consists of three implementation layers. The bottom layer is the GF grammar formalism (Ranta, 2009b). We use GF's multilingual resource grammar to define the different grammar modules (Ranta, 2009a). The surface strings are stored as GF syntax trees, and the GF linearisation algorithm is used for displaying the sentences to the user. We have no use of parsing the sentences, since the syntax trees are already known and there is no free text input.

On top of GF we implement an API for modifying syntax trees by specifying linearisation constraints. The API consists of functions that transform trees to obey the constraints, by using as few transformations as possible. An example of constraints can be that the linearisations of some given tree nodes must come in a certain order (e.g., when the user moves a word to a position between two other words). Another example is that the linearisation of a given node must be of a specified form (e.g., when the user selects a specific word form from the context menu).

The final layer is the graphical interface, which communicates with the API to decide which words can be moved where, and what their contextual menus should contain.

3 Grammatical Framework

GF is a two-level formalism, with an underlying abstract syntax and a surface concrete syntax (Ranta, 2009b). It is a high-level grammar formalism with good support for both multilingual and modular grammar writing (Ranta, 2009a). In this paper we focus on a simplified core language, which every grammar can be compiled into.

3.1 GF abstract syntax

The abstract syntax of a GF grammar consists of a finite number of typed functions. In the general framework, functions can be both higher-order and dependently typed, but most applications only use first-order functions with non-dependent types.

A GF function is declared by giving its type, $f : A_1 \dots A_n \to A$. If $n = 0$, the function has no arguments, and is called a constant. From the function f we can create a *term* of type A by applying it to n terms of type $A_1, \dots, A_n$. In other words, $f(t_1 \dots t_n)$ is a term of type A whenever $t_1, \dots, t_n$ are terms of types $A_1, \dots, A_n$, respectively.

This is similar to how syntax trees are licensed by a context-free grammar, but instead of using nonterminals in the tree nodes, we use function names. In fact, the abstract syntax is equivalent to a context-free grammar without terminal symbols, where the nonterminals correspond to GF types, and where the grammar rules have names. A simple example grammar is shown in Figure 1. Two terms of type S licensed by this grammar are:

$$sleep(npp(all(cat), in(this(house))))$$
$$spp(sleep(all(cat)), in(this(house)))$$

Terms can be drawn as context-free syntax trees, where the nodes contain function symbols instead of nonterminals. The trees corresponding to the example terms above are shown in Figure 2.

$$\begin{array}{rcl}
\mathit{cat}, \mathit{house} & : & \mathrm{N} \\
\mathit{this}, \mathit{all} & : & \mathrm{N} \to \mathrm{NP} \\
\mathit{sleep} & : & \mathrm{NP} \to \mathrm{S} \\
\mathit{in} & : & \mathrm{NP} \to \mathrm{PP} \\
\mathit{npp} & : & \mathrm{NP}, \mathrm{PP} \to \mathrm{NP} \\
\mathit{spp} & : & \mathrm{S}, \mathrm{PP} \to \mathrm{S}
\end{array}$$

Figure 1: Example abstract grammar

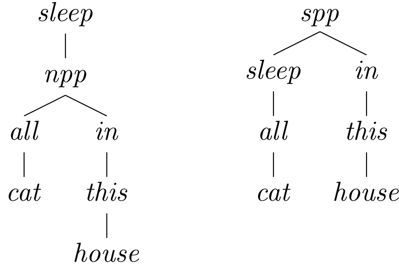


Figure 2: Trees licenced by the grammar

$$\begin{array}{rcl}
\mathit{cat}^\circ & = & \langle \text{“}\mathit{cat}\text{”}\,;\ \text{“}\mathit{cats}\text{”} \rangle \\
\mathit{house}^\circ & = & \langle \text{“}\mathit{house}\text{”}\,;\ \text{“}\mathit{houses}\text{”} \rangle \\
\mathit{this}^\circ(x) & = & \langle \text{“}\mathit{this}\text{”} + x\,!\,1\,;\ 1 \rangle \\
\mathit{all}^\circ(x) & = & \langle \text{“}\mathit{all}\text{”} + x\,!\,2\,;\ 2 \rangle \\
\mathit{sleep}^\circ(x) & = & \langle x\,!\,1 + \langle \text{“}\mathit{sleeps}\text{”};\ \text{“}\mathit{sleep}\text{”} \rangle\,!\,(x!2) \rangle \\
\mathit{in}^\circ(x) & = & \langle \text{“}\mathit{in}\text{”} + x\,!\,1 \rangle \\
\mathit{npp}^\circ(x,y) & = & \langle x\,!\,1 + y\,!\,1\,;\ x\,!\,2 \rangle \\
\mathit{spp}^\circ(x,y) & = & \langle x\,!\,1 + y\,!\,1 \rangle
\end{array}$$

Figure 3: Concrete syntax for the grammar

3.2 GF concrete syntax

The concrete syntax of a GF grammar is a compositional mapping from abstract terms to concrete terms, called the *linearisation*. The concrete terms can be quite complex and consist of strings, finite parameters, recursive records and inflection tables. We do not use the full concrete language in this paper, but instead use a simplified syntax which all GF grammars can be compiled into.

Every GF term is a tuple of strings and integer values. There are two operations; string concatenation and tuple selection:

$$\begin{array}{rcl}
\text{“}s_1 \ldots\text{”} + \text{“}s_2 \ldots\text{”} & = & \text{“}s_1 \ldots s_2 \ldots\text{”} \\
\langle t_1; \ldots; t_k; \ldots; t_n \rangle\,!\,k & = & t_k
\end{array}$$

This term language is similar to multiple context-free grammar (MCFG) (Seki et al., 1991), and indeed every GF grammar can be converted to an equivalent MCFG (Ljunglöf, 2004).

We write t° for the linearisation of t. Compositionality can then be formulated as,

$$f(t_1 \ldots t_n)^\circ \quad = \quad f^\circ(t_1^\circ \ldots t_n^\circ)$$

where f° is the linearisation function corresponding to the abstract function f. The concrete syntax of our example grammar is shown in Figure 3.

3.2.1 Example linearisation

To illustrate the linearisation algorithm, we here give the linearisation of the term $\mathit{sleep}(\mathit{all}(\mathit{cat}))$:

$$\begin{array}{rl}
& \mathit{sleep}(\mathit{all}(\mathit{cat}))^\circ \\
= & \mathit{sleep}^\circ(\mathit{all}^\circ(\mathit{cat}^\circ)) \\
= & \mathit{sleep}^\circ(\langle \text{“}\mathit{all}\text{”} + \mathit{cat}^\circ\,!\,2\,;\ 2 \rangle) \\
= & \mathit{sleep}^\circ(\langle \text{“}\mathit{all}\text{”} + \langle \text{“}\mathit{cat}\text{”};\ \text{“}\mathit{cats}\text{”} \rangle\,!\,2\,;\ 2 \rangle) \\
= & \mathit{sleep}^\circ(\langle \text{“}\mathit{all}\text{”} + \text{“}\mathit{cats}\text{”}\,;\ 2 \rangle) \\
= & \mathit{sleep}^\circ(\langle \text{“}\mathit{all\ cats}\text{”}\,;\ 2 \rangle) \\
= & \langle \langle \text{“}\mathit{all\ cats}\text{”}; 2 \rangle\,!\,1\, + \\
& \quad \langle \text{“}\mathit{sleeps}\text{”};\ \text{“}\mathit{sleep}\text{”} \rangle\,!\,(\langle \text{“}\mathit{all\ cats}\text{”}; 2 \rangle!2) \rangle \\
= & \langle \text{“}\mathit{all\ cats}\text{”} + \langle \text{“}\mathit{sleeps}\text{”};\ \text{“}\mathit{sleep}\text{”} \rangle\,!\,2 \rangle \\
= & \langle \text{“}\mathit{all\ cats}\text{”} + \text{“}\mathit{sleep}\text{”} \rangle \\
= & \langle \text{“}\mathit{all\ cats\ sleep}\text{”} \rangle
\end{array}$$

Note that the numbers have different meaning in different linearisation terms: Both 2's in all° denotes plural, one for selecting the plural form of the noun and the other for remembering that the resulting NP is in plural. On the other hand, the 2 in sleep° is used to select the number of the NP, and then use that number to select the corresponding verb form.

4 Trees and tree editing

Formally, an ordered tree is a connected directed acyclic non-empty graph, in which every node $v \in V$ (where V denotes the set of nodes) has exactly one parent node $\uparrow v$, except the root node which has no parent. Furthermore, there is a precedence relation ($\prec$) defined on sibling nodes.

Each abstract GF term t is a tree where each node v has a label $\widehat{v}$. The label values are GF functions. We write v° for the linearisation of the subtree rooted at v; i.e., $v^\circ = \widehat{v}^\circ(v_1^\circ \dots v_n^\circ)$ when $v_1 \dots v_n$ are the children of v. Note that this is only meaningful if the tree is type-correct.

An example tree representing the term $t_\mathsf{c} = \mathit{sleep}(\mathit{all}(\mathit{cat}))$ consists of the nodes a, b and c, where $\widehat{a} = \mathit{sleep}$, $\widehat{b} = \mathit{all}$, $\widehat{c} = \mathit{cat}$, $a = \uparrow b$, and $b = \uparrow c$.

4.1 Tree edit distance

The *tree edit distance* is a distance measure between trees (Tai, 1979). It is a modification of the well-known Levenshtein string edit distance; the distance between two trees is the number of edit operations required to transform one of them into the other. The allowed operations are insertion, deletion and replacement:

- $\mathit{insert}(v, f, p, j, k)$ inserts a new node v with label f as the jth child of the node p. Furthermore, the new node becomes the parent of p's existing children j to $k-1$.
- $\mathit{delete}(v)$ removes the node v. All children of v become children of v's parent node.
- $\mathit{replace}(v, f)$ replaces the label of v with f.

Note that the resulting tree after an editing operation is not guaranteed to be type-correct. In fact, for deletions and insertions, we always need at least two operations to get a new type-correct tree.

4.2 Constrained linearisation

In GF, not all strings in a linearisation of a subtree node have to be used in the linearisation of the full tree. In the example grammar, cat° contains two strings, but only one of them is used in $t = \mathit{sleep}(\mathit{all}(\mathit{cat}))^\circ$. In this paper we need to talk about only the parts of a linearisation that are used, and for this purpose we define the *constrained linearisation* $[\![v]\!]_t$ of a subtree node v in a tree t. The formal definition is a bit complex, but the intuition is that $[\![v]\!]_t$ consists of the strings in v° that are actually used when calculating t°. For the example tree t_c with the node c representing the child, we get the constrained linearisation $[\![c]\!]_{t_\mathsf{c}} = \langle \textit{“cats”} \rangle$.

4.3 Constraints for automatic tree editing

Each GF grammar rule $f : B_1 \dots B_n \to A$ can be seen as a constraint on f-labeled nodes and its children. Checking that a tree is grammatical according to the grammar, which in GF is the same as checking that the tree is type-correct, can then be implemented as a constraint satisfaction problem (Sulzmann and Stuckey, 2008). Furthermore, when we formulate the grammar as constraints on trees, we can add additional constraints for specifying in more detail how our intended tree should look like.

By using tree constraints and the notion of tree edit distance, we can describe a system for interactive tree editing. The system starts with a grammatical tree, and the user specifies additional constraints on the tree. Then the system searches for the closest grammatical tree (in terms of tree edit distance) that meets the constraints. This continues until the user is satisfied.

This approach lifts the level of tree editing from procedural to declarative: the user does not have to think about how to modify the tree, but instead what the tree should look like. First we have structural constraints on the tree:

- We can state that a node should be in the tree, $v \in V$, or should not be in the tree, $v \notin V$.
- We can state properties about node labels, $\widehat{v} = f$, and node parents, $\uparrow v = v'$; as well as the order between node siblings, $v \prec v'$.

Since our final goal is to allow for editing directly on the concrete surface strings, we also need some linearisation constraints:

- We can specify that the (constrained) linearisation of a node should be, or should not be, a string (tuple): $[\![v]\!] = s$, resp. $[\![v]\!] \neq s$.

  A special case is $\neg[\![v]\!]$, meaning that the node is not realised in the final sentence; this is true either if it linearises to the empty string: $[\![v]\!] = \epsilon$, or if the node is removed completely: $v \notin V$.
- We can specify a linear precedence constraint: $[\![v]\!] \prec [\![v']\!]$ means that the rightmost

word in $[\![v]\!]$ is adjacent to the leftmost word in $[\![v']\!]$; it also implies that both linearisations are non-empty.

Two special cases are $\epsilon \prec [\![v]\!]$ and $[\![v]\!] \prec \epsilon$, meaning that the linearisation comes first resp. last in the sentence.

4.3.1 Example: Modifying a phrase

The context-menu example, from section 2.2.1, can be explained like this. Assume that we start with the following tree t_{c}:

$$t_{\mathsf{c}} = \mathit{sleep}(\mathit{all}(\mathit{cat}))$$
$$t_{\mathsf{c}}^{\circ} = \text{“\textit{all cats sleep}”}$$

This tree has the nodes a, b, c with the labels sleep, all, cat, respectively. Now we want to say that the second word (whose corresponding node is c) should be in its singular form. This can be specified by the constraint $[\![c]\!] = \text{“\textit{cat}”}$. The system can then apply the tree editing operations to search for the closest type-correct tree t'_{c} which meets the constraint. In this case we need only one operation: we rename the b node from all to this:

$$t'_{\mathsf{c}} = \mathit{sleep}(\mathit{this}(\mathit{cat}))$$
$$t'^{\circ}_{\mathsf{c}} = \text{“\textit{this cat sleeps}”}$$

4.3.2 Example: Inserting a phrase

Our second example, introduced in section 2.2.2, is when the user wants to insert a prepositional phrase $t_{\mathsf{p}} = \mathit{in}(\mathit{this}(\mathit{house}))$ into the tree t_{c}. First we encode the whole subtree t_{p} as structural constraints:

$$\widehat{d} = \mathit{in} \qquad \uparrow e = d$$
$$\widehat{e} = \mathit{this} \qquad \uparrow f = e$$
$$\widehat{f} = \mathit{house}$$

Now we can specify where t_{p} should be inserted by giving linearisation constraints:

- If we state that the phrase should come right after *“sleeps”*, $[\![a]\!] \prec [\![d]\!]$, the system needs to first insert a spp-labeled node above a and then insert d as the 2nd child:

  $$t_{\mathsf{cp}} = \mathit{spp}(\mathit{sleep}(\mathit{all}(\mathit{cat})), \mathit{in}(\mathit{this}(\mathit{house})))$$
  $$t_{\mathsf{cp}}^{\circ} = \text{“\textit{all cats sleep in this house}”}$$

- If we instead state that the phrase should come directly before “sleeps”, $[\![d]\!] \prec [\![a]\!]$, the system inserts an npp-labeled node below a, and inserts d as the 2nd child:

  $$t'_{\mathsf{cp}} = \mathit{sleep}(\mathit{npp}(\mathit{all}(\mathit{cat}), \mathit{in}(\mathit{this}(\mathit{house}))))$$
  $$t'^{\circ}_{\mathsf{cp}} = \text{“\textit{all cats in this house sleep}”}$$

4.4 Fine-tuning the search

When the grammar is large, there might be several possible syntax trees that are equally close to the original tree. One possible solution to this problem is to use a more fine-grained distance measure, where the cost of the editing operations depend on the nodes and the labels that are involved.

If the grammar used in example 4.3.1 contains the singular determiner each in addition to this, then there will be two possible solution trees: $\mathit{sleep}(\mathit{this}(\mathit{cat}))$ and $\mathit{sleep}(\mathit{each}(\mathit{cat}))$. Our solution is to augment the grammar with distance values between different functions. In this case the grammar could state that replacing $\mathit{all} \mapsto \mathit{each}$ is cheaper than $\mathit{all} \mapsto \mathit{this}$, to force the resulting tree to be $\mathit{sleep}(\mathit{each}(\mathit{cat}))$.

We can introduce similar costs for deleting and inserting nodes; so that some functions prefer some other functions as parents, or siblings. This could be used, e.g., for PP attachment problems when inserting new phrases.

5 Syntactic editing of the surface string

Now we are ready to get rid of the syntax trees altogether, and introduce syntactic editing operations directly on the surface string. Our final goal is to implement a syntactic editor where the user does not need any knowledge of syntax trees. Therefore the text is presented to the user as a sequence of words, and in this section we define intuitive editing operations on the words.

To implement these operations, we only make use of three GUI “gestures”: *select-click*, *context-click* and *drag*. In a 2-button mouse interface, they are commonly implemented by left-click, right-click, and click-and-hold. In a touch-screen interface, they can be implemented by touch-and-release, touch-and-hold, and touch-and-drag; but there are of course other possibilities.

5.1 Editing operations

Since the user only modifies the surface string, we need a way of translating surface editing operations onto the underying syntax tree. We use the fact that in GF, each surface word belongs to one and only one node in the syntax tree. So, when the user makes a gesture on a word $w \in [\![v]\!]_t$, we interpret it as a gesture on the underlying node v.

During editing, there is an information state consisting of the current tree, and a single node which is called *the selected node* v^*. The selected

node is displayed to the user by highlighting the words in $[\![v^*]\!]_t$. Sometimes there are other nodes v having the same linearisation, $[\![v]\!]_t = [\![v^*]\!]_t$. In that case we always select the *maximal* node, such that $[\![v^*]\!]_t \neq [\![\uparrow v^*]\!]_t$ always holds. The nature of GF grammars ensures that there always exist a unique maximal node.

5.1.1 Selecting a phrase

There are two possibilities when the user select-clicks a word w:

- If the word is unselected, $w \notin [\![v^*]\!]_t$, or if all words in the sentence are selected, the interpretation is that the user wants to start over, and select another node v such that $w \in [\![v]\!]_t$. The node v will be the maximal node with the minimal linearisation covering w. By "minimal linearisation" we mean that there is no descendant v' such that $w \in [\![v']\!]_t \neq [\![v]\!]_t$

- If the word is already selected, $w \in [\![v^*]\!]_t$, the interpretation is that the user wants to increase the selection. We do this by selecting the closest maximal ancestor v such that $[\![v]\!]_t \neq [\![v^*]\!]_t$.

Phrases can also be selected by context-clicking and dragging; if the user performs an operation on an unselected word, its covering node becomes selected before the operation is performed.

5.1.2 Displaying a context menu

When the user context-clicks a word w, the system displays a modification menu for the selected node v^*. Let $s = [\![v^*]\!]_t$ be the currently highlighted phrase.

The modification menu is calculated like this: We search for nearby trees satisfying the constraint $[\![v^*]\!] \neq s$, i.e., so that v^* is linearised different from the current linearisation. For each of these trees, we display a menu item consisting of its linearisation of v^*. If there are no such alternative linearisations, increase the selection and try again.

When the user selects a menu item, the current tree is replaced by the corresponding new tree. The selected node v^* remains selected. The example in section 4.3.1 shows what happens when the user selects the menu item *"cat"* for the selected word *"cats"*.

5.1.3 Deleting a phrase

The user can delete the selected phrase by dragging it to the trash can. This introduces the linear constraint $\neg[\![v^*]\!]$, saying that either v^* should be removed, or that $[\![v^*]\!]$ should be empty. The system then searches for the closest tree satisfying the constraint.

5.1.4 Moving a phrase

The user can drag the selected phrase to another position in the sentence, which is interpreted as a linear precedence constraint on v^*. If the phrase is moved to between words w and w', we introduce the constraints $[\![v]\!] \prec [\![v^*]\!] \prec [\![v']\!]$, where $w \in [\![v]\!]_t$ and $w' \in [\![v']\!]_t$.

If the phrase is moved to the beginning or end of the sentence, instead of between two words, the constraints become $\epsilon \prec [\![v^*]\!] \prec [\![v']\!]$, or $[\![v]\!] \prec [\![v^*]\!] \prec \epsilon$, respectively

5.1.5 Inserting a phrase

We assume that somewhere on the screen there is a lexicon of phrases that the user can add to the sentence. If the user drags a phrase from the lexicon into the sentence between two words, we first deselect the currently selected node. Then we create new nodes and constraints representing the new phrase, as in the example in section 4.3.2, and select the topmost node. Finally we can add the same constraints as when moving a phrase, $[\![v]\!] \prec [\![v^*]\!] \prec [\![v']\!]$, but recall that v^* now denotes the topmost node in the inserted phrase, and not the previously selected phrase.

5.1.6 Replacing a phrase

Instead of inserting the user can replace phrases, by dragging a phrase from the lexicon and dropping it onto the selected phrase. As usual, if the user drops onto a currently unselected word, the system reselects it as explained in section 5.1.1.

All descendants $v_1, v_2, \ldots$ of the selected node v^* are removed by adding constraints $v_1, v_2, \ldots \notin V$. Furthermore, the new phrase should be added at v^*. We do this by letting v^* be the topmost node of the phrase, create new descendant nodes $v'_1, v'_2, \ldots$, and then add associated constraints:

$$\widehat{v^*} = f, \quad \widehat{v'_1} = a, \quad \uparrow v'_1 = v^*, \quad v'_1 \prec v'_2, \quad \ldots$$

If there is no nearby tree matching the constraints, the system can increase the selection and try again.

6 Discussion

6.1 Grammar formalism

The underlying grammar formalism is GF, but there are of course other formalisms that can be used in the same way. The most important feature is the separation of abstract and concrete syntax, which several formalisms have in different ways. Formalisms such as HPSG and LFG are probably also well suited for surface string editing, but the theory of editing presented in this paper must of course be adapted to the underlying formalism.

6.2 Example applications

We hope that our grammatical editing system can be a useful pedagogical tool for supporting language learning and training, for children with communicative disabilities, and for people learning a second language. We also believe that the ideas can be useful in touch screen devices, as an additional editing layer in text-based applications such as translation, email and chat.

6.2.1 Touch screen devices

An example application can be a translation tool for a touch screen device such as a mobile phone, as a kind of interactive phrasebook. This kind of application is already being developed in the MOLTO project, but currently it has very limited editing facilities (Angelov et al., 2010). Other touch screen possibilities include chat and email, where the user can create messages by dragging around text blocks instead of writing with a error-prone touch screen keyboard. It could also work together with speech recognition, to correct misrecognised phrases in a grammatical way.

6.2.2 Robust parsing

Another possible application can be robust parsing for limited-domain dialogue systems. It is possible to describe a dialogue system as a GF grammar (Ljunglöf, 2009), but the problem with GF is that the concrete syntax is not robust. Suppose that we use a statistical parser such as the MALT parser (Nivre et al., 2007). This returns a parse tree for every string, but in most cases, the tree is not grammatical. Then we can use the techniques in this paper for finding the closest grammatical tree, together with a confidence measure.

6.2.3 The GRASP project

The GRASP[1] project is developing another example application, an interactive system for Computer Assisted Language Learning (CALL). There are two intended target groups: one is children and adults trying to learn another language; another group is persons with communicative disabilities who are learning to read and write in their first language.

The idea of the final GRASP system is that it will work as an interactive textbook, where the user can read different texts and also experiment with and modify the texts. The system will be divided into modules dealing with different linguistic features, e.g., inflection, simple phrases and more advanced constructions. The modules can be used on their own, or can be combined for more advanced training.

The texts are stored as syntax trees in a multilingual GF grammar, which makes it possible to linearise the texts in parallel for several languages. This can be useful for second language learning, as the system can display the text in the user's first language in parallel. Multilinguality is also useful for first language learning, e.g., by displaying the parallel text in a symbol language such as Blissymbolics.

6.3 Current status

The GRASP system is work in progress, and not all features described in this paper are implemented, as of April 2011. There is a functioning demonstration system, which needs more work to be useful for the intended audience. In particular, the implementation is still too slow and the demonstration grammar needs to be expanded.

The current demostration grammar is a small monolingual Swedish grammar, and the module system is not fully developed. The grammar handles noun phrase inflection, fronting of noun phrases, and verb inflection.

7 Acknowledgements

The author would like to thank three anonymous reviewers for their valuable comments on an earlier version of this paper. The GRASP project is financed by Sunnerdahls Handikappfond.

[1]GRASP is an acronym for "grammatikbaserad språkinlärning" (grammar-based language learning).

References

Krasimir Angelov, Olga Caprotti, Ramona Enache, Thomas Hallgren, and Aarne Ranta. 2010. The MOLTO phrasebook. In *SLTC'10, 3rd Swedish Language Technology Conference*.

Trude Heift. 2001. Intelligent language tutoring systems for grammar practice. *Zeitschrift für Interkulturellen Fremdsprachenunterricht*, 6(2).

Peter Ljunglöf. 2004. *Expressivity and Complexity of the Grammatical Framework*. Ph.D. thesis, University of Gothenburg and Chalmers University of Technology, Gothenburg, Sweden.

Peter Ljunglöf. 2009. Dialogue management as interactive tree building. In *DiaHolmia'09, 13th Workshop on the Semantics and Pragmatics of Dialogue*, Stockholm, Sweden.

Joakim Nivre, Johan Hall, Jens Nilsson, Gülşen Eryiğit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. 2007. Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135.

Aarne Ranta. 2009a. The GF resource grammar library. *Linguistic Issues in Language Technology*, 2.

Aarne Ranta. 2009b. Grammatical Framework: A multilingual grammar formalism. *Language and Linguistics Compass*, 3(5):1242–1265.

Hiroyuki Seki, Takashi Matsumara, Mamoru Fujii, and Tadao Kasami. 1991. On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229.

Martin Sulzmann and Peter J. Stuckey. 2008. HM(X) type inference is CLP(X) solving. *Journal of Functional Programming*, 18(2):251–283.

Kuo-Chung Tai. 1979. The tree-to-tree correction problem. *JACM, Journal of the Association for Computing Machinery*, 26:422–433.

ISSN 1736-6305 Vol. 11
http://hdl.handle.net/10062/16955