Unify and Merge in Fluid Construction Grammar

Luc Steels[1,2] and Joachim De Beule[2]

[1] SONY Computer Science Laboratory - Paris
[2] Vrije Universiteit Brussel, Artificial Intelligence Laboratory
Pleinlaan 2, B-1050 Brussel

Abstract. Research into the evolution of grammar requires that we employ formalisms and processing mechanisms that are powerful enough to handle features found in human natural languages. But the formalism needs to have some additional properties compared to those used in other linguistics research that are specifically relevant for handling the emergence and progressive co-ordination of grammars in a population of agents. This document introduces Fluid Construction Grammar, a formalism with associated parsing, production, and learning processes designed for language evolution research. The present paper focuses on a formal definition of the unification and merging algorithms used in Fluid Construction Grammar. The complexity and soundness of the algorithms and their relation to unification in logic programming and other unification-based grammar formalisms are discussed.

1 Introduction

Computational research into the origins of language and meaning is flourishing. There is a growing set of experiments showing how certain aspects of human natural languages emerge in a population of agents endowed with specific cognitive components such as a bi-directional associative memory or an articulatory and auditory apparatus. (See overviews and representative samples of current work in [2], [3], [13], [24]). Most of the solid results so far have been reached for the emergence of lexicons and perceptually grounded categories. Although there have been a number of experiments on the role of syntax and grammar (see e.g. [9] [22], [1], [21]), there are as yet only very few demonstrations where non-trivial grammars arise in grounded situated interactions between robotic agents. Part of the reason is of course that the problem of grammar emergence is much more encompassing than that of lexicons and many fundamental questions remain unanswered. In addition, the world models of agents and the nature of their interactions needs to be much more complex than in lexical experiments. But another reason, we believe, has to do with the nature of the computational apparatus that is required to do serious systematic experiments. Whereas lexicon emergence can be studied with relatively standard neural networks, grammar requires much more powerful symbolic processing which falls outside the scope of connectionist modeling today.

Our group has therefore been working for many years on a formalism that would be adequate for handling phenomena typically found in natural language grammars, but that would at the same time support highly flexible parsing and production (even of ungrammatical sentences or partially unconventionalised meanings) and invention and learning operators that could lead to the emergence, propagation, and further evolution of grammar, all this in a multi-agent setting. Our formalism has been called Fluid Construction Grammar, as it is in line with the approaches advocated in usage-based cognitive approaches to language in general and construction grammar in particular, and because it is designed to support highly flexible language processing in which conventions are not static and fixed but fluid and emergent. At this point the FCG system is ready for use by others. An implementation on a LISP substrate has been released for free download through http://arti.vub.ac.be/FCG/. This site also contains very specific examples on how to do experiments in language evolution with FCG. The goal of the present paper is to define some core aspects of FCG more precisely, building further on earlier reports [7], [25]. The application of FCG to various issues in the emergence of grammar is discussed in some other papers (see [23], [6], [20]).

Fluid Construction Grammar (FCG) uses as much as possible existing widely accepted notions in theoretical and computational linguistics, specifically feature structures for the representation of syntactic and semantic information during parsing and production, and abstract templates or rules for the representation of lexical and grammatical usage patterns, as in [14] or [16]. Contemporary linguistic theories propose general operators for building up syntactic and semantic structures, such as Merge in Chomsky's Minimalist Grammar [4] or Unify in Jackendoff's framework [11]. Unification based grammars [14–16] are similarly based on a unification operator, and more generally many generic inference systems, particularly within the logic programming framework, use some form of unification [18]. There is also some research on finding the neural coordinates of unify and merge in language processing [10]. Unfortunately there are quite substantial differences between the use of the terms unify and merge in all these different frameworks and one of the goals of this paper is to clarify in detail the unify and merge operators that form the core of FCG. For this, we build further on the comparative analysis done by [17] who reformulated FCG in PROLOG.

The remainder of the paper has five parts. The next section defines first the requirements for grammar formalisms needed to do experiments in the emergence of non-trivial grammars and the basic ideas behind the Fluid Construction Grammar approach. Then there is a section with some preliminary definitions and background notions. The remainder of the paper focuses on a formal definition of the unify and merge operations in FCG. Section 4 defines the unification in general, and particularly the FCG extensions to standard unification, section 5 then applies this to FCG feature structures. Section 6 defines merging in general, and section 7 applies it to FCG feature structures. Section 7 contains a worked out example.

2 Fluid Construction Grammars

It is obvious that there many possible formalisms could be (and have been) invented for capturing aspects of language, depending on the nature of the linguistic theory and the types of processing one wants to study. For example, generative (derivational) grammar is adequate for studying ways to generate the set of all possible sentences of a language but it is not well suited for parsing or production, while a constituent structure grammar is adequate for studying syntactic structure but not helpful for investigating how case frames intervene between the mapping from meaning to form, etc. Fluid Construction Grammar takes a strong stance with respect to current linguistic theorising and attempts in addition to satisfy specific requirements which arise when one wants to do experiments in the emergence of grammar in grounded situated interactions.

2.1 Linguistic Assumptions

The linguistic perspective of FCG is in the general line of cognitive grammar [12] and more specifically construction grammar [8]. This means the following:

1. *FCG is usage-based*: The inventories available to speakers and hearers consist of templates which can be highly specialised, perhaps only pertaining to a single case, or much more abstract, covering a wide range of usage events. There is no sharp distinction therefore between idiomatic and general rules. New sentences are constructed or parsed by assembling the templates using the unify and merge operators defined later in this paper.

2. *The grammar and lexicon consist of symbolic units*: A symbolic unit associates aspects of meaning with aspects of form. The templates of FCG are all symbolic units in this sense. They feature a semantic pole and a syntactic pole. Templates are always bi-directional, and so are usable both for production *and* for parsing. This makes FCG unique not only with respect to derivational formalisms (like generative grammar or HPSG) but also with respect to other construction grammar formalisms which tend to be uni-directional (such as Embodied Construction Grammar).

3. *There is a continuum between grammar and the lexicon.* Not only can templates be of different levels of abstraction, but there is also no formal distinction in the structure or the processing of lexical and grammatical entries. In the case of lexical entries, the syntactic pole concerns mostly a lexical stem and the semantic pole tends to be equal to some concrete predicate-argument structure. In the case of grammatical constructions, the syntactic pole contains various syntactic categories constraining the sentence, and the semantic pole is based on semantic categories, but otherwise there is no formal difference between the two types of templates.

4. *Schematisation occurs through variables and categorisation.* A template has the same form as an association between a semantic structure and a syntactic structure, in other words both poles of a template are feature structures. However, templates are more abstract (or schematic) in three ways: Some parts of the semantic or syntactic structure are left out, variables are used instead of

units and values, and syntactic or semantic categories are introduced to constrain the possible values of the semantic and syntactic pole. These categories are often established by syntactic or semantic categorisation rules. In some experiments the categories have been defined using a memory-based approach with typical examples and prototypes but this is not yet embedded in the current release of FCG.

5. *Syntagmatic and Paradigmatic Compositionality*: To produce or parse a sentence, templates can be combined (several templates all matching with different parts of the meaning in production or with parts of the sentence in parsing are simply applied together) or integrated (using hierarchical templates that combine partial structures into larger wholes, possibly after a modification of the syntactic or semantic aspects of the component units). Apart from this syntagmatic composition, there is also the possibility that several templates are overlayed and each contribute additional constraints to the final sentence. This is paradigmatic compositionality. Both forms of compositionality are completely supported with the unify and merge operators defined later in the paper. Of particular importance is that the FCG unify and merge operators handle linking (resolving equalities introduced by separate lexical items) by unifying variables in merge. This topic is discussed more extensively in [25].

2.2 Additional Requirements

In addition to these characteristics, which we take to be general features of cognitive grammars and construction grammars, there are some additional requirements for a formalism if it is to be adequate for modeling emergent natural language like grammars.

1. *All inventory entries have scores*: We assume that speakers and hearers create new templates which are often competing with each other. Not all templates are widely accepted (entrenched) in the population and there is a negotiation process. To enable this capability we associate with every item in the lexicogrammar a score that reflects the degree of entrenchment of that item. It is based on feedback from success or failure in the language games in which the item has been used. We know from our other experiments in lexicon emergence that an appropriate lateral inhibition dynamics is the most adequate way for driving a population to a sufficiently co-ordinated repertoire to have success in communication and the same dynamics is part of FCG.

2. *The set of syntactic and semantic categories is open.* Often linguistic formalisms posit specific sets of semantic categories (for example semantic roles like agent, patient, etc.) or syntactic categories (such as parts of speech, syntactic features, and others). Because we are interested in how all these categories arise, we make the formalism completely open in this respect. Constraints on feature values are expressed as propositions or predicate- argument clauses so that the set of categories can be expanded at any time. In our experiments there are typically hundreds or even thousands of new categories built. This openness of categories is in line with the Radical Construction Grammar approach which argues that linguistic categories are not universal and subject to evolution [5].

3. *The multi-agent perspective*: In traditional Chomskyan linguistics only the grammar or lexicon of an 'idealised speaker or hearer' is considered. In contrast, when we want to study how grammar arises in a population we need to take a multi-agent perspective, where every agent possibly has a different inventory. We need to understand in particular how agents co-ordinate their inventories to be successful in communication. This raises a large number of computational issues (for example linguistic knowledge is always local to an agent) as well as issues in how to track and measure 'the grammar' in the population.

Although all these issues are dealt with in the current FCG design and implementation, the remainder of this paper focuses only on the core of the system, namely the unify and merge operators.

3 Preliminaries

This section starts the more formal discussion and assumes some familiarity with FCG (e.g. from [23], [25], [7] or the FCG website `http://arti.vub.ac.be/FCG`). We will adopt the standard terminology of logic, which is unfortunately different from the terminology often used in other unification-based approaches to natural language. In logic, the term 'unification' does not involve the merging of two structures or a change to a structure, so unification is a kind of 'matching' which yields a set of variable bindings. In contrast with simpler forms of matching (as used in many production rule systems), variables can be present both in the source and in the target. This is the meaning of 'unify' as used in the rest of the paper. Because we also want a way to extend or build up structure based on templates, we have an additional operator called 'merge' which takes a source and a target and combines them into a new structure.

We now introduce some definitions which are common in logic (see e.g. [18, 26]). **Symbols** are the atomic units. Symbols starting with a question mark are **variables**. $\mathcal{A}$ denotes the set of all symbols, $\mathcal{V}$ the set of all variables. Symbols that are not variables are **constants.** A **simple expression** is defined as a symbol or a list of one or more simple expressions. More formally, let (.|.) be the list-creator operator (similar to cons in LISP) and let () be an operator of arity 0 (similar to NIL in LISP).

Definition 1. *Let $\mathcal{E}_s$ denote the set of all* **simple expressions***:*

- *All elements of $\mathcal{A}$ as well as () are elements of $\mathcal{E}_s$.*
- *If $e_1 \in \mathcal{E}_s$ and $e_2 \in \mathcal{E}_s$ and if e_2 is not in $\mathcal{A}$ then $(e_1|e_2) \in \mathcal{E}_s$.*

In addition to simple expressions, we will consider later non-simple expressions which feature special operators and will be called general FCG-expressions or simply expressions.

Let us denote the set of variables in an expression e by **vars(e)**. Often an expression of the form $(e_1|(e_2|...(e_k|())...))$ is represented as $(e_1e_2...e_k)$ and is called a **list of length k**. The operator () is called a list of length 0.

A **binding** $[?x/X]$ specifies the value X of a variable $?x$. If X is itself a variable then such a binding is called an **equality**. A set of bindings B =

$\{[?x_1/X_1], ..., [?x_n/X_n]\}$ defines a function $\sigma_B : \mathcal{V} \rightarrow T$ such that $\sigma_B(?x) = ?x$ except for the variables $?x_1$ to $?x_n$ in B, called the **domain** of σ_B. For these variables it holds that $\sigma_B(?x_i) = X_i$. The set $\{X_1, ..., X_n\}$ is denoted by $\text{ran}(\sigma_B)$. Such a function is called a **substitution** and the set of bindings B is sometimes called the **graph** of σ_B and is written simply as $[?x_1/X_1, ..., ?x_n/X_n]$. Often a substitution σ_B is represented by its graph B. The empty substitution (the identity function) is denoted by ϵ. Furthermore define *fail* to be a special symbol which will be used to specify the failure to find a substitution.

The extension of a substitution to the domain $\mathcal{E}_s$ can be defined using structural induction. If σ_B is the substitution constructed from the set of bindings B then the application of σ_B to an expression e is written either as $\sigma_B(e)$ or as $[e]_B$.

Definition 2. *Two simple expressions x and y are said to be* **equal** *(written as $x = y$) iff either:*

1. *x and y are both the same atom*
2. *both $x = (x_1...x_n)$ and $y = (y_1...y_m)$ are lists of the same length ($m = n$) and for every $i = 1..n : x_i = y_i$.*

Substitutions can be ordered according to the pre-order $\preceq_{\mathcal{V}}$ which is defined as follows: If σ_1 and σ_2 are two substitutions then $\sigma_1 \preceq_{\mathcal{V}} \sigma_2$ iff there exists a substitution λ such that $\sigma_1(v) = (\sigma_2 \circ \lambda)(v)$ for each $v \in \mathcal{V}$. When clear from the context, $\sigma_1 \preceq_{\mathcal{V}} \sigma_2$ is written as $\sigma_1 \preceq \sigma_2$.

If $\sigma_1 \preceq_{\mathcal{V}} \sigma_2$ and $\sigma_2 \preceq_{\mathcal{V}} \sigma_1$ then we say that $\sigma_1 =_{\mathcal{V}} \sigma_2$.

Definition 3. *Two simple expressions e_1 and e_2 are said to* **unify** *iff there exists a substitution σ such that $\sigma(e_1) = \sigma(e_2)$. In this case the substitution σ is called a* **unifier** *of the two expressions and the set of all unifiers of e_1 and e_2 is written as $\bigcup T(e_1, e_2)$.*

Example 1. The expression $e = (a\ ?x)$ unifies with the expression $e' = (?y\ a)$ with unifier $[?x/a, ?y/a]$.

It is easy to see that the set of unifiers of two unifiable expressions e_1 and e_2 is infinite (if $\mathcal{V}$ is infinite.) Indeed, a unifier can always be extended with an additional binding for a variable that is not an element of either $vars(e_1)$ or $vars(e_2)$. In order to exclude these unifiers we only assume unifiers that satisfy the **protectiveness** condition. A unifier σ of two expressions e_1 and e_2 satisfies this condition iff $dom(\sigma) \subseteq vars(e_1) \bigcup vars(e_2)$ and $dom(\sigma) \cap vars(ran(\sigma)) = \emptyset$

A **complete set** of unifiers $c \bigcup T(e_1, e_2)$ is a subset of $\bigcup T(e_1, e_2)$ that satisfies the additional condition that for any unifier σ of e_1 and e_2 there exists a $\theta \in c \bigcup T(e_1, e_2)$ such that $\theta \preceq \sigma$.

The **set of most general unifiers** $\mu \bigcup T(e_1, e_2)$ is a complete set of unifiers that additionally satisfies the **minimality condition**: for any pair $\mu_1, \mu_2 \in \mu \bigcup T(e_1, e_2)$, if $\mu_1 \preceq \mu_2$ then $\mu_1 = \mu_2$.

It is well known that if two simple expressions x and y unify, then there always is only one most general unifier (up to a renaming of variables, see e.g. [18]).

Let $f(x, y, \{\epsilon\})$ be the function that computes the set containing only this most general unifier. Before showing how this function can be computed we first define the notion of valid extensions.

Definition 4. *The set of* **valid extensions** $\Xi(B, b)$ *of a set of bindings* $B = [?x_1/X_1, ..., ?x_n/X_n]$ *with a binding* $b = [?x/X]$ *is defined as follows:*

1. *If* $?x = ?x_i$ *for some* i *then* $\Xi(B, b) = f(X, X_i, \{B\})$.
2. *Else, if* $X \in \mathcal{V}$ *and* $X = ?x_j$ *for some* j *then* $\Xi(B, b) = f(?x, X_j, \{B\})$.
3. *Else* $\Xi(B, b) = \{[?x/X] \cup B\}$.

Definition 5. *The set of* **valid extensions** $\Xi(B, b)$ *of a set of bindings* $B = [?x_1/X_1, ..., ?x_n/X_n]$ *with a binding* $b = [?x/X]$ *is defined as follows:*

$$\Xi(B, b) = \begin{cases} f(X, X_i, \{B\}) & \text{if } ?x = ?x_i \text{ for some } i \in \{1, ..., n\} \\ f(?x, X_j, \{B\}) & \text{if } X \in \mathcal{C} \text{ and } X = ?x_j \text{ for some } j \in \{1, ..., n\} \\ \{[?x/X] \cup B\} & \text{otherwise.} \end{cases}$$

Using this definition, $f(x, y, Bs)$, with Bs a set of sets of bindings, can be computed as follows [29]:

1. if $x \in \mathcal{V}$ and $x \neq y$ and x does not occur in y then $f(x, y, Bs) = \{\Xi(B, [x/y]); B \in Bs\}$.
2. else if $y \in \mathcal{V}$ and $x \neq y$ and y does not occur in x then $f(x, y, Bs) = \{\Xi(B, [y/x]); B \in Bs\}$.
3. else if $x = (x_1|x_2)$ and $y = (y_1|y_2)$ then $f(x, y, Bs) = f(x_1, y_1, f(x_2, y_2, Bs))$
4. else if $x = y$ then $f(x, y, Bs) = Bs$
5. else $f(x, y, Bs) =$ fail

FCG unification of two *simple* expressions is equivalent to standard unification and thus returns the single most general in this case.

4 Unifying

FCG expressions are more extensive than simple expressions because they may include special operators such as the includes-operator '==' which specifies which elements should be included in a feature's value, or the J-operator which plays a key role in defining hierarchy.[3] Each of these operators has a dedicated function for defining how unification should be carried out.

Let $\mathcal{O}$ be the set of special operator symbols. It includes for example the symbol '=='.

Definition 6. *Let* $\mathcal{E}$ *denote the set of all* FCG **expressions**. *Then:*

1. *All elements of* $\mathcal{A}$ *and* $\mathcal{O}$ *as well as* () *are elements of* $\mathcal{E}$.
2. *if* $e_1 \in \mathcal{E}$ *and* $e_2 \in \mathcal{E}$ *and if* e_2 *is not in* $\mathcal{A} \cup \mathcal{O}$ *then* $(e_1|e_2) \in \mathcal{E}$.

[3] The J-operator is treated in a separate paper [7].

For every operator o in $\mathcal{O}$ and for all expressions $e_1 = (o|e_1')$ and e_2 and sets of bindings Bs a designated unification function $f_o(e_1, e_2, Bs)$ must be defined returning a set of unifiers for e_1 and e_2.

Definition 7. FCG **unification** $f_{\text{FCG}}(x, y, \text{Bs})$ *of two expressions* x *and* y*, given the sets of bindings* Bs*, is defined as follows:*

1. *if* $x = (o|x')$ *with* $o \in \mathcal{O}$ *then* $f_{\text{FCG}}(x, y, \text{Bs}) = f_o(x, y, \text{Bs})$
2. *else if* $y = (o|y')$ *with* $o \in \mathcal{O}$ *then* $f_{\text{FCG}}(x, y, \text{Bs}) = f_o(y, x, \text{Bs})$
3. *else if* $x \in \mathcal{V}$ *and* $x \neq y$ *and* x *does not occur in* y *then* $f_{\text{FCG}}(x, y, \text{Bs}) = \{\Xi(B, [x/y]); B \in \text{Bs}\}$.
4. *else if* $y \in \mathcal{V}$ *and* $x \neq y$ *and* y *does not occur in* x *then* $f_{\text{FCG}}(x, y, \text{Bs}) = \{\Xi(B, [y/x]); B \in \text{Bs}\}$.
5. *else if* $x = (x_1|x_2)$ *and* $y = (y_1|y_2)$ *then* $f_{\text{FCG}}(x, y, \text{Bs}) = f_{\text{FCG}}(x_1, y_1, f_{\text{FCG}}(x_2, y_2, \text{Bs}))$
6. *else if* x=y *then* $f_{\text{FCG}}(x, y, \text{Bs}) = \text{Bs}$
7. *else* $f_{\text{FCG}}(x, y, \text{Bs}) = fail$

Clearly FCG-unification is equivalent to standard unification if only simple expressions (and thus no special operators) are considered. Hence the properties of FCG-unification depend on the properties of the dedicated unification functions f_o. We now define some of these special operators so that FCG unification of FCG feature structures can be defined. The list of special operators can in principle be extended by defining the relevant unification functions.

4.1 The includes operator ==

Let us first define the notion of containment.

Definition 8. *An expression* x_i *is* **contained** *in a list iff it* FCG*-unifies with an element in this list.*

A list starting with the includes operator ($==\ x_1 ... x_n$) unifies with any source list that at least contains the elements x_1 to x_n. The order in which the elements occur in the source list is irrelevant, however every x_i should unify with a *different* element in the source, as in the following examples:

Example 2.

$$
\begin{aligned}
&f_{==}((==\ a\ a\ b), (a\ b), \{\epsilon\}) = fail \\
&f_{==}((==\ a\ a\ b), (a\ a\ b), \{\epsilon\}) = \{\epsilon\} \\
&f_{==}((==\ a\ a\ b), (b\ a\ a), \{\epsilon\}) = \{\epsilon\} \\
&f_{==}((==\ a\ ?x), (a\ b\ c), \{\epsilon\}) = \{[?x/b], [?x/c]\} \qquad (1)
\end{aligned}
$$

This is formalized in the following

Definition 9. *Let $p_n((e_1...e_m))$ with $n \leq m$ be the set of expressions $(e_{i_1}...e_{i_n})$ for every variation $(i_1, ..., i_n)$ of n elements out of $(1, ..., m)$.*[4] *Then*

$$f_{==}((== x_1...x_n), (a_1...a_m), \mathrm{Bs}) = \bigcup_{a \in p_n((a_1...a_m))} f_{\mathrm{FCG}}((x_1...x_n), a, \mathrm{Bs})$$

Example 3. Consider again the last example in (1). We have to consider all variations of two elements out of (abc), i.e. $(a\ b)$, $(b\ a)$, $(a\ c)$, $(c\ a)$, $(b\ c)$ and $(c\ b)$. Unifying these with $(a\ ?x)$ results in $\{[?x/b]\}$, *fail*, $\{[?x/c]\}$, *fail*, *fail* and *fail* respectively. Keeping only the successful ones indeed leads to $\{[?x/b], [?x/c]\}$.

4.2 Special cases of $f_{==}$

First of all, the unification of two include-lists $x = (==\ x_1...x_n)$ and $y = (== y_1...y_m)$ is not well defined. One possibility is to state that two such lists always unify. However, it is not clear what the resulting set of bindings should be. For simplicity we define $f_{\mathrm{FCG}}((o_1|x), (o_2|y), Bs)$ with $o_1, o_2 \in \mathcal{O}$ to *always be equal to fail.*

Second, the pattern $(x_1...x_k == y_1...y_l)$ with a source $(z_1...z_m)$, $m \geq k + l$ is well defined. More formally, the pattern should be written as

$$(x_1...x_k == y_1...y_l) \equiv (x_1|(...|(x_k|(== |(y_1|(...|(y_l|())...).$$

The f_{FCG} function will progressively unify the elements x_1 to x_k with the first k elements in the source. At this point f_{FCG} is recursively applied to the pattern $(== |(y_1|(...|(y_l|())...)$, which can be re-written as $(== y_1...y_l)$, and the source $(z_{k+1}...z_m)$.

Third, operators are treated as ordinary symbols if they are not the first element of a list:

$$f_{==}((==\ a\ ?x), (a\ ==\ b), \{\epsilon\}) = \{[?x/ ==], [?x/b]\}$$

4.3 Minimality, Completeness and Complexity of $f_{==}$

Theorem 1 (Completeness of $f_{==}$). *The function $f_{==}((== |X), Y, , \{\epsilon\})$ computes a complete set of unifiers $c\bigcup T((== |X), Y)$*

Proof. From the definition of $f_{==}$ it follows that, if a substitution τ is a unifier of $(== |X) = (== x_1...x_n)$ and $Y = (y_1...y_m)$, then it must be that $\tau(X)$ is a variation of n elements from Y. Let $\sigma \in f_{==}((== |X), Y, \{\epsilon\})$ be the substitution that computes this variation, so that $\sigma(X) = \tau(X)$. If X and Y do not contain any special operators then, from the completeness of F_{FCG} for simple expressions, it follows that $\sigma \preceq \tau$ and thus that $f_{==}(X, Y, \{\epsilon\})$ computes a complete set of unifiers. If some element x_i (y_i) of X (Y) is of the form $(== |z)$, with z not containing a special operator, then the same argument can be used recursively. Continuing in this way, it follows that $f_{==}$ computes a complete set of unifiers. □

[4] We intend here the set-theoretic notion of variation, i.e. all subsets of n elements from $(1, ..., m)$ where the order of the elements matters.

Theorem 2 (Non-minimality of $f_{==}$). *The function $f_{==}((== |X), Y, , \{\epsilon\})$ does not necessarily compute the most general set of unifiers $\mu \bigcup T((== |X), Y)$.*

Proof. It suffices to show that $f_{==}$ does not satisfy the minimality condition. Consider the unification of $(== ?x\ ?y)$ with $(?x\ ?y)$ which results in $\{\epsilon, [?x/?y]\}$. This is different from the minimal set of unifiers which consists of the empty substitution ϵ only (this example was taken from [17].) □

Theorem 3 (Complexity of $f_{==}$). *$f_{==}((== x_1...x_n), (y_1...y_{m \geq n})$ is of exponential complexity in n.*

Proof. Basically, the exponential complexity arises from the need to calculate the variations. Indeed, when $x_i \neq y_j$ for all possible combinations of i and j then $f_{==}((== x_1...x_n), (y_1...y_m))$ is equivalent with the subset unification of $\{x_i\}$ and $\{y_j\}$. General subset unification is exponential and the subset-unifiability problem is NP-complete [27, 28]. □

The implementation of $f_{==}((== x_1...x_n), (y_1...y_{m \geq n})$ can be made more efficient by calculating in advance the set of candidate expressions $C_i = \{y_j | y_j$ unifies with $x_i\}$ and by only considering combinations of n distinct elements out of each C_i. However the inherent exponential complexity cannot be improved upon in general.

In the following we introduce two additional special operators which are defined as special cases of the includes operator. This ensures that the completeness of f_{fcg} is maintained. However they can be computed more efficiently.

4.4 The permutation operator $==_p$

The permutation operator is like the includes operator except that the source should contain *exactly* the elements specified in the pattern. Thus we have:

Definition 10.

$$f_{==_p}((==_p x_1...x_n), (a_1...a_m), \mathrm{Bs}) = \begin{cases} f_{==}((== x_1...x_n), (a_1...a_m), \mathrm{Bs}) & \text{if } n = m, \\ \mathrm{fail} & \text{otherwise.} \end{cases} \quad (2)$$

4.5 The includes-uniquely operator $==_1$

The function of this operator will become more important in merging (see later.) However its behavior in unification must be specified because FCG-templates may contain this operator.

Definition 11. *Let $s = (y_1...y_m)$. Then $f_{fcg}((==_1 x_1...x_n), s)$ is the set $\{B\} \subset f_{fcg}((== x_1...x_n), s)$ of substitutions B that satisfy the following conditions*

1. *No two symbols $\sigma_B(y_i)$ and $\sigma_B(y_j)$ of $\sigma_B((y_1...y_n))$ with $i \neq j$ are allowed to unify: $f_{\mathrm{FCG}}(y_i, y_j, \{B\}) = \mathrm{fail}$ and*

2. *if* $\sigma_B(y_i) = \sigma_B((y_{i1}|y_{i2}))$ *and* $\sigma_B(y_j) = \sigma_B((y_{j1}|y_{j2}))$ *are two non-atomic elements of* $\sigma_B((y_1...y_n))$ *with* $i \neq j$ *then their first elements are not allowed to unify:* $f_{\text{FCG}}(y_{i1}, y_{j1}, \{B\}) = \text{fail}$

The above definition ensures that every element in $\sigma_B((y_1...y_n)), B \in Bs$ is distinct. It also implies that no element $\sigma_B(y_i)$ can be a variable or start with a variable if $n \geq 1$.

Example 4.

$$\begin{aligned}
&f_{==_1}((==_1\ ?x_1\ a), (?y_1\ (?y_2)\ b)) = \{[?y_1/a, ?x_1/(?y_2)], [?y_1/a, ?x_1/b]\} \\
&f_{==_1}((==_1\ ?x_1\ a), (?y_1\ ?y_2\ b)) = \mathit{fail} \\
&f_{==_1}((==_1\ ?x_1), (?y_1\ b)) = \mathit{fail}.
\end{aligned} \tag{3}$$

In contrast with the last of these examples, consider that:

Example 5.

$$f_{==}((==\ ?x_1), (?y_1\ b)) = \{[?x_1/b], [?x_1/?y_1]\}. \tag{4}$$

Note that some includes-uniquely patterns cannot be satisfied (e.g. $(==_1\ a\ a)$.)

5 Unifying Feature Structures

We are now ready to define the matching of two FCG structures. We begin by defining FCG feature structures which are more constrained than feature structures in other unification grammars [14], in the sense that they are not hierarchical. Hierarchy is represented instead by using the name of a unit as the definition of the syn-subunits or sem-subunits slots. This has many advantage, including that a unit can be the subunit of more than one other unit. It also simplifies computation enormously.

5.1 Feature structures in FCG

A syntactic or semantic structure in FCG consists of a set of units which each consist of a unique name and a set of feature-value pairs. A unit typically corresponds to a lexical item or to constituents like noun phrases or relative clauses. The name can be used to identify or refer to a unit and unit-names can be bound to variables. Feature-values cannot themselves be feature structures. We always introduce separate units with their own names and associate feature-value pairs with this new

Example 6. The following expression could be a syntactic structure in FCG. The structure contains three units named `sentence-unit`, `subject-unit` and `predicate-unit`. The `sentence-unit` has two features named `syn-subunits` and `syn-cat`, with respective values the lists `(subject-unit predicate-unit)` and `(SV-sentence)`.

```
((sentence-unit (syn-subunits (subject-unit predicate-unit))
                (syn-cat (SV-sentence)))
 (subject-unit (syn-cat (proper-noun (number singular)))
               (form John))
 (predicate-unit (syn-cat (verb (number singular)))
                 (form walks))).
```

Without separate units or unit names (as in other formalisms) this would look like:

```
(sentence-unit
   (syn-subunits
      ((syn-cat (proper-noun)
                (number singular))
       (form John))
      ((syn-cat (verb)
                (number singular))
       (form walks)))).
```

A template's pole has the same form as the feature structures shown above but typically contains variables as well as special operators (like '==').

Example 7. The following expression could be the syntactic pole of a template. Note how agreement in number between subject and verb is handled through the variable `?number` which will be bound to a specific number value.

```
((?sentence-unit
    (syn-cat (SV-sentence))
    (syn-subunits (?subject-unit ?predicate-unit))
    (form (== (precedes ?subject-unit ?predicate-unit))))
 (?subject-unit
    (syn-cat (== proper-noun (number ?number))))
 (?predicate-unit
    (syn-cat (== verb (number ?number))))).
```

The features that may occur are restricted to a limited set of symbols: {sem-subunits, referent, meaning, sem-cat} for semantic structures and {syn-subunits, utterance, form, syn-cat} for syntactic structures. The syntactic or semantic categories are completely open-ended, and so the example categories used here (like number, proper-noun, etc.) are just intended as illustration. Syntactic and semantic structures always come in pairs, and units in a syntactic structure are paired with those in the semantic structure through common unit names. More formally, we have the following

Definition 12. *a* **feature-value pair** *is an expression of the form* $(e_n\ e_v)$*. The expression* e_n *is called the feature name and* e_v *is the feature value. A* **unit** *is any expression of the form* $(e_n\ f_1...f_k)$ *with the expression* e_n *the unit's name and* $f_i, i = 1...k$ *the features. Unit names are usually but not necessarily symbols. Finally a* **unit structure** *(or feature structure) is any expression of the form* $(u_1...u_l)$ *with all of the* u_i *units.*

Thus, a unit structure can be represented by an expression of the form

$$\begin{aligned}&((u_1\ (f_{11}\ v_{11})...(f_{1n_1}\ v_{1n_1}))\\&\quad ...\\&\quad (u_m\ (f_{m1}\ v_{m1})...(f_{mn_m}\ v_{mn_m}))).\end{aligned} \tag{5}$$

Here is another (simplistic) example of a syntactic structure for the utterance "red ball":

```
((np-unit (syn-subunits (adjective-unit noun-unit)))
 (adjective-unit (syn-cat (adjective))
                 (form ((stem adjective-unit "red"))))
 (noun-unit (syn-cat (noun))
            (form ((stem noun-unit "ball")))),
```

which may be associated with the following semantic structure:

```
((np-unit (sem-subunits (adjective-unit noun-unit)))
 (adjective-unit (meaning ((color obj-1 red))))
 (noun-unit (referent obj-1)
            (meaning ((sphere obj-1)(used-for obj-1 play)
                      (mentioned-in-discourse obj-1))))).
```

The names of the units allow cross-referencing between the two structures.

Unification in FCG determines the applicability of templates. An example of an FCG template that should be triggered by the above semantic structure is shown below (the template's left (semantic) and right (syntactic) poles are separated by a double-arrow):[5]

```
((?unit (referent ?obj)
        (meaning (== (sphere ?obj) (used-for ?obj play)))))
<-->
((?unit (form (== (stem ?unit "ball")))))
```

However.it can be seen that the left pole of this template does *not* unify with the semantic structure above: only (part of) the `noun-unit` is specified by the pole but specifications for the other units are missing. Therefore no substitution can make the source structure equal to the pole or vice versa. If however the pole is changed to:

```
(==1 (?unit (referent ?obj)
            (meaning (== (sphere ?obj) (used-for ?obj play)))),
```

then this indeed unifies with the source structure to yield the bindings

```
[?obj/obj-1,?unit/noun-unit].
```

[5] These examples have all been simplified for didactic reasons.

5.2 The unification of feature structures

Definition 13. *The function* **unify-structures(P,S,B)** *takes a pattern structure P, a source structure S and a set of bindings B and can be computed as follows. If P is as represented in (5) then the pattern is first transformed to the pattern P':*

$$\begin{aligned}(==_1(u_1 \ ==_1 (f_{11} \ v'_{11})...(f_{1n_1} \ v'_{1n_1}))\\ ...\\ (u_m \ ==_1 (f_{m1} \ v'_{m1})...(f_{mn_m} \ v'_{mn_m}))),\end{aligned} \tag{6}$$

in which the new feature values v'_{ij} are determined as follows: Every non-atomic feature value $v_{ij} = (v_1|v_2)$ in the pattern for which v_1 is not a special operator is replaced by $v'_{ij} = (==_p |v_{ij})$. Atomic feature values remain unchanged: $v'_{ij} = v_{ij}$ if v_{ij} is atomic. Unify-structures(P,S,B) is then defined as $f_{\mathrm{FCG}}(P', S, \{B\})$.

6 Merging

Informally, merging a source expression s and a pattern expression p means changing the source expression such that it unifies with the pattern expression. Merging two general fcg-expressions is undefined in this paper, we only consider the case where at least the source is a simple expression (i.e. does not contain special operators.) We first examine the case where also the pattern is a simple expression.

6.1 Merging of Simple Expressions

Definition 14. *Let $g(p, s, B)$ denote the merge function that computes a set of tuples (s', Bs') of new source patterns s' and bindings sets Bs' such that $f_{\mathrm{FCG}}(p, s', \{B\}) = \mathrm{Bs}'$. $g(p, s, B)$ on simple expressions p and s is defined as follows:*

1. *If $\mathrm{Bs} = f_{\mathrm{FCG}}(p, s, \{B\}) \neq \mathrm{fail}$ then $g(p, s, B) = (s, \mathrm{Bs})$.*
2. *Else if $p = (p_1|p_2)$ and $s = (s_1|s_2)$ then let $G'_1 = g(p_1, s_1, B)$.*
   (a) If $G'_1 \neq \emptyset$ then

$$g(p, s, B) = \bigcup_{(s'_1, B_1) \in G'_1} \left(\bigcup_{g'_2 \in g(p_2, s_2, B_1)} \left(\bigcup_{(s'_2, \mathrm{Bs}') \in g'_2} \{((s'_1|s'_2), \mathrm{Bs}')\} \right) \right)$$

   (b) Else, if $\mathrm{length}(p) > \mathrm{length}(s)$ then let $G'_2 = g(p_2, s, B)$ and let

$$S'_1 = \bigcup_{(s'_2, \mathrm{Bs}') \in G'_2} \left(\bigcup_{B' \in \mathrm{Bs}'} \{\sigma_{B'}(p_1)\} \right).$$

Then

$$g(p,s,B) = \bigcup_{s'_1 \in S'_1} \left(\bigcup_{(s'_2,\mathrm{Bs}') \in G'_2} \{((s'_1|s'_2),\mathrm{Bs}')\} \right)$$

3. *Else if* $p = (p_1|p_2)$ *and* $s = ()$ *then* $g(p,s,B) = \{(\sigma_B(p),\{B\})\}$
4. *Else* $g(p,s,B) = \emptyset$

Let us clarify these steps. The first step is obvious and ensures that no unnecessary modifications are done: the merging of a pattern and a source that unify is equivalent to leaving the source unchanged and unifying them.

The second step consists of two possibilities. If the first element of the pattern merges with the first element of the source (case (a)) then the result is further completely determined by the results $g'_2 = (s'_2, Bs')$ of merging the remaining elements of the source and the pattern.

Else (case (b), the first elements do not merge), if the pattern is longer then the source we can consider extending the source with the first element of the pattern. The result is then further completely determined by the result G'_2 of merging the remaining elements of the pattern with the entire source. And because this might involve a set of bindings which could potentially lead to different expressions for the first element of the pattern p_1, the combinations of such distinct expressions and bindings need to be computed.

Theorem 4 ((Termination of $g(p,s,B)$)). *The definition above can be viewed as an algorithm to compute the value of $g(p,s,B)$. It is obvious that this algorithm will always terminate when called on a pattern of finite length: although it is called recursively in steps 2(a) and (b), it is always called on a pattern of smaller length. This can only continue until the pattern is of length 0 (i.e. is equal to $()$) in which case the algorithm always returns from steps 1 or 4.* □

Example 8. Let a and b be constants. Then:

$$\begin{aligned}
&g(a,a,\{\epsilon\}) = \{(a,\{\epsilon\})\}\\
&g((a\ b),(a),\{\epsilon\}) = \{((a\ b),\{\epsilon\})\}\\
&g((a\ b),(b),\{\epsilon\}) = \{((a\ b),\{\epsilon\})\}\\
&g((a\ ?y),(a),\{\epsilon\}) = \{((a?y),\{\epsilon\})\}\\
&g((?x\ b),(a),\{\epsilon\}) = \{((a\ b),\{[?x/a]\})\}\\
&g((?x\ ?y),(a),\{\epsilon\}) = \{((a\ ?y),\{[?x/a]\})\}.
\end{aligned} \tag{7}$$

6.2 Merging a general pattern

We now turn to the case where the pattern p can be any FCG-expression. As with unification, the merge function g is extended with specialized merge functions whenever the pattern is of the form $p = (o...)$ with $o \in \mathcal{O}$.

The includes operator Let us first look at the case where $p = (== e_1 ... e_n)$. The main differences with the simple case is that now neither the order nor the number of elements in the source matters:

Example 9.

$$g_{==}((== b\ a), (a\ b), \{\epsilon\}) = \{((a\ b), \{\epsilon\})\}$$
$$g_{==}((== b\ a), (a), \{\epsilon\}) = \{((a\ b), \{\epsilon\})\} \quad (8)$$

The algorithm presented above can be used to compute the merge of an includes list with only a minimal amount of changes. Let $p = (== |(p_1|p_2))$. First, in step 2, instead of trying to merge p_1 only to the first element of the source, all source elements must be considered. Every source element that merges with p_1 now leads to a case similar to 2(a). The computation of the union of the results for these cases is somewhat more complicated and requires some additional bookkeeping.

If no source element merges with the first pattern element then this leads to a case similar to 2(b). G'_2 is now computed as

$$G'_2 = g((== |p_2), s, B)$$

i.e. the includes operator must be propagated.

Merging an includes list also always terminates for the same reasons as why the merging of simple expressions terminates.

The permutation operator Merging a permutation pattern $p = (==_p e_1 ... e_n)$ is similar to simple merging except that the order of elements in the source is arbitrary. As in the case of the includes operator, this requires that in step 2 all elements in the source are considered instead of only the first. A more easy but possibly less efficient implementation would be to merge the pattern as if it is an includes pattern and only keep those results that are of the same length as the original pattern (without the permutation operator.)

The includes uniquely operator The includes uniquely operator can be used to block merging. Consider for example the patterns

p_1 =
```
((?unit  (form (== (string ?unit "car"))
         (syn-cat (== (number singular))))))
```
and

p_2 =
```
((?unit  (form (==1 (string ?unit "car"))
         (syn-cat (==1 (number singular))))))
```
and the source

s =
```
 ((unit   (form ((string unit "cars")))
          (syn-cat ((number plural))))).
```

The source represents (part of) a syntactic structure. The patterns represent template-poles that are tried to merge with the source to obtain a new syntactic

structure. In this case both patterns are intended to fail because a unit cannot be both singular (as specified by the patterns) and plural (as specified in the source.) However, merging p_1 and s results in

$$g(p_1, s, \epsilon) = \{(((\texttt{unit (form ((string unit "car") (string unit "cars"))) (syn-cat ((number singular) (number plural)))))}, \{[?\text{unit}/\text{unit}]\})\}$$

whereas p_2 and s do not merge: the merging is blocked by the includes uniquely operator.

An includes uniquely pattern can be merged with a source by first treating the pattern as a normal includes pattern and then filtering the result on the conditions of section 4.5. This can be made more efficient by checking whether it is allowed to add a new element to the source in step 2(b) of the merging algorithm.

7 Merging Feature Structures

As with unification, the merging of a pattern feature structure P with a source structure S will be defined as merging a transformed pattern P' with the source. The transformation consists of adding special operators to the pattern. However, the set of special operators defined so far does not suffice. Consider the merging of the pattern:

```
((?unit (sem-cat (== (agent ?e ?a) (human ?a)))),
```

with the following source:

```
((unit (sem-cat ((agent e a) (motion-event e))))).
```

The intended result with bindings $[?e/e, ?a/a]$ is clearly:

```
((unit (sem-cat ((agent e a) (motion-event e) (human a))))).
```

This solution requires that the first includes element is unified with the first source element and that the `human` part is added. However, the first includes element also merges with the second source element by adding `agent` to it, leading to the solution:

```
((unit (sem-cat ((agent e a) (agent motion-event e)
                 (human e))))),
```

with bindings $[?e/\text{motion-event}, ?a/e]$.

In this particular case the spurious solution can be ruled out by changing the includes operator == to an includes uniquely operator $==_1$. However, this is not always possible, and some more general mechanism is needed that allows to specify that feature values like `(motion-event e)` may not be modified during merging.

Therefore, for every special operator $o \in \mathcal{O}$ a non-destructive version $o!$ is defined which behaves the same in unification (i.e. $f_o{=}f_{o!}$) but which differs in merging such that the modification of candidate source elements for an element of a non-destructive pattern is prohibited. In terms of the merge algorithm g in section 6.1 this means that the recursive call to g in step 2 to determine G'_1 is replaced by a call to f_{FCG} and that steps 2(b) and step 3 are not allowed because they modify the source.

By using non-destructive special operators the modification of already present feature value elements can be prohibited. However, there is another problem. Consider the merging of the pattern

```
(==1 (unit1 ==1 (F1 V1))
     (unit2 ==1 (F2 V2))),
```

with the source

```
((unit1)
 (unit2)))
```

One expected result is

```
((unit1 (F1 V1))
 (unit2 (F2 V2))).
```

However, the following is also a valid merge:

```
((unit1 unit2 (F1 V1))
 (unit2 unit1 (F2 V2))).
```

Prohibiting this solution requires the introduction of a final special operator $==_{1l}$ which is equivalent to the includes uniquely operator except that it only allows its elements to be lists.

Definition 15. *The function* **expand-structure(P,S,B)** *which takes a pattern structure P, a source structure S and a set of bindings B is defined as follows. If P is as represented in (5) then the pattern is first transformed to the pattern P':*

$$\begin{aligned}(==_{1l}\ (u_1\ \ ==_{1l}\ (f_{11}\ v'_{11})...(f_{1n_1}\ v'_{1n_1}))\\ \ldots\\ (u_m\ \ ==_{1l}\ (f_{m1}\ v'_{m1})...(f_{mn_m}\ v'_{mn_m}))),\end{aligned} \tag{9}$$

with the new feature values determined as follows: Every non-atomic feature value $v_{ij} = (v_1|v_2)$ in the pattern for which v_1 is not a special operator is replaced by $v'_{ij} = (==!_p|v_{ij})$. If v_1 is a special operator then it is replaced by its non-destructive version. Atomic feature values are left unchanged: $v'_{ij} = v_{ij}$ if v_{ij} is atomic. Expand-structures(P,S,B) is then equal to $g(P', S, \{B\})$.

8 Examples

The examples presented in this section are simplified to focus on the unification and merging aspects of FCG-template application and do not take the J-operator into account.

8.1 Example of syntactic categorisation in parsing

Assume the following syntactic structure, which could be built based on the utterance "Mary walks":

```
Syn=((sentence-unit (syn-subunits (Mary-unit walks-unit)))
     (Mary-unit (form ((string Mary-unit "Mary"))))
     (walks-unit (form ((string walks-unit "walks")))))
```

The structure contains three units: one for both words ('strings') in the sentence, and one to keep these together in a sentence unit. The initial corresponding semantic structure might look like:

```
Sem=((sentence-unit (sem-subunits (Mary-unit walks-unit)))
     (Mary-unit)
     (walks-unit))
```

It does not yet contain any meanings because we are in the beginning of the parsing process before application of the lexical templates.

As explained elsewhere, the first type of templates that is applied during parsing in FCG is concerned with morpho-syntactic transformations and syntactic and semantic categorisations. In parsing, this phase is comparable to more traditional part-of-speech tagging. However, in FCG these templates can be applied both during production and in parsing and the set of form-constraints and syntactic categories (like parts of speech) is open-ended.

The following template categorises the string "walks" as the third-person singular form of the verb-stem "walk":

```
((?unit (form (== (stem ?unit "walk")))
        (syn-cat (==1 (number singular)
                      (person third)))))
<-->
((?unit (form (== (string ?unit "walks")))))
```

While producing, the same rule would be applied to establish the third-person singular form "walks" for the stem "walk".

To test the applicability of the above template while parsing, the right pole must be unified with the syntactic structure. As explained earlier, this requires first the transformation of the pole to the pattern R' (see equation 6):

```
R'=(==1 (?unit ==1 (form (== (string ?unit "walks"))))),
```

followed by the unification of this new pattern with the syntactic structure:

$$Bs = f_{\mathrm{FCG}}(R', Syn, \{\epsilon\}) = \{[?unit/\text{walks-unit}]\}$$

Because this yields a valid set of bindings, the template's right pole can be applied to compute a new, extended syntactic structure `Syn'` (syntactic categorisation rules always work only on syntactic structures). This requires that the template's left pole is merged with the syntactic structure. Therefore, it is first transformed to the pattern L' (see equation 9):

```
L'= (==1l (?unit ==1l (form (==! (stem ?unit "walk") ))
                      (syn-cat (==1! (number singular)
                                     (person third))))),
```

which then is merged with the syntactic structure: $g(L', Syn, Bs) = \{(Syn', Bs)\}$, yielding:

```
Syn'=((sentence-unit (syn-subunits (Mary-unit walks-unit)))
      (Mary-unit (form ((string Mary-unit "Mary"))))
      (walks-unit (form ((string walks-unit "walks")
                         (stem walks-unit "walk")))
                  (syn-cat ((number singular)
                            (person third))))).
```

8.2 Example of lexicon lookup in parsing

Here is next a lexical template associating a predicate-argument structure with the stem "walk":

```
((?unit (referent ?event)
        (meaning (== (walk ?event) (walker ?event ?person)))))
<-->
((?unit (form (== (stem ?unit "walk")))))
```

In parsing, this template is triggered by a successful unification of its right pole with the syntactic structure. Therefore, the pole is first transformed to the pattern R'':

```
R''=(==1 (?unit ==1 (form (== (stem ?unit "walk"))))).
```

It is easy to see that R'' indeed unifies with the syntactic structure `Syn'` from the previous example with unifier Bs'=[?unit/`walks-unit`].

Given successful unification, the left pole can be merged with the semantic structure, yielding the new semantic structure `Sem'`, with $g(L'', Sem, Bs') = \{(Sem', Bs')\}$,

```
L''=(==1l (?unit ==1l (meaning (==! (walk ?event)
                                    (walker ?event ?person)))))
```

and thus

```
Sem'=((sentence-unit (sem-subunits (Mary-unit walks-unit)))
      (Mary-unit)
      (walks-unit (referent ?event)
                  (meaning ((walk ?event)
                            (walker ?event ?person))))).
```

8.3 Example of construction application in production

Assume that conceptualization, lexicalisation and categorisation resulted in the following semantic and syntactic structures:

```
Sem=((sentence-unit (sem-subunits (Mary-unit walk-unit)))
     (Mary-unit (referent person-1)
                (meaning ((Mary person-1))))
     (walk-unit (referent ev-1)
                (meaning (walk ev-1)
                         (walker ev-1 person-1))
                (sem-cat (motion-event ev-1)
                         (agent ev-1 person-1))))
```

and

```
Syn=((sentence-unit (syn-subunits (Mary-unit walk-unit)))
     (Mary-unit (form ((stem Mary-unit "Mary")))
                (syn-cat ((person third)
                          (number singular))))
     (walk-unit (form ((strem walk-unit "walk"))))).
```

The above syntactic structure specifies that there are two lexical items involved (the stems "Mary" and "walk"), reflecting the fact that the meaning to express involves some person `person-1` (Mary) and some walk event `ev-1`. However it is not yet specified that it is Mary who fulfills the role of walker (agent) in the walk event. The following simple SV-construction template can be used for this and uses word order and agreement as would be the case in English:

```
((?SV-unit (sem-subunits (?subject-unit ?predicate-unit)))
 (?subject-unit (referent ?s))
 (?predicate-unit (referent ?p)
                  (sem-cat (==1 (agent ?p ?s)))))
<-->
((?SV-unit (syn-subunits (?subject-unit ?predicate-unit))
           (form (== (precedes ?subject-unit ?predicate-unit))))
 (?subject-unit (syn-cat (==1 NP
                              (number ?n)
                              (person ?p))))
 (?predicate-unit (syn-cat (==1 verb
                              (number ?n)
                              (person ?p))))).
```

Many other syntactic constraints can easily be incorporated into this kind of template. The above template's left pole unifies with the semantic structure `Sem`, with unifier

```
Bs=[?SV-unit/sentence-unit, ?subject-unit/Mary-unit,
    ?predicate-unit/walk-unit, ?s/person-1, ?p/event-1].
```

Thus, a new syntactic structure `Syn'` can be computed by merging the template's right pole with the structure `Syn`: $g(R', Syn, Bs) = \{(Syn', Bs')\}$, with

```
R'=(==1l
    (?SV-unit ==1l
              (syn-subunits (==p! ?subject-unit ?predicate-unit))
              (form (==! (precedes ?subject-unit ?predicate-unit))))
    (?subject-unit
              ==1l
              (syn-cat (==1! NP
                             (number ?n)
                             (person ?p))))
    (?predicate-unit
              ==1l
              (syn-cat (==1! verb
                             (number ?n)
                             (person ?p)))),
```

,

```
Bs'=[?SV-unit/sentence-unit, ?subject-unit/Mary-unit,
     ?predicate-unit/walk-unit, ?s/person-1, ?p/event-1,
     ?n/singular, ?p/third]
```

and

```
Syn'=((sentence-unit
         (syn-subunits (Mary-unit walk-unit))
         (form ((precedes Mary-unit walk-unit))))
       (Mary-unit (form ((stem Mary-unit "Mary")))
                  (syn-cat (NP
                             (person third)
                             (number singular))))
       (walk-unit (form ((strem walk-unit "walk")))
                  (syn-cat (verb
                             (person third)
                             (number singular)))))
```

9 Conclusion

Experiments in the emergence of grammatical languages require powerful formalisms that support the kind of features that are typically found in human natural languages. Linguists have been making various proposals about the nature

of these formalisms. Even though a clear consensus is lacking, most formalisms today use a kind of feature structure representation for syntactic and semantic information and templates with variables and syntactic and semantic categories. There are also several proposals on how templates are to be assembled, centering around concepts like match, unify, merge, etc. although the proposals are often too vague to be operationalised computationally. We argued that computer simulations of the emergence of grammar have some additional technically very challenging requirements: the set of linguistic categories must be open-ended, templates can have various degrees of entrenchment, and inventories and processing must be distributable in a multi-agent population with potentially very diverse inventories.

Fluid Construction Grammar has been designed to satisfy these various requirements and the system is now fully operational and has already been used in a number of experiments.

In this document the unification and merging algorithms used in FCG were formally defined as they form the core of the system. It was shown that FCG unification is a special type of multi-subset-unification, which is inherently of exponential complexity in the length of the expressions that are unified. FCG unification always returns a complete but not necessarily minimal set of unifiers. FCG merging was properly defined and it was shown that it always terminates.

The unification of a source with an includes list ($==$) was formally defined and the unification of a permutation list ($==_p$) and of an includes-uniquely list ($==_1$) were shown to be special cases hereof. These made it possible to define the matching of structures, needed for FCG template application in terms of the general unification function. Non-destructive versions of these operators were introduced to enable the definition of FCG structure merging in terms of the general merging function.

FCG can be used without considering all the technicalities discussed in the present paper, but these details are nevertheless of great importance when constructing new implementations.

10 Acknowledgment

The research reported here has been conducted at the Artificial Intelligence Laboratory of the Vrije Universiteit Brussel (VUB) and at the Sony Computer Science Laboratory in Paris. Joachim De Beule was funded as a teaching assistant at the VUB. Additional funding for the Sony CSL activities has come from the EU FET-ECAgents project 1170. Many other researchers have been important in shaping FCG. We are particularly indebted to Nicolas Neubauer for early work on the unification and merge algorithms, Josefina Sierra for an early re-implementation in Prolog, and to Martin Loetzsch for recent contributions towards making FCG a more professional software engineered artifact. Other contributions have come from Benjamin Bergen, Joris Bleys, Remi Van Trijp, and Pieter Wellens.

References

1. Batali, J. (2002) The negotiation and acquisition of recursive grammars as a result of competition among exemplars. In Ted Briscoe, editor, Linguistic Evolution through Language Acquisition: Formal and Computational Models. Cambridge University Press.
2. Bergen, B.K. and Chang, N.C.: Embodied Construction Grammar in Simulation-Based Language Understanding. In: Ostman, J.O. and Fried, M. (eds): Construction Grammar(s): Cognitive and Cross-Language Dimensions. John Benjamin Publ Cy., Amsterdam (2003)
3. Briscoe, T. (ed.) (2002) Linguistic Evolution through Language Acquisition: Formal and Computational Models. Cambridge University Press, Cambridge, UK.
4. Cangelosi, A. and D. Parisi (eds.) (2001) Simulating the Evolution of Language. Springer-Verlag, Berlin.
5. Chomsky, N.: Logical Structure of Linguistic Theory. Plenum (1955)
6. Croft, William A. (2001). Radical Construction Grammar; Syntactic Theory in Typological Perspective. Oxford: Oxford University Press.
7. De Beule, J. and B. Bergen (2006) On the emergence of compositionality.Accepted for the sixth evolution of language conference, Rome, 2006
8. De Beule, J. and Steels, L. (2005) Hierarchy in Fluid Construction Grammar. In Furbach U., editor, Proceedings of KI-2005, pages 1–15. Berlin: Springer-Verlag.
9. Degyarev, A., Voronkov, A.: Equality Elimination for Semantic Tableaux. Tech. report **90**, Computer science department, Uppsala University, Upsalla, Sweden (1994)
10. Dovier, A., Pontelli, E., Rossi, G.: Set Unification. arXiv:cs.LO/0110023v1 (2001)
11. Goldberg, A.E. (1995) Constructions: A construction grammar approach to argument structure. University of Chicago Press, Chicago.
12. Hashimoto, T. and Ikegami, T. (1996) Emergence of net-grammar in communicating agents. Biosystems, 38(1):1–14.
13. Hagoort, P.: On Broca, brain and binding: a new framework. Trends in Cognitive Science **9(9)** (2005) 416–423
14. Jackendoff, R.: Foundations of Language: Brain, Meaning, Grammar, Evolution. Oxford University Press (2002)
15. Kay, M.: Functional unification grammar: A formalism for machine translation. Proceedings of the International Conference of Computational Linguistics (1984)
16. Kapur, D. and Narendran, P.: NP-completeness of the set-unification and matching problems. In: Proceedings of the Eighth International Conference on Automated Deduction. Springer Verlag, Lecture Notes in Computer Science **230** (1986) 289–495
17. Langacker, R.W. (2000) Grammar and Conceptualization. Mouton de Gruyter, Den Haag.
18. Minett, J. W. and Wang, W. S-Y. (2005) Language Acquisition, Change and Emergence: Essays in Evolutionary Linguistics. City University of Hong Kong Press: Hong Kong.
19. Pollard, C. and Sag, I.: Head-driven phrase structure grammar. University of Chicago Press (1994)
20. Russell S.J., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd edition. Upper Saddle River, New Jersey 07458, Prentice Hall, Inc. (2003)
21. Sierra-Santibàñez, J.: Prolog Implementation of Fluid Construction Grammar. Presented at the first FCG workshop, Paris (2004)

22. Sterling, L. and Shapiro, E.: The art of PROLOG. MIT Press, Cambridge, Massachusetts (1986)
23. Steels, L.: Self-organizing vocabularies. In: Langton, C. (ed.): Proceedings of the Conference on Artificial Life V (Alife V) (Nara, Japan) (1996)
24. Steels, L., M. Loetzsch and B. Bergen (2005) Explaining Language Universals: A Case Study on Perspective Marking. [submitted]
25. Smith, K., Kirby, S., and Brighton, H. (2003) Iterated Learning: a framework for the emergence of language. Artificial Life, 9(4):371–386
26. Steels, L. (1998) The origins of syntax in visually grounded robotic agents. Artificial Intelligence, 103(1-2):133–156.
27. Steels, L. (2004) Constructivist Development of Grounded Construction Grammars Scott, D., Daelemans, W. and Walker M. (eds) (2004) Proceedings Annual Meeting Association for Computational Linguistic Conference. Barcelona. p. 9-19.
28. Steels, L. (2005) The emergence and evolution of linguistic structure: from lexical to grammatical communication systems. Connection Science, 17(3-4):213–230.
29. Steels, L., De Beule, J., Neubauer, N.: Linking in Fluid Construction Grammar. In: Transactions Royal Flemish Academy for Science and Art. Proceedings of BNAIC-05. (2005) p. 11-18.