

Basics of Fluid Construction Grammar

Paper under review for Constructions and Frames

Luc Steels
Institute for Advanced Studies (ICREA)
IBE (UPF-CSIC), Barcelona

August 30, 2016

Abstract

Fluid Construction Grammar (FCG) is a fully operational computational platform for developing grammars from a constructional perspective. It contains mechanisms for representing grammars and for using them in experiments in language understanding, production and learning. FCG can be used by grammar writers investigating some aspect of language and by computational linguists implementing practical language processing systems or exploring how machine learning algorithms can acquire grammars. This paper introduces some of the basic mechanisms of FCG, starting from the more familiar formalism of rewrite grammars. It focuses in particular on the structure of construction schemas and how they are applied in the process of comprehension and production.

Keywords: Construction Grammar, Computational Construction Grammar, Fluid Construction Grammar.

1 Introduction

Despite the great interest and growing success of construction grammar in empirical linguistics, diachronic linguistics, language teaching, child language research, and other fields of language studies, there is still no widely accepted formalism for construction grammar nor a robust easily accessible computational implementation that operationalises how constructions are used in parsing or producing utterances. But the good news is that there are a number of active research programs trying to fill this gap. Some of the main examples (in alphabetic order) are: Embodied Construction Grammar [Bergen and Chang(2003)], Fluid Construction Grammar [Steels(2011)], Sign-based Construction Grammar [Boas and Sag(2012)] and Template Construction Grammar [Barres and Lee(2014)]. These various computational platforms have obviously many things in common, such

as the fundamental principle that the core unit of grammar are signs, i.e. pairings of forms with meanings or functions. But there are also significant differences, often due to the computational mechanisms with which they have been built. Each of these formalisms is also still undergoing substantial evolution as their developers try to tackle new aspects of language or new application areas. This paper focuses exclusively on Fluid Construction Grammar (FCG). It introduces some of its basic principles as succinctly as possible. Comparisons to other formalisms are found in [Chang and Micelli(2011)] and [van Trijp(2013)].

Due to the rich nature of the structures underlying human languages, a computational platform for construction grammar requires a lot of expressive power and must support complex structural operations. It should support tracing language processing and provide tools for managing large inventories of constructions. It is therefore inherently difficult to summarise any constructional platform in a single paper. Here I just try to give a very brief summary of the main ideas behind FCG, abstracting as much as possible away from implementation details in order to make the paper accessible to linguists. To this purpose, I also introduce a new notation compared to earlier publications on FCG, which has now been fully operationalized as well.

If you are curious for more, I refer to the growing literature on FCG which exposes the underlying design principles, describes the basic mechanisms more fully, and provides many concrete examples [Steels(2011)], [Steels(2012b)] and the other papers in this special issue. A recent paper [Steels and Szathmary(2016)] sketches the broader view on FCG as a biological system. Studying the available web resources, including demonstrations and an on-line course, through <http://www.fcg-net.org/> is also strongly recommended. The present paper has also an associated web demonstration (accessible through <http://www.fcg-net.org/demos/basics-of-fcg>) which implements all the constructions used in the paper. In any case, the only way to really understand FCG is to download the FCG software, which can be done from the same site, and start writing grammars yourself, just like you only learn how to swim by jumping in the water. But keep in mind that FCG is still undergoing profound evolution.

Here is an important preliminary point: FCG is as neutral as possible with respect to the linguistic theory a linguist may want to explore, as long as the analysis adopts a constructional perspective. The set of possible grammatical categories, types of units, relations between units, possible constructions, relations between constructions, etc. is entirely open. For example, one can explore a verb template approach just as easily as a constructionist approach to argument-structure [Goldberg(2015)], or emphasize phrase structure versus dependency grammar. When I show examples in this paper, they should be seen as no more than examples, more precisely didactic, highly simplified examples. They are not claims that this is the only or

the best way in which a particular phenomenon has to be handled.

There are two reasons for taking this stance:

(i) The history of formal and computational analyses shows that agreement between linguists is hard to come by. This might simply be because there are often many ways to handle the same phenomena, just like the same computational function can often be achieved through many different algorithms and implemented in very different programming styles.

(ii) We are ultimately interested in how grammars are built and acquired autonomously by learning systems. Different learning algorithms will undoubtedly come up with different grammars depending on what learning operators they use or what sequences of examples they get. So a formalism has to be as open as possible.

Of course, all this does not mean that we should refrain from performing detailed linguistic analysis and formalisation to see whether the representational and computational mechanisms of FCG are adequate, efficient, and ‘user-friendly’ enough to build large-scale natural grammars.

The rest of the paper introduces first the semiotic cycle (section 2) and then the basic units of FCG: transient structures (section 3) and constructions (section 4). Section 5 adds meaning to this picture and section 6 introduces a slightly more complex example of the double object construction.

2 The Semiotic Cycle

2.1 Components of the semiotic cycle

A language interaction requires that speaker and hearer go through a set of processes, called the *semiotic cycle* [Steels(2015)]. To produce an utterance, the speaker needs to achieve the following tasks [Levelt(1989)]:

- **Perception and Action:** Both speaker and hearer must perceive the shared context to build and maintain a model of this environment and the actions that make sense in their interaction.
- **Conceptualization:** The speaker must decide which communicative goal should be achieved and has to conceptualize reality in order to come up with the meanings he could convey to achieve that goal. For example, the speaker might want to draw the attention of the hearer to an object in the shared context.
- **Production:** The speaker must translate meaning into an utterance based on the lexicon and grammar of the language, which implies building a set of intermediary semantic and syntactic structures. Input to production is the meaning to be expressed.

- **Rendering:** The speaker uses the result of the formulation-process to render the utterance through written or spoken forms.

To achieve understanding, the hearer needs to accomplish the following tasks:

- **De-rendering:** The hearer must extract basic observable features from the sounds emitted by the speaker or from the written text. Output is the sequence of phonemes, intonation and stress structures, or the set of words ordered sequentially.
- **Comprehension:** The hearer must reconstruct the meaning using his own lexicon and grammar. This includes not only recovering the grammatical structures underlying the utterance, a process usually called parsing, but also recovering from these structures the meaning. In construction grammar there is no strict distinction between these two processes.
- **Interpretation:** The hearer must interpret this meaning in terms of his own world model and possibly act on it. For example, look at the object described by the speaker or point at it.

These various tasks are not performed in a sequential order. The speaker may already start rendering part of an utterance while still working out the grammar and the meaning to be expressed. He may even have to focus his perceptual attention on some additional aspect of the scene to find distinctive properties for introducing one of the objects appearing later in the utterance. The hearer is already interpreting meanings reconstructed by the comprehension processes and may start steering his perception to find which objects in the world are being talked about, while still having to process a large part of the utterance.

Control should be able to flow in a top-down and bottom-up manner. Speakers often self-monitor their own speech. They simulate how hearers would comprehend and interpret fragments of the utterance they are about to render in order to gauge the potential for communicative success, while hearers may simulate how the speaker conceptualizes and expresses meaning, in order to predict what the speaker is about to say or in order to learn new constructions.

It is also often the case that there is more than one hypothesis of an utterance being worked on. There are usually multiple ways to achieve a communicative goal or to express a particular meaning and there are often multiple meanings and functions for the same words or grammatical patterns. There is unavoidably a search space. So the different processes in the semiotic cycle should be conceived as operating in parallel and are highly interlaced. FCG is primarily concerned with production and comprehension but can call upon the other processes in the semiotic cycle at any time.

2.2 Linguistic Pathways

It views production and comprehension in terms of a chain of consecutive operations over a linguistic structure, called the *transient structure*. A sequence of transient structures on a particular execution chain is called a *linguistic pathway*. The FCG-engine orchestrates the whole process using an inventory of construction schemas called an FCG-grammar. The FCG-grammar contains both lexemes (construction schemas for lexical items) and grammatical constructions (construction schemas for grammatical constructions) and there is no formal or computational difference between the two, so an FCG-grammar includes both the lexicon, idiomatic expressions, as abstract grammatical constructions. A graphical user-interface (the FCG-interface) is available for tracing operations and for inspecting transient structures and construction schemas.

In comprehension, the initial transient structure contains all information derived from the input utterance by derendering: which words, morphemes, ordering relations, intonation and stress patterns are present. Operations then expand the transient structure until the meaning can be derived. It is possible to supply information about the utterance in a step-wise fashion from left to right and each time the FCG-engine does as much as it can with the available construction inventory and the input so far. It is also possible to provide all information at once. In production, the initial transient structure contains the meaning to be expressed and then consecutive operations expand this structure until all the information is there to render an utterance. Again, the target meaning could be supplied all at once, or it could be provided step-wise as the conceptualization process unfolds and partially based on demands from the production process.

Operations that expand transient structures are of three types:

1. **Application of a construction.** The FCG-engine has access to an inventory of *construction schemas* (usually simply called constructions). Each schema defines a particular construction by specifying a set of constraints. Construction schemas are applied by the FCG-engine using an operation called *U-Merge*. Each construction schema contains a *lock* and the transient structure must fit in the lock. The FCG-engine figures this out using a *matching process*, using a function called *U-Match* because it is based on unification. If there is a fit, then information from the contributing part of a construction, called the *contributor*, is added to the transient structure, through a *completion process*, called *U-Complete*, also operationalized using unification. The combination of first using U-match and U-complete together constitute *U-merge*. Often U-merge is simply called merge and U-match match.

We will see later that there are in fact two locks, a comprehension

and a production lock, so that constructions can be applied in both directions: from form to meaning (typical for comprehension) and from meaning to form (typical for production).

2. **Expansion based on semantics.** This happens in comprehension when a fitting construction schema is not (yet) available to the FCG-engine and the interpretation process is called upon to suggest possible ways in which words or phrases could be linked together based on the shared context or ‘common sense’ knowledge that suggests possible relations between meanings. Expansion based on semantics make it possible to continue a comprehension process even if construction schemas are missing or the input is ungrammatical, and it can help to prune possible hypotheses.
3. **Activation of a meta-operator.** Meta-operators allow speakers and listeners to expand or stretch their grammars in order to deal with gaps or unexpected situations. The meta-operators consist of diagnostics which track whether production or comprehension is running smoothly and signal a problem when it appears, for example, when there is a word or grammatical construction missing to express a particular meaning, when all words have been parsed but the derived syntactic structure fragments cannot be fully integrated, whether there is combinatorial search in parsing, etc. Repairs become active based on the output from these diagnostics and they introduce new constructions, recategorize existing words, etc.

In this paper I focus exclusively on (1), i.e. the application of constructions, although the FCG-interpreter smoothly handles (2) and (3) as well. FCG has various utilities to write these diagnostics and repairs, including an *anti-unification* operator that discovers commonalities between constructions and differences between constructions. These are discussed in [Beuls, van Trijp and Wellens(2012)] and [Steels and Van Eecke(2016)].

3 Transient Structures

One of the characteristics of construction grammar is its “insistence on simultaneously describing grammatical patterns and the semantic and pragmatic purposes to which they are dedicated” [Fillmore(1988)][p.36] so transient structures, as well as constructions, need to be able to represent information from a multitude of different perspectives. If we see a complete language system (relating sound to meaning) as a big cake, than traditionally linguistics cuts the cake in different horizontal layers: phonology, morphology, syntax (with phrase structure grammar, functional grammar, dependency grammar, case grammar), semantics (with propositional structure, argument structure, temporal structure, modality) and pragmatics

(including information structure and dialog management) Each layer has its own autonomy, its own representational structures, and its own set of rules with modular interfaces between the layers. The distinction between the c-structure (phrase structure) and the f-structure (functional structure) in Lexical Functional Grammar is a good example [Bresnan(2001)]. Instead, construction grammarians like to cut the cake in a vertical fashion, freely combining information across different layers. For example, the same construction may contain phonological constraints, syntactic constraints, meanings and some pragmatics. The layers are seen as perspectives on the utterance and constructions can cross different perspectives.

3.1 Feature Structures

FCG represents transient structures using (extended) feature structures, familiar from many other contemporary grammar formalisms. An FCG *feature structure* consists of a set of units, that may correspond to lexical items (words, stems, affixes) or groupings of these items (for example phrases but other groupings are possible as well). Each unit has a name. The name has to be unique within the context of a particular transient structure.

Each unit has a *feature-set* which is a set of features and values. There is no ordering assumed among the features. The features provide information from different linguistic perspectives. Aspects of the meaning or of the form of an utterance are represented as features and sprinkled over different units in comprehension or production and collected at the end of processing to provide the total meaning or total form of the utterance.

Units. Feature structures are familiar from other grammar formalisms such as Unification Grammar [Kay(1984)] or HPSG [Sag et al.(2003)], but there are some small but important differences in their usage here. In classical feature structure formalisms [Shieber(1986)], there are no explicit units, only feature-sets, and consequently no names to address units directly, only path descriptions. As we will see, reifying units makes it possible to obtain clearer structure and names of units make it possible to directly refer to units, for example to represent what the subunits are of a unit or which dependency relations hold between them.

It is entirely open what counts as a unit and what the possible features and values of units are. It depends on the linguistic theory the linguist wants to experiment with, or what categories and structures a learning algorithm might come up with. Moreover the units in a transient structure form a set and each unit is accessible independently within that set through its name, without traversing a path in a tree. This gives enormous flexibility to deal with languages that do not have a strict phrase structure or use strict sequential ordering, and it allows multiple structures to be imposed on the utterance (e.g. a dependency structure *and* a constituent structure).

Symbols. Feature structures use a lot of symbols: for unit-names,

feature-names, values, predicates, objects in the domain of discourse, etc. Some symbols are special and known to the FCG-interpreter, however most of them are specific to an FCG-grammar. The name of these symbols does not (indeed should not) play any role in its functioning. Their meaning is uniquely determined by its occurrence in all the feature structures and construction schemas in which they appear. For example, it is not by calling a unit *subject* or a feature *number* that the FCG-interpreter knows what these symbols stand for. A unit which is acting as the subject could just as well be called *u-85773* and gender *b444adb*, as long as the same symbol-name is used everywhere else where reference is made to the same unit. But it is of course much better to use symbol-names that make sense *for us*, because that makes it easier to follow what a symbol is about. Once grammars become more serious, there will be tens of thousands of symbols, and even when processing a moderate-sized utterance, hundreds of novel symbols are created and used within a transient structure. Choosing good symbol-names is one of the first rules of good programming practice and it applies also here.

Often we need many tokens of the same type, for example, there may be several adjectives in an utterance, in which case we use a type-name followed by an index like *adjective-1*, *adjective-2*, etc. The indices, as well as the indices of variables, categories, objects or other symbols of which there are many tokens, are automatically constructed by the FCG-engine.

Values. Values of features can be of different types: an atomic symbol, a set, a sequence, a feature-set, a Boolean expression, a formula (i.e. predicate-argument expression), or an expressions with special operators, to be defined soon. The type of value assumed for a particular feature is declared explicitly, and the default is an atomic value or feature-set. Only atomic symbols, sets, formulas and feature-sets are used in this paper to simplify the discussion. Sets are written with curly brackets { and }. Sequences are written with square brackets [and]. The elements are written between commas. A formula has a predicate and a number of arguments as in: *predicate(arg₁, ... , arg_n)*.

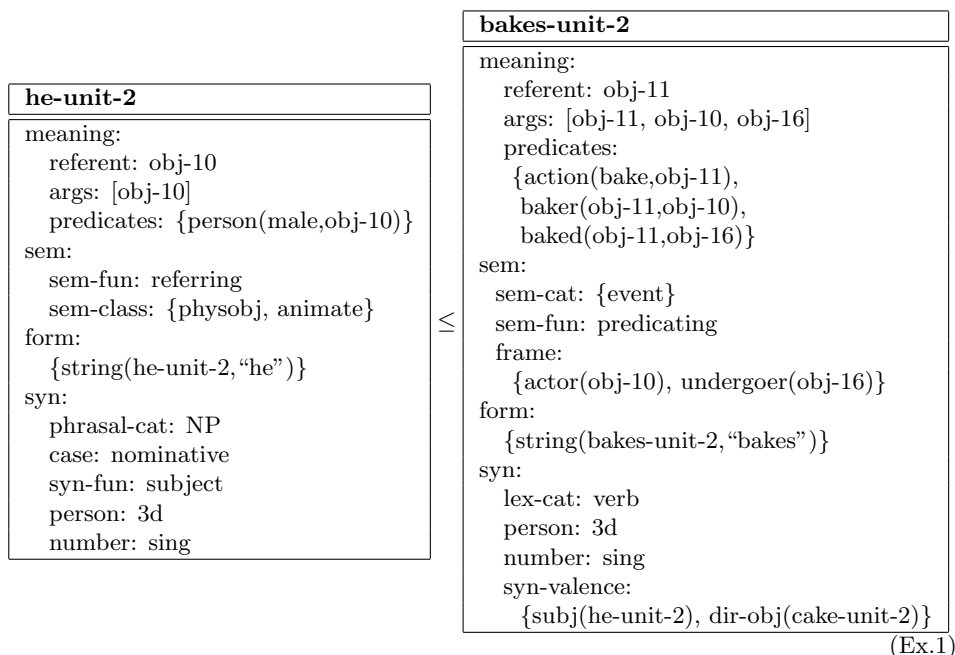
Variables. Variables are written with a question-mark as the first character of the symbol, so that we see immediately that the symbol is a variable. Examples are: *?NP-unit* or *?number*. Any element in a feature structure (symbols, units, sets, formulas, feature-sets) can be a variable, except feature-names. It is recommended that variables have understandable rather than arbitrary names. FCG variables have the same behavior as logic variables in logic programming languages such as PROLOG, and they are bound and references through the unification operation, which is one of the fundamental algorithms of computer science [Knight(1989)].

We follow tradition by representing a list of variable bindings, also called a substitution, as a set of dotted pairs:

$$\{(variable_1 . binding_1) \dots (variable_n . binding_n)\}$$

Each unit in a transient structure is represented as a box with the name of the unit on top and the feature-set of the unit below it. The semantically-oriented features, called the *semantic pole*, are usually written above the syntactically-oriented features, called the *syntactic pole*, but there is no fixed ordering of the features and no *a priori* defined set of features that must occur obligatory. The FCG user interface can display the syntactic and semantic poles of units separately, which is helpful when inspecting complex transient structures, but in this paper they are always displayed as one set.

Ex.1 is a transient structure related to the utterance “he bakes”:



(Ex.1)

There are two units, one for each word. The units are called *he-unit-2* and *bakes-unit-2*. (Recall that these indices are made up by the FCG-engine. They could just as well have been *he-unit-5* and *bakes-unit-123*.) We see here meaning and semantics-oriented features, such as *meaning* with a *referent*, *args*, and *predicates* and *sem* (semantic properties) with *sem-cat* (semantic categories), *sem-fun* (semantic function) and *frame* (semantic valence frame). We also see form and syntactic-oriented features, such as *form* which specifies the *string*, and *syn* (syntactic properties) with *syn-cat* (syntactic categories), *syn-valence* (syntactic valence) and *syn-fun* (syntactic function). *he-unit-2* similarly represents various aspects of an occurrence of the word “he”, including its meaning, semantic features, form, and syntactic features. Note that this is just one way to capture all this information. In principle a grammar engineer is entirely free about which features to use or how to structure them.

Units may have a form-feature which collects information about the form

of the unit in the form of a set of predicates. One of these predicates is *string* (in the case of the comprehension or production of written language). It takes two arguments: the unit and the string itself, as in

form: {*string*(*he-unit-2*, “*he*”)}

where *he-unit-2* is the name of a unit.

The same form-feature also stores information about sequential ordering using two predicates: *precedes* and *meets*. These predicates have three arguments, two units which are left and right from each other and a unit which acts as a context. A unit u_1 *precedes* a unit u_2 iff the wordform(s) of u_1 precede those of u_2 . A unit u_1 *meets* a unit u_2 iff the wordform(s) of u_1 immediately precede those of unit u_2 . For example, to represent that the form of *he-unit-2* (namely “*he*”) immediately precedes the form of *bakes-unit-2* (namely “*bakes*”) we write:

form: {*string*(*he-unit-2*, “*he*”),
string(*bakes-unit-2*, “*bakes*”),
meets(*he-unit-2*, *bakes-unit-2*, *np-unit-5*)}

np-unit-5 is the context in which the *meets* relation holds. In the simplified notation used in this paper, the ordering relations are not represented explicitly but a <-sign is put between the relevant units for the *precedes*-relation and a ≤-sign for the *meets*-relation.

The units in a transient structure form an unordered set. Various relations can be imposed on the units and they are explicitly represented using features and values. For example, hierarchical phrase structure can be represented with a feature *constituents* which contains as its values the list of phrasal constituents of a unit, or if you want to represent dependency relations, you can do it with a feature *dependents*. All this is up to the grammar designer. FCG does not insist on trees (and certainly not on binary trees). Some perspectives (such as phrase structure) impose a tree on units but other perspectives (such as dependency structure) impose a graph, and sometimes (particularly while formulating) it is not yet known how units relate to each other, so it would not be possible to draw a tree or a graph.

Ex.1 is simplified. Dozens more features are typically needed to represent all information needed in comprehension and production. The different features have been structured here into different subsets and features like meaning, sem, syn, form, etc., that have as value another set of features. However, in the examples later in the paper, I will use a single feature-set to simplify the examples so that they fit on a page. For example, *he-unit-2* (Ex.1) is represented as:

he-unit-2
referent: obj-10
args: [obj-10]
predicates: {person(male,obj-10)}
sem-fun: referring
sem-cat: {physobj, animate}
form: {string(he-unit-2, "he")}
phrasal-cat: NP
case: nominative
syn-fun: subject
person: 3d
number: sing

(Ex.2)

What features are used, how they are combined into feature-subsets and how they are labeled is entirely in the hands of the grammar designer, although obviously teams of grammar designers should try to reach a consensus to make exchange and combination of grammar fragments possible.

4 Construction schemas

A *construction schema* is an abstract schema that can be used to expand any aspect of a transient structure from any perspective and it can consult any aspect of this transient structure to decide how to do so. Construction schemas are not merely intended to be descriptive, they should provide enough information to operationalize comprehension and production.

FCG construction schemas use the same representation as transient structures, i.e. they are formulated in terms of named units, features, and values. Any symbol in a construction schema may be variable, including the name of a unit, except feature names.

Although construction schemas are represented using feature structures, they are (more) abstract than transient structures because many of its elements are variables instead of concrete elements, some of the features are not specified or only partially specified, and a construction schema may contain fewer units. The inventory of construction schemas available to a language user is sometimes called his FCG-grammar *constructome*. As a first approximation, we assume that all construction schemas form an unordered set (but see section 7).

To understand better how the FCG-engine uses construction schemas, I will start from rewrite rules, a formalism that every linguist is familiar with, and progressively introduce innovations to arrive at FCG construction schemas. I will also use first a simple example from phrase structure grammar, again, because phrase structure is familiar to all linguists. But keep in mind that functional grammar, dependency grammar, case grammar, information structure or any other grammatical perspective can be accommodated and used just as easily, and that construction schemas can

combine information from any of these perspectives.

4.1 Rewrite Rules using Feature Structures

A typical set of rules in a very simple phrase structure grammar for English might be:

- [1] NP \rightarrow art noun
- [2] art \rightarrow the
- [3] noun \rightarrow girl

Such rules are usually interpreted in a *generative* way, namely, if you have the symbol(s) on the left-hand side of a rule then you can replace them by the symbols on the right-hand side. For example, if you start from NP as the initial symbol, you can derive the sentence “the girl” by application of [1] then [2] and then [3]:

$$\text{NP} \xrightarrow{1} \text{art noun} \xrightarrow{2} \text{“the” noun} \xrightarrow{3} \text{“the girl”}$$

FCG is not intended for generating random sentences but for modeling comprehension and production processes. Even though the term generation is often used for production as well, we make a strict distinction here. Production is coming up with an utterance (and its underlying structure) that expresses a particular meaning according to the constraints imposed by the grammar. Generation is coming up with a random example utterance (and its structure) that conforms with the constraints defined by the grammar.

A rewrite grammar can be used in the parsing phase of comprehension by applying rules in the reverse direction: An occurrence of the symbol(s) on the right-hand side of a rule is replaced by the symbol on the left-hand side. For parsing “the girl”, we get the following chain of rule applications:

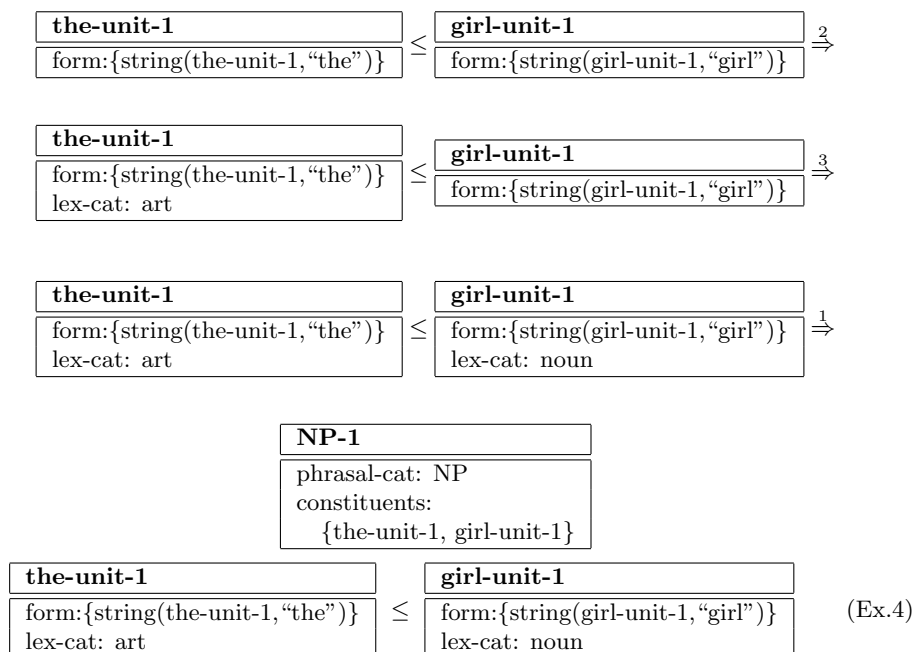
$$\text{“the girl”} \xrightarrow{2} \text{art “girl”} \xrightarrow{3} \text{art noun} \xrightarrow{1} \text{NP} \quad (\text{Ex.3})$$

However a rewrite grammar is too underconstrained to be used in production, as soon as the same symbol occurs more than once on the right-hand side of a rule. For example, NP can be rewritten in hundreds of different ways (NP \rightarrow N, NP \rightarrow Art N, NP \rightarrow NP prep NP, etc.). A generator can just pick randomly one of these, but a formulator has to choose one option, based on semantics and many other criteria that are not expressed in a rewrite grammar.

In the late nineteen seventies, almost all computational formalisms started to replace atomic symbols by feature structures, as pioneered in logic-based (definite clause) grammars [Pereira and Warren(1980)] and unification-based grammars [Kay(1984)]. FCG does this too. For each atomic symbol there will be a unit. The feature-set associated with this unit specifies its properties. To represent phrase structure, the relevant properties are the string and lexical category for word units, and the phrasal category, the subunit-relations and their ordering for phrasal units. The application of a rule

no longer replaces the occurrence of symbol(s) on its right-hand side by the symbol on the left-hand side, but rather adds the information on the left-hand side to the transient structure.

Starting from two units `the-unit-1` and `girl-unit-1` representing “the” and “girl” respectively, the same parsing sequence as in Ex.1 then looks like this:



Keep in mind that the units in a transient structure form a set and that sequential orderings are written using the \leq -sign.

The use of feature structures as a way to represent the current state of parsing has a number of significant advantages. A parser can dispense with an external memory device like a stack because all information to continue the computation is contained in the transient structure itself. It is also easy to set up a search space. Each node in the space simply consists of a transient structure. If more than one construction can expand a transient structure, then several nodes are added to the search space and different strategies (depth-first, best-first, etc.) can be applied. The FCG-engine is able to handle such search spaces and comes with an initial set of strategies.

4.2 Representing construction schemas

The next step is to translate rewrite rules into feature structures as well. I start calling rules from now on construction schemas or simply constructions and talk about construction application instead of rule application. I will also change the direction of the arrow so that it is clear that we are no longer talking about standard generative rewrite rules. The right hand side of the construction schema is called the *lock* because the transient structure has to

fit in the lock as established by the matching process. The left hand side is called the *contributor* because it contains information that will be added to the transient structure by the U-Complete operation, once the lock has been opened.

The first step, to translate the rules themselves, is straightforward. The atomic symbols in a rule are replaced by units with names, features and values. Instead of names of specific units (as in a transient structure), the unit-names are variables that are to be bound to the concrete units found in a transient structure, because the construction schema is an abstract schema that should be applicable to many different transient structures.

The second step is to add more sophistication to the application process. Instead of simply comparing two symbols, the interpreter has to find for every unit on the right-hand side of a construction a matching unit in the transient structure, and, if this is the case, the information on the left-hand side is added to the information already present in the transient structure. As mentioned earlier, information is added. The matching units are not replaced, as in parsing with traditional rewrite rules.

The details of the matching and merging processes are tricky, partly because of the logic variables, but they are entirely similar to unification with logic variables as is the basis of logic programming languages. These details are presented in the appendix.

Con.1a is a translation of rule (1) in the rewrite grammar given earlier. The unit-names are ?NP-unit, ?art-unit, and ?noun-unit. Although units in constructions have the same characteristics as in transient structures, I will use left and right square brackets (as in Con.1a) instead of boxes (as in Ex.4) for the different units, to make the distinction visually clear.

$$\left[\begin{array}{l} \text{?NP-unit} \\ \hline \text{phrasal-cat: NP} \\ \text{constituents: \{?art-unit, ?noun-unit\}} \\ \text{number: ?number} \end{array} \right] \leftarrow \left[\begin{array}{l} \text{?art-unit} \\ \hline \text{lex-cat: article} \\ \text{number: ?number} \end{array} \right] \leq \left[\begin{array}{l} \text{?noun-unit} \\ \hline \text{lex-cat: noun} \\ \text{number: ?number} \end{array} \right] \quad (\text{Con.1a})$$

The right-hand side defines two units, ?art-unit and ?noun-unit, which have the values article and noun for their lex-cat feature. The left-hand side introduces a new unit for a noun-phrase, which has a subset with two units: ?art-unit and ?noun-unit. The ?NP-unit has the value NP for its phrasal-cat feature.

Feature structures have become so widespread because they can easily handle all sorts of phenomena which defy traditional rewrite rules, for example, representing grammatical features (such as gender, number, case, tense, aspect, etc.) and dealing with grammatical agreement and percolation of feature values from constituents to their parent phrasal units.

For example, English requires agreement for number and person between subject and verb and then percolation of number, person and gender from the head of a noun-phrase to the noun-phrase as a whole. We can express this easily by using the same variables in each unit participating in agreement or percolation. This is illustrated in Con.1a. The ?NP-unit, the ?art-unit and the ?noun-unit all have the same number-value: ?number. This constrains the application of this construction schema. It now can become active only when the article and the noun bind to the same number-value ?number and this value then percolates as the number-value of the NP in the merge phase.

Now we make another seemingly trivial but very important step: We allow that a unit appearing on the right-hand side re-appears on the left-hand side. In this case, no new unit needs to be made but the information on the left-hand side will be added to the already existing unit when the construction schema is applied.

This is illustrated in Con.1b. The ?art-unit and ?noun-unit not only appear on the right-hand side but also on the left-hand side with additional information about the function of these units, namely that they are the determiner of the noun-unit and the head of the noun-phrase respectively:

$$\left[\begin{array}{l} \text{?NP-unit} \\ \hline \text{phrasal-cat: NP} \\ \text{constituents:} \\ \quad \{?art-unit, ?noun-unit\} \\ \text{number: ?number} \end{array} \right] \left[\begin{array}{l} \text{?art-unit} \\ \hline \text{syn-fun:} \\ \quad \text{determiner: ?noun-unit} \end{array} \right] \left[\begin{array}{l} \text{?noun-unit} \\ \hline \text{syn-fun:} \\ \quad \text{head: ?NP-unit} \end{array} \right] \leftarrow$$

$$\left[\begin{array}{l} \text{?art-unit} \\ \hline \text{lex-cat: article} \\ \text{number: ?number} \end{array} \right] \leq \left[\begin{array}{l} \text{?noun-unit} \\ \hline \text{lex-cat: noun} \\ \text{number: ?number} \end{array} \right] \quad (\text{Con.1b})$$

This constructions builds both the phrase structure and the functional structure (using determiner and head). The fact that units on the left-hand side can add information to existing units has an important consequence. The existing unit might be incompatible with what the construction schema would like to add to the transient structure. The U-Match operation, which compares the right-hand side with the transient structure, could either block or allow a construction schema to be considered further. Now the merging operation, which adds information from the left-hand side to the transient structure, can also block further application of the construction schema as well.

By allowing the same units to appear on both sides of the arrow, we break away from a possible generative usage of the grammar. Generation requires that the symbols on the left-hand side are either the initial symbol or symbols that have been added earlier. This is no longer the case here. So $X \leftarrow Y$ can no longer be interpreted as “ X can be rewritten as Y ” but rather as “ X can be projected (added/imposed) on the transient structure

which is subsumed by Y ".

4.3 Interfacing

The de-rendering task in the semiotic cycle transforms the input into a list of elements which are sequentially ordered. These elements are usually words or possibly smaller units like morphemes or even phonemes. There could also be gestures or features that involve several words such as intonation contours. All this information is put in a unit called ‘root’ (also called sometimes the base, shell, or input buffer in other FCG publications). This root-unit is a unit in every transient structure and it is unique.

De-rendering creates a unit-name for each of the elements in the input, but no real units yet. The root-unit has a form-feature that contains a description in terms of predicates of the form of the utterance. Which predicates are used is entirely up to the grammar designer. Typical example predicates are *string*, *meets*, *preceeds*, etc. For the phrase “the girl”, the root looks as follows:

root	(Ex.5)
form:{string(the-unit-1,“the”), string(girl-unit-1,“girl”), meets(the-unit-1, girl-unit-1, ?scope)}	

Note that the scope of the meets-predicate is a variable, i.e. undefined yet. The scope will become bound to a concrete phrasal unit once that is known.

Constructions occasionally need to check whether features are present in the input. For example, lexical construction schemas typically look out whether a particular string is present in the input. These features are marked with the special operator # (pronounced hash), which forces the FCG-engine to look in the root-unit. If matching predicates are found, a new unit u is created, added to the transient structure, and u gets the predicates that were matching.

This sounds more complicated than it really is. An example clarifies. Here is the lexical construction for the word “the”, which looks for a string in the input, such as the one in Ex.5.

$$\left[\begin{array}{l} \text{?the-unit} \\ \hline \text{referent: ?obj} \\ \text{lex-cat: article} \\ \text{number: ?number} \end{array} \right] \leftarrow \left[\begin{array}{l} \text{?the-unit} \\ \hline \# \text{ form:}\{\text{string(?the-unit, “the”)}\} \end{array} \right] \quad (\text{Con.2a})$$

When the FCG-engine is trying to find a unit in the transient structure that matches with ?the-unit as defined on the right-hand side of Con.2a, it will not find it. However it will find a matching predication in the root-unit in Ex.5 for the form-feature:

form:{string(?the-unit, “the”)}

?the-unit then gets bound to the-unit-1. A full-fledged unit is created because it did not exist yet and the matching predicates are added as in Ex. 6.

the-unit-1	(Ex.6)
form:{string(the-unit-1,“the”)}	

The information on the left-hand side of Con.2a is then added, yielding Ex.7:

$\xRightarrow{2a}$	the-unit-1	(Ex.7)
	form:{string(the-unit-1,“the”)}	
	referent: ?obj-1	
	lex-cat: article	
	number: ?number-1	

Note that the FCG-engine has renamed the variable names in the construction schema (?number and ?obj) to ?number-1 and ?obj-1 respectively (we stress again that these names are entirely arbitrary). This is necessary to keep variables apart when there are multiple applications of the same construction schema, for example if the same word appears more than once two in a single utterance.

Here is another lexical construction, for the word “girl”:

<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="border-bottom: 1px solid black; padding: 2px;">?girl-unit</td> </tr> <tr> <td style="padding: 2px;">referent: ?y</td> </tr> <tr> <td style="padding: 2px;">sem-cat: {animate, feminine}</td> </tr> <tr> <td style="padding: 2px;">lex-cat: noun</td> </tr> <tr> <td style="padding: 2px;">number: singular</td> </tr> </table>	?girl-unit	referent: ?y	sem-cat: {animate, feminine}	lex-cat: noun	number: singular	←	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="border-bottom: 1px solid black; padding: 2px;">?girl-unit</td> </tr> <tr> <td style="padding: 2px;"># form:{string(?girl-unit,“girl”)}</td> </tr> </table>	?girl-unit	# form:{string(?girl-unit,“girl”)}	(Con.3a)
?girl-unit										
referent: ?y										
sem-cat: {animate, feminine}										
lex-cat: noun										
number: singular										
?girl-unit										
# form:{string(?girl-unit,“girl”)}										

Using the lexical constructions Con.2a and Con.3a as well as the noun-phrase construction Con.1b, the complete chain shown in Ex.4 can be achieved (as discussed in more detail later).

5 Constructional Processing

One of the main features of construction grammar is that the basic unit of grammar, the construction, deals not only with syntactic issues but also with semantics, pragmatics and meaning. After all, a construction is a pairing of form and meaning/function. To incorporate this principle, we first need a way to represent meaning. Then we can address the question how meaning gets introduced in the comprehension process and how production can convert meaning into its grammatical expression.

5.1 Adding Meaning

FCG can be used with any kind of representation of meaning. In the work of our group on grounding language in the sensori-motor states of physical robots, we usually employ procedural semantics [Spranger et al.(2012)]. But

in what follows, I will use a predicate calculus style notation for meaning because that is more familiar to linguists. In fact, I will use a very minimal lightweight form of the predicate calculus, using only predicates with arguments that are either constants or variables, and conjunctive combinations. A meaning then consists of a referent and a set of predications that implicate the referent as one of the arguments. All predicates are typed and we write: `type(predicate,arg)`. This notation is illustrated in the following example:

```
referent: obj-2
predicates: {state(sleepy,obj-2), person(girl,obj-2)}
```

It describes a concrete object, `obj-2`, for which it is true that the predicate `sleepy` with type `state` is valid and the predicate `girl` with type `person`.

The meanings in the lexicon and grammar typically have variables because we do not yet know what the concrete objects are - indeed it is the task of the interpretation process to find bindings between meaning variables and entities in the world model. Here is an example, with `?obj` being a variable:

```
referent: ?obj
predicates: {state(sleepy,?obj), person(girl,?obj)}
```

N-ary predicates are decomposed into subpredicates. For example, the meaning of the word “bakes”, which might also be represented with a single predicate, as in `bake(obj-6, obj-17, obj-15)`, is represented here with three predicates: `bake`, `baker`, and `baked`. The type of `bake` is action:

```
referent: obj-6
predicates: {action(bake,obj-6), baker(obj-6,obj-17), baked(obj-6,obj-16)}
```

Such a decomposition is quite common in computational linguistics and useful because it makes it clear what role the arguments play. It also becomes easier to add other predicates about the same event, such as time or location.

In the above example, the referent of “bakes” is the `bake`-event itself. But it is possible to have other referents, for example, the `baker` (`obj-17`), as in “the baker of the cake”:

```
referent: obj-17
predicates: {action(bake,obj-6), baker(obj-6,obj-17), baked(obj-6,obj-16)}
```

In production, the input consists of meaning, rather than form. But we use the same idea of a root-unit. The meaning that needs to be expressed is added to this unit. For example, to start up the production of the sentence “he bakes a cake”, the root-unit looks as follows:

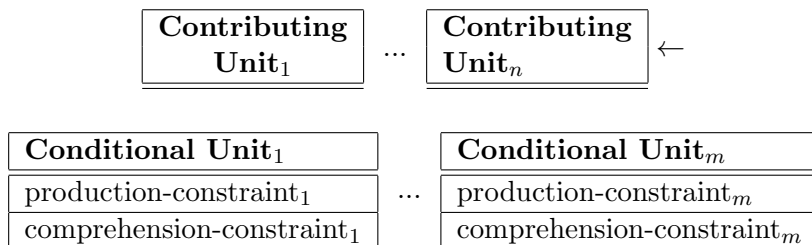
root	
predicates:	(Ex.8)
{person(male,obj-17), action(bake,obj-6), baker(obj-6,obj-17), baked(obj-6,obj-16)}	

5.2 The Production and Comprehension Lock

It is highly desirable that the same construction, without change, can be used for comprehension AND production. But production requires that constructions trigger mainly on the basis of semantic criteria, whereas comprehension requires that they trigger based mainly on syntactic criteria. This leads naturally to the idea that the constraints on each conditional unit should be split into two: a production constraint and a comprehension constraint. The production-lock then consists of all the production constraints and the comprehension-lock of all the comprehension constraints.

- The *production-lock* is the gate-keeper in production. When the transient structure fits with the production-lock, all information from the comprehension lock is added as well as the information in the contributor (the left hand side of the construction). So the comprehension-lock is not the gate-keeper in deciding whether the construction should be used in production, but its information will become part of the new transient structure.
- The *comprehension-lock* is the gate-keeper in comprehension. When the transient structure fits with the comprehension-lock, information from the production-lock as well as from the contributor are merged with the transient structure.

The production-constraint for each conditional unit is written above the comprehension-constraint. When there is no constraint for a unit (either for production or comprehension), the symbol for an empty set, i.e. \emptyset , is used. So we get the following schematic:



Here is an example of the lexical construction for “the”, which now has not only comprehension but also production constraints for ?the-unit on the right hand side:

$$\left[\begin{array}{l} \text{?the-unit} \\ \hline \text{referent: ?obj} \\ \text{lex-cat: article} \\ \text{number: ?number} \end{array} \right] \leftarrow \left[\begin{array}{l} \text{?the-unit} \\ \hline \# \text{ meaning: } \{\text{definite(?obj)}\} \\ \# \text{ form: } \{\text{string(?the-unit, "the")}\} \end{array} \right] \quad (\text{Con.2b})$$

Here is the lexical construction, for the word “girl”:

$$\left[\begin{array}{l} \text{?girl-unit} \\ \hline \text{referent: ?obj} \\ \text{sem-cat: } \{\text{animate, feminine}\} \\ \text{lex-cat: noun} \\ \text{number: singular} \end{array} \right] \leftarrow \left[\begin{array}{l} \text{?girl-unit} \\ \hline \# \text{ predicates: } \{\text{person(girl,?obj)}\} \\ \# \text{ form: } \{\text{string(?girl-unit, "girl")}\} \end{array} \right] \quad (\text{Con.3b})$$

And here is a (simplified) example of a noun-phrase construction. The conditional meaning is rather limited here. Basically it establishes that the referent of article and the noun are the same by using the same variable-name for their referent (namely ?obj), and this is also the referent of the ?NP-unit.

$$\left[\begin{array}{l} \text{?NP-unit} \\ \hline \text{referent: ?obj} \\ \text{phrasal-cat: NP} \\ \text{constituents:} \\ \quad \{\text{?art-unit, ?noun-unit}\} \\ \text{agreement: ?number} \end{array} \right] \left[\begin{array}{l} \text{?art-Unit} \\ \hline \text{syn-fun:} \\ \quad \text{determiner: ?noun-unit} \end{array} \right] \left[\begin{array}{l} \text{?noun-unit} \\ \hline \text{syn-fun:} \\ \quad \text{head: ?NP-unit} \end{array} \right] \leftarrow$$

$$\left[\begin{array}{l} \text{?art-unit} \\ \hline \text{referent: ?obj} \\ \text{lex-cat: article} \\ \text{number: ?number} \end{array} \right] \leq \left[\begin{array}{l} \text{?noun-unit} \\ \hline \text{referent: ?obj} \\ \text{lex-cat: noun} \\ \text{number: ?number} \end{array} \right] \quad (\text{Con.1c})$$

5.3 An example of comprehension

Suppose that we now try to parse “the girl”, starting from the input in Ex.5. The application of the lexical constructions 2b and 3b yields the following:

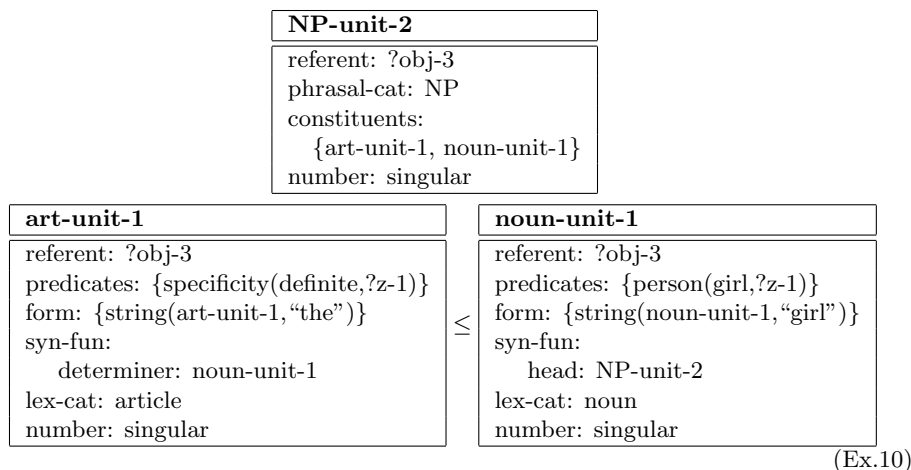
$$\xrightarrow{2b,3b} \left[\begin{array}{l} \text{the-unit-1} \\ \hline \text{referent: ?obj-1} \\ \text{predicates: } \{\text{specificity(definite,?obj-1)}\} \\ \text{form: } \{\text{string(the-unit-1, "the")}\} \\ \text{number: ?number-1} \\ \text{lex-cat: article} \end{array} \right] \leq \left[\begin{array}{l} \text{girl-unit-1} \\ \hline \text{referent: ?obj-2} \\ \text{predicates: } \{\text{person(girl,?obj-2)}\} \\ \text{form: } \{\text{string(girl-unit-1, "girl")}\} \\ \text{number: singular} \\ \text{lex-cat: noun} \end{array} \right] \quad (\text{Ex.9})$$

Two new units have been built and both the production lock and the contributor (left-hand side) of the construction have supplied information

for them. Now the noun-phrase construction (Con.1c) applies because its comprehension-lock matches, assuming the following binding-set:

{(?art-unit . the-unit-1)(?noun-unit . girl-unit-1)
 (?obj . ?obj-1) (?obj . ?obj-2)}

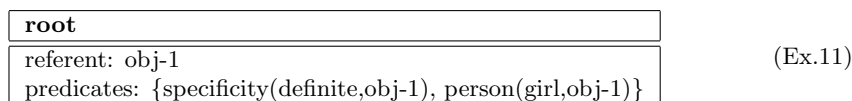
After merging we get the following transient structure:



Note that the referent of the noun-phrase has been made equal to the arg-values of the constituents: ?obj was bound to ?obj-1 and to ?obj-2 and hence all occurrence of ?obj, as well as ?obj-1 and ?obj-2, have been replaced by a single new variable ?obj-3. Note also how the number-value singular coming from “girl” has propagated to the article and NP-units. The number-value could have come also from the article (as would be the case in “a sheep”) or from the NP, due to agreement with the verb (as in “the sheep sleeps”). Variable-bindings flow in all directions and that makes them so powerful.

5.4 An example of Production

Exactly the same constructions can now be used to go from meaning to form, except that the production-locks are now the gate-keepers. The starting point is the following root-unit:



The first step is the application of the lexical constructions. A lexical construction applies in production if its production-lock matches with predicates in the root-unit. If this is the case, then a new unit is created for the corresponding word. Information from the comprehension-lock on the right-hand side of the construction is merged into this unit and all other information from the contributor (on the left-hand side) is added as well. So we get the following two new units, with so far no ordering relations between them:

the-unit-2	girl-unit-2
referent: obj-1 predicates: {specificity(definite,obj-1)} form: {string(the-unit-2, "the")} lex-cat: article number: ?number-2	referent: obj-1 predicates: {person(girl,obj-1)} form: {string(girl-unit-2, "girl")} lex-cat: noun number: singular

(Ex.12)

Now the noun-phrase construction Con.1c can apply. The binding set is
 $\{(?art-unit . the-unit-2) (?noun-unit . girl-unit-2)$
 $(?number . singular) (?number . ?number-2) (?obj . obj-1)\}$
and it produces a similar structure as we saw in Ex.10:

NP-unit-3
referent: obj-1 phrasal-cat: NP constituents: {the-unit-2, girl-unit-2} number: singular

the-unit-2	girl-unit-2
referent: obj-1 form: {string(the-unit-2, "the")} syn-fun: determiner: girl-unit-2 lex-cat: article number: singular	referent: obj-1 form: {string(girl-unit-2, "girl")} syn-fun: head: NP-unit-3 lex-cat: noun number: singular

(Ex.13)

Note how ?number-2 could be replaced by singular in the-unit-2 and this value percolated up to NP-unit-3. The referent of NP-unit-3 has become obj-1.

This particular example is also shown through the web demonstration as Example I. (see [link](#)) Other examples shown there involve intransitive (Example II) and transitive (example III) clauses.

6 Argument structure constructions

We now look at another, still highly simplified, example to further illustrate the basic mechanisms of FCG. The example concerns argument structure constructions, a topic that is closer to the interests of many construction grammarians than phrase structure. The example focuses on the double-object construction and is based on a detailed example worked out by Remi van Trijp [van Trijp(2010)].

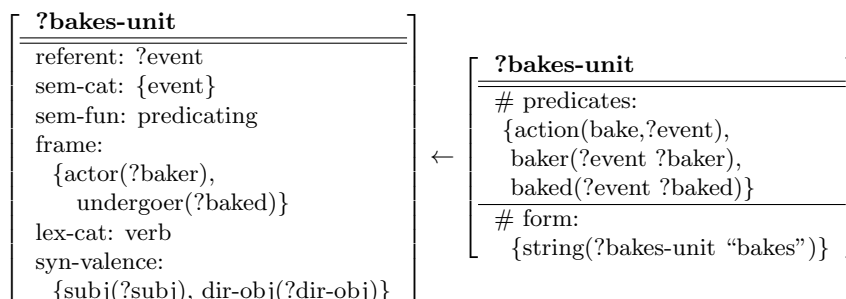
6.1 The double object construction

The double-object construction has two noun-phrases after the verb, as in: “He gave her the book.” It imposes the meaning of ‘causing to receive’, even

if the verb itself does not express this already as part of its lexical meaning, as in “He baked her a cake.” [Goldberg(1995)].

I will here ignore many subtleties of this construction, for example that the indirect object is usually animate or that the first noun phrase is necessarily shorter (a phonological constraint) or more topical (a pragmatic constraint) than the second. I will also ignore tense, aspect and agreement phenomena (although they are all worked out in van Trijp’s implementation) and morphology. Instead I focus only on the verb and how the construction can impose meaning beyond the meaning provided by lexical items.

Lexical constructions for “he”, “her”, and “(a) cake” are similar to what I discussed earlier. Here is the lexical construction for “bakes”:



(Con.4)

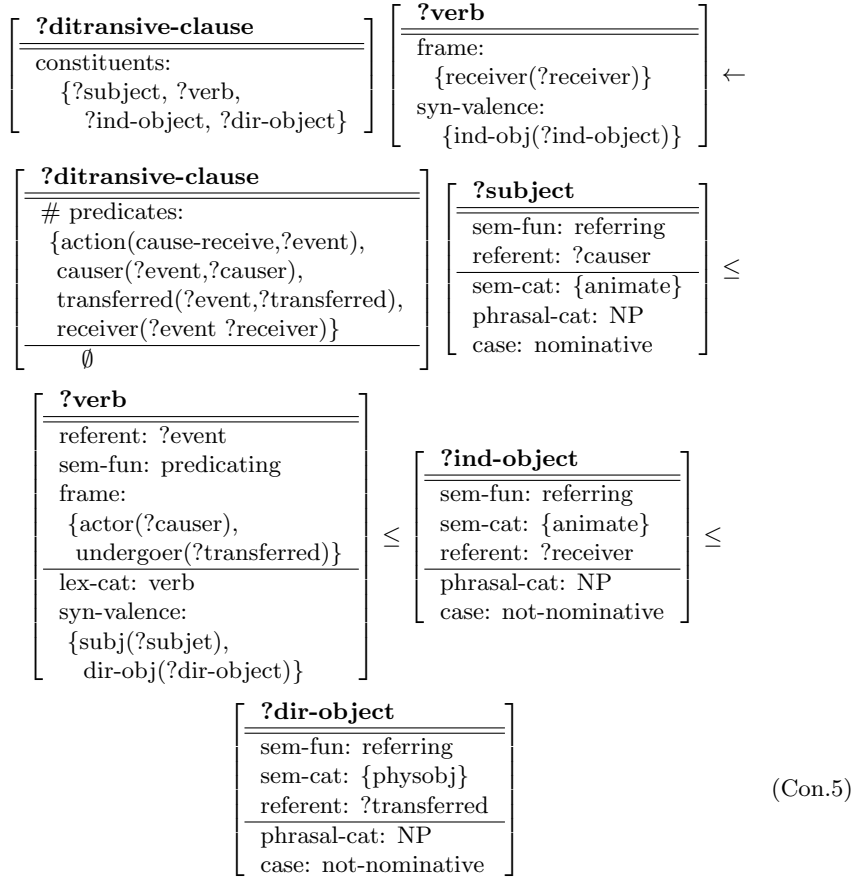
As expected, the lock, on the right-hand side, contains a verb-unit whose production-constraint requires the predicates describing a bake event:

action(bake,?event), baker(?event,?baker), and baked(?event,?baked)

and whose comprehension-constraint requires the string “bakes” (in fact it should be the stem “bake-” but I gloss over morphological details here to simplify the example).

The contributor, on the left-hand side, imposes a semantic valence structure (a case frame) for the same verb-unit with an actor and an undergoer. The actor is ?baker and the undergoer is ?baked. These variables are the same as those in the lock, so that if they get bound, they are also bound in the case frame as well. The contributor also introduces a syntactic valence for the verb-unit with roles for a subject and an object. The subject or direct-object cannot be known based on this construction alone, and so variables, ?subj and ?obj, are introduced whose values will have to be bound by other constructions.

Next, Con.5 is the definition of the double object construction itself. It has to handle phrases like “she gave him a book”, where the receiver is already included in the syntactic and semantic valence of the verb but also “she baked him a cake” where the receiver has to be added by the construction. The presence of a receiver in the semantic valence frame of the verb is therefore not obligatory, but it is added by the double object construction.



When the comprehension or the production constraint of a conditional unit is empty, we write the empty set symbol \emptyset . This is the case here for the ?ditransitive-clause unit, which has only a production-constraint and no comprehension-constraint. This is typical for phrasal constructions, or other types of grammatical constructions, that add new meaning to a syntactic usage pattern, because the properties that the construction is looking for are syntactically expressed by properties of the conditional units and their ordering relations.

6.2 A production example

The double-object construction triggers in production when the predicate cause-receive and its arguments have to be expressed and when there are other units with the appropriate semantic characteristics as specified in the production-locks of the respective units on the right-hand side of the double object construction. Here is a concrete example, starting from the following meaning: There is a cake (obj-16), there is a bake-event (obj-6), with a baker (obj-17) and a baked object (obj-16). There is a cause-receive event (obj-6), with an entity causing this event (obj-17) who is a male person,

an object that is being transferred (namely the case, obj-16), and an entity receiving this object (obj-18) who is a female person. The root-unit is:

root	
referent: obj-6 predicates: {physobj(cake,obj-16),action(bake,obj-6), baker(obj-6,obj-17), baked(obj-6,obj-16), action(cause-receive,obj-6), causer(obj-6,obj-17) transferred(obj-6,obj-16), receiver(obj-6,obj-18), person(male,obj-17), person(female,obj-18)}	(Ex.14)

After the application of lexical constructions the transient structure contains the following units. There are no ordering constraints between them yet.

he-unit-1	bakes-unit-1	
referent: obj-17 predicates: {person(male,obj-17)} sem-fun: referring sem-cat: {physobj, animate} phrasal-cat: NP string: "he" case: nominative	referent: obj-6 predicates: {action(bake,obj-6), baker(obj-6,obj-17), baked(obj-6,obj-16)} sem-cat: {event} sem-fun: predicating frame: {actor(obj-17), undergoer(obj-16)} string: "bakes" lex-cat: verb syn-valence: {subj(?subj-1), dir-obj(?dir-obj-1)}	
her-unit-1	cake-unit-1	(Ex.15)
referent: obj-18 predicates: {person(female,obj-18)} sem-cat: {physobj, animate} sem-fun: referring phrasal-cat: NP string: "her" case: not-nominative	referent: obj-16 predicates: {person(cake,obj-16)} sem-fun: referring sem-cat: {physobj} phrasal-cat: NP string: "(a) cake" case: ?case-1	

Now the double-object construction Con.5 can apply. The production-lock of the ?ditransive-clause unit in this construction requires that the meaning includes:

cause-receive(?event), causer(?event,?causer), transferred(?event,?transferred), receiver(?event,?receiver)

This is the case here, assuming the following binding-set:

{(?event . obj-6)(?causer . obj-17)(?transferred . obj-16)
(?receiver . obj-18) (?case-1 . not-nominative)}

The right hand side of this construction contains a unit for a ditransitive-clause which is now created:

ditransitive-clause-1	
predicates: {action(cause-recv, obj-6), causer(obj-6, obj-17), transferred(obj-6, obj-16), receiver(obj-6, obj-18)}	(Ex.15)

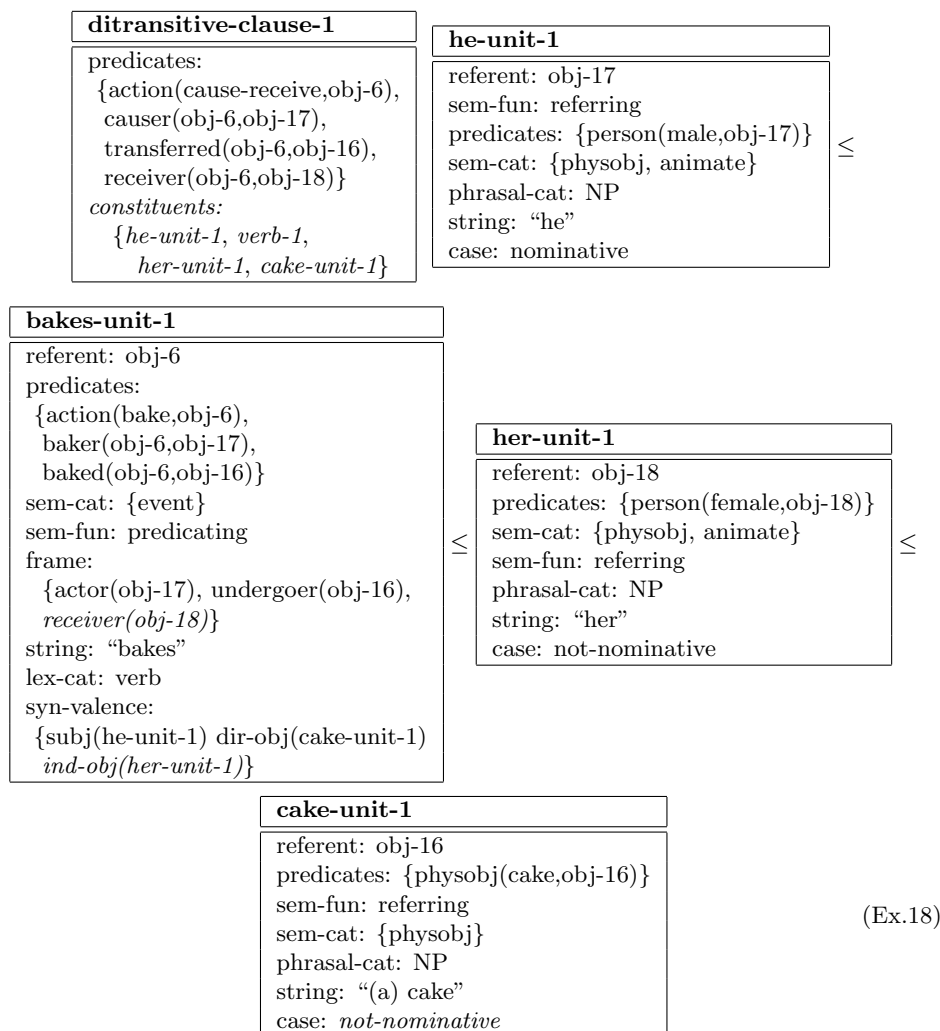
The other units on the right hand side of Con.5 match and so no new units have to be created but their constraints can be merged into the existing units in the transient structure. Variables are substituted for their bindings to construct the new transient structure after construction application. So we see new information in the verb-unit (“bakes”) because the variables ?subj-1 and ?dir-obj-1 have been replaced by their bindings: *he-unit-1* and *cake-unit-1* respectively, and for the noun phrase “(a) cake” where ?case-1 is replaced by its binding, namely not-nominative.

bakes-unit-1		cake-unit-1
referent: obj-6 predicates: {action(bake, obj-6), baker(obj-6, obj-17), baked(obj-6, obj-16)} sem-cat: {event} sem-fun: predicating frame: {actor(obj-17), undergoer(obj-16)} string: “bakes” lex-cat: verb syn-valence: {subj(he-unit-1), dir-obj(cake-unit-1)}	referent: obj-16 predicates: {physobj(cake, obj-16)} sem-fun: referring sem-cat: {physobj} phrasal-cat: NP string: “(a) cake” case: not-nominative	(Ex.16)

Two things should be noted. (i) As mentioned earlier, units can receive extra information when the constraints in the conditional units are merged with the transient structure, but this merging process will block the application of the construction when incompatibilities are detected. For example, in “he bakes she (a) cake” “she” is nominative which clashes with the constraint on ?ind-obj that the case-value is not-nominative. (ii) Ex.15 and Ex.16 show that variables (here ?case-1 which is the case-value of case-unit-1) can appear in transient structures as well and they can get bound when a construction provides a concrete value for them (in this case not-nominative). This illustrates again the power of using logic variables. In effect, they cast a net of constraints over the transient structure and when a variable gets bound at some point, its value propagates in this net to all other occurrences.

Next we turn to the contributor. The constituents-feature of ditransitive-clause-1 are filled in with the different NPs and the verb and obj-17 has been

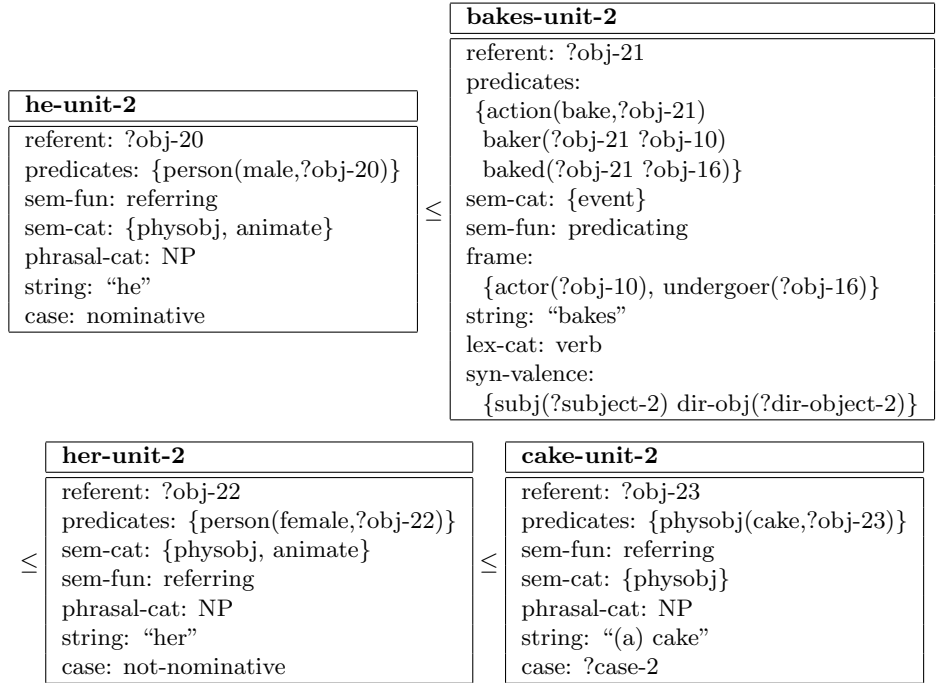
added as receiver in the frame of the verb and the ind-obj in the syn-valence. These additions are highlighted in italics in the final transient structure:



From this transient structure, information about the form of the utterance gets extracted, yielding the following ordered set of strings: "he baked her (a) cake".

6.3 A comprehension example

Here is a concrete example of comprehension, starting from the utterance "he baked her (a) cake". After the application of the lexical constructions, we get the following initial transient structure:



(Ex.19)

The referents in all the NP units (such as ?obj-23 in *cake-unit-2*) are all variables, and the roles in the bake-predicate (the baker to be bound to ?obj-10, or the baked to be bound to ?obj-16) or its semantic valence (the actor ?obj-10 or the undergoer ?obj-16) are not related yet to these referents. Moreover there are several other variables not yet determined, such as the subject ?subject-2 or direct-object ?dir-object-2 in the syntactic valence frame of the verb, or the case-value ?case-2 of "(a) cake". Notice also that there is no mention of an indirect object in the valence frame of "bake", nor of a receiver in the predicates defining the predicates of the verb.

Now the comprehension-constraints of the conditional units on the right-hand side of the double-object construction are matched against this transient structure. This is successful with the following binding-list:

(?subject-2 . he-unit-2) (?verb . bakes-unit-2) (?NP2 . her-unit-2)
(?dir-object-2 . cake-unit-3) (?case-2 . not-nominative)
(?event . ?obj-21) (?causer . ?obj-20) (?causer . ?obj-10)
(?receiver . ?obj-22) (?transferred . ?obj-16) (?transferred . ?obj-23)

Variables in the predicates and in the frames are linked to the referents of the NP-units by the chain of bindings to the same variables: ?causer = ?obj-20 = ?obj-10, ?transferred = ?obj-23 = ?obj-16.

There are no comprehension-constraints for the the ditransitive-clause unit. This means that this unit can simply be created as part of the matching phase of the construction, and using the bindings already known from

matching. The unit is called ditransitive-clause-2. All variables bound to each other are replaced by a single variable, as in Ex.20:

ditransitive-clause-2	
predicates: {action(cause-receive,?obj-21), causer(?obj-21,?obj-20), transferred(?obj-21,?obj-23), receiver(?obj-21,?obj-22)}	(Ex.20)

Then merging takes place. First the production constraints are added to each conditional unit. The units which are subject ?subject-2 and direct-object ?dir-object-2, can be substituted by their bindings, namely *he-unit-2* and *cake-unit-3*. Second, information from the contributor is added, which includes additions to the syntactic and semantic valence of the verb and the phrase structure. We obtain the following final transient structure with the final changes highlighted:

ditransitive-clause-2		he-unit-2	
predicates: {action(cause-receive,?obj-21), causer(?obj-21,?obj-20), transferred(?obj-21,?obj-23), receiver(?obj-21,?obj-22)} constituents: { <i>he-unit-2</i> , <i>verb-2</i> , <i>her-unit-2</i> , <i>cake-unit-2</i> }	≤	referent: ?obj-20 sem-fun: referring predicates: {person(male,?obj-20)} sem-cat: {physobj, animate} phrasal-cat: NP string: "he" case: nominative	≤
bakes-unit-2 referent: ?obj-21 predicates: {action(bake,?obj-21), baker(?obj-21,?obj-20), baker(?obj-21,?obj-22)} sem-cat: {event} sem-fun: predicating frame: {actor(?obj-20), undergoer(?obj-23), receiver(?obj-22)} form: {string(bakes-unit-2,"bakes")} lex-cat: verb syn-valence: {subj(<i>he-unit-2</i>), dir-obj(<i>cake-unit-2</i>), ind-obj(<i>her-unit-2</i>)}	≤	her-unit-2 referent: ?obj-22 predicates: {person(female,?obj-22)} sem-cat: {physobj, animate} sem-fun: referring phrasal-cat: NP string: "her" case: not-nominative	≤
		cake-unit-2 referent: ?obj-23 predicates: {action(cause,?obj-23)} sem-fun: referring phrasal-cat: NP string: "(a) cake" case: not-nominative	(Ex.21)

The constituent structure tree that is present in this feature structure is shown in Figure 1

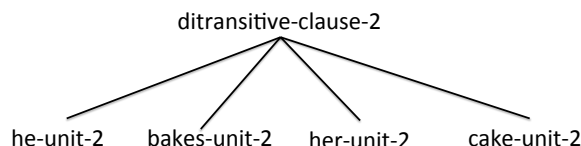


Figure 1: Constituent structure tree represented explicitly in the value of the constituents-feature of ditransitive-clause-2 in Ex.21.

This example is still extremely simple compared to the constructions we need for handling ‘realistic’ human language. Usually there are a lot more features that play a role in determining whether a construction should apply or what new features and units a construction adds to the transient structure. But the mechanisms sketched out here scale up to as many units and features as required.

The ditransitive examples of production and comprehension are shown in the web demonstration (see [link](#)) as Example IV.

7 Other grammatical perspectives

Fluid Construction Grammar integrates different views on the utterance, and we have already seen some of them:

- Phrase structure is realized using the feature *constituents* which stores the hierarchical relations between the different units, and the features *lex-cat* and *phrasal-cat* which contain the parts of speech or phrasal categories of each unit.
- Functional structure is realized both in terms of semantic functions and syntactic functions. Units have a *sem-fun* feature with values like predicating, referring, etc. These indicate how the predications contained in the meaning of the unit should be used to achieve communicative goals. The *syn-fun* feature has values like subject, determiner, modifier, etc. They are the syntactic reflections of the semantic functions.
- Argument structure is realized by the frame attached with words, such as ‘bakes’, that defines the possible semantic roles, by the semantic classes (*sem-cat*) that are associated with units, such as *physobj* or *animate*, and by defining the mapping between lexical meaning and argument structure in the construction.

To extend the ditransitive example so that it can deal with dependencies is straightforward. It is just necessary to add a feature to store the dependents in the transient structure and to keep track of both the head and the dependents. This approach is shown in full detail in the web demonstration associated with this paper, as part of Example IV.

8 Conclusions

This paper introduced some of the basic ideas of Fluid Construction Grammar, illustrated with simplified examples. FCG was explained starting from rewrite grammars because this is a formalism that many linguists are familiar with. There are many topics related to FCG that have not been discussed such as: How language processing can be fluid, by accepting partial matches or coercing words or structures in roles they did not have yet [Steels and van Trijp(2011)]. How to deal with a very large number of constructions both from the viewpoint of retrieving rapidly the most relevant construction and of damping combinatorial search through the space of possible transient structures? How can grammatical constructions be learned? It is possible to use FCG with inductive machine learning algorithms operating over large corpora, but most of the work so far has focused on using FCG to learn grammars in situated embodied interactions [Spranger et. al.(2016)] using insight learning [Garcia-Casademont and Steels(2016)]. Another topic is how constructions can change in a population of language users. Much work has already been done in this area as well, some of it inspired by phenomena observed in actual language change [Steels(2012)], [Beuls and Steels(2013)].

9 Acknowledgement

The design and implementation of a computational platform for comprehension and production of language is obviously a major undertaking that takes decades, both for design and for implementation, typically going through several iterations. Creating a community of users equally takes a long time. FCG is still in a development phase and many people have helped to get it to this point. Most of the activity has been taking place at the AI Laboratory of the University of Brussels (VUB), the Sony Computer Science Laboratory in Paris, and more recently the Institute for Evolutionary Biology (UPF-CSIC) in Barcelona.

The first implementations of components leading to FCG were started in the late nineties by Luc Steels, Angus McIntyre and Joris Van Looveren, later joined by Nicolas Neubauer. Around 2002, a new ambitious implementation effort started with Joachim De Beule as main developer of a new unification engine and Andreas Witzel as developer of the first web-based FCG interface. The FCG-system became more stable, the first FCG work-

shop was held in Paris in September 2004, and the first papers started to appear [Steels(2004)].

Thanks to the EU-FP6 ECAgents project that started in January 2004, a new team of Ph.d. students could be assembled, which included Joris Bleys, Martin Loetzsch, Remi van Trijp and Pieter Wellens. This expansion led to many important innovations in terms of the representational power of the formalism, the processing engine and the general experimental environment. Thanks to the EU-FP7 ALEAR project, which started in 2008, a variety of case studies could be launched by an additional team of researchers, including Katrien Beuls, Nancy Chang, Sebastian Hofer, Kateryna Gerasymova, and Vanessa Micelli. Significant progress was also made in parallel on embodied semantics, particularly by Wouter Van den Broeck, Martin Loetzsch, Simon Pauw, and Michael Spranger. This had in turn an important impact on grammar design, with major contributions, such as a new FCG-interface by Martin Loetzsch.

More recently, Remi van Trijp and Katrien Beuls have been making important new contributions to the FCG-core by tackling well-known issues in grammar (such as long-distance dependencies or more flexible constituent ordering), which lead to further improvements in FCG itself. Michael Spranger has constructed a new object-oriented implementation, and additional case studies have been developed by a new generation of Ph.d students including Emilia Garcia Casademont, Miquel Cornudella, Yana King, Tânia Marques, and Paul Van Eecke. Paul van Eecke made major contributions to the new notation used in this paper and its mapping to FCG-2011 used in [Steels(2011)]. The writing of this paper was possible thanks to an ICREA fellowship to the author, and partially financed by the EU FP7 project Insight.

References

- [Barres and Lee(2014)] V. Barres and J. Lee. Template construction grammar: from visual scene description to language comprehension and agrammatism. *Neuroinformatics*, 12(1):181–208, 2014.
- [Bergen and Chang(2003)] B.K. Bergen and N.C. Chang. Embodied construction grammar in simulation-based language understanding. In J.O. Ostman and M. Fried, editors, *Construction Grammar(s): Cognitive and Cross-Language Dimensions*. John Benjamin Publ Cy., Amsterdam, 2003.
- [Beuls(2011)] K. Beuls. Construction sets and unmarked forms: A case study for Hungarian verbal agreement. In Luc Steels, editor, *Design Patterns in Fluid Construction Grammar*, pages 237–264. John Benjamins, Amsterdam, 2011.

- [Beuls, van Trijp and Wellens(2012)] K. Beuls, R. van Trijp, and P. Wellens. Diagnostics and Repairs in Fluid Construction Grammar. In Luc Steels and Manfred Hild, editors, *Language Grounding in Robots*, pages 215–234. New York: Springer, 2012.
- [Beuls and Steels(2013)] K. Beuls and L. Steels. Agent-Based Models of Strategies for the Emergence and Evolution of Grammatical Agreement *Plos One*, 8(3):e58960.
- [Boas and Sag(2012)] H. Boas and I. Sag, editors. *Sign-based Construction Grammar*. CSLI Publications, Palo Alto Ca, 2012.
- [Bresnan(2001)] J. Bresnan. *Lexical Functional Syntax*. Blackwell Pub., Oxford, 2001.
- [Chang and Micelli(2011)] J. De Beule Chang, N.C. and V. Micelli. Computational construction grammar: Comparing ecg and fcg. In L. Steels, editor, *Design Patterns in Fluid Construction Grammar*, pages 259–288. John Benjamin Publ Cy., Amsterdam, 2011.
- [Fillmore(1988)] C. J. Fillmore. The mechanisms of “Construction Grammar”. In *Proceedings of the Fourteenth Annual Meeting of the Berkeley Linguistics Society*, pages 35–55, Berkeley CA, 1988. Berkeley Linguistics Society.
- [Goldberg(2015)] A. Goldberg. Fitting a slim dime between the verb template and argument structure construction approaches. *Theoretical Linguistics*, to appear, 2015.
- [Goldberg(1995)] Adele E. Goldberg. *A Construction Grammar Approach to Argument Structure*. Chicago UP, Chicago, 1995.
- [Jackendoff(1997)] R. Jackendoff, editor. *The architecture of the language faculty*. The MIT Press, Cambridge Ma, 1997.
- [Kay(1984)] M. Kay. Functional unification grammar: A formalism for machine translation. In *Proceedings of the International Conference of Computational Linguistics*, 1984.
- [Knight(1989)] K. Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, 1989.
- [Levelt(1989)] W. Levelt, editor. *Speaking: From Intention to Articulation*. The MIT Press, Cambridge Ma, 1989.
- [Martelli and Montanari(1982)] A. Martelli and U. Montanari. An efficient unification algorithm. *Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

- [Pereira and Warren(1980)] F. Pereira and D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.
- [Sag et al.(2003)] I. Sag, T. Wasow, and E. Bender. *Syntactic theory: a formal introduction*. University of Chicago Press, Chicago, 2003.
- [Shieber(1986)] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*, volume 4 of *CSLI Lecture Notes Series*. Center for the Study of Language and Information, Stanford, CA, 1986.
- [Spranger et al.(2012)] M. Spranger, S. Pauw, M. Loetzsch, and L. Steels. Open-ended procedural semantics. In *Language grounding in robots*, pages 153–172. Springer Verlag, New York, 2012.
- [Spranger et. al.(2016)] Michael Spranger, Simon Pauw, and Martin Loetzsch. Open-ended semantics co-evolving with spatial language. In A.D.M. Smith, M. Schouwstra, B. de Boer, and K. Smith, editors, *The Evolution of Language (EVOLANG 8)*, pages 297–304, Singapore, 2010. World Scientific.
- [Steels(2004)] L. Steels. Constructivist development of grounded construction grammars. In D. Scott, W. Daelemans, and M. Walker, editors, *Proceedings of the Annual Meeting of the Association for Computational Linguistic Conference.*, pages 9–19. Barcelona, 2004.
- [Steels(2005)] L. Steels. The emergence and evolution of linguistic structure: from lexical to grammatical communication systems. *Connection Science*, 17(3-4):213–230, December 2005.
- [Steels(2011)] L. Steels, editor. *Design Patterns in Fluid Construction Grammar*. John Benjamins, Amsterdam, 2011.
- [Steels(2012)] L. Steels. *Experiments in cultural language evolution*. John Benjamins Pub., Amsterdam, 2012.
- [Steels(2012b)] Luc Steels, editor. *Computational Issues in Fluid Construction Grammar*, volume 7249 of *Lecture Notes in Computer Science*. Springer, 2012
- [Steels(2015)] L. Steels. *The Talking Heads Experiment. Origins of Words and Meanings.*, volume 1 of *Computational Models of Language Evolution*. Language Science Press, Berlin, 2015.
- [Steels and De Beule(2006)] L. Steels and J. De Beule. Unify and merge in fluid construction grammar. In P. Vogt, Y. Sugita, E. Tuci, and C. Nehaniv, editors, *Symbol Grounding and Beyond: Proceedings of the Third*

International Workshop on the Emergence and Evolution of Linguistic Communication, LNAI 4211, pages 197–223. Springer-Verlag, Berlin, 2006.

- [Steels and Szathmáry(2016)] L. Steels and E. Szathmáry Fluid Construction Grammar as a Biological System. *Linguistics Vanguard*, 5(2):
- [Steels and Van Eecke(2016)] L. Steels and P. Van Eecke *Insight language learning with Pro- and Anti-Unification*. *Frontiers in Psychology*, submitted.
- [van Trijp(2013)] R. van Trijp. A comparison between Fluid Construction Grammar and sign-based construction grammar. *Constructions and Frames*, 5(1):88–116, 2013.
- [Garcia-Casademont and Steels(2016)] Garcia-Casademont, E. and L. Steels. Grammar learning as insight problem solving. *The Journal of Cognitive Science*, 5(17):27–62,
- [van Trijp(2010)] Remi van Trijp. Argument realization in Fluid Construction Grammar. In Hans C. Boas, editor, *Computational Approaches to Construction Grammar and Frame Semantics*. John Benjamins, Amsterdam, 2010.
- [van Trijp(2011)] Remi van Trijp. Feature matrices and agreement: A case study for German case. In Steels, L., editor, *Design Patterns in Fluid Construction Grammar*, pages 205–235. John Benjamins, Amsterdam, 2011.
- [Steels and van Trijp(2011)] Remi van Trijp. How to make construction grammars fluid and robust. In Steels, L., editor, *Design Patterns in Fluid Construction Grammar*, pages 301–330. John Benjamins, Amsterdam, 2011.
- [Wellens(2011)] Pieter Wellens. Organizing constructions in networks. In *Design Patterns in Fluid Construction Grammar*, pages 181–201. John Benjamins, Amsterdam, 2011.

10 Appendix I: Glossary (alphabetic)

Anti-unification: Whereas unification tries to see whether and how one structure (e.g. a transient structure) can be made similar to another one (the lock or contributor of a construction schema), anti-unification does the opposite. It finds what is common and hence what is different between two structures. This is a fundamental operation in FCG to make construction

application more fluid and for learning grammar but this is not further elaborated in this paper.

(Atomic) symbol: The primary building block of feature structures. They are either constants or variables. The meaning of a symbol does not come from its name but from its role in all structures where it appears.

Comprehension-constraint: Features and values of a unit that have to match in comprehension.

Construction schema: A construction schema (also called a construction) is an abstract schema that can be used to expand any aspect of a transient structure from any perspective and it can consult any aspect of the transient structure to decide how to do so. A construction associates a contributor (the left-hand side) with a lock or conditional part (the right-hand side).

Construction-score: Is a numerical value (between 0.0 and 1.0) that is associated with a construction. It reflects the success of the construction in past interactions and is used by the FCG-engine in case there is a competition between different constructions all matching with the same transient structure at the same step in the search space.

Contributor: Also called contributing part or left-hand side of a construction schema. It contains the set of units that are to be added to the transient structure, if they are not yet there, and their associated features and values.

Feature: A feature has a name and a value. Features represent properties of units and relations between units.

Feature-set: A set of features with their respective feature-names and values.

Feature structure: Datastructure for representing information about an utterance. It consists of a set of units with names and feature-sets.

Lock: Also called conditional part or right-hand side (of a construction schema). It contains the set of units that have to be present in the transient structure in order for the construction schema to be applicable, together with their production and comprehension constraints. The production lock is the set of conditional units with their production constraints and the comprehension lock the set of conditional units with their comprehension constraints.

Pathway: Also called linguistic pathway. A chain of transient structures obtained by consecutive application of constructions.

Production-constraint: Features and values of a unit that have to match in production.

Transient structure: The feature structure built up during the comprehension or production of a concrete utterance.

U-Complete: This operation takes a transient structure and the contributor of a construction and adds all information missing from the contributor to the transient structure. It uses a version of the unification algorithm to do this.

U-Match: The basic function for testing whether a transient structure fits with a construction lock. This operation is based on a variant of the unification operation, widely used in logically inference systems. It takes a transient structure, a lock, and an initial set of bindings as argument and then decides whether the transient structure fits in the lock, returning an extended set of bindings in case of success.

U-merge: The basic operation for applying a construction to a transient structure. It consists of two parts: a U-Match phase in which the lock of a construction is compared with the transient structure, and a U-Complete phase in which information from the contributor of the construction is added to the transient structure.

Unit: A unit has a name and a feature-set. It contains all information about a particular linguistic unit, e.g. a word, a morpheme, a phrase.

Value: The value of a feature can take the form of an atomic symbol, a formula (i.e. a set of predicates), a feature-set, or a set of values.

Variable: An atomic symbol which can get bound to another FCG-element (constants, sets, feature-sets, formulas, expressions). Variables are written as symbol of which the first character is ?, as in ?subject, which is a variable that gets bound to a unit.

11 Appendix II: Definition of U-Match and U-Merge

This appendix defines the basic functions of FCG used in this paper with more precision. For a fully formal definition, the reader should consult [Steels and De Beule(2006)]. Note that the following definition is not an algorithm but a formal specification which algorithms have to satisfy. It does not give an effective procedure for how the binding-sets are computed, only which binding-sets are needed. Of course, an effective algorithm has been used in the actual FCG implementation, which is similar to the algorithms used in the unification component of logic programming languages (such as [Martelli and Montanari(1982)]) but its exposition goes beyond the scope of this paper.

FCG conceptualizes a construction schema in terms of a contributing part (written on the left-hand side of the arrow) and a lock or conditional part (written on the right-hand side). The feature-sets of the units on the right-hand side is split into two: a production-constraint and a comprehension-constraint. So a construction c is a triple:

$$c = \langle cont_c, prod_c, comp_c \rangle$$

where $cont_c$ is the set of units forming the contributing part, $prod_c$ is the set of conditional units with their production constraints and $comp_c$ is the set of conditional units with their comprehension constraints. All of these are feature structures, i.e. lists of units with names and feature-sets.

The core of construction application relies on two functions: U-Match and U-Complete, which are combined into U-Merge. In the following definitions, l stands for the lock and k stands for the key (the transient structure). Both of them are feature structures. m stands for the result which is true (T) or false (F) and b is used for the binding-set.

- U-Match checks whether a production or comprehension lock l matches with a transient structure k given a particular binding-set b . It returns a truth-value $m \in \{T, F\}$:

$$\text{U-Match}(l, k, b) \rightarrow m.$$
- U-Merge combines a transient structure k_{in} with p , the information in a production or comprehension lock or the contributing part of a construction. It returns both a truthvalue $m \in \{T, F\}$ if U-Merge is possible given a binding-set b , and an expanded transient structure k_{out} :

$$\text{U-Merge}(p, k_{in}, b) \rightarrow m, k_{out}$$

FCG uses logic variables and therefore the FCG-engine builds and utilizes a binding-set or substitution $b \in V \times E$ where V is a set of variables and E the allowed elements of FCG: constants, variables, sets, formulas, and feature sets. Binding-sets are written as a set of dotted pairs. Because variables may be bound to variables, a binding-set can contain a chain, as in the following example:

$$b = \{(?x . ?y) (?y . ?z) (?x . ?u) (?z . \text{singular}) (?a . ?b)\}$$

The *grounded binding* of a variable v is the non-variable binding of v in the chain, so the variables involved in the chain $?x = ?y = ?z = ?u$, have all the element singular as their grounded binding. A variable may occur more than once in a binding-set, as is the case here for $?x$ but there can be only one grounded-binding for each variable. Some variables, such as $?a$ and $?b$ in the above example, may remain unbound, i.e. they have no grounded binding (yet).

We define the function $inst(k, b) \rightarrow k'$ as the instantiation of a feature structure k given a binding-set b . It is equal to a feature structure k' where all variables which have grounded-bindings are replaced by their grounded-binding, and all unbounded variables of the same chain are replaced by a single newly constructed variable, so that k' reflects that these variables are co-referential.

Given a transient structure k , then the following functions define the application of a single construction c in production or comprehension, assuming a binding-set b :

1. **Production:** $produce(c, k_{in}, b) \rightarrow k_{out}$

The production process first tries to match the production-lock $prod_c$

with the transient structure k_{in} , given a binding-set b . If the match is successful, then the comprehension-lock $comp_c$ is completed with k_{in} yielding k' . If this was possible, then k' is completed with the contributing part $cont_c$ to obtain k'' . If this was possible as well, k_{out} is the result of instantiating k'' with b , i.e. $k_{out} = inst(k'', b)$. If any of these steps failed, the construction does not apply.

More formally *produce* is defined as follows:

If U-Match($prod_c, k_{in}, b$) $\rightarrow T$ then

 If U-Merge($comp_c, k_{in}, b$) $\rightarrow T, k'$ then

 If U-Merge($cont_c, k', b$) $\rightarrow T, k''$ then

 The new transient structure $k_{out} = inst(k'', b)$.

 Otherwise c cannot be applied to k_{in} and U-Merge fails.

2. **Comprehension:** $comprehend(c, k_{in}, b) \rightarrow k_{out}$

Try to match the comprehension-lock $comp_c$ with the transient structure k_{in} , given a binding-set b . If they match, then complete the production-lock $comp_c$ with k yielding k' . If this was possible, then complete k' with the contributing part $cont_c$ to obtain k'' . If this was possible as well, $k_{out} = inst(k'', b)$. If any of these steps failed, the construction does not apply.

More formally:

If U-Match($comp_c, k_{in}, b$) $\rightarrow T, b$ then

 If U-Merge($prod_c, k, b$) $\rightarrow T, k'$ then

 If U-Merge($cont_c, k', b$) $\rightarrow T, k''$ then

 The new transient structure is $k_{out} = inst(k'', b)$.

 Otherwise c does not apply to k .

I now define the U-Match and U-Merge functions.

Matching: U-Match(l, k, b) $\rightarrow m$

1. A feature structure l matches with a feature structure k iff there is a set of bindings b such that for every unit in l there is a corresponding unit in k that matches, i.e. $\exists b(\forall u_l \in l, \exists u_k \in k \text{ U-Match}(u_l, u_k, b) \rightarrow T)$.
2. For a given binding-set b , a unit u_l with name n_l and feature-set g_l matches with a unit u_k with name n_k and feature-set g_k iff the names of u_l and u_k are the same, i.e. $n_l = n_k$, and both feature-sets match, i.e. U-Match(g_l, g_k, b) $\rightarrow T$. If n_l or n_k are variables, then they must be part of the same chain in b or have the same grounded-binding.
3. For a given binding-set b , a feature-set g_l matches with a feature-set g_k iff every feature $f_l \in g_l$ matches with a corresponding feature $f_k \in g_k$, i.e. $\forall f_l \in g_l, \exists f_k \in g_k \text{ U-Match}(f_l, f_k, b) \rightarrow T$.
4. For a given binding-set b , a feature f_l matches with a feature f_k iff the feature-names and values of these features match. In other words,

given $f_l = [fn_l : v_l]$ and $u_k = [fn_k : v_k]$ then $\text{U-Match}(f_l, f_k, b) \rightarrow T$ iff $fn_l = fn_k$ and $\text{U-Match}(v_l, v_k, b) \rightarrow T$.

5. To determine whether two values $L = v_l$ and $K = v_k$ match given a binding-set b , we have the following cases:
 - (a) L is an atomic constant symbol and K is also an atomic symbol, then they have to be equal, i.e. $L = K$.
 - (b) L is a variable, then either (i) $K = L$, (ii) K is also a variable, in which case their grounded-bindings have to match or they have to be part of the same chain in b , or (iii) K is not a variable (in other words a constant symbol, set, formula, or feature-set) in which case the grounded-binding of L in b has to match with K .
 - (c) L is a set. In that case, K has to be a set as well and all elements of L have to match with an element in K : $\forall e_l \in L, \exists e_k \in K$ where $\text{U-Match}(e_l, e_k, b) \rightarrow T$. Note that this implies a subset relation. The two sets do not have to have exactly the same elements.
 - (d) L is a predicate with its arguments $L = p_l(\text{arg}_{1,l} \dots \text{arg}_{n,l})$. In that case, (i) the value K has to be a predicate with arguments as well, i.e. $K = p_k(\text{arg}_{1,k} \dots \text{arg}_{n,k})$, (ii) the predicates have to be equal, i.e. $p_l = p_k$, and (iii) the arguments have to match, i.e. $\forall \text{arg}_{i,l} \in v_l, \exists \text{arg}_{i,k} \in v_k$ where $\text{U-Match}(\text{arg}_{i,l}, \text{arg}_{i,k}, b) \rightarrow T$ and $\forall \text{arg}_{i,k} \in v_k, \exists \text{arg}_{i,l} \in v_l$ where $\text{U-Match}(\text{arg}_{i,l}, \text{arg}_{i,k}, b) \rightarrow T$.
 - (e) L is a feature-set. In that case, K has to be a feature-set as well and both feature-sets have to match, i.e. $\text{U-Match}(L, K, b) \rightarrow T$.

In all other cases, $\text{U-Match}(l, k, b) \rightarrow F$.

Merging: $\text{U-Merge}(p, k_{in}, b) \rightarrow m, k_{out}$

1. A feature structure p can U-Merge with a feature structure k_{in} iff $\text{U-Match}(p, k_{in}, b) \rightarrow T$. The units in the merged feature structure k_{out} are either the result of merging the units of p with an existing unit in k or of adding a missing unit of p to k_{out} . So, for every unit $u_p \in p$ either there is a unit $u'_k \in k_{out}$ such that u'_k is the result of merging u_p with its matching unit $u_k \in k$, or, if no such merging was possible, u_p is additional to k_{out} , i.e. $u_p \in k_{out} \setminus k_{in}$.
2. Given a set of bindings b , the merging of a unit p with name n_p and feature-set g_p with a unit k with name n_k and feature-set g_k is possible iff the names of p and k are the same, i.e. $n_p = n_k$, and their feature-sets can be merged, i.e. $\text{U-Merge}(g_p, g_k, b) \rightarrow T, g'_k$. If n_p or

n_k are variables, than they must be part of the same chain or have the same grounded-binding. The new unit k' has as its name n_k if n_k is a constant, the grounded-binding of n_k if it is a variable, or a new variable which replaces all occurrences of n_p and n_k . k' has as its feature-set g'_k .

3. The merging of g_l with a feature-set g_k given a binding-set b is a new feature-set g'_k which is the union of all features $f_l \in g_l$ that merged with a corresponding feature $f_k \in g'_k$ and all features $f_l \in g_l$ that could not be merged with a corresponding unit in g_k .
4. Merging a feature f_l with a feature f_k , given a binding-list b , is possible iff $\text{U-Match}(f_l, f_k, b) \rightarrow T$. Suppose $f_l = [fn_l : v_l]$ and $f_k = [fk : v_k]$ then the merged feature is $f'_k = [fn_k : v'_k]$ where v'_k is the result of merging the two values v_l and v_k .
5. K' is the merge of two values P and K , which are matching given a binding-set b . We have the following cases:
 - (a) If P, K are atomic constant symbols, then $K' = K$. (We already know they are equal because otherwise the features of which they are values would not match.)
 - (b) If P, K are variables, then a merge is possible iff $\text{U-Match}(P, K, b) \rightarrow T$. $K' = K$. (The variables form a chain in b so it does not matter which one we choose.)
 - (c) If P and K are sets, then K' is equal to the set of all merged elements of P and K and those elements in P or K that did not match (and therefore cannot be merged):
$$K' = \{y'|x \in P \text{ and } \exists y \in K, \text{U-Merge}(x, y, b) \rightarrow T, y'\} \cup \\ \{x|x \in P \text{ and } \neg \exists y \in K, \text{U-Match}(x, y, b) \rightarrow F\} \cup \\ \{x|y \in K \text{ and } \neg \exists x \in P, \text{U-Match}(x, y, b) \rightarrow F\}.$$
 - (d) If P, K are predicates with their arguments
 $P = K = p_p(\text{arg}_{1,l} \dots \text{arg}_{n,l})$. Assuming that $\text{U-Match}(P, K, b) \rightarrow T$, then $K' = K$.
 - (e) If P, K are feature-sets then K' is equal to the union of all features that could be merged and the remaining features in P or K that did not match (and therefore could not be merged):
$$K' = \{y'|x \in P \text{ and } \exists y \in K, \text{U-Merge}(x, y, b) \rightarrow T, y'\} \cup \\ \{x|x \in P \text{ and } \neg \exists y \in K, \text{U-Match}(x, y, b) \rightarrow F\} \cup \\ \{x|y \in K \text{ and } \neg \exists x \in P, \text{U-Match}(x, y, b) \rightarrow F\}.$$

In all other cases, $\text{U-Merge}(p, k_{in}, b) \rightarrow F, nil$.