A Reflective Architecture for Robust Language Processing and Learning

Remi van Trijp

Sony Computer Science Laboratory Paris, France

Abstract. Becoming a proficient speaker of a language requires more than just learning a set of words and grammar rules, it also implies mastering the ways in which speakers of that language typically innovate: stretching the meaning of words, introducing new grammatical constructions, introducing a new category, and so on. This paper demonstrates that such meta-knowledge can be represented and applied by reusing similar representations and processing techniques as needed for routine linguistic processing, which makes it possible that language processing makes use of computational reflection.

1 Introduction

When looking at natural language, two striking observations immediately jump to the mind. First, there is an extraordinary amount of diversity among the world's languages [10] and 'almost every newly described language presents us with some "crazy" new category that hardly fits existing taxonomies' [9, p. 119]. Secondly, languages are not static homogeneous entities, but rather complex adaptive systems [24] that dynamically change over time and in which new forms are forever emerging [11]. These observations pose strong requirements on linguistic formalisms, which need to support an enormous amount of variety while at the same time coping with the open-ended nature of language [33].

Both requirements may seem overwhelming for anyone who wants to develop operational explanations for language, but two formalisms within the cognitive linguistics tradition have nevertheless accepted the challenge: Fluid Construction Grammar (FCG; for handling parsing and production; see the remainder of this volume and [26]) and Incremental Recruitment Language (IRL; a constraint language that has been proposed for operationalizing embodied cognitive semantics [20]). Both IRL and FCG have the necessary expressivity for capturing the myriad of conceptual and grammatical structures of language. FCG is based on *feature structures* and *matching and merging* (i.e. unification) [6, 18, 28, 32], whereas IRL is based on *constraints* and *constraint propagation.*[1]

[1] In order to fully appreciate the technical details of this paper, the reader is expected to have a firm grasp of the FCG-system and a basic understanding of IRL. Interested readers are also advised to first check [33] for learning about how problems concerning robustness and fluidity are typically handled in FCG, and [4, 15] for how that approach is implemented.

Both FCG and IRL are embedded in a double-layered *meta-level architecture* for handling unforeseen problems and inventing novel solutions [4, 33]. This meta-level architecture allows the implementation of *computational reflection* [19], which is commonly defined as "the activity performed by a system when doing computation about (and by that possibly affecting) its own computation" [16, p. 21]. The architecture is illustrated in Figure 1. On the one hand, there is a *routine layer* that handles habitual processing. On top of that, a *meta-layer* tries to detect and solve problems that may occur in routine processing through *diagnostics and repairs* (also called *meta-layer operators*). For instance, a diagnostic may detect that the listener encountered an unknown word, while a repair can ask the language system to perform a top-down prediction on what kind of word it is and what its meaning might be (see [4, 33]; and the remainder of this paper for more concrete examples). Once a repair is made, computation resumes at the routine-layer.

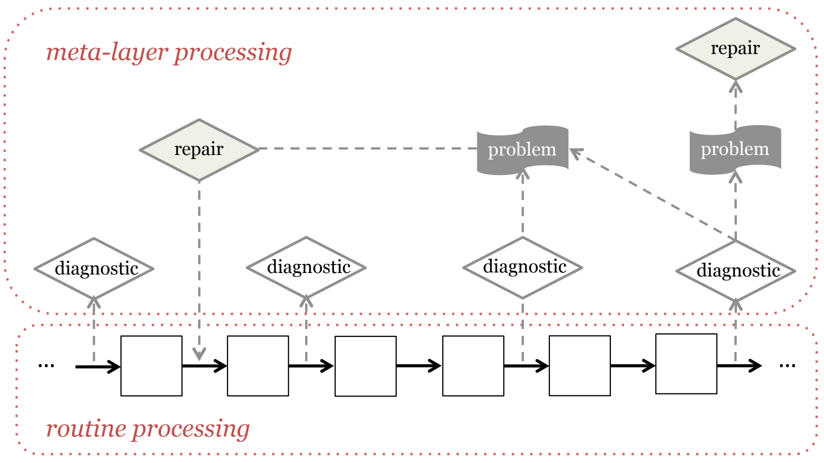

Fig. 1. FCG and IRL feature a double-layered meta-level architecture. Besides a layer for routine processing, a meta-layer diagnoses and repairs problems that may occur in the routine layer [4].

Recent studies on the evolution of language [29, 30] have identified numerous meta-layer operators that operationalize open-ended language processing for specific domains such as agreement [3], tense-aspect [7], event structure [34, 36], space [14, 21], and quantifiers [17]. However, most operationalizations implement the *functions* of the diagnostics and repairs in LISP code, so the language processing system is only reflective in a weak sense. Figure 2 shows a typical example of this approach in pseudo-code. The diagnostic shown in the Figure tries to detect unknown words by looking for unprocessed strings in each last search node (or 'leaf') of linguistic processing. If the set of those strings contains exactly one word, the diagnostic reports an `unknown-word` problem.

diagnostic (detect-unknown-word)

```
when SEARCH-NODE in PARSING is a LEAF
then get the UNPROCESSED-STRINGS from the SEARCH-NODE
     when the UNPROCESSED-STRINGS contain a SINGLE-WORD
     then report UNKNOWN-WORD problem
```

Fig. 2. Most diagnostics and repairs for IRL and FCG are implemented directly as computational functions, whereas full reflection requires that IRL and FCG are their own meta-language.

This paper demonstrates that the same representation and application machinery now used at the routine language processing layer can also be used for the meta-layer, which makes the whole system reflective in a strong sense. By doing so, this paper paves the way towards experiments in which novel language strategies emerge and become culturally shared. More specifically, this paper proposes the following approach:

1. Diagnostics can be represented as *feature structures*, which can be processed by the FCG-interpreter. Problems are detected by *matching* these feature structures against other feature structures.
2. Repairs can be represented as *constraint networks*, which can be configured, executed and chunked by IRL.
3. Diagnostics and repairs that exclusively operate on the linguistic level can be associated to each other in the form of *coupled feature structures* and thus become part of the linguistic inventory in their own right.

I use the term *language strategy* for a particular set of diagnostics and repairs (see [31], for a more complete definition of a language strategy). Language strategies are processed in the meta-layer and allow a language user to acquire and expand a *language system*, which are the concrete choices made in a language for producing and comprehending utterances, such as the English word-order system. Language systems are processed in the routine layer.

2 Diagnostics Based on Matching

This section demonstrates through a series of examples how feature structures, which are used for representing linguistic knowledge in FCG [22], can represent diagnostics. When they are able to match with a transient structure they detect a specific problem.

2.1 A First Example: Detecting an Unknown Word

Let's start with one of the most common problems in deep language processing: unknown words [1]. The key to diagnosing this problem through the FCG machinery (rather than by using a tailored LISP function) is to understand how routine processing handles familiar words. As explained in more detail by [27], FCG processing first involves a *transient structure*, which is a temporary data structure that contains all the information of the utterance that a speaker is producing, or that a listener is parsing. The transient structure consists of a semantic and a syntactic pole. Both poles comprise a set of units, which have feature structures associated with them. The following transient structure represents the initial structure of the utterance *the snark* at the beginning of a parsing task:

Example 1.

```
((top-unit))
<-->
((top-unit
  (form ((string the-unit "the")
         (string snark-unit "snark")
         (meets the-unit snark-unit)))))
```

As can be seen, both the semantic pole (above the double arrow symbol) and the syntactic pole have a unit called `top-unit`. The top-unit on the semantic pole is still empty because we're at the beginning of a parsing task hence the listener has not analyzed any of the words yet. The top-unit of the right pole has a `form` feature, whose value contains two words and an ordering constraint (`meets`) between the words. During parsing, the FCG-interpreter then tries to apply linguistic constructions to the transient structure and, by doing so, modifying it. For example, the following lexical entry applies for the word *the*:

Example 2.

```
((?top-unit
  (tag ?meaning (meaning (== (unique-entity ?entity))))
  (footprints (==0 the-lex)))
 ((J ?the-unit ?top-unit)
  ?meaning
  (args (?entity))
  (footprints (the-lex lex))))
<-->
((?top-unit
  (tag ?form (form (== (string ?the-unit "the"))))
  (footprints (==0 the-lex)))
 ((J ?the-unit ?top-unit)
  ?form
  (footprints (the-lex lex))))
```

The above lexical construction is kept simple for illustration purposes. In parsing, all it does is look for any unit whose `form` feature contains the string "the" in its value. If such a unit is found, the construction builds a new unit for the article and moves the form to this new unit. The construction also leaves a footprint that prevents it from applying a second time. On the semantic pole, the construction builds a corresponding unit for the article and it adds the article's meaning to this new unit. So when applying the lexical construction of example 2 to the transient structure in example 1, the transient structure is modified into the following structure:

Example 3.

```
((top-unit
  (sem-subunits (the-unit)))
 (the-unit
  (meaning ((unique-entity ?entity)))
  (args (?entity))
  (footprints (the-lex lex))))
<-->
((top-unit
  (syn-subunits (the-unit))
  (form ((string snark-unit "snark")
         (meets the-unit snark-unit))))
 (the-unit
  (form ((string the-unit "the")))
  (footprints (the-lex lex))))
```

It is common design practice in FCG to consider the `top-unit` as a buffer that temporarily holds all data concerning meaning (on the semantic pole) or form (on the syntactic pole) until they are moved into their proper units by lexical constructions. If all lexical constructions abide by this design rule, all meanings and strings that are left in the top-unit after all constructions have been tried can be considered as unprocessed and therefore problematic. For detecting whether any unknown words are left, we can thus simply define a *meta-level feature structure* that matches on any string in the top-unit:

Example 4.

diagnostic (string)

```
((?top-unit
  (form (== (string ?any-unit ?any-string)))
  (footprints (==0 lex))))
```

This diagnostic looks exactly like the 'conditional' features of a lexical construction (i.e. units that need to be 'matched' before the other features are merged by the FCG-interpreter), except for the fact that there is a variable `?any-string` instead of an actual string, and that the feature structure cannot trigger if the footprint `lex` has been left in the unit by a lexical construction.

Assume now that the FCG-system did not have a lexical construction for the word `snark`, which is likely because it is an imaginary word invented by Lewis Carroll for his 1876 poem *The Hunting of the Snark*, so routine processing gets stuck at the transient structure of example 3. The FCG-interpreter can now *match* the meta-level feature structure of example 4 with the syntactic pole of that transient structure, which yields the following bindings:

```
((?any-string . "snark") (?any-unit . snark-unit) (?top-unit . top-unit))
```

In other words, the meta-level feature structure matches with the `top-unit` and finds the unknown string *snark*. Here, there is only one unknown word, but if there would be multiple unknown strings, matching would yield multiple hypotheses. This example, which is kept simple for illustration purposes, achieves the same result as the diagnostic that was illustrated in Figure 2.

By exploiting the same feature structure representation as FCG uses for transient structures and linguistic constructions, the FCG-interpreter can be reused without resorting to tailor-made functions. Moreover, the diagnostic explicitly uses the design pattern shared by all lexical constructions, namely that strings and meanings are taken from the buffer-like top-unit and put into their own unit, whereas the tailored function keeps this information implicit.

2.2 Internal Agreement

A common feature of language is ‘internal agreement’, which involves two or more linguistic units that share semantic or syntactic features such as gender or number [3]. For example, in the French noun phrase *la femme* (‘the woman’), the singular-feminine definite article *la* is used in agreement with the gender and number of *femme*, as opposed to the singular-masculine article *le* as in *le garçon* (‘the boy’).

Assume now a grammar of French that uses phrasal constructions for handling agreement between an adjacent determiner and noun, using the design pattern for phrasal constructions as proposed by [25]. The schema in Figure 3 illustrates how a DetNP-construction modifies the transient structure on the left to the resulting transient structure on the right: the construction takes the units for the determiner and the noun and groups them together as subunits of a new NP-unit, which has the same number and gender features as its two subunits.

Just like with lexical constructions, we can exploit the design pattern captured in phrasal constructions for detecting problems. The diagnostic in example 5 detects whether the DetNP-construction applied successfully or not. It looks for any unit that contains at least two subunits, and which does not contain the feature-value pair `(phrase-type nominal-phrase)` (ensuring that the phrasal construction did not apply on this unit). The two specified subunits should ‘meet’ each other (i.e. be adjacent), and they should be a determiner and a noun. Both the determiner and the noun have a `number` and `gender` feature, but their values are unspecified through unique variables for each unit, which allows the actual values to differ from each other.

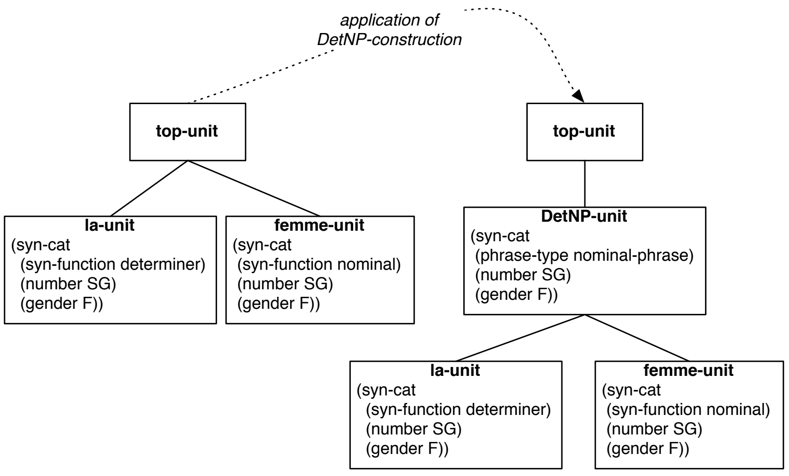

Fig. 3. The DetNP-construction groups a determiner- and noun-unit together as subunits of a phrasal unit.

Example 5.

diagnostic (internal-agreement)

```
((?top-unit
  (syn-subunits (== ?determiner-unit ?noun-unit))
  (form (== (meets ?determiner-unit ?noun-unit)))
  (syn-cat (==0 (phrase-type nominal-phrase))))
 (?determiner-unit
  (syn-cat
   (==1 (gender ?determiner-gender)
        (number ?determiner-number)
        (syn-function determiner))))
 (?noun-unit
  (syn-cat
   (==1 (gender ?noun-gender)
        (number ?noun-number)
        (syn-function nominal)))))
```

Suppose the FCG-interpreter has to parse the ungrammatical utterance **le femme*. Matching the meta-level feature structure would yield the following bindings for those unique variables:

```
((?determiner-number . SG) (?determiner-gender . M)
 (?noun-number . SG) (?noun-gender . F))
```

From these bindings, we can infer that there is a problem with the `gender` feature, because there are two different values for both units: masculine and feminine. The `number` feature, on the other hand, is singular for both units because the variables `?determiner-number` and `?noun-number` are both bound to the same value. Similarly, the diagnostic can detect a problem of number (but not of gender) if it would be matched against a feature structure for the utterance **la femmes*:

```
((?determiner-number . SG) (?determiner-gender . F)
 (?noun-number . PL) (?noun-gender . F))
```

Again, the diagnostic in example 5 does not require a special function, but simply reuses the FCG-interpreter for discovering problems and providing the FCG-system with details about where the problem is found. The diagnostic is, however, specific to a language such as French that has internal agreement of gender and number, but it would be almost useless for English, which does not mark gender agreement between an article and the adjacent noun, and which also does not mark number agreement between a definite article and a noun.

2.3 Word Order

Another widespread language strategy is based on using word order for marking various kinds of grammatical or pragmatic functions, but there is a wide variety in which word order constraints are applied by particular languages. For example, Dutch is fairly free in its word order constraints but it is a so-called V2 (verb second) language, which means that (with the exception of certain constructions), the inflected verbal unit of a Dutch utterance has to be in second position in the main clause. For example, a Dutch speaker would translate 'Yesterday, I went walking' as *Gisteren ging ik wandelen*, literally 'yesterday went I walk'. The following meta-level feature structure is able to diagnose violations of the Dutch V2-constraint:

Example 6.

diagnostic (V2-constraint)

```
((?top-unit
   (syn-subunits
    (== ?unit-a ?unit-b ?verbal-unit))
   (form (== (meets ?unit-a ?unit-b)
             (meets ?unit-b ?verbal-unit))))
 (?verbal-unit
   (syn-cat (==1 (syn-function verbal)
                 (inflected? +)))))
```

The diagnostic uses two adjacency constraints to check whether there is any 'illegal' unit that precedes the verbal unit, which itself is identified through its syntactic function and the inflection constraint. Thus if an English speaker who learns Dutch would say **Gisteren ik ging wandelen*, the diagnostic would find matching variables for `?unit-a` (the adverbial phrase *gisteren* 'yesterday') and `?unit-b` (the subject *ik* 'I'), which means that at least one of these two units is in the wrong position.

2.4 Unexpressed Meanings

Diagnostics can not only be of lexical or morphosyntactic nature, but also target semantic properties. In dialogues it often happens that a language user cannot remember a particular word for the meaning he wishes to express, especially when speaking in a foreign language. This problem is equivalent to detecting unknown words, hence we can use a similar solution on the meaning side. The meta-level feature structure of example 7 triggers on all meanings that have been left unprocessed by lexical constructions.

Example 7.

diagnostic (meaning)

```
((?top-unit
  (meaning (== (?predicate . ?args)))
  (footprints (==0 lex))))
```

Suppose that FCG-processing got stuck at a transient structure that includes the following top-unit, which contains an unprocessed temporal meaning that states that one event happened before another one:

```
(top-unit
 (meaning ((before ev-1 ev-2))))
```

Matching the meta-level structure yields the following bindings:

```
((?predicate . before) (?args ev-1 ev-2) (?top-unit . top-unit))
```

The diagnostic finds out that the unexpressed meaning predicate (if one uses a predicate calculus for handling meaning) is `before`. The diagnostic also uses a dot before the variable `?args`, which is a Common Lisp notation that corresponds to 'the rest of the list', so `?args` is bound to the remainder of the meaning element `(ev-1 ev-2)`. For each unexpressed meaning predicate, the diagnostic thus returns a binding for the predicate and its arguments.

2.5 Valence

A major challenge for linguistic formalisms is the distribution of verbs (i.e. which argument structures are compatible with which verbs). Usually, a verbal lexical entry contains a `valence` feature that states which semantic (and syntactic) roles the verb can assign to its arguments. Problems, however, arise when speakers wish to override those constraints, as when for instance using the intransitive verb `sneeze` in a Caused-Motion frame as in *Pat sneezed the napkin off the table* [8, p. 3]. Assume the following meaning and semantic valence for the verb *sneeze*:

```
(meaning
 (== (sneeze ?ev)
     (sneezer ?ev ?sneezer)))
(sem-cat
 (==1 (sem-valence
       (==1 (agent ?ev ?sneezer)))))
```

Using a predicate calculus for meaning, the verb contains one predicate for the event itself and one predicate for the participant who's sneezing (the sneezer). The semantic valence of the verb states that the sneezer can be categorized in terms of the semantic role `agent` by repeating the variable `?sneezer` for its argument, thereby making the verb compatible with the intransitive construction (see [35] for a detailed discussion of valence and argument structure constructions in FCG).

The Caused-Motion construction, however, requires not only an Agent but also a Patient and a Direction, as found in the valence of verbs that typically express caused motion such as *push* and *pull*. There is thus a mismatch between the valence of *sneeze* and the requirements of the Caused-Motion construction, which means that the argument structure construction will not be triggered during routine processing.

According to Goldberg [8], this problem can be solved through *coercion*. Goldberg argues that the Caused-Motion construction only specifies that the Agent is obligatory and that the other roles can be added by the construction itself on the condition that there are no conflicts with the semantics of the verb. If we want to operationalize this hypothesis, an adequate diagnostic thus needs to figure out the following two things. First, it has to detect that the Caused-Motion construction failed to apply, and if so, it has to check whether the verb can be coerced into the Caused-Motion frame if necessary. The meta-level feature structure shown in example 8 achieves both goals.

Example 8.

diagnostic (Caused-Motion)

```
((?top-unit
   (sem-subunits
    (== ?agent-unit ?verbal-unit ?patient-unit
        ?direction-unit))
   (meaning
    (== (cause-move ?ev) (causer ?ev ?agent)
        (moved ?ev ?patient)
        (direction ?ev ?direction)))
   (footprints (==0 arg-cxn)))
  (?verbal-unit
   (sem-cat (==1 (sem-valence
                  (==1 (agent ?ev ?agent)))))
   (args (?ev)))
  (?agent-unit
   (args (?agent)))
  (?patient-unit
   (args (?patient)))
  (?direction-unit
   (args (?direction))))
```

The diagnostic in example 8 matches if there is a unit whose `meaning` feature contains a Caused-Motion frame. The `footprints` ensure that no argument-structure construction has applied on this unit. The diagnostic also verifies whether all necessary units are present, and whether the semantic valence of the verb contains at least the obligatory Agent-role. When matched against a transient structure that contains the verb *sneeze*, the diagnostic would thus immediately be capable of linking the sneezer-role to the causer-role in the Caused-Motion frame through the variable `?agent`.

3 Diagnostics Based on Merging

The examples in the previous sections have all used the *matching* facility of the FCG-interpreter. A second way in which meta-level feature structures can be exploited is to use *merging*. The merge-operation is more permissive than matching and hence should only be used as a test for conflicts. In this usage, the meta-level feature structure does not represent a particular problem, but rather captures certain 'felicity conditions' of a language. This means that a failure in merging it with the transient structure reveals a violation of the constraints of that language. Let's take the example of internal agreement in French again. The following meta-level feature structure checks whether the `number` and `gender` features of a determiner and an adjacent noun agree with each other when it is merged with a transient structure:

Example 9.

diagnostic (internal-agreement-2)

```
((?top-unit
  (syn-subunits (== ?determiner-unit ?noun-unit))
  (form (== (meets ?determiner-unit ?noun-unit))))
 (?determiner-unit
  (syn-cat
   (==1 (gender ?gender)
        (number ?number)
        (syn-function determiner))))
 (?noun-unit
  (syn-cat
   (==1 (gender ?gender)
        (number ?number)
        (syn-function nominal)))))
```

The meta-level feature structure uses the same variables `?gender` and `?number` for both units, indicating that the determiner and the noun need to have the same value for both features. Merging the meta-level feature structure with the transient structure of *la femme* ('the woman') would thus succeed because both forms are feminine-singular. However, merging would fail for utterances such as **le femme* because the two words have different gender values.

However, using FCG's merging operation for diagnosing problems is less powerful than using the matcher because there is less feedback: merging simply fails without providing more information about what caused the failure. So when the diagnostic of example 9 reports a problem, it cannot say whether the problem is caused by mismatches in gender, number or both.

4 Exploiting Constraint Networks for Repairs

As explained in more detail by [4], *repairs* are powerful operations that try to solve the problems detected by diagnostics. Repairs are able to modify the inventory used in routine processing, and to restart or even repurpose a parsing or production task. As is the case for diagnostics, most currently implemented repairs for FCG are specific functions that look as the repair shown in Figure 4.

repair (add-meta-level-cxn problem parsing-task)

```
If there is an UNKNOWN-WORD in PROBLEM
then add a META-LEVEL CONSTRUCTION of UNKNOWN-WORD to GRAMMAR
   and restart PARSING-TASK
else return FAILURE
```

Fig. 4. Most current FCG repairs are implemented as specific LISP functions.

The repair handles unknown words by inserting a meta-level construction in the linguistic inventory. Interested readers are kindly referred to [33] to learn more about how the solution works. Roughly speaking, the meta-level construction creates a new unit for the unknown word and makes it compatible with any semantic and syntactic specification, which may trigger the application of other constructions that were previously blocked. Returning to our example of *the snark*, for instance, a DeterminerNominal construction can now treat *snark* as a noun because it is immediately preceded by a determiner. FCG can thus overcome the problem and continue parsing until a final transient structure is found that passes all the ‘goal tests’ that decide on the adequacy of a parse result [5].

Writing a specific function for each repair is useful for fast prototyping, but soon becomes problematic. First, there is no uniform and coherent way of representing and processing repairs. As a result, it largely depends on the grammar engineer’s appreciation of particular problems whether the repair is adequate or not. Secondly, there is no ‘safe’ way of testing the consequences of a repair: repairs have to ‘commit’ their changes to the linguistic inventory and then restart a process before any hidden effects may pop up. Needless to say, when complex problems need to be solved, writing adequate repairs soon involves a lot of trial and error, even for experienced grammar engineers.

This paper proposes that the implementation of repairs should be treated as *constraint satisfaction problems*, which allows the grammar engineer to define constraints that need to be satisfied before a repair is allowed to commit its changes to the inventory. For this purpose, repairs can be formulated as *constraint networks* using IRL, a constraint language that has been proposed for handling conceptualization and interpretation [20], and which can be considered as FCG’s sister formalism. The next subsection shows a first example of an ‘IRL repair’. The subsequent section then goes a step further and shows the full power of IRL for implementing repairs.

4.1 A First Example: Repairing an Unknown Word

Assume that a diagnostic has detected the unknown word *snark* in the utterance *the snark*. We now need to define a network of constraints that performs the same operations as the repair function in Figure 4 and that first tests the solution instead of immediately committing any changes and restarting the parsing task. Every IRL repair network consists of the following elements:

– *An object store*, which contains ‘linguistic types’ such as transient structures, constructions, meanings and strings. Specific instantiations of each type are called ‘linguistic instances’.
– *A component store*, which contains ‘linguistic operators’, which are the building blocks of each network. These operators perform specific operations on linguistic instances, such as applying a construction, fetching the inventory of constructions, and so on.

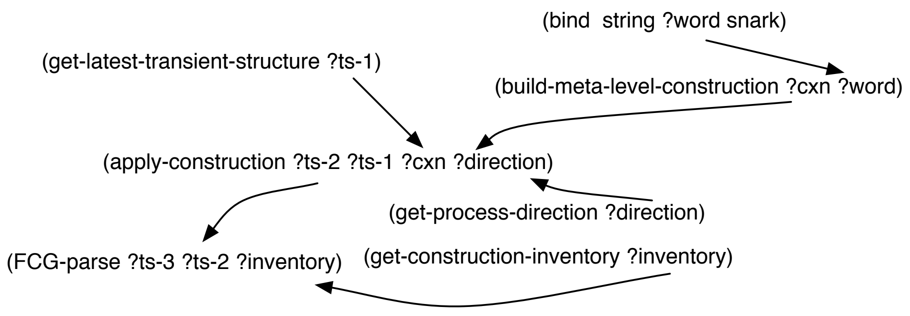
(bind string ?word snark)
(get-latest-transient-structure ?ts-1)
(build-meta-level-construction ?cxn ?word)
(apply-construction ?ts-2 ?ts-1 ?cxn ?direction)
(get-process-direction ?direction)
(FCG-parse ?ts-3 ?ts-2 ?inventory)
(get-construction-inventory ?inventory)

Fig. 5. A possible IRL repair network for handling unknown words. This first example does not yet exploit the full power of IRL and uses a sequential order for the execution of linguistic operations (with precedence relations indicated by the arrows).

A possible repair network is shown in Figure 5. Every node in a repair network evokes a specific linguistic operation, represented by the name of the operator and a list of its arguments, for instance `(get-construction-inventory ?inventory)`. The arguments are variables (indicated by a question mark) that are or will be bound to a linguistic instance. Binding variables to a specific linguistic instance of a certain type is done by the special operator `bind`. For example, the operation `(bind string ?word snark)` binds the word *snark* (which is of type `string`) to the variable `?word`. Different operations in the network are linked to each other through the variables. For example, the variable `?inventory` is shared by two operations. In its list notation, the network looks as follows:

```
((bind string ?word snark)
 (get-construction-inventory ?inventory)
 (get-latest-transient-structure ?ts-1)
 (get-process-direction ?direction)
 (build-meta-level-construction ?cxn ?word)
 (apply-construction ?ts-2 ?ts-1 ?cxn ?direction)
 (FCG-parse ?ts-3 ?ts-2 ?inventory))
```

An explanation of how each operator can be implemented in IRL can be found in [20]. When described in words, the network performs the following operations:

1. Take the unknown word *snark* of type `string` (provided by a diagnostic) and bind it to the variable `?word`.
2. Get the construction inventory from the parsing task in which the problem was detected and bind it to the variable `?inventory`.

3. Get the latest transient structure from the parsing task in which the problem was detected and bind it to the variable `?ts-1` (transient structure 1).
4. Get the processing direction (parsing or production) from the task in which the problem was detected and bind it to the variable `?direction`.
5. Build a meta-level construction using the linguistic instance bound to the variable `?word`. Bind the meta-level construction to the variable `?cxn` (construction).
6. Take the linguistic instances bound to the variables `?cxn` (of type 'construction'), `?ts-1` (of type 'transient structure') and `?direction` (of type 'process-direction'). Apply the construction to the transient structure to obtain a new transient structure. Bind this new transient structure to the variable `?ts-2`.
7. Take the linguistic instances bound to `?inventory` (of type 'construction-inventory') and `?ts-2` (of type 'transient structure'). Perform a parsing task with these linguistic instances in order to obtain a new transient structure. Bind the final transient structure to `?ts-3`.

Particular constraints can be defined for each operation. For example, the operation `FCG-parse` only succeeds if the final transient structure passes all the goal tests that the grammar engineer defined for obtaining adequate results. Likewise, applying a single construction is only successful if (a) the construction indeed applies on a particular transient structure, and (b) if the resulting transient structure passes all the 'node tests' defined in the FCG-system [5].

4.2 Dataflow Repairs

The example of the previous section showed how IRL can provide the grammar engineer with a safer way of testing the adequacy of repairs. However, the applied IRL-network still featured a simple sequential control flow in which there is a fixed order in which the network's constraints are executed, which does not necessarily warrant a constraint propagation approach. This section justifies the use of IRL by showing how the formalism allows the operationalization of *dataflow repairs*, in which information can propagate in any direction depending on instance availability [23]. The main advantage of dataflow repairs is that they can be used for both parsing and production at the same time.

Apply-construction. Using dataflow instead of control flow makes it possible to develop more powerful linguistic operators. For instance, in a control flow approach, the operator `apply-construction` (which takes four arguments: a resulting transient structure, a source transient structure, a construction and a process-direction; see Figure 4) requires three available instances (the source transient structure, construction and direction) before it can return a new transient structure. In a dataflow approach, at least the scenarios listed in Table 1 become possible.

In principle, it is also possible to implement a scenario in which the operator has the output transient structure and the construction as available instances in

Available instances	**Computation**
- Resulting transient structure - Source transient structure	The operator can compute which construction needs to be applied (and in which direction) in order to go from the source to the resulting transient structure.
- Resulting transient structure - Source transient structure - Direction	The operator can compute which construction needs to be applied in order to go from the source to the resulting transient structure.
- Resulting transient structure - Source transient structure - Construction	The operator can compute the direction of application.
- Resulting transient structure - Source transient structure - Construction - Direction	The operator can perform a 'sanity check' to see whether application of the construction on the source structure indeed leads to the resulting structure.

Table 1. In a dataflow approach, the operator *apply-construction* can perform different computations depending on the availability of linguistic instances.

order to compute what the source transient structure was. This scenario however requires the retroactive application of constructions, which is currently not supported in FCG.

In sum, depending on the particular configuration of instance availability, the operators can already perform various computations and pass the results to other operators instead of waiting for other procedures to finish their work. IRL keeps cycling through each operator until no more computation can be achieved.

Build-meta-level-construction. With the power of IRL's dataflow approach comes the possibility of anticipating different scenarios of instance availability depending on the task that the FCG-interpreter needs to perform. For example, when building a meta-level construction for an unknown word during parsing, the FCG-interpreter already has the form of the new construction at its disposal. The same function, however, would also be useful for a problem in production in which the FCG-interpreter has a novel meaning to express but no corresponding form yet. A new 'call pattern' for the operator that allows the meaning and form to be specified looks as follows:

```
(build-meta-level-construction ?cxn ?meaning ?word)
```

The operator now needs to be implemented in such a way that it can handle at least the situations listed in Table 2. Now that the principle and power of dataflow repairs are clear, it is time to change the IRL repair of Figure 5 into a repair that can be applied in both parsing and production. Such a repair is shown in Figure 6. The linking lines between operations in the figure do not have

Available instances	Computation
- String (parsing)	The operator assumes a meta-level meaning and builds a meta-level construction. The results are bound to the variables `?meaning` and `?cxn`.
- Meaning (production)	The operator assumes a meta-level form and creates a new construction. The results are bound to the variables `?word` and `?cxn`.
- Meaning - String	The operator builds a meta-level construction and binds it to the variable `?cxn`.

Table 2. Different scenarios of instance availability for the linguistic operator *build-meta-level-construction.*

arrows anymore, which illustrates the dataflow approach. The bind operations are shown in dark grey and italics to indicate that their availability depends on whether FCG is producing or parsing an utterance.

Apart from the new call pattern for `build-meta-level-construction`, the operator `FCG-parse` has been replaced by the more general operator `FCG-apply`. As opposed to `FCG-parse`, this operator takes a fourth argument, which is the direction of processing: from meaning to form, or from form to meaning. The direction is provided by the operator `get-process-direction`, which fetches the direction from the task that the FCG-interpreter is performing. In its list notation, the network looks as follows (the bind operations are left out; these have to be provided by the diagnostics):

```
((get-construction-inventory ?inventory)
 (get-latest-transient-structure ?ts-1)
 (get-process-direction ?direction)
 (build-meta-level-construction ?cxn ?meaning ?word)
 (apply-construction ?ts-2 ?ts-1 ?cxn ?direction)
 (FCG-apply ?ts-3 ?ts-2 ?inventory ?direction))
```

5 Diagnostics and Repairs as Coupled Feature Structures

The previous two sections have shown how FCG and IRL can be exploited for representing and processing meta-level operators. The approach can readily be applied in the current formalisms without needing any extensions. One limitation of the current implementation, however, is that the repair of a problem involves a lot of (computational) red tape: first, a diagnostic needs to be implemented that operates on certain predefined situations. If the diagnostic detects issues in processing, it needs to instantiate a problem, which subsequently triggers one or more repairs [4]. Using problems as 'mediators' between diagnostics and

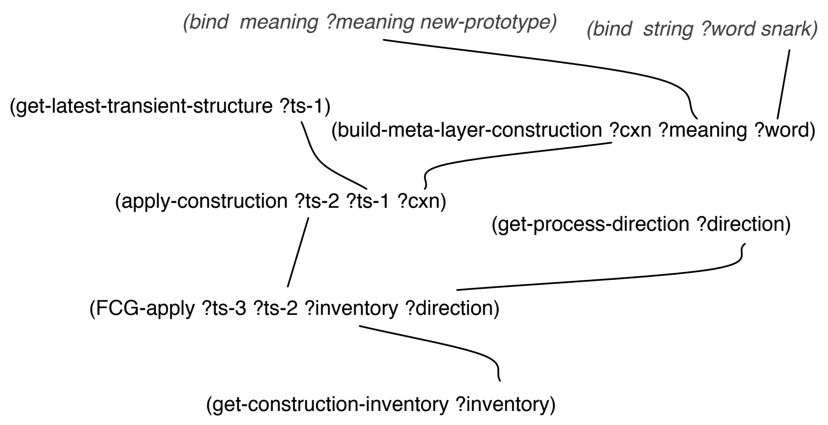
(bind meaning ?meaning new-prototype)
(bind string ?word snark)
(get-latest-transient-structure ?ts-1)
(build-meta-layer-construction ?cxn ?meaning ?word)
(apply-construction ?ts-2 ?ts-1 ?cxn)
(get-process-direction ?direction)
(FCG-apply ?ts-3 ?ts-2 ?inventory ?direction)
(get-construction-inventory ?inventory)

Fig. 6. A dataflow repair. This IRL network implements the solution of a meta-level construction for both parsing and production. The bind operations for meaning and form are shown in grey italic because their availability depends on the direction of processing.

repairs allows for a lot of flexibility, but it would often be much more efficient to implement a 'quick fix' by directly coupling a repair to a diagnostic.

This section explores a way in which meta-level operators can be directly associated to each other in the form of *coupled feature structures*, thereby representing them in the same way as transient structures and constructions. In the remainder of this paper, I will use the term *fix* for the association of a diagnostic and an IRL-repair, because they are meant to be efficient solutions that blend in seamlessly with routine processing. All of the examples presented in this section have been computationally implemented in a proof-of-concept fashion and are therefore not (yet) part of the current FCG implementation.

5.1 Coupled Feature Structures

In FCG, both transient structures and constructions use the same feature structure representation, which is implemented in CLOS (Common Lisp Object System; see [13]). A `coupled-feature-structure` is the base class for both, which has a left pole and a right pole:

Definition 1.

class	**coupled-feature-structure**
Description	An association of two feature structures.
Slots	`left-pole`
	`right-pole`

Transient structures are direct instantiations of coupled feature structures. For each pole, it can be specified which domain it belongs to: semantic or syntactic. By default, the left pole is semantic and the right pole is syntactic. A `construction` is a subclass of a `coupled-feature-structure` that contains additional slots that are relevant for their application, but which do not matter for our current purposes. The FCG-interpreter uses the domain of a pole of a construction to decide whether it should operate on the semantic or on the syntactic pole of a transient structure.

In order to integrate a meta-level fix into FCG-processing, we need to define another subclass of a `coupled-feature-structure`, which inherits a left pole and a right pole. Diagnostics are contained in the right pole (as they can be considered as the 'form' of a problem) and repairs go in the left pole (as they are the 'meaning' of a problem). In principle, no additional slots are required, but here we include three slots called `name`, `domain` and `score`:

Definition 2.

class	**fix**	subclass of coupled-feature-structure
Description	A coupling of a diagnostic and a repair.	
Slots	`name`	
	`domain`	
	`score`	

The `name` of a fix is a symbol for identifying it. The slot `score` could potentially be exploited to orchestrate a competition between different fixes if there are multiple ways of repairing the same problem, or if there are different problems that try to exploit the same repair. The `domain` slot specifies whether the diagnostic of the fix should operate on the semantic or the syntactic pole of a transient structure.

5.2 Extending FCG-apply

Let's return to the problem of *Pat sneezed the napkin off the table*, where the speaker wishes to express a Caused-Motion frame using the intransitive verb *sneeze*, whose valence is incompatible with the requirements of the Caused-Motion construction. However, coercing verbs into the Caused-Motion frame is a recurrent and productive pattern in English [8], hence it is worthwhile to implement a `fix` that first decides whether coercion is needed and indeed possible (i.e. whether the verb can be coerced), and if so, immediately performs coercion.

We have already defined a meta-level feature structure in example 8 that is capable of detecting the need and opportunity for coercion: if no argument structure construction has been applied, the diagnostic checks whether the speaker wishes to express the Caused-Motion frame and whether the verb can at least assign the Agent-role to one of its arguments, which is the only obligatory semantic role. If so, the diagnostic is supposed to report a problem. In a `fix`, however, the repair is immediately triggered. Figure 7 shows how the diagnostic and repair are coupled to each other, with the diagnostic on the right and the repair on the left (both in a graphical representation). In order for the `fix` to be applied, the method `fcg-apply` (which is used for applying constructions)

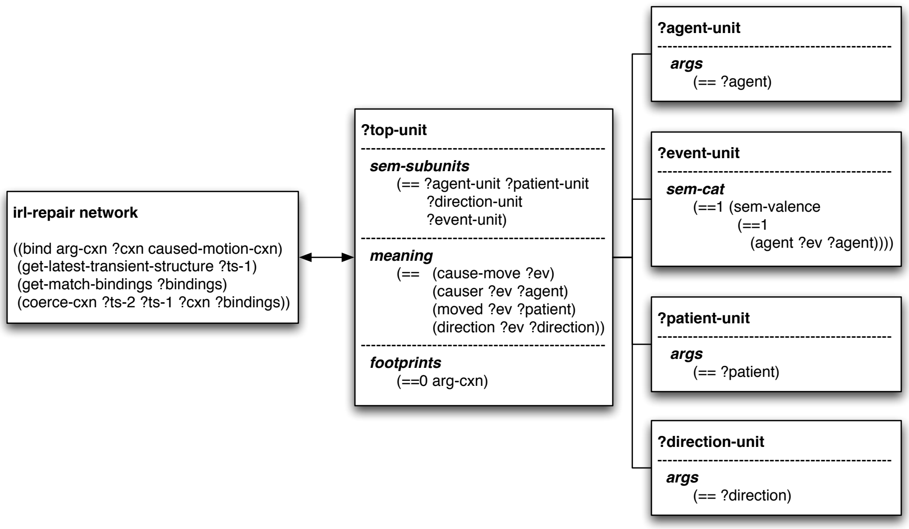
irl-repair network
((bind arg-cxn ?cxn caused-motion-cxn)
(get-latest-transient-structure ?ts-1)
(get-match-bindings ?bindings)
(coerce-cxn ?ts-2 ?ts-1 ?cxn ?bindings))
?top-unit
sem-subunits
(== ?agent-unit ?patient-unit ?direction-unit ?event-unit)
meaning
(== (cause-move ?ev) (causer ?ev ?agent) (moved ?ev ?patient) (direction ?ev ?direction))
footprints
(==0 arg-cxn)
?agent-unit
args
(== ?agent)
?event-unit
sem-cat
(==1 (sem-valence (==1 (agent ?ev ?agent))))
?patient-unit
args
(== ?patient)
?direction-unit
args
(== ?direction)

Fig. 7. This 'fix' associates a diagnostic (right pole) with a repair (left pole). The diagnostic first checks the need and opportunity for coercion. The repair then performs coercion, thereby exploiting the bindings obtained by matching the diagnostic.

needs to specialize on applying a `fix` to a transient structure. This specialized method looks as follows:

FCG-apply `(fix transient-structure)`

```
if the DOMAIN of FIX is SEMANTIC
then MATCH the DIAGNOSTIC of FIX
       with SEMANTIC-POLE of TRANSIENT-STRUCTURE
else MATCH it with SYNTACTIC-POLE of TRANSIENT-STRUCTURE
if MATCH is found
then EXECUTE the IRL REPAIR NETWORK of FIX and RETURN RESULT
else do nothing
```

All fixes can be applied using this specialized method. Interested readers can check a detailed discussion of how FCG achieves coercion in [33]. Summarizing in words, the repair needs to perform the following linguistic operations:

1. `Bind` the Caused-Motion construction to the variable `?cxn`. This is already possible because the `fix` implements a construction-specific solution.
2. Get the latest transient structure (against which the diagnostic was matched) and bind it to `?ts-1`.
3. Get the bindings obtained from matching the diagnostic pole, and bind that information to the variable `?bindings`.

4. Coerce the construction bound to `?cxn` into the transient structure bound to `?ts-1`, using the bindings bound to `?bindings`, to obtain a new transient-structure. Bind that structure to `?ts-2`.
5. If the repair is successful, return the result as a new search node so FCG can continue routine processing.

5.3 Advantages and Issues of Fixes

Besides efficiency and the blending of fixes with routine processing, one of the main advantages of a `fix` is that the bindings obtained from matching the diagnostic against the transient structure can be passed to other linguistic operators. For example the operator `coerce-cxn` performs a powerful operation that skips FCG's matching phase and tries to merge both poles of a construction with a transient structure [33]. Without the inhibitive constraints of matching, however, there is always the danger of an explosion of the possible merge-results, which is exactly the reason why other precision-grammar formalisms that do not have a matching phase implement additional constraints on the unification of feature structures [12, p. 437]. By passing the match-bindings to `coerce-cxn`, improbable coercions can be ruled out. Moreover, if the IRL-repairs are treated as feature structures, they can be matched and merged as well, which opens up the possibility of enriching the repairs with additional information in the form of feature structures.

The most natural way of applying a meta-level fix is to first detect problems by matching the diagnostic against a transient structure and then solving the problem by executing the IRL repair network. However, just like linguistic constructions are bidirectional, fixes can in principle be applied in the other direction as well. For instance, if a speaker introduces a novelty in conversation, the listener might infer why he did so (i.e. detect what the speaker's problem was) by recognizing which repair was performed by the speaker. The prospects of reasoning over fixes for learning and robust language processing is in itself an exciting new research avenue that needs closer examination in future work.

In sum, it is technically speaking possible to blend meta-level fixes with routine processing: all that is required is an additional class `fix` and an `fcg-apply` method that specializes on this new class. However, blending fixes with routine processing requires more control on *when* a fix is allowed to fire: the fix in Figure 7 should only be applied after routine processing has tried all argument structure constructions, and likewise, a fix that would insert a meta-level construction for handling unknown words should only be executed after the application of 'normal' lexical constructions. Fortunately, FCG already provides various ways in which the application of constructions (and fixes) can be regulated, such as specializing the search algorithm [5], using dependency networks [37], or by treating fixes as 'defaults' using construction sets [2].

6 Discussion and Conclusion

This paper has demonstrated how the same representations and processing techniques that support routine language processing can be reused for implementing diagnostics and repairs, which makes it possible that language processing makes use of strong computational reflection. All of the discussed examples have been fully implemented; most of them without needing any extensions to the existing computational frameworks of FCG and IRL. The approach adopted in this paper has both practical and scientific merits: it offers new ways for grammar engineers to operationalize their hypotheses, and it paves the way to research on how language strategies can be culturally acquired by autonomous agents.

The first set of examples demonstrated how diagnostics can be represented as feature structures, which can be matched against transient structures by the FCG-interpreter. Using feature structures provides a uniform way of representing linguistic knowledge in transient structures, constructions and diagnostics, which potentially allows diagnostics to be directly abstracted from recurrent patterns and structures of a particular language.

Next, I have shown how repairs can be represented as constraint networks using IRL. Such repairs consist of linguistic operators (such as coercion) that perform operations on linguistic instances (such as constructions). There are three main advantages of using this approach. First, the dataflow of IRL constraints allows the same repair to work for both production and parsing. Secondly, IRL provides grammar engineers with a coherent and safer way of implementing and testing adequate repairs. Finally, the use of IRL allows future research to focus on the origins of culturally acquired repairs by letting IRL autonomously compose networks of linguistic operations and chunking successful networks.

The final part has shown how diagnostics and repairs can be coupled to each other and become a meta-level 'fix' for processing problems. Using a small extension to the FCG-system for handling associations of diagnostics and repairs, fixes can blend in with routine processing and efficiently handle problems immediately as they occur. Future research needs to focus on how associations of diagnostics and repairs can marry the strengths of constraint networks (IRL) and feature structures (FCG) in a more powerful way.

Acknowledgements

This research was conducted at and funded by the Sony Computer Science Laboratory Paris, with additional funding from the EU FP7 Alear project. I would like to thank Luc Steels, director of the VUB AI-Lab at the Vrije Universiteit Brussel and Sony CSL Paris, for his invaluable feedback and pioneering ideas. I also thank my colleagues for their support, in particular Katrien Beuls for her courtesy in allowing me to use Figure 1, Kevin Stadler for his comments, and Joachim De Beule (whose earlier work on FCG included a system of 'fixes' that inspired this research). All remaining errors are of course my own.

Bibliography

[1] Baldwin, T., Beavers, J., Bender, E.M., Flickinger, D., Kim, A., Oepen, S.: Beauty and the beast: What running a broad-coverage precision grammar over the BNC taught us about the grammar – and the corpus. In: Kepser, S., Reis, M. (eds.) Linguistic Evidence: Empirical, Theoretical, and Computational Perspectives, pp. 49–69. Mouton de Gruyter, Berlin (2005)

[2] Beuls, K.: Construction sets and unmarked forms: A case study for Hungarian verbal agreement. In: Steels, L. (ed.) Design Patterns in Fluid Construction Grammar. John Benjamins, Amsterdam (2011)

[3] Beuls, K., Steels, L., Höfer, S.: The emergence of internal agreement systems. In: Steels, L. (ed.) Experiments in Cultural Language Evolution. John Benjamins, Amsterdam (2012)

[4] Beuls, K., van Trijp, R., Wellens, P.: Diagnostics and Repairs in Fluid Construction Grammar. In: Steels, L., Hild, M. (eds.) Language Grounding in Robots. Springer, New York (2012)

[5] Bleys, J., Stadler, K., De Beule, J.: Search in linguistic processing. In: Steels, L. (ed.) Design Patterns in Fluid Construction Grammar. John Benjamins, Amsterdam (2011)

[6] De Beule, J.: A formal deconstruction of Fluid Construction Grammar. In: Steels, L. (ed.) Computational Issues in Fluid Construction Grammar. Springer Verlag, Berlin (2012)

[7] Gerasymova, K., Spranger, M.: An Experiment in Temporal Language Learning. In: Steels, L., Hild, M. (eds.) Language Grounding in Robots. Springer, New York (2012)

[8] Goldberg, A.E.: A Construction Grammar Approach to Argument Structure. Chicago UP, Chicago (1995)

[9] Haspelmath, M.: Pre-established categories don't exist – consequences for language description and typology. Linguistic Typology 11(1), 119–132 (2007)

[10] Haspelmath, M., Dryer, M.S., Gil, D., Comrie, B. (eds.): The World Atlas of Language Structures. Oxford University Press, Oxford (2005)

[11] Hopper, P.: Emergent grammar. BLC 13, 139–157 (1987)

[12] Jurafsky, D., Martin, J.H.: Speech and Language Processing. An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Prentice Hall, New Jersey (2000)

[13] Keene, S.: Object-Oriented Programming in Common Lisp: A Programmar's Guide to CLOS. Addison-Wesley, Boston (MA) (1988)

[14] Loetzsch, M., van Trijp, R., , Steels, L.: Typological and computational investigations of spatial perspective. In: Wachsmuth, I., Knoblich, G. (eds.) Modeling Communication with Robots and Virtual Humans, pp. 125–142. LNCS 4930, Springer, Berlin (2008)

[15] Loetzsch, M., Wellens, P., De Beule, J., Bleys, J., van Trijp, R.: The babel2 manual. Tech. Rep. AI-Memo 01-08, AI-Lab VUB, Brussels (2008)

[16] Maes, P.: Issues in computational reflection. In: Maes, P., Nardi, D. (eds.) Meta-Level Architectures and Reflection, pp. 21–35. Elsevier, Amsterdam (1988)

[17] Pauw, S., Hilferty, J.: The emergence of quantifiers. In: Steels, L. (ed.) Experiments in Cultural Language Evolution. John Benjamins, Amsterdam (2012)

[18] Sierra, J.: A logic programming approach to parsing and production in Fluid Construction Grammar. In: Steels, L. (ed.) Computational Issues in Fluid Construction Grammar. Springer Verlag, Berlin (2012)

[19] Smith, B.C.: Procedural Reflection in Programming Languages. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge MA (1982)

[20] Spranger, M., Pauw, S., Loetzsch, M., Steels, L.: Open-ended Procedural Semantics. In: Steels, L., Hild, M. (eds.) Language Grounding in Robots. Springer, New York (2012)

[21] Spranger, M., Steels, L.: Emergent functional grammar for space. In: Steels, L. (ed.) Experiments in Cultural Language Evolution. John Benjamins, Amsterdam (2012)

[22] Steels, L., De Beule, J., Wellens, P.: Fluid Construction Grammar on Real Robots. In: Steels, L., Hild, M. (eds.) Language Grounding in Robots. Springer, New York (2012)

[23] Steels, L.: The emergence of grammar in communicating autonomous robotic agents. In: Horn, W. (ed.) Proceedings of the 14th European Conference on Artificial Intelligence (ECAI). pp. 764–769. IOS Press, Berlin, Germany (August 2000)

[24] Steels, L.: Language as a complex adaptive system. In: Schoenauer, M., Deb, K., Rudolph, G., Yao, X., Lutton, E., Merelo, J., Schwefel, H.P. (eds.) Parallel Problem Solving from Nature. pp. 17–28. LNCS 1917, Springer-Verlag, Berlin (2000)

[25] Steels, L.: A design pattern for phrasal constructions. In: Steels, L. (ed.) Design Patterns in Fluid Construction Grammar. John Benjamins, Amsterdam (2011)

[26] Steels, L. (ed.): Design Patterns in Fluid Construction Grammar. John Benjamins, Amsterdam (2011)

[27] Steels, L.: A first encounter with Fluid Construction Grammar. In: Steels, L. (ed.) Design Patterns in Fluid Construction Grammar. John Benjamins, Amsterdam (2011)

[28] Steels, L.: Design methods for Fluid Construction Grammar. In: Steels, L. (ed.) Computational Issues in Fluid Construction Grammar. Springer Verlag, Berlin (2012)

[29] Steels, L. (ed.): Experiments in Cultural Language Evolution. John Benjamins, Amsterdam (2012)

[30] Steels, L.: Modeling the cultural evolution of language. Physics of Life Reviews (2012)

[31] Steels, L.: Self-organization and selection in cultural language evolution. In: Steels, L. (ed.) Experiments in Cultural Language Evolution. John Benjamins, Amsterdam (2012)

[32] Steels, L., De Beule, J.: Unify and merge in Fluid Construction Grammar. In: Vogt, P., Sugita, Y., Tuci, E., Nehaniv, C. (eds.) Symbol Grounding and Beyond. pp. 197–223. LNAI 4211, Springer, Berlin (2006)

[33] Steels, L., van Trijp, R.: How to make construction grammars fluid and robust. In: Steels, L. (ed.) Design Patterns in Fluid Construction Grammar. John Benjamins, Amsterdam (2011)
[34] van Trijp, R.: Grammaticalization and semantic maps: Evidence from artificial language evolution. Linguistic Discovery 8(1), 310–326 (2010)
[35] van Trijp, R.: A design pattern for argument structure constructions. In: Steels, L. (ed.) Design Patterns in Fluid Construction Grammar. John Benjamins, Amsterdam (2011)
[36] van Trijp, R.: The emergence of case marking systems for marking event structure. In: Steels, L. (ed.) Experiments in Cultural Language Evolution. John Benjamins, Amsterdam (2012)
[37] Wellens, P.: Organizing constructions in networks. In: Steels, L. (ed.) Design Patterns in Fluid Construction Grammar. John Benjamins, Amsterdam (2011)