PLANNING WHAT TO SAY: Second Order Semantics for Fluid Construction Grammars.

Luc Steels[1,2] and Joris Bleys[1]

[1] Vrije Universiteit Brussel, Belgium,
[2] Sony Computer Science Laboratory, Paris, France
{steels, jorisb}@arti.vub.ac.be

Abstract. Research in the origins and evolution of language has now reached a level where languages with grammatical structures are emerging in computer simulations and robotic experiments based on situated embodied language games played by populations of agents. This paper focuses on some of the technical AI issues related to this research. Specifically, we report on a system for planning complex meanings (IRL) and on their grammatical expression in Fluid Construction Grammar.

1 Introduction

There have been a number of very important advances lately concerned with modeling the emergence of communication systems with natural language like properties in artificial agents (see recent overviews in [17], [14]). The goal of this research is partly to understand how the evolution of human language has been possible, but also to create new technologies that allow agents to self-organise their own communication systems [12]. Research is not only studying how communication conventions might emerge but also how the conceptual repertoires that underly the meanings being expressed could originate and get coordinated among the agents.

In the work of our group, we have focused first on the emergence of lexicons, initially just for naming individual objects by playing Naming Games [8]. Then we studied the emergence of perceptually grounded categories and how they could be expressed by single words [10], using Guessing Games in which the hearer has to guess the topic chosen by the speaker. These experiments not only showed that lexical convergence could be orchestrated quite effectively, but also that agents could develop their own conceptual repertoires and coordinate them (see [16] for a systematic study). The next step in our research considered combinations of categories and hence the potential for using multiple-word utterances [11], but still without syntax. The experiments showed that even for a large group of agents interacting with an open-ended physical world through vision, it was possible to self-organise a repertoire of concepts and words expressing these concepts.

More recently we started to focus on how grammar might emerge. After some preliminary investigations [10] we focused first on the development of a new formalism for representing grammars, called Fluid Construction Grammar (FCG) [13]. This formalism attempts to capture a new trend in linguistics towards construction grammars ([4], [7]) and is generally in the line of unification-based feature structure grammars. But it has some characteristics which make it suited for experiments in the emergence of grammar, specifically bidirectional rule application for parsing and production, fluidity in rule application, and competition between rules for dominance in the population. Some more technical papers describing this formalism, focusing in particular on the syntax-semantics interface [15] and on hierarchy [2], have recently been published and form a background for the present paper. Here we focus on the problem of a richer semantics which is more realistic compared to human languages and also necessary for achieving the emergence of more complex grammatical language.

In almost all earlier work on the emergence of language, meanings are either conjunctions of propositions or conjunctions of clauses with arguments. However natural languages can clearly express much more powerful semantic structures, including quantification of variables and second order predicates that modulate other predicates, as abundantly shown in Montague grammar [3]. For example, an adverb like "very" in "very big" modulates the meaning of "big".

Given that we are interested in grounded language use, in the sense that meanings are to be conceptualised or interpreted by robots in terms of their own sensori-motor perception of the world, we also need to achieve a procedural semantics: the meanings are viewed as programs that have an effective on the mental state of the listener [18]. Thus, the 'meaning' of the phrase "the big block" will be construed as a program that filters the objects in the perceived world scene by retaining those that are similar to the prototype of a block, then filtering out the remaining objects by taking those whose size is greater than the average size, and then (assuming only a singleton is left) take the unique element out of the resulting set. We find in this semantic program a series of components, which have various associated slots that are filled by values. For example, one primitive component is the operation of filtering a set of objects to retain those similar to a prototype and it has slots for the source set (to be filtered), the target set (after filtering), and the prototype. Some of these slots are called arguments because they are supplied by the speaker. For example in the phrase "the block" the speaker supplies the prototype "block". Although natural languages typically make it possible to explicitly convey arguments, they do not specify what kind of operations have to be invoked. This information is conveyed by the grammar. Thus the syntactic structure of a noun phrase like "the block" suggests that the context should be filtered with the prototype "block" and this will result in a unique referent.

In summary, the meaning of an utterance will be viewed as a (semantic) program, conceptualisation will be viewed as a problem in automatic programming, and interpretation as the application of a semantic program to the real world data resulting from perception. Grammar hints at what semantic programs are to be invoked, and the lexicon introduces arguments to these programs. The remaining sections of this paper provide specific technical details how we are implementing this vision.

2 The Constraint Language IRL

The first thing we need is a programming language which is adequate to act as the internal representation language for formulating meanings as semantic programs. This programming language should be flexible with respect to control flow, because we know from natural language that often information to find the interpretation of an utterance comes in a non-systematic order. For example, in "the ball which is rolling towards the green block at the edge of the table", the referent of "ball" can only be determined after the last word has been heard, as there might be two rolling balls and two green blocks. This suggests strongly that the required programming language should be a constraint language [5].

Each agent a maintains a world model W_a which contains a set of facts, represented in the standard logic-based way with predicates and arguments. The world model is directly derived from perception or through additional inference. We assume next a set of components C_a which each handle particular constraints over the objects of this world model and over temporary objects (such as the chosen topic, a set of objects in the context, subsets of objects, combinations of predicates, etc.). These objects are all typed. Each agent also maintains a set of knowledge items K_a which are additional objects needed by the components, such as a set of prototypes, a set of comparators for comparing objects, etc. These knowledge items are also typed and the activation of a component may lead to the expansion of the set of knowledge items maintained by the agent.

2.1 Definition of Components

A component has a call pattern with a set of arguments written as follows:

```
(<constraint> <arg-1> ... <arg-n>)
```

Each argument consists of a typed variable and refers to a slot used by the component. Slots may be filled before the constraint is invoked or bound as a side effect of the invocation of the component. The first argument is known as the result slot because it is logically speaking the result, however components are viewed as constraints, which means that they may just as well find fillers for other slots when the result slot is already bound.

There are three outcomes when a component is activated for a particular set of slots (bound or not): (1) Satisfied: means that all slots are filled and the constraint handled by the component could be satisfied. (2) Suspended: means that some of the slots are not filled and there is not enough information to further the computation. (3) Failure: means that there is a constraint violation.

Here is an example of a constraint: COMPARE-AVERAGE, which is used to handle the semantics of adjectives like 'big' or 'quick'. It has three *slots*: `filtered-set` (of type object-set), `source-set` (of type object-set) and `comparison` (of type comparison). A comparison contains a target-dimension and a relation, for example size and greater-than. COMPARE-AVERAGE will take such a comparator and retain from the `source-set` only those that satisfy the relation for the given dimension compared to the average, for example all the objects whose size is greater than the average.

Implementing a primitive constraint amounts to defining what should happen for each constellation of slot fillings. For COMPARE-AVERAGE, this is the following:

– **slots `source-set` and `comparison` have a filler** and **slot `filtered-set` has not**: This is the standard case in which the elements of the filler for `filtered-set` are computed by averaging over all elements of the filler for `source-set` in the *target-dimension*. Only the elements which obey the *relation* are retained and stored in the `filtered-set` slot.
– **slots `filtered-set` and `source-set` have a filler and slot `comparison` has not**: This is a case where the constraint should try to find possible comparisons that could perform the right sort of filtering. Otherwise, it returns a failure stating it could not find any valid comparison.
– **slot `source-set` has a filler and slots `comparison` and `filtered-set` have not**: There is even less information available here, but a guess can be made by calculating all possible subsets, and test for every combination whether there is a distinctive filler for `comparison`.
– **slots `filtered-set`, `source-set` and `comparison` have a filler**: Here all slots are filled so it can be tested whether the constraint holds. If it does, the component reports 'satisfied', otherwise 'failure'.
– **any other combination of fillers/no-fillers**: The component is suspended and reports that at least a filler for `source-set` is needed to be able to calculate any other fillers.

2.2 Constraint Networks

Individual components may be combined into a constraint network based on sharing slots and such a combination may itself be abstracted into a (non-primitive) component with its own call-pattern:

```
(def-network (<component> <arg-k> ...)
  (<constraint-1> <arg-1> ... <arg-n>)
  (<constraint-2> <arg-1> ... <arg-n>) ...)
```

There is no explicit control flow defined between the different component constraints. To interpret such a complex constraint, the agent cycles through each individual component trying to advance the computation as much as possible but possibly leaving it suspended until other constraints have supplied more information. So the interpretation process follows the principle of data-driven computation familiar from other constraint programming languages [5]. Cycling continues until no more computation could be achieved.

Here is an example of a constraint network called IDENTIFY-OBJECT-3 that evokes three primitive constraints:

```
(def-network (identify-object-3 object comparison)
  (equal-to-context object-set-1)
  (compare-average object-set-2 object-set-1 comparison)
  (unique-element object object-set-2))
```

The calling pattern of this network has itself two slots: `object` and `comparison`. A schematic representation of the network is given in Figure 1. The slots `object-set-1` and `object-set-2` are shared between several components. The fillers for these slots are constrained by all components they are connected to. Slots `comparison` and `object` are constrained by only one component. The above represents the

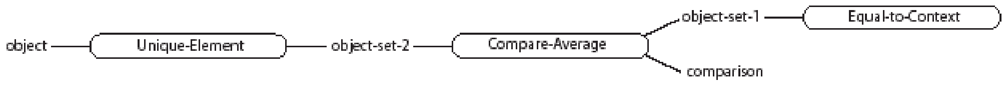

Fig. 1. Schematic representation of a constraint network for a phrase like "the small one".

meaning of a phrase like "the small one". First the object-set with all elements in the context is set as filler for the `object-set-1` slot by EQUAL-TO-CONTEXT. Next COMPARE-AVERAGE fills `object-set-2` with the object-set containing the elements which validate the comparison (in this case the relation should be $<$ and the dimension `size`). Next UNIQUE-ELEMENT checks whether there is only one such element and if so, binds it to `object` (reflected by the word "one").

Constraint networks can be used in multiple directions. Thus if there is a filler for `object-set-1` (because of the context) and if `object` is known (because the agent already knows what object he wants to refer to), then `object-set-2` can be computed. Next COMPARE-AVERAGE will be able to find a possible `comparison` given `object-set-1` and `object-set-2`. This ability to use constraints and constraint networks in multiple ways is important both for the speaker, because he can use the networks backward, and for the hearer, because he can reconstruct the meaning of unknown words by trying to find out what knowledge-items have been used to make certain constraints (signalled by the grammar) work.

The execution of a constraint network often involves the exploration of multiple alternatives. For example, COMPARE-AVERAGE will generate more than one possible comparison given the object-sets involved. We have implemented an efficient search algorithm, similar to those adopted in other logical inference systems [6], which develops multiple hypotheses at the same time by tagging in which 'possible world' a hypothesis is true. The technical details are complex and will be reported in another paper.

3 Configuring Constraint Networks

We now turn to the problem how agents could autonomously develop constraint networks to solve communicative problems. Earlier research on automatic programming (see e.g. [1]) has shown that complex programs can only be derived fast enough if there is a set of powerful building blocks, and if the system progressively develops a library of rich subprograms and templates that are re-used or further extended, possibly aided by heuristics. We have followed the same strategy for planning constraint networks in IRL and introduced a datastructure (the extension tree) that stores the search space for each communicative goal. Rather than starting a new search space every time a new problem needs to be solved, this extension tree is elaborated further. When a solution is found, the solution path is 'chunked' into a new constraint network and it can become another node in the solution tree.

Agents start with a library of primitive constraints (like Compare-average or Unique-element), which are available for being recruited into larger constraint networks that satisfy communicative goals. When no solution components are available, agents begin to develop a search space for that goal by systematically trying to use existing components in a breadth-first manner (as shown in figure 2). The typing constraints associated with the calling pattern of every constraint guide the search process.

For example, to identify an object in the scene, agents may first try the primitive Unique-element, because it yields a unique object, but this component itself requires a set from which to take the unique element. Such a set can be produced by other components, including Compare-Average and Equal-to-Context. Suppose Equal-to-Context fails because there is more than one object in the context, then Compare-Average is tried but, it requires itself yet another source-set, which could again be produced either by Compare-Average and Equal-to-Context. Suppose that by using Equal-to-Context a set of fillers can be found satisfying all constraints, then this is a possible solution, then it can be chunked into a new constraint network (equal to the Identify-Object-3 shown earlier). This network is now itself a possible component that can be used to expand the search space.

Thanks to chunking, the search for a solution becomes progressively more efficient because more complex components are readily available. The chunking process combines all the constraints that have been encountered on the way into a network. Thus Identify-Object-3 combines together the solution path into the network shown in figure 1. The chunking process must also identify what slots should be associated with the calling pattern of the new complex component. These are on the one hand the result slot that formed part of the initial goal, and the knowledge-items that are needed by the various constraints. In the example, this is the case for `object` (which is the result slot) and `comparison`. The other slots in the network can be regarded as internal variables and therefore do not form part of the calling pattern.

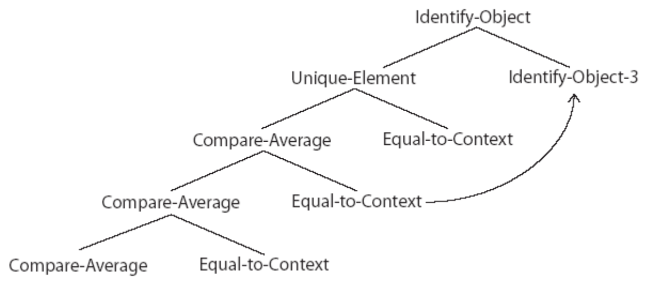

Fig. 2. Extension tree for the goal Identify-object. The left subtree contains the extension tree built by trying out primitives. A solution path has been chunked to form the new component IDENTIFY-OBJECT-3 on the right.

4 Expressing Semantic Programs in FCG

The mechanisms outlined in the previous paragraph, give agents a way to build up a library of complex constraint networks assembled by combining more primitive ones. For each network, there is a calling pattern that contains the result object as well as the knowledge-items that are needed. We now turn to the question how these networks are expressed grammatically. We have adopted a construction grammar framework, where grammar rules consist of two poles: a semantic pole which constrains the semantic applicability of a pattern, and a syntactic pole which constrains the syntactic structure. We use the Fluid Construction Grammar (FCG) framework, described in more technical detail in [15], and [2]. FCG is within the tradition of unification-based feature-structure grammars but features some additional formal properties, specifically a construction in FCG is viewed as a template which is always applicable in two directions: both for parsing and production.

In order to map IRL constraints to FCG constructions, we assume that there is a construction for every component and units for the different arguments in the call-pattern of the component (except the result argument which corresponds to the referent of the top-unit in the construction). The semantic pole of the construction contains the type constraints or stronger constraints on the fillers of each slot. The syntactic pole contains syntactic ingredients that are used to hint which component should be invoked. Knowledge-items appear as explicit objects, similar to Montague grammar [3].

We have only space here to briefly develop one example using English to illustrate the main idea. We focus on the component IDENTIFY-OBJECT-5 which is assumed to have been self-configured by the agent as:

```
(def-network (identify-object-5 object prototype determiner)
  (equal-to-context object-set-1)
  (filter-prototype object-set-2 object-set-1 prototype)
  (retrieve-element object object-set-2 determiner))
```

It should be invoked by a phrase like “the block” where “block” is the prototype needed for filtering the context and “the” determines how the element should be retrieved. This determiner can be ‘unique’ (if the set has a unique element which then has to be retrieved) or ‘random’ (if a random object can be chosen), as in the case of indefinite articles. The FCG construction to express this network similar to the way this is done in English looks as follows:

```
(def-con "identify-object-5-template"
  ((?top-unit
    (meaning (== (identify-object-5 ?object ?prototype ?determiner)))
    (sem-subunits (== ?sub-unit-1 ?sub-unit-2)))
   ((J ?np-unit ?top-unit)
    (referent ?object))
   (?sub-unit-1
    (referent ?prototype)
    (sem-cat (== (prototype ?prototype))))
   (?sub-unit-2
    (referent ?determiner)
    (sem-cat (== (determiner ?determiner)))))
  <-->
  ((?top-unit
    (form (== (precedes ?sub-unit-2 ?sub-unit-1)))
    (syn-subunits (== ?sub-unit-1 ?sub-unit-2)))
   ((J ?np-unit ?top-unit)
    (syn-cat (== (noun-phrase ?np-unit))))
   (?sub-unit-1
    (syn-cat (== (noun ?sub-unit-1))))
   (?sub-unit-2
    (syn-cat (== (article ?sub-unit-2))))))
```

All symbols preceded by question marks are variables. The left pole contains units that have to match (in production) or will be merged (in parsing) to the semantic structure and the right pole contains units that have to match (in parsing) or will be merged (in production) to the syntactic structure. An initial meaning to be expressed could look as follows:

```
((identify-object-5 obj1 proto1 det1) (block proto1) (unique det1))
```

This is decomposed into a number of units, partly by lexical templates that lexicalise ‘(block proto1)’ as ”block” and ‘(unique det1)’ as ”the”. The units appear both on the semantic and syntactic side. `proto1` is categorised as a prototype and `det1` as a determiner by semantic categorisation rules, so that the construction can trigger. In parallel, the syntactic pole merges with the syntactic structure to add the ordering constraint and build the new np-unit.

The semantic and syntactic structure at the end of production are as follows:

```
((top-unit
  (sem-subunits (np-unit)))
 (np-unit
  (referent obj1)
  (meaning
   ((identify-object-5 obj1 proto1 det1)))
  (sem-subunits
   (noun-unit determiner-unit)))
 (noun-unit
  (referent proto1)
  (meaning ((block proto1)))
  (sem-cat ((prototype proto1))))
 (determiner-unit
  (referent det1)
  (meaning ((unique det1)))
  (sem-cat ((determiner det1)))))
```

,

```
((top-unit
  (syn-subunits (np-unit)))
 (np-unit
  (syn-cat ((noun-phrase np-unit)))
  (syn-subunits
   (determiner-unit noun-unit))
  (form
   ((precedence determiner-unit noun-unit))))
 (noun-unit
  (form ((stem noun-unit "block")))
  (syn-cat ((noun noun-unit))))
 (determiner-unit
  (form ((stem determiner-unit "the")))
  (syn-cat ((article determiner-unit)))))
```

Parsing works in an analogous way but now starts from the units that correspond to the words and a top-unit to cover the whole utterance. Then the intermediary noun-phrase units gets created and in parallel the semantic structures, including the 'identify-object-5' component with its corresponding arguments.

At present we have working prototype implementations of all the elements discussed here: IRL, FCG, and the coupling between the two. We have also experimented with learning operators [13], but large-scale experiments with populations of agents are still ongoing.[3]

5 Discussion

This paper provides some technical details related to our ongoing work on the origins and evolution of language. We focused in particular on how second order meanings in the form of IRL constraint networks could be automatically invented by agents and how these meanings could be transformed into language expressions using the FCG framework. Much detail of course could not be reported and many open issues could not be discussed. Nevertheless we hope that the reader has now a better notion in which direction we are pursuing this research.

6 Acknowledgement

This research has been conducted at the Sony Computer Science Laboratory in Paris and the University of Brussels VUB Artificial Intelligence Laboratory. It is partially sponsored by the ECAgents project (FET IST-1940). We are indebted to stimulating discussions about FCG with Benjamin Bergen, Joachim De Beule, and Martin Loetzsch.

[3] More detail and demonstrations of the parsing and production for the examples given here is in http://arti.vub.ac.be/FCG/supporting/irl-fcg, in addition to other examples.

References

1. Barstow, D. (1979) Knowledge-based Program Construction. Elsevier, New York.
2. De Beule, J. and L. Steels (2005) Hierarchy in Fluid Construction Grammar. In: Furbach, U. (eds) (2005) Proceedings of KI-2005. Lecture Notes in AI. Vol 3550 (p. 1-15). Springer-Verlag, Berlin.
3. Dowty, D., R. Wall, and S. Peters (1981) Introduction to Montague Semantics. D. Reidel Pub. Cy., Dordrecht.
4. Goldberg, A.E. 1995. *Constructions.: A Construction Grammar Approach to Argument Structure.* University of Chicago Press, Chicago
5. Marriott, K. and P.J. Stuckey (1998) Programming with Constraints: An Introduction. The MIT Press, Cambridge Ma.
6. Martins, J.P., and Shapiro, S.C. (1983) Reasoning in multiple belief spaces. In Proceedings of the 8th International Joint Conference on Artificial Intelligence (Karlsruhe, W. Germany, Aug.). International Joint Conference on Artificial Intelligence, 1983, pp. 370-372
7. Michaelis, L. 2004. *Entity and Event Coercion in a Symbolic Theory of Syntax* In J.-O. Oestman and M. Fried, (eds.),Construction Grammar(s): Cognitive Grounding and Theoretical Extensions. Constructional Approaches to Language series, volume 2. Amsterdam: Benjamins.
8. Steels, L. (1995) A self-organizing spatial vocabulary. Artificial Life, 2(3):319–332.
9. Steels, L. (1997) Constructing and sharing perceptual distinctions. In M. van Someren and G. Widmer, editors, Proceedings of the European Conference on Machine Learning. Berlin: Springer-Verlag
10. Steels, L. (1998b) The origins of syntax in visually grounded robotic agents.. Artificial Intelligence, 103:1-24 1998.
11. Steels, L., Kaplan, F., McIntyre, A., and Van Looveren, J. (2002) Crucial Factors in the Origins of Word-Meaning. In Alison Wray, editor, The Transition to Language. Oxford: Oxford University Press.
12. Steels, L. (2003) Evolving grounded communication for robots. Trends in Cognitive Science. Volume 7, Issue 7, July 2003 , pp. 308-312.
13. Steels, L. (2004) Constructivist Development of Grounded Construction Grammars Scott, D., Daelemans, W. and Walker M. (eds) (2004) Proceedings Annual Meeting Association for Computational Linguistic Conference. Barcelona. p. 9-1
14. Steels, L. (2005) The Emergence and Evolution of Linguistic Structure: From Lexical to Grammatical Communication Systems. Connection Science 17(3).
15. Steels, L., J. De Beule and N. Neubauer (2005) Linking in Fluid Construction Grammar. Proceedings of BNAIC. Transactions of the Belgian Royal Society of Arts and Sciences. Brussels.
16. Steels, L. and T. Belpaeme (2005) Coordinating Perceptually Grounded Categories through Language. A Case Study for Colour. Behavioral and Brain Sciences, 28(4) 469-524.
17. Wagner, K., J.A. Reggia, J. Uriagereka, G.S. Wilkinson (2003) Progress in the Simulation of Emergent Communication and Language. Adaptive Behavior. Volume 11 (1): 37-69.
18. Winograd, T (1972) Understanding Natural Language. Academic Press, 1972.