Chapter 8
Open-ended Procedural Semantics

Michael Spranger[1,2], Simon Pauw[3], Martin Loetzsch[4], and Luc Steels[1,5]

This paper is the authors' draft and has been officially published as:
M. Spranger, S. Pauw, M. Loetzsch and L. Steels (2012). Open-Ended Procedural Semantics. In Luc Steels and Manfred Hild (Eds.), *Language Grounding in Robot*, 153-172. New York: Springer.

Abstract This chapter introduces the computational infrastructure that is used to bridge the gap between results from sensorimotor processing and language. It consists of a system called Incremental Recruitment Language (IRL) that is able to configure a network of cognitive operations to achieve a particular communicative goal. IRL contains mechanisms for finding such networks, chunking subnetworks for more efficient later reuse, and completing partial networks (as possibly derived from incomplete or only partially understood sentences).

Key words: Incremental Recruitment Language, cognitive semantics, procedural meaning, flexible interpretation, open-ended conceptualization

8.1 Introduction

Research in cognitive semantics (Talmy, 2000) has abundantly shown that human speakers need to conceptualize the world in terms of a rich repertoire of categories, relations, sets, sequences, perspectives, etc., before their meanings can be translated into language. For example, the phrase "the yellow block right of you", which intends to draw the attention to some object in the world, presupposes that the world is categorized using colors (yellow) and prototypical objects (block), that spatial relations are imposed (right of) and a spatial perspective introduced (you). The conceptualizations found in human languages are known to be language-specific. They have to be learned and are changing in a process of cultural evolution, which implies that experiments in artificial language evolution not only have to explain how

[1]Sony Computer Science Laboratory, Paris, e-mail: spranger@csl.sony.fr
[2]Systems Technology Laboratory, Sony Corporation, Tokyo
[3]ILLC, University of Amsterdam, Amsterdam
[4]AI Lab, Vrije Universiteit Brussel, Brussels
[5]ICREA Institute for Evolutionary Biology (UPF-CSIC), Barcelona

lexicons and grammars arise but also how the conceptual building blocks used by a language may arise and propagate in a population.

Cognitive semantics is usually a purely descriptive endeavor, without any formalization and without any computational operationalization (with the exception of early work by Holmqvist, 1993). Moreover, although there is a consensus that cognitive semantics has to be grounded through an embodied sensorimotor system (Lakoff, 1987), it is actually not common to employ cognitive semantics on physically embodied robots. This chapter reports on a computational system that has been developed to fill this gap. The system is called IRL (Incremental Recruitment Language). The meaning of a sentence is captured in terms of a network of cognitive operators that are dynamically assembled for reaching certain communicative goals. These cognitive operators perform operations either directly on the sensorimotor streams or on semantic entities such as sets, sequences, and the like. IRL is implemented in Common LISP and runs on any Common LISP environment. Interfaces have been constructed with Fluid Construction Grammar for mapping to sentences on the one hand and with sensorimotor systems on the other.

The early development of IRL took place at the end of the nineties. A first implementation by Luc Steels was used in experiments in grammar emergence (Steels, 2000). A second implementation was made by Wouter Van den Broeck (Van Den Broeck, 2008). This chapter is based on a new more recent implementation by Martin Loetzsch, Simon Pauw and Michael Spranger (Spranger et al, 2010a). The current implementation has already been used in language game experiments targeting various domains including color (Bleys, 2008), spatial language (Spranger, 2011), quantifiers (Pauw and Hilfery, 2012) and temporal language (Gerasymova and Spranger, 2012, this volume) on different robotic platforms, including the Sony humanoid (Fujita et al, 2005) and the Humboldt MYON (Hild et al, 2012) robot discussed in earlier chapters of this volume.

8.2 Motivating Example

To illustrate the key notions of IRL we use an actual IRL-based robotic experiment. There are two humanoid robots located in an environment that contains colored bricks and artificial landmarks (Spranger et al, 2010b, see Figure 8.1). The robots play a *language game*.

The perceptual systems for recognizing and tracking the objects in their environment are described in an earlier chapter in this volume (Spranger et al, 2012). The world model produced by the vision systems consists of a set of objects that are characterized by *continuous* real-valued features such as color, position and orientation but also width, height and length. Conceptualization must handle the transition from these continuous values to the discrete categorizations used in language.

In order to achieve this, the speaker should try to find a particular set of operations that, when executed by the hearer, will satisfy his communicative goal. Consequently, the meaning of an utterance is a set of *cognitive operations* that the speaker

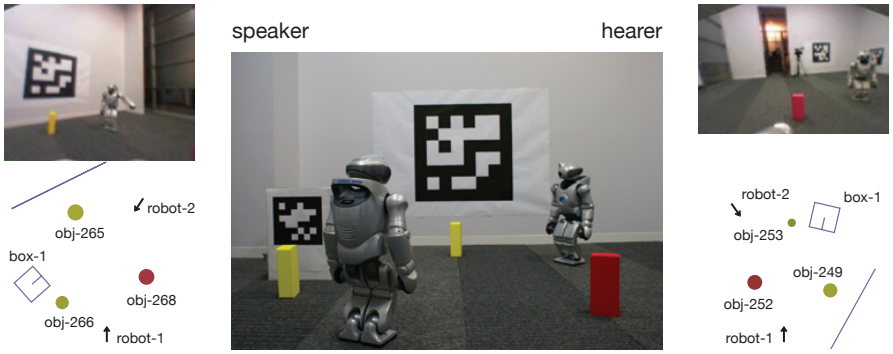

Fig. 8.1 Robots scan the shared environment (center image) with the cameras in their heads (images top left and top right) and construct world models from these data streams (images bottom left and bottom right). The robot at the left has the role of the speaker and tries to draw the attention of the other robot to the red block by means of the utterance "the red block").

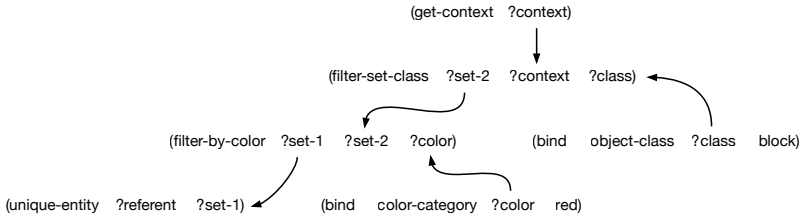

Fig. 8.2 An IRL network representing the meaning of "the red block". When executed by the hearer in the interaction shown in Figure 8.1 (right robot), the variable `?referent` (the referent of the utterance) becomes bound to the object `obj-252`.

wants the hearer to execute. Such an approach to meaning is often referred to as *procedural semantics* (see Winograd, 1971; Johnson-Laird, 1977; Woods, 1981, for original ideas).

More specifically, we assume that an utterance encodes a *network* of cognitive operations as well as the relationships between their arguments. An example network for the utterance "the red block" is shown in Figure 8.2. It includes operations such as filtering the context for blocks (`filter-set-class`) or finding red objects (`filter-by-color`). Every node in the network evokes a cognitive operation, represented by its name and its list of arguments, for example `(filter-set-class ?set-2 ?contex ?class)`, evokes the `filter-set-class` operation. The arguments act as variables or slots that are bound to or will contain specific values. The variables are represented as names preceded by question marks. The same variable can re-occur in different cognitive operations, represented by arrows in the network.

There is a special operation called `bind` which introduces concrete *semantic entities* of a certain type and binds them to variables in the network. Semantic entities

are categories in the conceptual inventory of the agents. For instance, the statement `(bind color-category ?color red)` in the example above binds the color category *red* to the variable `?color`. The color category itself has its own grounded representation. These bind-operations are typically supplied by the lexicon. Information about the connections in the network are typically derived from the syntax of the sentence. For example the first argument of `filter-set-class` is linked to the second argument of `filter-by-color` through the variable `?set-2`, for the phrase "red block".

8.3 Building Blocks of IRL

Cognitive operations and semantic entities loosely resemble an often made distinction in computer science between algorithms and procedures on the one side and data structures on the other side. The IRL core system does not come with any built-in cognitive operations or semantic entities. It instead provides an interface for defining cognitive operations and semantic entities for a particular language game and sensorimotor embodiment.

Cognitive operations

A cognitive operation implements a specific cognitive function or task, for example filtering a set with a color category, picking elements from a set, categorizing an event, performing a spatial perspective transformation, taking the union of two sets, and so on. Here is an example of how an operation for color categorization can be declared in IRL (we will show the full implementation later):

```
(defoperation filter-by-color ((target-set entity-set)
                               (source-set entity-set)
                               (color color-category))
  ;; .. implementation of the operation
  )
```

This operation has the three *arguments* `target-set`, `source-set` and `color`, which are of type: `entity-set`, `entity-set` and `color-category`.

The operation `filter-by-color` can perform a classic categorization by finding those elements in `source-set` that are closest to the `color` category (i.e. by applying a nearest neighbor algorithm) and returning them in `target-set`. For example if the color category is `yellow`, then all yellow objects in `source-set` close to the prototype will end up in `target-set`.

In many ways, cognitive operations behave like functions in the sense that they compute a set of output arguments from a set of input arguments. However, cognitive operations differ from normal functions in that they are *multi-directional*, so that IRL can in fact be seen as a constraint language (as pioneered in early programming language designs of Borning, 1981; Sussman and Steele, 1980; Steels, 1982).

For example the operation `filter-by-color` can also infer a `color` category from a `target-set` of classified objects and a `source-set`. Or it can compute combinations of `color` categories and resulting `target-set` values from a `source-set`. This ability to operate in multiple directions is crucial for flexible conceptualization and interpretation of semantic structures.

When an operation is executed, some of its arguments are *bound* to a value. This value can be any *semantic entity* (see next subsection) with a type that is compatible to the type of the argument specified in the operation. Whether an argument then is input or output of the operation depends on whether it is bound or not. Here is a concrete example implementation for the `filter-by-color` operation:

```
(defoperation filter-by-color ((target-set entity-set)
                               (source-set entity-set)
                               (color color-category))
  ;; Case 1
  ((source-set color => target-set)
   (let ((filtered-set (apply-color-category
                        color source-set
                        (color-categories ontology))))
     (when filtered-set
       (bind (target-set 1.0 filtered-set)))))

  ;; Case 2
  ((target-set source-set => color)
   (loop for category in (all-color-categories ontology)
      when (equal-entity target-set
                         (apply-color-category
                          category source-set
                          (color-categories ontology)))
      do (bind (color 1.0 category))))

  ;; Case 3
  ((source-set => color target-set)
   (loop for category in (color-categories ontology)
      for filtered-set = (apply-color-category
                          category source-set
                          (color-categories ontology))
      when filtered-set
      do (bind (target-set 1.0 filtered-set)
               (color 1.0 category))))

  ;; Case 4
  ((target-set source-set color =>)
   (let ((filtered-set (apply-color-category
                        (color-categories ontology))))
     (equal-entity filtered-set target-set))))
```

The IRL-specific code is underlined. There are four cases, which each implement the behavior of the operation for a different combination of bound/ unbound arguments. Each case starts with a pattern that defines its applicability: when all arguments before the `=>` symbol are bound and all arguments after `=>` unbound, then the code

below the pattern is executed. For example, *Case 1* specifies the operation of the primitive when `source-set` and `color` are bound, but `target-set` is still unbound.

Each case 'returns' values for all its unbound arguments with the `bind` command. For example in the first case, `(bind (target-set 1.0 filtered-set))` assigns the computed value `filtered-set` with a score of `1.0` to the argument `target-set`. An operation can call the `bind` command multiple times and thereby create *multiple hypotheses*. For example in the third case, the operation computes all possible pairs of values for the `color` and `target-set` arguments when only the `source-set` is bound.

It is also possible that an operation does not compute a value for an output argument. For example in the second case above, it can happen that the operation is not able to infer a `color` category which can account for a categorization of `source-set` into `target-set`. The operation will then simply not call the `bind` command, which *invalidates* the values bound to its input arguments. Finally, when all arguments of an operation are bound, then the operation does not bind any values at all but returns information on whether its arguments are *consistent*. In the fourth case, the operation checks whether the `color` category applied to the `source-set` is indeed the same as the given `target-set`.

Semantic entities

The "data" that can be bound to the arguments of cognitive operations are called *semantic entities*. These can be any kind of data representations, including items in the conceptual inventory of an agent (e.g. image schemata, categories, prototypes, relations, roles, etc.), representations of the current context (e.g. the world model, discourse information, etc.), and intermediate data structures that are exchanged between cognitive operations (e.g. sets of things, constructed views on a scene, etc.).

In the example above, a semantic entity of type `color-category` consists of three numeric values that represent a prototypical point in the YC_bC_r color space. The memory of the agent contains several instances of `color-category`, for example `red` is represented by the point in the color space [16, 56, 248]. A semantic entity of type `entity-set` represents a list of objects, which each again contain numerical values computed by the vision system.

The 'meaning' of a semantic entity depends on how it is used in a network of operations. For example the meaning of "red" will be different depending the operation that the semantic entity `red` is used in. In "the red block", the `filter-by-color` operation will return all objects from the set of blocks that are closer to the prototype of red than any another prototype. However, in "the red most block", the prototype red requires another operation. (E.g., `identify-by-extreme-color` which returns the single object that is simply closest to the color, even if it is not red at all but, for example, orange.)

IRL makes no specific assumptions about the nature of semantic entities and it depends on the agent architecture of the application scenario how they are stored in the conceptual inventories of an agent. When existing category systems are not suffi-

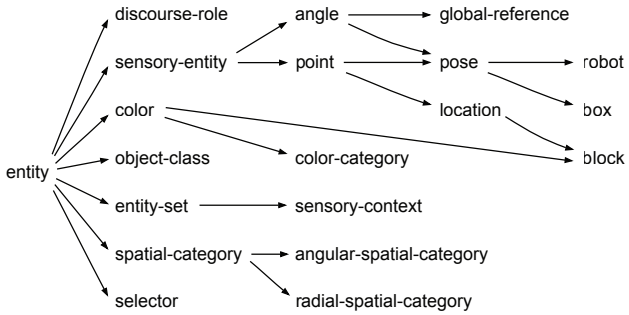

Fig. 8.3 Example of a type hierarchy of semantic entities.

cient, or in order to optimize conceptual inventories, cognitive operations can create new or adapt existing semantic entities. For instance distinctions such as green vs. red, left vs right or even walk vs. run are growing out of the agents interaction with the environment and are constantly shaped and updated by agents based on their use in communication.

Semantic entities are typed, which makes it possible to explicitly model intuitive distinctions between different cognitive representations. Such distinctions could for example be rooted in a perceptual system which already distinguishes between objects and events because they are recognized by different sub-systems. Or it could be the difference between a color category and a discourse role, which clearly are meant to operate in different domains. Furthermore, types can be organized in hierarchies, which allows it to treat entities with a common super-type the same. Technically, type hierarchies are represented using the standard class inheritance system of Lisp (Kiczales et al, 1991), that is new types are defined by creating classes that are directly or indirectly derived from the IRL class `entity`. Using class inheritance additionally allows it to inherit properties from other classes of semantic entities and can be used for software engineering, in particular, designs with reuse. An example of such a type hierarchy is shown in Figure 8.3. It shows semantic entities that were chosen for the examples in this chapter.

Type information is used in IRL in three different ways. First, it constrains what semantic entities can be bound to arguments of cognitive operations: only entities of the same type or of a sub-type of the type of the argument or can be bound to the argument of an operation. Second, it constrains the way in which cognitive operations can be combined in networks (see Section 8.5.1). And third, they can provide a seed for semantic and syntactic categories in the grammar that expresses semantic structures: an distinction on the semantic level between objects and events could be reflected in categories such as noun and verb (see Bleys, 2008; Spranger and Steels, 2012, for experiments in this direction).

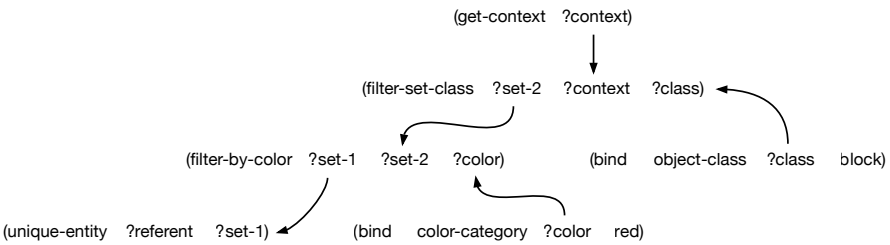

Fig. 8.4 Graphical representation of an IRL network underlying "the red block". The `get-context` operation binds the set of all objects contained in the world model to the variable `?context`. Then `filter-set-class` filters this set for all objects of class `block` and binds the result to `?set-2`. This set is then filtered by `filter-by-color` for objects that match the `red` color category into `?set-1`. Finally, `unique-entity` checks whether `?set-1` contains only one object and binds the result to `?referent`.

8.4 Representation of Compositional Meanings

8.4.1 Networks of Cognitive Operations and Semantic Entities

The IRL network for the utterance "the red block" shown earlier (Figure 8.2) is repeated below as an S-expression. We continue with this example to explain the mechanisms for evaluation and construction of networks.

```
((unique-entity ?referent ?set-1)
 (filter-by-color ?set-1 ?set-2 ?color)
 (bind color-category ?color red)
 (filter-set-class ?set-2 ?context ?class)
 (bind object-class ?class block)
 (get-context ?context))
```

It contains four cognitive operations: `unique-entity`, `filter-by-color`, `filter-set-class` and `get-context`, and two semantic entities: `red` and `block`. The arguments of the operations are connected via *variables* (starting with a `?`). Two or more operations are linked when they share the the same variable. For example in the network above the argument `target-set` of the `filter-set-class` operation is connected to the `source-set` argument of `filter-by-color` through the variable `?set-2`.

Semantic entities are introduced in a network with *bind statements* (starting with the `bind` symbol) and they are also linked to cognitive operations through variables. For example `(bind color-category ?color red)` binds the `red` color category to the `color` argument of `filter-by-color` via the `?color` variable. The first parameter of the bind statement (here: `color-category`) declares the type of the semantic entity, which is information needed for interfacing with language processing.

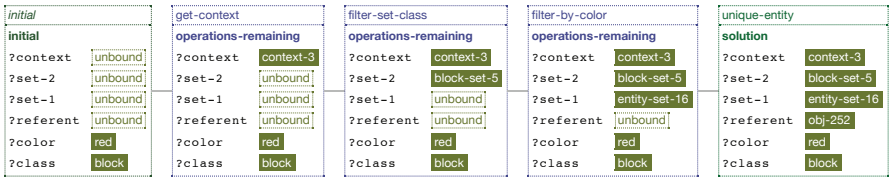

Fig. 8.5 Example of an execution process. The network from Figure 8.4 is executed by the hearer in the interaction of Figure 8.1 (right robot). From left to right, each node represents a step in the execution process. From top to bottom, the executed operation, the node status, and the current list of bindings of each node are shown. A consistent solution with bindings for all variables is found in the last node, and the value `obj-252` is indeed a unique red block (compare Figure 8.1).

Figure 8.4 (repeated from Figure 8.2) shows the graphical representation of the network, with the links between operations and bind statements are drawn as arrows. Although the arrows suggest directionality, they only represent a 'default' direction of execution, which could be different from the actual data flow in the network. Furthermore, the order of operations and bind statements in a network is not meaningful at all. It is only important how operations and semantic entities are linked. Two networks are equivalent when both have the same set of operations and bind statements and when the structure of the links between them is the same.

8.4.2 Execution of IRL Networks

Which particular red block in the example above is referred to, i.e. which object is bound to the variable `?referent`, is found by *executing* the network within the current context of the interaction. Execution is the process by which values are bound to the variables in the network. A set of variable-value bindings is considered a *solution*, if it is *complete* and *consistent*. Complete means that all variables are bound. A set of bindings is consistent if all operations in the network have been executed.

The execution process starts by executing the bind statements to yield a list of initial bindings. The semantic entities expressed in bind statements are retrieved via their id and bound to the respective variables in the network. All other variables are assigned an empty value (unbound). As shown in the leftmost node of Figure 8.5, the initial bindings for the execution of our example network map the semantic entity `red` to the `?color` variable and `block` to `?class`, with the rest of the variables remaining unbound.

Execution of the network proceeds by executing all cognitive operations in the network. In each step, a random operation is picked from the list of not yet executed operations and it is checked whether the operation can be executed given the current set of bindings for its arguments, i.e. whether it has implemented a case for that particular combination of bound and unbound arguments. If such a case exists, then

the operation is executed (see Section 8.3) and newly established bindings are added to the list of bindings. If not, then another operation is tried. A consequence of this procedure is that the particular order in which operations are executed, the control flow, can not be determined by the structure of a network alone. Rather, IRL execution is data-flow driven and execution order depends on how data spreads between cognitive operations.

In the example of Figure 8.5, the only operation that can be executed given the initial bindings is `get-context` (it doesn't require bound input arguments) and it introduces the entity set `context-3` as a value for `?context`. Then `filter-set-class` can be run, and so on. Each added binding enables the execution of more operations, until the `unique-entity` adds a binding for the last remaining unbound variable `?referent`. The set of bindings in the right-most node of Figure 8.5 is a consistent solution of the execution process, because all operations in the network have been successfully executed and all variables are bound.

Of course there can be also other outcomes of executing operations than in the example above (see Section 8.3). First, it can happen that an operation returns multiple bindings for its unbound arguments. IRL will then add each hypothesis to a copy of the current bindings list and then further process these lists in parallel. Second, when all arguments of an operation are bound, then its execution amounts to a verification or checking of consistency. If that fails, then the complete set of bindings is invalidated and not further processed. And third, when an operation is not able to bind a value for an unbound argument, then the whole bindings set is also invalidated.

To illustrate this, we will now look at the execution of a second network. It has the same operations and the same connections between them as the previous example, but does not contain bind statements for `?color` and `?class`. Instead, the `?referent` variable is bound to object `obj-268` (the red block in the world model of the speaker, see Figure 8.1):

Example 1.

```
((bind sensory-entity ?referent obj-268)
 (unique-entity ?referent ?set-1)
 (filter-by-color ?set-1 ?set-2 ?color)
 (filter-set-class ?set-2 ?context ?class)
 (get-context ?context))
```

It is unlikely that such a semantic structure will be the result of parsing an actual utterance, but as we will see in the next section, the execution of such networks is heavily used in conceptualization to try out constructed networks. The execution process for this network in the world model of the speaker in Figure 8.1 is shown in Figure 8.6. Execution again starts with the `get-context` operation, but then another case of `filter-set-class` is executed: because both its `source-set` and `class` arguments are unbound, the operation creates combinations of object classes and resulting filtered sets, which leads to a branching of the execution process. The first two of these branches (Figure 8.6 top) immediately become invalidated, because `filter-by-color` cannot apply color categories to boxes and

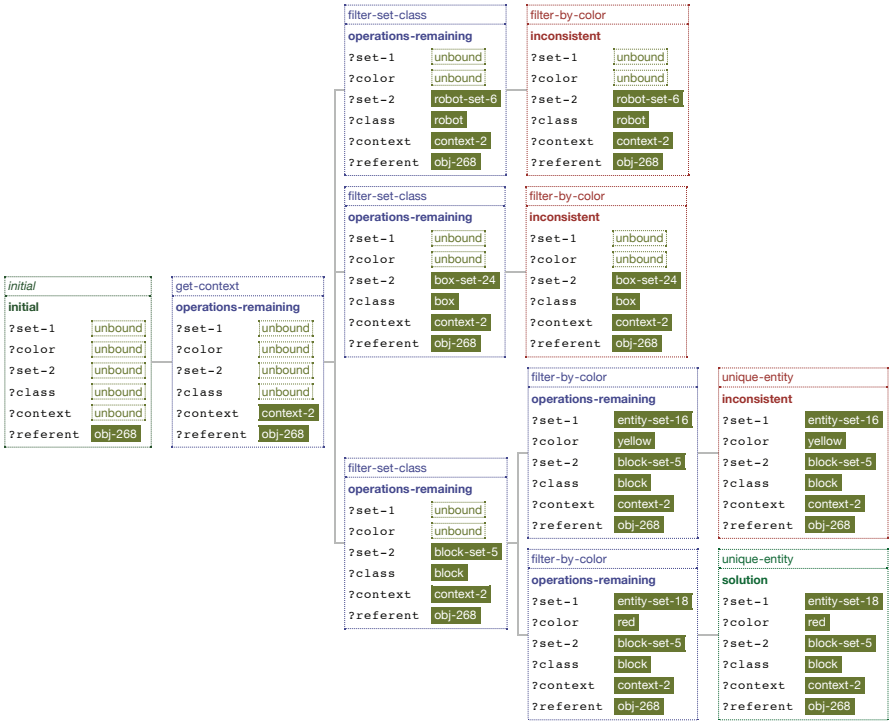

Fig. 8.6 Example of an execution process with parallel processing of multiple hypotheses.

robots. The third case, however, is further branched by `filter-by-color`, because the set `block-set-5` bound to `set-2` contains both yellow and red objects. The first of these two hypotheses is then invalidated by `unique-entity`, because `entity-set-16` contains more than one object. A consistent solution is then found with the node at the bottom right of Figure 8.6.

8.5 Conceptualization and Interpretation

We have seen how compositional semantics are represented and executed in IRL and will now turn to the use of these mechanisms in communicative interactions, i.e. how meanings are constructed and interpreted and how underspecified semantic structures can be completed.

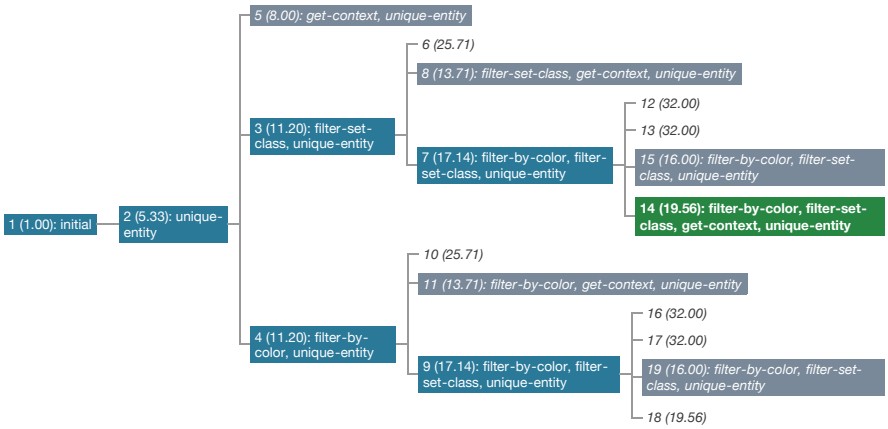

Fig. 8.7 Example of a search process involved in the construction of an IRL program. Analogous to previous examples, the goal for this conceptualization process is to find a program that can identify the red block in the scene of Figure 8.1. Each node represents one processing step and branches in the tree indicate multiple possibilities for expansion. Node labels show the order in which nodes were created, a score that determines which node should be expanded next, and a list of the cognitive operations that have been incorporated into the network so far. Starting from an empty network (node 1), cognitive operations are recursively added and the resulting programs are tried out (nodes 2-3, 7, 9), until finally a solution is found that can achieve the goal of identifying the red block (node 14). By then, some nodes have not been tested yet (nodes 6,10, 12, 13, 16-18) and some can not be further expanded (nodes 5, 8, 11, 15, 19).

8.5.1 Conceptualization

For structured procedural meanings such as IRL programs, conceptualization is the process of constructing a network that, when executed, can achieve a specific communicative goal. For instance, the communicative goal can be to discriminate `obj-268` in Figure 8.1 (i.e. the red block). This goal can be achieved by the following network:

Example 2.

```
((unique-entity ?referent ?set-1)
 (filter-by-color ?set-1 ?set-2 ?color-prototype)
 (filter-set-class ?set-2 ?context ?class)
 (get-context ?context)
 (bind object-class ?class block)
 (bind color-category ?color-prototype red))
```

The mechanism that takes care of finding such a network is called the *composer*. The composer is implemented as a standard best first search algorithm. Starting from an initial (usually empty) network, cognitive operations are recursively added and linked until a useful network is found. Moreover, the composer can also use complete or incomplete networks in the process of composition.

An example of such a composition search process is shown in Figure 8.7. Each node in the search tree contains an (intermediate) IRL program together with a *target variable* and a set of *open variables* and a number indicating the *cost* of that node.

The target variable of the chunk in composition is the variable that is linked to the first slot of the first operation that is added by the composer (thus there is always only one target variable per network). Open variables are all other variables in the network that don't link cognitive operations. Additionally, the types of the slots of cognitive operations that are connected to target variables and open variables are also stored with the network. The cost of a node is used to determine which node to expand next. The one with the lowest cost is the first to be expanded. Example 3 shows the internal network of node '4' in Figure 8.7:

Example 3. Node 4

```
((unique-entity ?topic ?set-1)
 (filter-by-color ?set-1 ?set-2 ?color))
```

The target variable is `?topic` (of type `sensory-entity`) and the open variables are `?set-2` (type `object-set`) and `?color` (type `color-category`).

The search process starts from an initial node. The content of the initial node depends on the communicative goal but should always contain at least one open variable. In our example the first node contains nothing but the open variable `?topic`.

Every iteration of the search procedure consist of two phases. In a first phase the composer checks if the current networks can achieve the communicative goal. For this, the conceptualizing speaker takes itself as a model for the hearer and executes the program using his own set of categories and his own perception of the world. This is a form of re-entrance (Steels, 2003). If one of the current networks can achieve the communicative goal then the composer is done and the solution is returned. The execution of a network can generate additional bindings. These additional bindings become part of the solution. Most of the time the networks will not provide a solution.

If no solution has been found yet, the composer tries to *extend* the network of the node with the lowest cost. The composer tries to add a cognitive operation to the existing network and links the target slot of the cognitive operation to one of the open variables of the node. This variable can only be linked if its type is compatible with the type of the target slot. For each possible extension, a child node is created with the extended network, the now connected variable is removed from the list of open variables and new open variables for the other slots of the added operation are created.

A solution of the conceptualization process is found when the execution of a node's network results in a set of bindings. The processing of nodes stops and the found program together with the bindings from execution is returned. However, often there is more then one solution and sometimes the first solution found is not

the best solution. Therefore it is possible to ask the composition engine for multiple (or even all) solutions up to a certain search depth.

We will turn to an example to illustrate the expansion of a node. In Example 3, the open variable `?set-2` (of type `object-set`) of node '4' can be connected to the three different operations `filter-set-class`, `get-context` and `filter-by-color` because their target slot is of type `object-set` (which is the same as and thus compatible with the type of `?set-2`). Consequently, three child nodes are created for the three resulting networks (nodes 9-11). Node 9 contains the following expansion:

Example 4. Node 9

```
((unique-entity ?topic ?set-1)
 (filter-by-color ?set-1 ?set-2 ?color))
 (filter-set-class ?set-2 ?set-3 ?class))
```

Its open variables are `?color` (of type `color-category`), `?set-3` (type `object-set`) and `?class` (type `object-class`). The expansion of node '4' removed `?set-2` from the list of open variables but added `?set-3` and `?class`. This network does can not yet compute the topic. In order for the operation `filter-set-class` to compute something it requires a value for `?set-3`. But, `?set-3` is still an open variable. However, a further expansion of node 9 into node 19 does give a solution:

Example 5. Node 19

```
((unique-entity ?topic ?set-1)
 (filter-by-color ?set-1 ?set-2 ?color))
 (filter-set-class ?set-2 ?set-3 ?class))
 (get-context ?set-3))
```

For the topic of this example (the red ball – `obj-268` in Figure 8.1). IRL finds an unambiguous set of bindings containing the values `red` and `block` for the variables `?color` and `?class` respectively, which was already hinted at in Example 2.

The composition process of IRL is highly customizable to the specific needs of particular learning scenarios. Most importantly, the order in which nodes are processed can be influenced by providing a function that ranks them depending on the composed program and their depth in the search tree. Nodes with a lower rank will be processed first (see the second number in the node representations in Figure 8.7). By default, networks with a low depth in the tree, few duplicate cognitive operations and a smaller number of open variables are preferred, resulting in a 'best first' search strategy. But this scoring mechanisms can also be used to implement depth-first or breadth-first searches.

8.5.2 Relation to Language

There are two basic ideas for mapping IRL networks to language: 1) Lexical constructions typically encode bind statements using words, and 2) grammatical constructions typically convey which cognitive operations are used in the meaning and

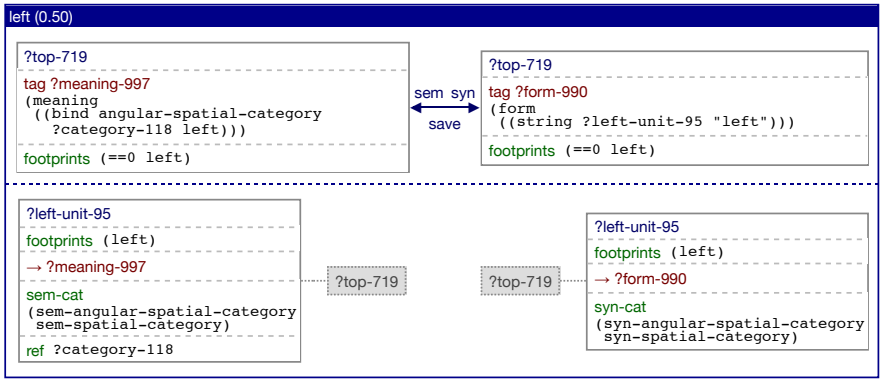

Fig. 8.8 Example of a lexical construction in Fluid Construction Grammar. This construction maps the word 'left' to the appropriate bind statement. Application of many such rules including grammatical ones leads to a network, that can be interpreted by IRL.

how they are linked by applying constraints on grammatical relations, e.g. word order.

Constructions provide bidirectional mapping between semantic structure and language. Figure 8.8 shows a lexical construction which maps the bind statement `(bindangular-spatial-category?category-118left)` to the word "left". In production, when such a bind statement is in the meaning, the construction applies and provides the string. In parsing, the process is reversed. Upon observing the string "left" the construction provides the part of the meaning that is the bind statement. Many of such constructions typically apply in production and parsing. Progressively building syntactic and semantic structure.

More elaborate examples how to map IRL to FCG and syntactic knowledge can be found in (Gerasymova and Spranger, 2012) for temporal language, in (Spranger and Loetzsch, 2011) for spatial language and in (Bleys, 2008) for color.

8.5.3 Flexible Interpretation of Partial Plans

One of the most essential parts of the IRL framework is the flexible interpretation. The flexible interpretation mechanism ensures that IRL is robust with respect to missing lexical items or grammatical constructions. The extend to which IRL can be robust depends, of course, on how much information is present in the context and the utterance.

The role of language in IRL is to transfer the IRL network from the speaker to the hearer. When the message (the network) can not perfectly be coded by the utterance, the hearer winds up with an incomplete network. It can be that some

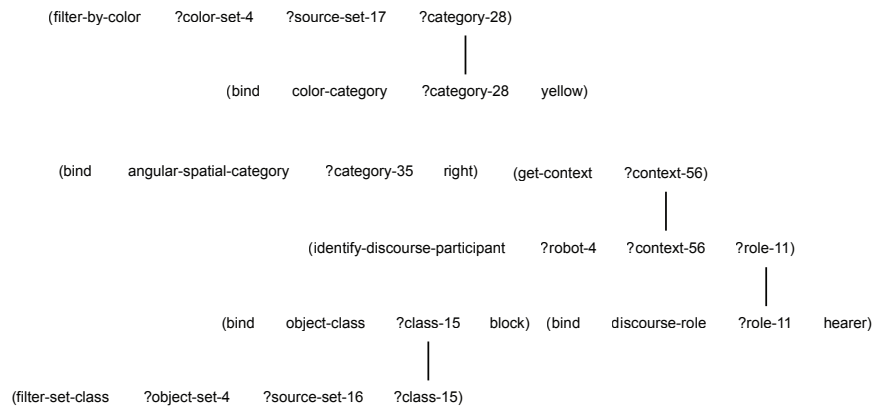

Fig. 8.9 A partial network for "block yellow you".

bind-statements are missing, some of the cognitive operations are missing or that the links between cognitive operations are underspecified. To deal with these forms of missing information IRL does not simply execute a network that comes from the language system, but *searches* for a network that *matches* the information from the language system and at the same time can be successfully executed. The search process is the same as the one that is used for composition: It looks for a network to fulfill a specific communicative goal. For the hearer, however, the information guiding the search process is different. The hearer is constrained in his interpretation by the information decoded from the utterance and the context.

Let us consider an example. Two agents are communicating and the hearer hears the speaker say the following phrase: “grrgh yellow block right krkks you ”. When the hearer knows English, this utterance has some recognizable elements, like “block”, “yellow”, “you” and “right”, but misses “the”, and “of”. Nevertheless the language system may still retrieve some of the intended network, such as the one in Figure 8.9. Executing this network leads to no result (solution). However, the hearer can actively reconstruct possible meanings using the composer. If an IRL network can be found that *matches* with the meaning obtained so far, than it is a possible interpretation of the phrase and if it can be properly executed, it is considered a solution.

To understand this process, recall that, during parsing, language constructions add meaning to the overall interpretation in the form of 1) bind statements, 2) cognitive operations, and 3) variable links. A solution in interpretation found by the composer can include additional information, but must preserve these three aspects from the parsed meaning. Consequently, the composer has to find a network that contains at least the cognitive operations and the variable links of the meaning. In addition, the open variables have to match the bind statements of the meaning. These intuitions are captured by the following definition:

- A meaning n **trivially matches** an IRL network c iff (1) for each bind statement `(bind type ?variable entity)` in n there is a open variable `(?variable . type)` in c and (2) every primitive p in n is in c.
- A meaning n **matches** IRL network c iff there is a function f from the variables in n to the variables in c such that $n' = f(n)$ trivially matches c.

where $f(n)$ is the meaning n' that is the result of substituting every variable x in n for $f(x)$.

For example, the parsed meaning for the utterance "block" is `(bind object-class ?class block)`. This matches Network 6, but not Network 7. This is because the open variable `?class` in Network 6 is of type `object-class` which matches the object class of the bind statement. The type of the open variable `?color` in 7 is not correct.

Example 6.

```
((unique-entity ?referent ?set-1)
 (filter-set-class ?set-1 ?context ?class)
 (get-context ?context))
```

Example 7.

```
((unique-entity ?referent ?set-1)
 (filter-by-color ?set-1 ?set-2 ?color)
 (get-context ?context))
```

8.6 Open-Ended Adaptation of Semantic Structure

Search in conceptualization and interpretation is costly (namely, $O(|P|^l)$ with l the size of the network and $|P|$ the amount of primitives to consider). For real, continuous interaction many thousands of the search trees need to be build. Therefore, it becomes quite quickly unfeasible to build large networks from scratch all the time. So a solution is needed to keep complexity in check.

Our solution is based on the observation that humans clearly have conventionalized ways of construing semantics. An English speaker is more likely to say "the ball" then "the red" even if the color of the object is more salient than its shape in a certain context. Keeping track of such conventions in IRL drastically optimizes the search process.

Conventionalization in IRL is done by storing IRL networks that have proven to be successful in communication. The successful programs are encapsulated as *chunks* also called "stereotype plans" or "schematized operations". A chunk contains an *IRL network* (the stereotype plan), *open variables* and a *target variable*. The open variables and the target variables are variables that occur only once in the network. Given values for the open variables, the chunk computes a value for the target variable. Consider the IRL network in Example 8.

Example 8.

```
((get-context ?context)
 (unique-entity ?referent ?set-1)
 (filter-by-color ?set-1 ?set-2 ?color))
 (filter-set-class ?set-2 ?set-3 ?class)
```

In an experiment it can happen that the combination of the `filter-set-class`-primitive and the `filter-by-color`-primitive is a particularly successful combination. In this case IRL can chunk this part of the network.

Example 9.

```
((filter-set-class ?internal ?source ?class)
 (filter-by-color ?target ?internal ?color))
```

The chunking mechanism in IRL automatically determines the open variables by looking at which variables only occur once in the network and the target variable by taking the open variable that occurs as the first argument of a cognitive operation. The type of the open variables and the target variable are then determined by looking at the type from the primitive which the variable occurs. In the network in Example 9 the target variable is `?target-set` and the open variables are `?class`, `?color-category` and `?source-set`.

This chunk can now be used as if it where a cognitive operation. It gets a name and the target variable and open variables as possible arguments. The target variable is conventionally always in front.

Example 10.

```
(colored-object ?target ?source ?class ?color)
```

Furthermore, the chunk is added to the list of operations so that it can be used for future conceptualizations. Chunks reduce the length of the networks that the composer needs to find which has a significant impact on the performance of the composer (for the network length is an exponent in the complexity $O(|P|^l)$).

8.7 Discussion

IRL has been built to support the embodied, multi-agent experiments in language evolution. This paper discusses the mechanisms provided for autonomous conceptualization and interpretation. Namely, a mechanism for the evaluation (or execution) and composition of semantic structure, a mechanism to reconstruct incomplete semantic structure, and a mechanism that can constructs and track conventionalized semantic structure. IRL, thus, provides the needed connection between the sensorimotor systems and the language systems, at the same time allowing for learning and open-ended adaptation.

This paper dealt with general machinery. Further details on how to implement cognitive operations in concrete language game scenarios can be found in the next paper of this volume (Spranger and Pauw, 2012).

Acknowledgements

The research reported here was carried out at the Sony Computer Science Laboratory in Paris and partially funded by the FP7 EU Project ALEAR. The authors are indebted to Joris Bleys for his many experiments in the use of the current IRL implementation.

References

Bleys J (2008) Expressing second order semantics and the emergence of recursion. In: The Evolution of Language: Proceedings of the 7Th International Conference, World Scientific Publishing, pp 34–41

Borning A (1981) The programming language aspects of thinglab, a constraint-oriented simulation laboratory. ACM Trans Program Lang Syst 3(4):353–387

Fujita M, Sabe K, Kuroki Y, Ishida T, Doi TT (2005) Sdr-4x ii: A small humanoid as an entertainer in home environment. In: Dario P, Chatila R (eds) Robotics Research, Springer Tracts in Advanced Robotics, Springer, pp 355–364

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

Hild M, Siedel T, Benckendorff C, Thiele C, Spranger M (2012) Myon, a New Humanoid. In: Steels L, Hild M (eds) Language Grounding in Robots, Springer, New York

Holmqvist K (1993) Implementing Cognitive Semantics: Image Schemata, Valence Accommodation and Valence Suggestion for AI and Computational Linguistics. Lund University Cognitive Studies 17, University of Lund, Lund

Johnson-Laird PN (1977) Procedural semantics. Cognition 5(3):189–214

Kiczales G, Des Rivieres J, Bobrow D (1991) The art of the metaobject protocol. The MIT Press

Lakoff G (1987) Women, Fire, and Dangerous Things: What Categories Reveal about the Mind. The University of Chicago Press, Chicago

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

Spranger M (2011) The evolution of grounded spatial language. PhD thesis, Vrije Universiteit Brussels (VUB), Brussels, Belgium

Spranger M, Loetzsch M (2011) Syntactic Indeterminacy and Semantic Ambiguity: A Case Study for German Spatial Phrases. In: Steels L (ed) Design Patterns in Fluid Construction Grammar, John Benjamins, Amsterdam

Spranger M, Pauw S (2012) Dealing with Perceptual Deviation: Vague Semantics for Spatial Language and Quantification. In: Steels L, Hild M (eds) Language Grounding in Robots, Springer, New York

Spranger M, Steels L (2012) Emergent Functional Grammar for Space. In: Steels L (ed) Experiments in Cultural Language Evolution, John Benjamins, Amsterdam

Spranger M, Loetzsch M, Pauw S (2010a) Open-ended Grounded Semantics. In: Coelho H, Studer R, Woolridge M (eds) Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010), IOS Press, Amsterdam, NL, Frontiers in Artificial Intelligence and Applications, vol 215, pp 929–934

Spranger M, Pauw S, Loetzsch M (2010b) Open-ended semantics co-evolving with spatial language. In: Smith ADM, Schouwstra M, de Boer B, Smith K (eds) The Evolution of Language (Evolang 8), World Scientific, Singapore, pp 297–304

Spranger M, Loetzsch M, Steels L (2012) A Perceptual System for Language Game Experiments. In: Steels L, Hild M (eds) Language Grounding in Robots, Springer, New York

Steels L (1982) Constraints as consultants. In: Proceedings of the European Conference on Artificial Intelligence (ECAI), Orsay, France, pp 75–78

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

Steels L (2003) Language re-entrance and the 'inner voice'. Journal of Consciousness Studies 10(4-5):173–185

Sussman G, Steele G (1980) Constraints - a language for expressing almost-hierarchical descriptions. Artif Intell 14(1):1–39

Talmy L (2000) Toward a Cognitive Semantics, Concept Structuring Systems, vol 1. MIT Press, Cambridge, Mass

Van Den Broeck W (2008) Constraint-based compositional semantics. In: DM Smith A, Smith K, Ferrer i Cancho R (eds) Proceedings of the 7th International Conference on the Evolution of Language (EVOLANG 7), World Scientific Publishing, Singapore, pp 338–345

Winograd T (1971) Procedures as a representation for data in a computer program for understanding natural language. PhD thesis, MIT

Woods WA (1981) Procedural semantics as a theory of meaning. In: Joshi AK, Webber BL, Sag IA (eds) Elements of Discourse Understanding, Cambridge University Press, pp 300–334