Discourse Representation by Hypergraphs

Doctoraalscriptie voor de studie Cognitieve Kunstmatige Intelligentie

Sander Bruggink

Begeleider:
Albert Visser

November 6, 2001

Contents

List of Figures

Chapter 1

Introduction

1.1 The Topic of this Thesis

A very important question in such diverse research areas as philosophy, linguistics, psychology and artificial intelligence is how we can determine the meaning of a sentence or discourse. In this thesis, I will take a look at a part of this problem from the perspective of dynamic semantics.

In most linguistic theories, the process of deriving the meaning of a discourse is divided in three stages:

1. A superficial parsing of the language. In linguistics the most commonly used format is syntax trees. In this thesis, however, I will use Context Modification Logic (CML), a dynamic logic designed by Albert Visser. I will use this logic, because it shares a few important properties of natural language: most notably, its formulas are, like natural language, linear strings, and the order of the symbols is quite similar, too.

2. The construction of some conceptually plausible, intermediary representation of the meaning of the discourse. In Montague Grammar, formulas of Intensional Logic (IL) are used, and in Dynamic Semantics the most common form of representation is a Discourse Representation Structure (DRS). I, too, will use DRS's.

3. The derivation of the actual meaning of the sentence from the intermediary representation. In Dynamic Semantics, meaning is represented by a set of assignments, usually combined with a representation of the context.

In this thesis, I will only address the second stage.

Before one can even begin an attempt to find the meaning of a discourse, one has to give a clue what meaning is. Without deluding myself into the belief that I know exactly what meaning is (a convincing definition of meaning will perhaps never be given), I would like to remark the following.

When trying to give an intuitive image of the meaning of a discourse, most people draw graphs. The nodes of the graphs denote objects, and the edges denote the properties, actions, etc. that the objects have or carry out. For example, the meaning of the sentence "Harry met Sally" is intuitively represented by drawing two nodes, and one edge between those nodes with a label 'met'. The fact that graphs are such an intuitive method of representing meaning suggest that they are, at least, a metaphor of how real people store meanings of discourses.

Therefore, I found it very interesting to try to use graphs for dynamic semantics in a more formal way. For this reason, I want to do the second stage of the process above with graph rewriting, a formal computational framework based on graphs. More formal reasons for using graph rewriting are:

- We want to focus on the *process* of determining the meaning of a sentence, which can be very well expressed using a computational framework like graph rewriting.
- Graph rewrite rules operate on a local level, and they do not modify irrelevent parts of the graph. Words and sentences, too, do not modify the parts of the information state that they are not about.
- The duality between nodes and edges resembles the duality between objects and predicates.

I will not pay any attention to negation, implication and universal quantification; although very interesting these are not within the scope of this thesis. So, to summarize, in this thesis, I will formulate an answer to the following research question:

> How can we use techniques from graph rewriting to calculate a DRS for a formula of (the positive fragment of) CML.

The answer to this question is of particular interest to the research area of artificial intelligence. Because of the characteristics of CML, it is supposed to be relatively easy to translate an utterance of natural language into CML. In this research, I make a start in finding a method of deriving (a representation of) meaning that is both intuitively plausible (cf. strong AI) and computationally solvable. So, the results of this research could eventually be used to write a computer program that really 'understands' natural language.

1.2 Research method

To find an answer to the research question, I have first studied the literature on both dynamic semantics and graph rewriting, and I used the outcomes of this preliminary study to design two methods of doing ‘Discourse Representation by Hypergraphs’.

1.3 Prerequisites

I will assume that the reader has some basic logical as well as linguistic skills, and possesses some knowledge of computational techniques and structures. In particular, students who successfully finished the ‘basisdoctoraal’ phase of CKI, should be able to follow this text. Some interest in logic and linguistics will certainly help in obtaining a good understanding of the matter.

1.4 Overview

Below I will give an overview of the chapters to come. In general, we can say that chapters 2 and 3 outline the preliminary notions and formalizations, while chapters 4 and 5 discuss the results of my project.

Chapter 2. In this chapter I will briefly discuss two theories from the field of Dynamic Semantics, viz. Discourse Representation Theory (DRT) and Context Modification Logic (CML).

Chapter 3. In this chapter I will define hypergraphs and give an introduction into (hyper)graph rewriting. In the last section I will discuss graph rewriting with (negative) application condition also, an extension to graph rewriting which will be used in section 5.4

Chapter 4. In this chapter we will design a way of using techniques from graph rewriting for representing discourse. As the basis we will take the language of CML. We will define a stack-based graph to represent discourse. We will then associate to each of the symbols of CML a graph rewrite rule, and apply these rules to an initial graph in sequence.

Chapter 5. In the system developed in the previous chapter we had to do a lot of work: we had to select the graph rewrite rules that we wanted to apply. In this chapter we will build a graph rewrite system which will do this work for us. In section 5.4 we will extend our system to automatically create stacks when needed.

Chapter 6. In this chapter the conclusions of my research are stated, and an answer is given to the question raised in section 1.1. Also, some directions for further research are given.

Now, let's do some work!

Chapter 2

Dynamic Semantics

2.1 Introduction

The meaning of a sentence is traditionally equated with its truth conditions, in the form of a formula in first-order predicate logic, or one of its higher order pendants (see e.g. [Can93]). In the early 80s, however, it was discovered that taking only truth conditions as the basis of meaning introduces some problems, most of which are related to anaphora (see [GS98] for an overview). In response to these problems, a number of theories were formalized, which, together, are called Dynamic Semantics. In Dynamic Semantics, meaning is taken to consist of both information about the world (truth conditions), and information about the current context of the discourse.

In this chapter, I will first give a general account of Dynamic Semantics. Then, we will take a more detailed look at two of the theories within this field: Discourse Representation Theory (DRT), and Context Modification Logic (CML). I will pay a little more attention to the second one, because it is less well-known than DRT.

2.2 Discourse

More traditional linguistic theories often focus on single sentences, and try to formulate truth-conditions for them. Some of those theories yield good results on some intra-sentential phenomena. However, when we try to apply the same theories to pieces of text larger than one sentence, they often make wrong predictions [GS98].

Theories of Dynamic Semantics, on the other hand, usually focus on larger pieces of natural language, called *discourses*. The discourses under consideration typically consist of one or more sentences, that depend on each other through anaphora. Generally, the sentences under consideration in Dynamic Semantics are of a less complex structure, but this is made up for by the better results on inter-sentential phenomena.

2.3 Definites and Indefinites

In traditional linguistics all noun phrases (NP's) are considered to be similar: they refer to one or more distinct objects in the domain of discourse. As a result, it is very difficult to describe the difference between, for example, the NP's “the man” and “a man”. Both refer to exactly one object, which is a man. But, obviously, the two have different meanings.

In the Russellian tradition, which is more or less adopted in Montague Grammar and Categorial Grammar, uttering “The author of Waverley was Scotch.” is equivalent to uttering the following three utterances [Rus96]:

- At least one person wrote Waverley.
- At most one person wrote Waverley.
- Whoever wrote Waverley was Scotch.

Even when this is applied to “The author of Waverley”, it is a bit awkward, but it gets worse when we are applying the same rules to an NP like “the man”. Then “the man” would mean something like “a man, and there exists only one man”. This is of course plain untrue (there certainly exist more than one men), but even if we perform the obvious repairs and restrict ourselves only to the objects in the current context, it is utterly unintuitive. Also, it misses the crucial point: “the man” refers back to a man that was talked about earlier, while the object to which “a man” refers has not necessarily been talked about before.

In Dynamic Semantics there is a principal difference between definite NP's, like “the man”, and indefinite NP's, like “a man”. The basic assumption is: when we hear an indefinite NP, we introduce a new discourse referent, and when we hear a definite NP, we look through the discourse referents previously introduced by an indefinite NP and use the one that best matches the definite NP. This referent is selected in different ways in all theories, but they share the above basic assumption.

2.4 Discourse Representation Theory

Discourse Representation Theory (DRT) is currently the major branch of research within the field of Dynamic Semantics. In DRT, discourses are given graphical representations, so called Discourse Representation Structures (DRS's). A detailed discussion of DRT and DRS's can be found in [KR93]. Here, I will only give a brief overview of DRT.

2.4.1 Discourse Representation Structures

Although DRS's are of a graphical nature, I will first give a 'linear' definition [KR93, Gam91], for two reasons: firstly, it saves room; and secondly, it will make more clear how I will formally treat a DRS in this thesis, namely as a tuple of two sets.

Definition 2.4.1 (DRT-signature). A signature $\mathcal{S}$ for DRT is a structure $\langle \mathcal{P}, \mathsf{arity}, C, X \rangle$, in which $\mathcal{P}$ is a non-empty, finite set of predicate symbols, arity is a function from $\mathcal{P}$ to $\mathbb{N}$, C is a finite set of constants, and X is a non-empty (possibly infinite) set of discourse referents. If $P \in \mathcal{P}$ and $\mathsf{arity}(P) = n$, we will call P an n-ary predicate.

Definition 2.4.2 (DRS). Let $\mathcal{S}$ be a DRT-signature. A term is either a discourse referent $x \in X$ or a constant $c \in C$. Then, an $\mathcal{S}$-DRS can be defined by simultaneous induction:

1. If P is an n-ary predicate symbol, and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is a condition.
2. If t and t' are terms, then $t = t'$ is a condition.
3. If Φ is an $\mathcal{S}$-DRS then $\neg\Phi$ is a condition.
4. If Φ and Ψ are $\mathcal{S}$-DRS's, then $\Phi \vee \Psi$ and $\Phi \rightarrow \Psi$ are conditions.
5. If $x_1, \ldots, x_n$ are discourse referents, and $\varphi_1, \ldots, \varphi_m$ conditions, then $\langle \{x_1, \ldots, x_n\}, \{\varphi_1, \ldots, \varphi_m\} \rangle$ is an $\mathcal{S}$-DRS.
6. Nothing else is either a condition or an $\mathcal{S}$-DRS.

In the rest of this section, the signature is usually left implicit. We will use lowercase roman letters from the latter half of the alphabet for discourse referents, lowercase roman letters from the first half of the alphabet for constants, and uppercase roman letters for predicates.

As was mentioned above, DRS's are usually represented graphically. A DRS is written as a box, with its discourse referents at the top, and the conditions below. Consider, for example, the DRS of the discourse "John owns a car."

$$\langle \{x, y\}, \{x = John, Car(y), Owns(x, y)\} \rangle$$

$$\begin{array}{|l|}\hline x, y \\ \hline x = \mathit{John} \\ \mathit{Car}(y) \\ \mathit{Owns}(x, y) \\ \hline \end{array}$$

Figure 2.1: DRS of “John owns a car.”

$$\begin{array}{|c|}\hline \\ \hline \begin{array}{|l|}\hline x, y \\ \hline \mathit{Farmer}(x) \\ \mathit{Donkey}(y) \\ \mathit{Owns}(x, y) \\ \hline \end{array} \quad \Rightarrow \quad \begin{array}{|l|}\hline \\ \hline \mathit{Beats}(x, y) \\ \hline \end{array} \\ \\ \hline \end{array}$$

Figure 2.2: DRS of “If a farmer owns a donkey, he beats it.”

This DRS would be graphically represented as in figure 2.1. Also more complex DRS’s are easily represented graphically, such as the DRS for the classical donkey sentence “If a farmer owns a donkey, he beats it.”:

$$\langle \emptyset, \{\langle \{x, y\}, \{\mathit{Farmer}(x), \mathit{Donkey}(y), \mathit{Owns}(x, y)\}\rangle \rightarrow \langle \emptyset, \{\mathit{Beats}(x, y)\}\rangle\}\rangle$$

of which the graphical representation is shown in figure 2.2.

2.4.2 Obtaining a DRS

Kamp & Reyle define an algorithm to derive a DRS from a given discourse in English. They assume that the discourse can be easily parsed into a series of linguistic trees. Below I will give a very brief, informal sketch of how this algorithm works, conveniently skipping disjunction, implication and negation. For a full account, refer to [KR93].

The algorithm’s input consists of a series of sentences, parsed into syntactical trees. In the process of the algorithm’s execution, a DRS Φ of the input is built. Initially, Φ is the empty DRS, $\langle \emptyset, \emptyset \rangle$. All the sentences of the input are sequentially processed. For all the NP’s that a sentence contains, a new referent is added to the DRS, and a condition that constrains the referent to ‘hold’ only objects of the specified type. If the NP is definite, an extra condition is added that equates the referent with the referent that best matches the NP (which referent that is, is deliberately kept quite vague).

Let me illustrate this with an example. Consider the discourse “A man sees a dog. It barks.”. When the algorithm encounters the first indefinite NP, “a man”, it adds a new referent to the empty DRS that is a man, as follows:

$$\begin{array}{|l|}\hline x \\ \hline man(x) \\ \hline\end{array}$$

The same happens when the algorithm encounters the second NP, “a dog”, and when the conditions are added we have the following DRS:

$$\begin{array}{|l|}\hline x, y \\ \hline man(x) \\ dog(y) \\ sees(x, y) \\ \hline\end{array}$$

Now in the second sentence, a definite NP is found that refers to an earlier referent. So, a new referent is added, that is equated with the referent to which it refers:

$$\begin{array}{|l|}\hline x, y, z \\ \hline man(x) \\ dog(y) \\ sees(x, y) \\ z = y \\ barks(z) \\ \hline\end{array}$$

Obviously, this DRS is equivalent to the following one:

$$\begin{array}{|l|}\hline x, y \\ \hline man(x) \\ dog(y) \\ sees(x, y) \\ barks(y) \\ \hline\end{array}$$

2.4.3 Assignments and Truth

Given a DRS and a model of the world, we want to determine whether the DRS is true. In order to do this formally, we have to define what a model looks like. We will use the same models as first order predicate logic [Vis01].

Definition 2.4.3 (Model). A model $\mathcal{M}$ is a structure $\langle D, I\rangle$ in which D is a non-empty set of objects (the domain of discourse) and I the interpretation function, such that: for all constants a of the language, $I(a) \in D$; and for all predicates P of the language, $I(P) \subseteq D^{\mathsf{arity}(P)}$.

Before we can see if a DRS is true, we must assign an object in the domain to each discourse referent. This is done by an assignment, a function from discourse referents to the domain. Now we can verify, given an assignment, whether the DRS is true in a model. What we are interested in, is which assignments make the DRS true. First, however, it is convenient to define some notational conventions:

$$[\![t]\!]_{\mathcal{M},g} = \begin{cases} I_{\mathcal{M}}(t) & \text{if } t \text{ is a constant} \\ g(t) & \text{if } t \text{ is a discourse referent} \end{cases}$$

$h[x_1, \ldots, x_n]g$**:** Assignment h differs from assignment g at most in the values it assigns to the discourse referents $x_1, \ldots, x_n$.

Now we proceed by defining the following two notions simultaneously:

$\models_{\mathcal{M},g} \varphi$ Condition φ is true in model $\mathcal{M}$ with respect to assignment g.

$h \models_{\mathcal{M},g} \Phi$ Assignment h is a verifying embedding for DRS Φ in model $\mathcal{M}$ with respect to assignment g.

Definition 2.4.4. Let $\mathcal{M}$ be a model and g be an assignment. Then:

(i) $h \models_{\mathcal{M},g} \langle \{x_1, \ldots, x_m\}, \{\varphi_1, \ldots, \varphi_n\} \rangle$ iff $h[x_1, \ldots, x_m]g$ and $\models_{\mathcal{M},h} \varphi_1 \wedge \ldots \wedge \models_{\mathcal{M},h} \varphi_n$

(ii) $\models_{\mathcal{M},g} P(t_1, \ldots, t_n)$ iff $\langle [\![t_1]\!]_{\mathcal{M},g}, \ldots, [\![t_n]\!]_{\mathcal{M},g} \rangle \in I_{\mathcal{M}}(P)$.
$\models_{\mathcal{M},g} t = t'$ iff $[\![t]\!]_{\mathcal{M},g} = [\![t']\!]_{\mathcal{M},g}$.
$\models_{\mathcal{M},g} \neg\Phi$ iff for no assignment h: $h \models_{\mathcal{M},g} \Phi$.
$\models_{\mathcal{M},g} \Phi \rightarrow \Psi$ iff for each assignment h: if $h \models_{\mathcal{M},g} \Phi$ then there is a k such that $k \models_{\mathcal{M},h} \Psi$.
$\models_{\mathcal{M},g} \Phi \vee \Psi$ iff there is an assignment h such that $h \models_{\mathcal{M},g} \Phi$ or there is an assignment h such that $h \models_{\mathcal{M},g} \Psi$.

A DRS Φ is true (in a model $\mathcal{M}$) with respect to an assignment g iff there exists an assignment h such that $h \models_{\mathcal{M},g} \Phi$. The DRS Φ is true, if there exists an assignment k such that Φ is true with respect to k.

2.5 Context Modification Logic

Context Modification Logic CML is a formal, dynamic logic designed by Albert Visser [Vis01]. It was designed in such a way that the order of the symbols of the formal logic resembles the order of their counterparts in natural language as closely as possible. Thus, parsing a string of natural language into CML is supposed to be relatively easy.

In this section I will briefly discuss the positive fragment of CML, since that is the fragment I will focus on in this thesis. Also, I will not pay much attention to the determiners and, conveniently, consider the determiner and the noun as a whole. The language of CML that I am going to use is an adaption of Visser's example language CML_0 [Vis01]. I will call the modified language CML_1. The most important difference between CML_0 and CML_1 is the following: CML_1 uses different markers for the term and predicate level, where CML_0 uses the same marker. Also, where Visser likes the referents to be typed, I use a typeless version of CML.

2.5.1 Informal Introduction to CML

The syntax of CML is very loose. There is a number of symbols, and a CML-formula consists of an arbitrary sequence of those symbols (hence we will speak of CML-strings in the sequel). Some symbols represent words or groups of words from natural language, and other symbols add structure to the string; they specify where sentences start or end, what parts of the string are terms rather than predicates, which of the terms is the subject of the predicate, etc. .

We use '[' and ']' to denote the beginning and the end of predicates (at any level of a sentence), and '⟨' and '⟩' to denote the beginning and the end of terms. Furthermore, the symbol '*ob*' means that the current term is the object, and '*sub*' that the current term is the subject of the current predicate. This allows us to write the following CML string for the discourse "A man sees a dog."

$$[\langle \text{a-man } \mathit{sub}\rangle\ \mathit{sees}\ \langle \text{a-dog } \mathit{ob}\rangle]$$

When analyzing this discourse, we begin with an empty information state. Then we look at all the symbols of the CML-string one by one, from left to right. Each of the symbols has an effect on the information state, just like in Dynamic Predicate Logic [GS91]. Before we go into what these effects are, we need to know what an information state in CML looks like.

Information states consist of stacks of referents, and assignments that assign objects in the real word to these referents. There are stacks for noun-like words like '`father`', '`dog`' and '`he`', there are stacks for predicate-level arguments (roles) like '`sub`', '`ob`' and '`with`', and there are stacks for term-level arguments like '`val`'.

Now what effect do the different symbols have on the information state? Let's find out by looking at the symbols of the example of sequentially.

The symbol '[' prepares the information state to receive a new predicate. It does this by pushing new referents (to be precise, empty places that can be filled with referents at a later time) to the stacks of the predicate-level arguments (‘sub’, ‘ob’, etc.). Next, the ‘$\langle$’ symbol prepares the information state to receive a new term, by pushing new referents to the term-level stacks (‘val’).

The next symbol is ‘a-man’. It denotes an indefinite NP, so it creates a new referent. We want to be able to refer to this referent later by using either ‘the-man’ or ’he’. So, we push the new referent to the tops of both the ‘man’ and the ‘he’-stacks. And, since the man is the ‘value’ of the term, we identify the referent with the top referent of the ‘val’ stack. Furthermore, it updates the assignments so that all assignments assign to the new referent an object that is a man. The ‘*sub*’ symbol identifies the referent with the top of the ‘sub’ stack.

The next symbol, ‘$\rangle$’, removes the top elements from the term-level stacks. Then, ‘*sees*’ changes the information state in such a way that all assignments assign to the top referent of the subject stack an object that sees the object assigned to the top referent of the object stack.

The next few symbols work the same as the first few, only they push a new referent to the ‘dog’ and ‘it’ stacks and identify it with the ‘ob’ stack rather than the ‘sub’ stack. And we finish the first sentence of our discourse with the ‘]’ symbol, which restores the predicate-level stacks.

Remark. An important thing to note is, that not every CML-string has a meaning. For example, the CML-string,

$$[\textit{sub}\ \langle\text{a-man}\rangle\ \textit{enters}]$$

has no meaning, because the ‘sub’ symbol occurs in a location where there is no sub stack (yet).

2.5.2 Labelled Sets and Information States

The stacks in CML$_1$ are implemented by using labelled sets. Labelled sets are like normal sets, only some of their elements may have a label (two different elements never have the same label, but one element may have more than one label). Formally:

Definition 2.5.1 (Labelled Set). Let a set Σ of labels be given. Then, a labelled Σ-set (ℓ,Σ-set for short) is a triple $\langle V, \mathsf{lab}, X\rangle$, in which $V \subseteq \Sigma$ is a set of labels, X an arbitrary set, and lab a function from V to X. The empty labelled set $\emptyset$ is the triple $\langle \emptyset, \emptyset, \emptyset\rangle$.

In the following Σ will sometimes be left implicit. Note that I will use much of the same notation for normal sets and labelled sets, but it will always be clear what kind of set a symbol represents.

We will assume, without loss of generality, that for all labelled sets $\langle V, \mathsf{lab}, X\rangle$, $X \subseteq \{0, \dots, n\}$ for some $n \in \mathbb{N}$. Then we can use the following style to represent labelled sets: of each element (starting with 0) we write the labels in arbitrary order, between ()'s (if an element has no labels, we write ϵ). Thus, a labelled set $\langle\{a, b, c\}, \mathsf{lab}, \{0, 1, 2\}\rangle$, with $\mathsf{lab}(a) = 0$, $\mathsf{lab}(b) = 0$ and $\mathsf{lab}(c) = 1$, can be represented as $(ab)(c)(\epsilon)$.

The labels of a labelled set are used as an interface to the objects themselves. We can access the objects through the labels, and thus, objects without a label cannot be accessed. For this reason, I have chosen to ignore unlabelled objects: the 'sum' operation just identifies all unlabelled objects with one 'garbage' object.

Definition 2.5.2 (Morphism).

(i) Consider two labelled sets $\chi = \langle V_\chi, \mathsf{lab}_\chi, X_\chi\rangle$ and $\xi = \langle V_\xi, \mathsf{lab}_\xi, X_\xi\rangle$, with $V_\chi \subseteq V_\xi$. Then a morphism φ from χ to ξ is a function from X_χ to X_ξ such that for all $v \in V_\chi$: $\varphi(\mathsf{lab}_\chi(v)) = \mathsf{lab}_\xi(v)$.

(ii) Two labelled sets χ and ξ are isomorphic if there exists a bijective morphism from χ to ξ.

Definition 2.5.3 (Sum). Let χ and ξ be labelled sets. Then η is a sum of χ and ξ iff there exists a morphism φ_χ from χ to η and a morphism φ_ξ from ξ to η, such that for all labelled sets δ for which there exist a morphism ψ_χ from χ to δ and a morphism ψ_ξ from ξ to δ, there exists a unique morphism θ from η to δ such that $\theta \circ \varphi_\chi = \psi_\chi$ and $\theta \circ \varphi_\xi = \psi_\chi$.

Note that if η and δ are both sums of χ and ξ, then they are isomorphic. So, we will write $\eta = \chi \oplus \xi$ if η is a sum of χ and ξ. Calculating the sum of two labelled sets has the effect of disjointly putting all objects together, but identifying those objects that have a label in common.

CML uses labelled sets of referents. In our version of CML, the labels will specify in what stack(s) a referent is (if any), and which places it occupies within those stacks. To define the set of possible labels we need to specify in advance three pairwise disjoint sets, a set ARG of predicate-level argument roles, a set TERM of term-level argument roles, and a set VAR of global variables. We assume that $\{\mathtt{arg}, \mathtt{term}\} \cap \mathsf{VAR} = \emptyset$. In our example we will take:

- $\mathsf{ARG} = \{\mathtt{sub}, \mathtt{ob}\}$
- $\mathsf{TERM} = \{\mathtt{val}\}$
- $\mathsf{VAR} = \{\mathtt{man}, \mathtt{he}, \mathtt{it}, \mathtt{dog}\}$

Now we can define our label space, $\Lambda_{\mathsf{ARG},\mathsf{TERM},\mathsf{VAR}}$, as follows:

- $\Lambda^{\mathrm{a}}_{\mathsf{ARG}} = \{\langle \mathtt{arg}, a, i\rangle \mid a \in \mathsf{ARG}, i \in \mathbb{N}\}$
- $\Lambda^{\mathrm{t}}_{\mathsf{TERM}} = \{\langle \mathtt{term}, t, i\rangle \mid t \in \mathsf{TERM}, i \in \mathbb{N}\}$
- $\Lambda^{\mathrm{v}}_{\mathsf{VAR}} = \{\langle v, \star, i\rangle \mid v \in \mathsf{VAR}, i \in \mathbb{N}\}$
- $\Lambda_{\mathsf{ARG},\mathsf{TERM},\mathsf{VAR}} = \Lambda^{\mathrm{a}}_{\mathsf{ARG}} \cup \Lambda^{\mathrm{t}}_{\mathsf{TERM}} \cup \Lambda^{\mathrm{v}}_{\mathsf{VAR}}$

In the following we will speak of Λ, and leave the parameters VAR, TERM and ARG implicit. We define $\mathsf{VAR}^+ = \mathsf{VAR} \cup \{\mathtt{arg}, \mathtt{term}\}$. Also, it will be convenient later on to introduce the following abbreviations for elements of a stack, which specify both the stack name and the positions within the stack:

- a^i (with $a \in \mathsf{ARG}$) for $\langle \mathtt{arg}, a, i\rangle$.
- t^i (with $t \in \mathsf{TERM}$) for $\langle \mathtt{term}, t, i\rangle$.
- v^i (with $v \in \mathsf{VAR}$) for $\langle v, \star, i\rangle$.

We only employ specific subsets of Λ, viz. the subsets that contain stacks with no 'holes'. To accomplish this, we will often use a function ν that assigns to each of the elements of VAR^+ a natural number, that represents the height of the specific stack. We define:

$$\Lambda^{\nu} = \{\langle s, o, i\rangle \in \Lambda \mid i < \nu(s)\}$$

Such a function to the natural numbers can be seen as a multiset over the elements of VAR^+ (in the rest of this section will confuse functions from VAR^+ to the natural numbers with multisets). A multiset is an in-between between sets and sequences. The order of the elements does not matter, but there can be duplicates within a multiset. A multiset that contains `arg` twice, and `man` once (and nothing else) will be denoted by: $(\mathtt{arg}, \mathtt{arg}, \mathtt{man})$, or $(\mathtt{arg}^{(2)}, \mathtt{man})$. We can add two multisets ν and μ by taking:

$$(\nu + \mu)(v) = \nu(v) + \mu(v)$$

for all v. The multiset that assigns 0 to all elements will be denoted by $\emptyset$.

As I mentioned above, an information state consists of stacks of referents (formally implemented by a labelled set) and a set of assignments. This part of the information state is modelled by a content (later on we will see that the formal definition of an information state consists of some extra information as well). Let the domain of discourse D be fixed in advance. Then we can define assignments and contents as follows:

Definition 2.5.4 (Assignment). Let a labelled set $\xi = \langle V, \mathsf{lab}, X\rangle$ be given. Then a ξ-assignment is a total function from X to D (the assignment assigns an object to each referent).

Definition 2.5.5 (Content). A content is a tuple $\langle \xi, F\rangle$ such that ξ is a labelled set and F is a set of ξ-assignments.

The definition of the sum of two contents is quite straightforward. We put everything together, but we ignore the 'assignments' that are not assignments anymore after the union (i.e. they assign multiple objects to a single referent). Formally:

Definition 2.5.6 (Sum of Contents). Let $S = \langle \xi, F\rangle$ and $T = \langle \chi, G\rangle$ be contents. Then the sum of S and T (written: $S \oplus T$) is the content $\langle \eta, H\rangle$ such that:

- $\eta = \xi \oplus \chi$
- Let φ_S be the morphism from ξ to η and φ_T be the morphism from χ to η (see definition 2.5.3). We define the following auxiliary operation on assignments $f \in F$ and $g \in G$:

  $$f \star g = \{\langle \varphi_S(x), f(x)\rangle \mid x \in X_\xi\} \cup \{\langle \varphi_T(x), g(x)\rangle \mid x \in X_\chi\}$$

  Then: $H = \{f \star g \mid f \in F, g \in G, f \star g \text{ is a function}\}$.

2.5.3 Relabellings

Pushing and popping referents to and from stacks is done by relabelling the labelled sets. The sets are relabelled in such a way that the referents in the sets have now labels that correspond to their new position.

Definition 2.5.7. Consider two label spaces, V_1 and V_2, and let E be a (possibly partial) injective function from V_1 to V_2. We can relabel a ℓ, V_1-set $\xi = \langle V_\xi, \mathsf{lab}_\xi, X_\xi\rangle$ via E to ξ' (written: $\xi \star E = \xi'$) by taking:

- $\xi' = \langle E[V_\xi], \mathsf{lab}_\xi \circ E^{-1}, X\rangle$
  where $E[V_\xi] = \{E(v) \mid v \in V_\xi \text{ and } E(v) \text{ is defined}\}$.

The $\star$ operation will be used on contents, too. For a content $S = \langle \xi, F \rangle$ we define $S \star E = \langle \xi \star E, F \rangle$.

We will use special relabellings. Let's define the injective functions $\mathsf{S}_{a,\nu}$ from Λ^{ν} to $\Lambda^{\nu+(a)}$, for all $a \in \mathsf{VAR}^{+}$:

$$\mathsf{S}_{a,\nu}(\langle s, o, i \rangle) = \begin{cases} \langle s, o, i+1 \rangle & \text{if } a = s \\ \langle s, o, i \rangle & \text{if } a \neq s \end{cases}$$

$\mathsf{S}_{a,\nu}$ will be used to mimic a 'push' operation, while $\mathsf{S}_{a,\nu}^{-1}$ (its inverse function) will be used to mimic a 'pop' operation.

2.5.4 The Language of CML

In order to process a CML-string, we have to know what symbols we may expect. There are two types of CML symbols. (Note that these types do not correspond to the types I used in section 2.5.1, where I talked about symbols that correspond to natural language, and symbols that add structure to the CML-string.)

The first type of symbols, the predicate symbols, form, in a sense, the static part of CML. They are used to 'add content' to the information state, but do not modify the context. In other words, they change the referents and the assignments, but not the labels. The predicate symbols are specified by the signature.

Definition 2.5.8 (CML-signature). A CML-signature is a structure $\langle \mathcal{P}, \mathcal{I}, \mathsf{ind}, \mathsf{lab}, \mathsf{ref} \rangle$, in which $\mathcal{P}$ is a set of predicate symbols, $\mathcal{I}$ is a partial order of indices, ind is a function that assigns an index to each predicate (we write ind_P for $\mathsf{ind}(P)$), lab is a function that assigns to each index $i \in \mathcal{I}$ a label space $\mathsf{lab}_i \subseteq \Sigma$, and ref is a function that assigns to each predicate $P \in \mathcal{P}$ a finite $\ell, \mathsf{lab}(\mathsf{ind}_P)$-set.

The indices are used to identify a state of the stacks. In our set-up, we use the multisets over VAR^{+} as indices, ordered in the following way:

$$\nu \leq \mu \text{ iff } \nu(v) \leq \mu(v) \text{ for all } v$$

Furthermore, we take: $\mathsf{lab}(\nu) = \Lambda^{\nu}$

The predicates are verified in a model. Note that the below definition differs from the one we used in DRT. This is necessary because in CML the arguments of a predicate have a label, rather than a syntactical order.

Definition 2.5.9 (CML-model). A model $\mathcal{M}$ for a signature $\mathcal{S} = \langle \mathcal{P}, \mathcal{I}, \mathsf{ind}, \mathsf{lab}, \mathsf{ref} \rangle$ is a tuple $\langle D, I \rangle$, where D is a non-empty domain of discourse, and I, the interpretation function, assigns to each predicate $P \in \mathcal{P}$ a set of functions from the referents of $\mathsf{lab}(\mathsf{ind}_P)$ to D.

In our example we will use the following predicates:

- $man \in \mathcal{P}$
  $\mathsf{ind}(man) = (\mathtt{term}, \mathtt{man}, \mathtt{he})$
  $\mathsf{ref}(man) = (\mathtt{val}^0\mathtt{man}^0\mathtt{he}^0)$
  $I(P) =$ The functions f such that $f(0) \in D$ is a man.

- $sub \in \mathcal{P}$
  $\mathsf{ind}(sub) = (\mathtt{term}, \mathtt{arg})$
  $\mathsf{ref}(sub) = (\mathtt{val}^0\mathtt{sub}^0)$
  $I(P) =$ All functions f such that $f(0) \in D$.

- $sees \in \mathcal{P}$
  $\mathsf{ind}(sees) = (\mathtt{arg})$
  $\mathsf{ref}(sees) = (\mathtt{sub}^0)(\mathtt{ob}^0)$
  $I(P) =$ All functions f such that $f(0) \in D$ sees $f(1) \in D$.

- $that \in \mathcal{P}$ (such as in "a dog *that* barks")
  $\mathsf{ind}(that) = (\mathtt{term}^{(2)})$
  $\mathsf{ref}(that) = (\mathtt{val}^0\mathtt{val}^1)$
  $I(P) =$ All functions f such that $f(0) \in D$.

The definitions of the other predicate symbols can be easily derived from the above ones.

The second type of symbols are the relabelling symbols. They add dynamic behavior to CML. To each of the relabelling symbols A, we assign an indexed set of relabellings J_A, in the following way. For each $i \in \mathcal{I}$, $J_A(i)$, if defined, is an index, and $J_{A,i}$ is a relabelling from $\mathsf{lab}(i)$ to $\mathsf{lab}(J_A(i))$. We require that if $i \leq i'$ and J_A is defined, then $J_A(i')$ is defined and $J_A(i) \leq J_A(i')$ and $J_{a,i}$ extends $J_{a,i'}$.

In CML$_1$, we define the set RELAB of relabelling symbols as follows:

$$\mathsf{RELAB} = \{\lceil_{v},_{v}\rceil \mid v \in \mathsf{VAR}^+\}$$

We will write '[' and ']' for '$\lceil_{\mathtt{arg}}$' and '$_{\mathtt{arg}}\rceil$', respectively, and '$\langle$' and '$\rangle$' for '$\lceil_{\mathtt{term}}$' and '$_{\mathtt{term}}\rceil$'. We define:

- $J_{\lceil_a}(\nu) = \nu + (a)$

- $J_{\lceil_a,\nu} = \mathsf{S}_{a,\nu}$

- $J_{_a\rceil}(\mu) = \begin{cases} \nu & \text{if } \nu = \mu + (a) \\ \text{undefined} & \text{otherwise} \end{cases}$

- $J_{{}_a\rceil,\nu+(a)} = \mathsf{S}_{a,\nu}^{-1}$

NP's in natural language have both static and dynamic components. Since all atomic CML-symbols are either static or dynamic, the CML-symbols that correspond to natural language NP's, are molecular symbols. We define:

- a-man = $\lceil_{\texttt{man}}\lceil_{\texttt{he}} man$
- the-man = $\lceil_{\texttt{he}} man$

The equivalents for the molecular symbols corresponding to other NP's, are easily derived.

Note that the language of CML lacks constants. Constants are considered to be a special kind of predicate, that only applies to one object. For example: in DRT 'John' would be considered a constant; in CML it is a predicate. I will not go into the advantages and disadvantages of this approach in this thesis.

2.5.5 Updating Information States

In this subsection we describe how we determine if a given CML-string is true in a model. Like in all dynamical logics, the symbols of CML update a certain information state. As said, an information state consists of a content. But besides that, we record the state of the stacks.

Definition 2.5.10 (Information State). Let $\mathcal{S} = \langle \mathcal{P}, \mathcal{I}, \mathsf{ind}, \mathsf{lab}, \mathsf{ref} \rangle$ be a CML-signature. Then, an information state σ is a tuple $\langle i, S \rangle$, where i is an index and $S = \langle \xi, F \rangle$ is a content, with ξ an ℓ, lab_i-set.

The initial information state, $\langle \emptyset, \langle \emptyset, \{\emptyset\} \rangle \rangle$, contains no information at all. The symbols of the strings, however, modify the information state, as follows:

Definition 2.5.11 (Update). Let $\mathcal{S} = \langle \mathcal{P}, \mathcal{I}, \mathsf{ind}, \mathsf{lab}, \mathsf{ref} \rangle$ be CML-signature, $\mathcal{M} = \langle \mathcal{D}, \mathcal{I} \rangle$ a CML-model for $\mathcal{S}$, and $\sigma = \langle i, S \rangle$ an information state.

(i) Let $A \in \mathcal{P} \cup \mathsf{RELAB}$ be a CML-symbol. Then:

- If $A \in \mathcal{P}$:
  $$\sigma[A] = \begin{cases} \langle i, S \oplus \mathsf{cont}_P \rangle & \text{if } \mathsf{ind}_P \leq i \\ \text{undefined} & \text{otherwise} \end{cases}$$
  where $\mathsf{cont}(P) = \langle \mathsf{ref}(P), I(P) \rangle$
- If $A \in \mathsf{RELAB}$:
  $$\sigma[A] = \begin{cases} \langle J_A(i), S \star J_{a,i} \rangle & \text{if } J_A(i) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

(ii) Let $\varphi, \psi \in (\mathcal{P} \cup \mathsf{RELAB})^*$ be CML-strings. Then:

- $\sigma[\epsilon] = \sigma$
- $\sigma[\varphi\psi] = \sigma[\varphi][\psi]$

We can see if the discourse that has been analyzed so far is true by looking at the set of assignments: it is true if the set of assignments is non-empty.

Remark. In the previous sections, we have implicitly assumed that the term-level and the predicate level markers alternate. However, the following CML-string is, of course, not forbidden:

$$[\langle\langle \text{the-man } \textit{sub}\rangle \text{ a-dog } \textit{ob}\rangle \textit{sees}]$$

However, in all normal discourses that are translated from natural language, the predicate-level and term-level arguments do alternate. I leave it to the reader to calculate the final information state of the discourse above.

2.5.6 An example

Now, let's work out an example. We are going to analyze the discourse "A man sees a dog that barks.". This discourse is represented in a CML-string as follows:

$$[\langle \text{a-man } \textit{sub}\rangle \ \textit{sees} \ \langle \text{a-dog } \textit{ob} \ [\langle \textit{that sub}\rangle \ \textit{barks}] \ \rangle]$$

We are going to update the empty information state σ with this CML-string. We assume that there are two men, Art and Ben, and two dogs, Pluto and Mars, and that Art sees Pluto, Mars and Ben, and Ben sees Art. Furthermore, we assume that Pluto is the only one of the three who barks. Then: (note that I occasionally skip a few steps)

- $\sigma[`[\text{'}] = \langle(\mathtt{arg}), \langle\emptyset, \{\emptyset\}\rangle\rangle$
- $\sigma[`[\langle\text{'}] = \langle(\mathtt{arg}, \mathtt{term}), \langle\emptyset, \{\emptyset\}\rangle\rangle$
- $\sigma[`[\langle\lceil_{\mathtt{man}}\text{'}] = \langle(\mathtt{arg}, \mathtt{term}, \mathtt{man}), \langle\emptyset, \{\emptyset\}\rangle\rangle$
- $\sigma[`[\langle\lceil_{\mathtt{man}}\lceil_{\mathtt{he}}\text{'}] = \langle(\mathtt{arg}, \mathtt{term}, \mathtt{man}, \mathtt{he}), \langle\emptyset, \{\emptyset\}\rangle\rangle$
- $\sigma[`[\langle\lceil_{\mathtt{man}}\lceil_{\mathtt{he}}\mathit{man}\text{'}] = \langle(\mathtt{arg}, \mathtt{term}, \mathtt{man}, \mathtt{he}), \alpha_0\rangle$, where α_0 is given by the following table (the top line specifies the labels of a referent, and the lines below specify the assignments):

$\mathtt{val}^0$ $\mathtt{man}^0$ $\mathtt{he}^0$
Art
Ben

- $\sigma[$‘$[\langle\lceil_{\mathtt{man}}\lceil_{\mathtt{he}}\mathit{man\ sub}$’$] = \langle(\mathtt{arg}, \mathtt{term}, \mathtt{man}, \mathtt{he}), \alpha_1\rangle$, where α_1 is given by the following table:

$\mathtt{val}^0$ $\mathtt{man}^0$ $\mathtt{he}^0$ $\mathtt{sub}^0$
Art
Ben

- $\sigma[$‘$[\langle\lceil_{\mathtt{man}}\lceil_{\mathtt{he}}\mathit{man\ sub}\rangle$’$] = \langle(\mathtt{arg}, \mathtt{man}, \mathtt{he}), \alpha_2\rangle$, where α_2 is given by the following table:

$\mathtt{man}^0$ $\mathtt{he}^0$ $\mathtt{sub}^0$
Art
Ben

- $\sigma[$‘$[\langle\lceil_{\mathtt{man}}\lceil_{\mathtt{he}}\mathit{man\ sub}\rangle$ *sees* ’$] = \langle(\mathtt{arg}, \mathtt{man}, \mathtt{he}), \alpha_3\rangle$, where α_3 is given by the following table:

$\mathtt{man}^0$ $\mathtt{he}^0$ $\mathtt{sub}^0$	$\mathtt{ob}^0$
Art	Ben
Art	Pluto
Art	Mars
Ben	Art

- $\sigma[$‘$[\langle\lceil_{\mathtt{man}}\lceil_{\mathtt{he}}\mathit{man\ sub}\rangle$ *sees* $\langle\lceil_{\mathtt{dog}}\lceil_{\mathtt{it}}$*a-dog*’$] =$ $\langle(\mathtt{arg}, \mathtt{man}, \mathtt{he}, \mathtt{term}, \mathtt{dog}, \mathtt{it}), \alpha_4\rangle$, where α_4 is given by the following table:

$\mathtt{man}^0$ $\mathtt{he}^0$ $\mathtt{sub}^0$	$\mathtt{ob}^0$	$\mathtt{val}^0$ $\mathtt{dog}^0$ $\mathtt{it}^0$
Art	Ben	Pluto
Art	Ben	Mars
Art	Pluto	Pluto
Art	Pluto	Mars
Art	Mars	Pluto
Art	Mars	Mars
Ben	Art	Pluto
Ben	Art	Mars

- $\sigma[`[\langle\lceil_{\texttt{man}}\lceil_{\texttt{he}}\textit{man sub}\rangle\ \textit{sees}\ \langle\lceil_{\texttt{dog}}\lceil_{\texttt{it}}\textit{a-dog ob}'] =$ $\langle(\texttt{arg},\texttt{man},\texttt{he},\texttt{term},\texttt{dog},\texttt{it}),\alpha_5\rangle$, where α_5 is given by the following table:

$\texttt{man}^0$ $\texttt{he}^0$ $\texttt{sub}^0$	$\texttt{ob}^0$ $\texttt{val}^0$ $\texttt{dog}^0$ $\texttt{it}^0$
Art	Pluto
Art	Mars

Note how the last two referents are identified.

- $\sigma[`[\langle\lceil_{\texttt{man}}\lceil_{\texttt{he}}\textit{man sub}\rangle\ \textit{sees}\ \langle\lceil_{\texttt{dog}}\lceil_{\texttt{it}}\textit{a-dog ob}\ [\langle'] =$ $\langle(\texttt{arg}^{(2)},\texttt{man},\texttt{he},\texttt{term}^{(2)},\texttt{dog},\texttt{it}),\alpha_6\rangle$, where α_6 is given by the following table:

$\texttt{man}^0$ $\texttt{he}^0$ $\texttt{sub}^1$	$\texttt{ob}^1$ $\texttt{val}^1$ $\texttt{dog}^0$ $\texttt{it}^0$
Art	Pluto
Art	Mars

- $\sigma[`[\langle\lceil_{\texttt{man}}\lceil_{\texttt{he}}\textit{man sub}\rangle\ \textit{sees}\ \langle\lceil_{\texttt{dog}}\lceil_{\texttt{it}}\textit{a-dog ob}\ [\langle\ \textit{that sub}'] =$ $\langle(\texttt{arg}^{(2)},\texttt{man},\texttt{he},\texttt{term}^{(2)},\texttt{dog},\texttt{it}),\alpha_7\rangle$, where α_7 is given by the following table:

$\texttt{man}^0$ $\texttt{he}^0$ $\texttt{sub}^1$	$\texttt{ob}^1$ $\texttt{val}^1$ $\texttt{dog}^0$ $\texttt{it}^0$ $\texttt{val}^0$ $\texttt{sub}^0$
Art	Pluto
Art	Mars

Note how the stacks make it possible to distinguish between the subject of the main sentence and the subject of the subordinate clause.

And, finally:

- $\sigma[`[\langle\text{a-man}\ \textit{sub}\rangle\ \textit{sees}\ \langle\text{a-dog}\ \textit{ob}\ [\langle\textit{that sub}\rangle\ \textit{barks}]\ \rangle]'] =$ $\langle(\texttt{man},\texttt{he},\texttt{dog},\texttt{it}),\alpha_6\rangle$, where α_6 is given by the following table:

$\texttt{man}^0$ $\texttt{he}^0$	$\texttt{dog}^0$ $\texttt{it}^0$
Art	Pluto

The set of assignments is non-empty, so the discourse is true.

2.6 Summary

In this chapter I have discussed two theories within the field of Dynamic Semantics.

The first theory I discussed is Discourse Representation Theory (DRT). In DRT, discourses are given graphical representations, so called Discourse Representation Structures (DRS's). DRS consist of two parts: in the top, there is a box with discourse referents. And below that, there is collection of conditions. Like:

$$\begin{array}{|l|}\hline x, y, z \\ \hline man(x) \\ dog(y) \\ sees(x, y) \\ barks(y) \\ \hline \end{array}$$

Intuitively, a DRS is true, if there exists an assignment that assigns to each of the referent in the top box an object in the domain of discourse, in such a way that the conditions are satisfied.

The second theory is Context Modification Logic (CML). CML is a dynamic logic. Like natural language, CML-formulas are strings of symbols, like the following:

$$[\langle \text{a-man } \mathit{sub}\rangle\ \mathit{sees}\ \langle \text{a-dog } \mathit{ob}\ [\langle \mathit{that}\ \mathit{sub}\rangle\ \mathit{barks}]\ \rangle]$$

An information state consists of a labelled set of referents, organized in stacks, and a set of assignments that assign an object in the domain to each referent. We begin with the empty information state. Each of the symbols has a certain effect on the information state, and these effects are applied in sequence. In the end, a CML-string is true, if the set of assignments is non-empty.

Chapter 3

Hypergraphs and Graph Rewriting

3.1 Introduction

Rewriting is a computational framework in which data objects are transformed into other objects of the same kind by applying rewrite rules. Many different mathematical objects may be used; in mathematics the most common form of rewriting is term rewriting, and in linguistic research phrase structure grammars, which are essentially string rewrite systems, have been very popular.

In this paper we use a different form of rewriting, (hyper)graph rewriting. In graph rewriting the objects that are rewritten are not strings or terms, but hypergraphs. In this chapter we will explore the basics of graph rewriting. People with little interest in the formal treatment of graph rewriting, need only read section 3.2 to get the general idea.

Formalizations and ideas set out in this chapter were partly based on [BCK01], [HHT96], [Plu98], [KKSdV93] and [SS94].

3.2 Informal Introduction to Graph Rewriting

An ordinary graph is a set of nodes and (labelled) edges. Each edge connects exactly two nodes. For our purposes this is not sufficient: we want the edges to represent relations between objects, and relations are not necessarily binary. Therefore, in this paper, we use an extended type of graph: a *hypergraph*. The difference between an ordinary graph and a hypergraph

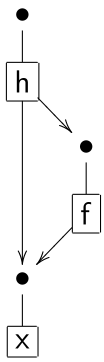


Figure 3.1: An example hypergraph.

is the 'arity' of its edges: in a hypergraph an edge may connect any positive number of nodes. We denote the nodes of a graph by bullets, and the edges by boxes, with the labels of the edges written inside. An example of a hypergraph can be found in figure 3.1.

So how are we going to do rewriting with hypergraphs? It is not that different from term and string rewriting. A graph rewrite rule consists of a left-hand side and a right-hand side. When the rule is applied to a graph, we search for a subgraph of the graph that matches the left-hand side of the rule, and replace it with the right hand side.

However, terms and strings are linear. Thus, in terms, when we delete a subterm from the term we can also delete its subterms (we even must do so). In hypergraphs, however, some nodes of the subgraph may be connected by edges of the graph that are not in the part matched by the left-hand side of the rule. To prevent such edges from pointing to nothing when the rule has been applied, a graph rewrite rule has a third component, the identification function. Assume a node a in the left-hand side is identified to a node b in the right-hand side. Then, when the rewrite is applied to a graph, all edges that were connected to a are, in the result graph, connected to b instead.

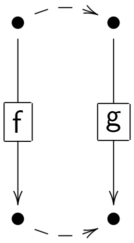


Figure 3.2: An example graph rewrite rule.

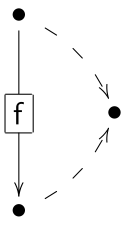
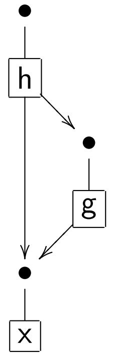


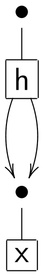
Figure 3.3: An non-injective graph rewrite rule.

An example of a graph rewrite rule is given in figure 3.2. The identification function is given by the dashed arrows. This graph rewrite rule searches for a binary edge labelled f, and replaces it with a binary edge labelled g. As an example, consider the graph of figure 3.1. We are going to apply the graph rewrite rule to this graph. We search for an occurrence of the left-hand side of the rule in this graph, and replace it with an occurrence of the right-hand side. The identification function tells us what nodes are identified. So, we get the following result:



Note, that the edge which was previously labelled 'f', is now labelled 'g'.

In the graph rewrite rule of figure 3.2, all nodes of the left-hand side of the rule are identified with a unique node of the right-hand side of the rule (i.e. the identification function is injective). However, we can also formulate a non-injective rewrite rule, such as the one of figure 3.3. Now let's see what happens if we apply this rule to the graph of figure 3.1. First, we seek an occurrence of the left-hand side of the rule, and we find the same part as in the previous example. Only now we replace this part by a single node, and identify the two nodes of the left-hand side with this node. Thus, we get the following result:



Note that this graph has one node less than the original graph, since the application of the graph rewrite rule had the effect of identifying two nodes of the graph.

In the next few sections we will formalize the idea presented in this section.

3.3 Hypergraphs and Morphisms

3.3.1 Hypergraphs

Definition 3.3.1 (Graph signature). A graph signature $\mathcal{S}$ is a tuple $\langle \Sigma, \mathsf{arity} \rangle$, in which Σ is a finite set of function symbols, and arity a function from Σ to $\mathbb{N} - \{0\}$.

Definition 3.3.2 (Hypergraph). Let $\mathcal{S} = \langle \Sigma, \mathsf{arity} \rangle$ be a graph signature. Then, a (labelled) hypergraph G is a structure $\langle V, E, \mathsf{lab}, \mathsf{att} \rangle$, in which: V is a non-empty finite set of nodes (vertices); E is a finite set of hyperedges; lab is a function from E to Σ; and att (the attachment function) is a function from E to V^+ such that $|\mathsf{att}(e)| = \mathsf{arity}(\mathsf{lab}(e))$ for all $e \in E$.

In the sequel, the words *graph* and *edge* will refer to hypergraphs and hyperedges, respectively. Also, the graph signature will usually be left implicit.

Since the nodes that are connected to an edge are ordered, it is tempting to view the graphs as being 'directed'. We adopt the following terminology: consider an edge e with $\mathrm{att}(e) = v_0 \ldots v_n$, then we will call v_0 the *source node* of e, and $v_1, \ldots, v_n$ the *target nodes*. Note that an edge always has exactly one source node, but may have any number of target nodes or none at all.

We use the following style to graphically represent graphs. Nodes are represented by bullets. Edges are drawn as rectangles, with the label of the edge written inside, and lines exiting to the attached nodes, ordered counter-clockwise. The line to the source node of the edge will be the only one without an arrow on the far end. The graph in figure 3.1, for example, could be given by: $V = \{a, b, c\}$, $E = \{d, e, f\}$, $\mathsf{att}(d) = acb$, $\mathsf{att}(e) = bc$,

$\mathsf{att}(f) = c$, $\mathsf{lab}(d) = \mathsf{h}$, $\mathsf{lab}(e) = \mathsf{f}$, and $\mathsf{lab}(f) = \mathsf{x}$. The codes of the nodes and edges (a, b and c for the nodes, and d, e and f for the edges) do not carry any information and are therefore not in the graphical representation of the graph.

Definition 3.3.3 (Reachability). Let $G = \langle V, E, \mathsf{lab}, \mathsf{att}\rangle$ be a graph.

(i) A node $w \in V$ is *directly reachable* from a node $v \in V$ iff there exists an edge $e \in E$ with $\mathsf{att}(e) = v_0 \dots v_n$ such that $v = v_0$ and $w \in v_1, \dots, v_n$.

(ii) A node $w \in V$ is *reachable* from a node $v \in V$ iff either $w = v$ or there is a $u \in V$ such that u is reachable from v, and w is reachable from u.

(iii) The *restriction* of graph G by a node $v \in V_G$ (written: $G|_v$), is the graph that consists only of nodes that are reachable from v in G, and the edges between them.

3.3.2 Homomorphisms and Isomorphisms

Definition 3.3.4 (Homomorphism). Let G and H be graphs. Then:

(i) A *homomorphism* from G to H is a function φ from $V_G \cup E_G$ to $V_H \cup E_H$, such that:

- $\varphi(x) \in V_H$ iff $x \in V_G$;
- for all edges $e \in E_G$:
  - $\mathsf{lab}_G(e) = \mathsf{lab}_H(\varphi(e))$;
  - $\varphi^*(\mathsf{att}_G(e)) = \mathsf{att}_H(\varphi(e))$.

We will call G *homomorphic* to H, if there exists a homomorphism from G to H.

(ii) φ is an *isomorphism*, iff its inverse φ^{-1} is a homomorphism from H to G. We will say that G and H are *isomorphic*, if there exists an isomorphism between them.

Because isomorphicity is much more interesting than exact equality (we only have to change the coding for the nodes, for example, to get a 'different' graph), in the following we will just write $G = H$ if G and H are 'only' isomorphic.

3.4 Graph Rewriting

In this section the following notation will be useful. Let $f : A \to B$ be a function. Then we define:

- if $S \subseteq A$ is a set, then: $f[S] = \{f(x) \mid x \in S\}$;
- $\text{Dom}(f) = A$ (the domain of f);
- $\text{Rng}(f) = f[\text{Dom}(f)]$ (the range of f).

Note that, if f is not surjective, then $\text{Rng}(f) \neq B$.

Another useful notation is the following: if A is a set of sets, then we write $\bigcup A$ for $\{x \mid \exists B \in A : x \in B\}$.

3.4.1 Disjoint Union

In this paper, when we write $A \oplus B$, we will assume that $A \cap B = \emptyset$, so that we can take:

$$A \oplus B = A \cup B.$$

This assumption can be made without loss of generality, because we are working with graphs: if for some reason A and B would have an element in common, we replace one of the graphs of which A or B is a part, by an isomorphic one.

3.4.2 Graph Rewrite Rules

Definition 3.4.1 (Graph Rewrite Rule). A graph rewrite rule is a structure $\langle L, R, \alpha \rangle$, in which L (the left-hand side) and R (the right-hand side) are graphs, and α (the identification function) is a total function from V_L to V_R.

When representing graph rewrite rules graphically, we will draw the left-hand side and the right-hand side next to each other, and denote the identification function with dotted arrows. Note that we will often omit the isolated nodes in the right-hand side and the dotted arrows of the identification function (α) that point to them. In fact, if you see a node v in the left-hand side of the graphical representation of a rule, that has no exiting dotted arrow, then there is a unique isolated node w in the right-hand side such that $\alpha(v) = w$.

Definition 3.4.2. A graph rewrite rule $\langle L, R, \alpha \rangle$ is injective (surjective) iff its identification function α is.

3.4.3 Applying a Rule to a Graph

A graph rewrite rule $r = \langle L, R, \alpha \rangle$ can be applied to a graph G if a homomorphism φ exists from L to G. Intuitively the result graph H is the same as G, except that the part to which the left-hand side of the rule was mapped is replaced by the right-hand side of the rule. Now, we proceed by looking at a rewrite step more formally.

To simplify the definition, we will assume, without loss of generality, that $\mathrm{Rng}(\alpha) \subseteq \wp(V_L)$, $(V_R - \mathrm{Rng}(\alpha)) \cap \wp(V_L) = \emptyset$, and that for all $w \in \mathrm{Rng}(\alpha)$: $w = \{v \in V_L \mid \alpha(v) = w\}$.

Definition 3.4.3 (Graph Rewrite Step). Let $r = \langle L, R, \alpha \rangle$ be a graph rewrite rule. Then, if a homomorphism φ exists from L to a graph G, there exists a proper graph rewrite step $G \rightarrow_{r,\varphi} H$, in which H is constructed as follows:

- $E_H = (E_G - \varphi[E_L]) \oplus E_R$
- $\mathsf{lab}_H(e) = \begin{cases} \mathsf{lab}_G(e) & \text{if } e \in E_G - \varphi[E_L] \\ \mathsf{lab}_R(e) & \text{if } e \in E_R \end{cases}$
- Let: $V'_G = \{\varphi[w] \mid w \in \mathrm{Rng}(\alpha)\}$.

  Then:
  $V_H = V'_G \oplus \{\{v\} \mid v \in V_G - \bigcup V'_G\} \oplus (V_R - \mathrm{Rng}(\alpha))$
- Let $\alpha' : V_G \rightarrow V_H$ be given by: $\alpha'(v) = w$ iff $v \in w$ (note that this w is unique), and $\psi(v) : V_R \rightarrow V_H$ by: $\psi(v) = \varphi[v]$.

  Then:
  $\mathsf{att}_H(e) = \begin{cases} \psi^*(\mathsf{att}_R(e)) & \text{if } e \in E_R \\ \alpha'^*(\mathsf{att}_G(e)) & \text{if } e \in E_G - \varphi[E_L] \end{cases}$

We will write $G \rightarrow_r H$ if $G \rightarrow_{r,\varphi} H$ for some homomorphism φ.

Note that nodes are never destroyed. The number of nodes can only decrease if two or more nodes are identified.

A graph rewrite step can also be described in category theory by a double pushout. In this sense, a graph rewrite rule is represented by a pair $\langle \varphi_l, \varphi_r \rangle$ of homomorphisms, such that $L \overset{\varphi_l}{\leftarrow} K \overset{\varphi_r}{\rightarrow} R$. Here, L and R are the left and right-hand side of the rewrite rule, respectively, and K is the so-called *interface graph*, which specifies what nodes must be identified. When applying this rule to a graph G, we construct the following diagram:

$$\begin{tikzcd}
L \arrow[d] & K \arrow[l] \arrow[r] \arrow[d] & R \arrow[d] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

in which the two squares are required to be pushouts. Then H will be the result of the rewrite step.

We can convert a graph rewrite rule $s = \langle L_s, R_s, \alpha_s \rangle$, as given in definition 3.4.1, to a category theoretic rewrite rule by taking $L = L_s$, $R = R_s$, $K = \langle \mathrm{Dom}(\alpha_s), \emptyset, \emptyset, \emptyset \rangle$, $\varphi_l = \mathsf{id}$ (the identity), and $\varphi_r = \alpha_s$. Now, the category theoretic notion of graph rewriting will yield the same results as definition 3.4.3 (modulo isomorphicity).

3.4.4 Graph Rewrite Systems

Definition 3.4.4 (Graph Rewrite System). A graph rewrite system (GRS) S is a finite set of graph rewrite rules. We will write $G \Rightarrow_S H$ if $G \rightarrow_r H$ for some $r \in S$. The reflexive and transitive closure of $\Rightarrow_S$ will be denoted by $\Rightarrow_S^*$.

Definition 3.4.5 (Normal Form). A graph G is a *normal form* of a GRS S iff there is no graph H such that $G \Rightarrow_S H$.

Usually, normal forms are what we are interested in. They are, in a sense, the 'output' of the rewrite system. We give a very crude algorithm to find a normal form of a graph rewrite system, starting with an arbitrary graph G.

Algorithm 3.4.1. Let S be a GRS, and G be a graph. Then a normal form H of S from G is found as follows:

```
(i) $H \leftarrow G$.
(ii) While there are rules in $S$ that can be applied to $H$:
    a. Non-deterministically select a rule $r \in S$ and a homomorphism $\varphi$ from $L_r$ to $H$ such that $H \rightarrow_{r,\varphi} K$ exists.
    b. $H \leftarrow K$.
(iii) $H$ is the result graph.
```

Let a GRS S be given. Then this algorithm will always terminate (for S), if the following condition holds:

> There is an $n \in \mathbb{N}$ such that for all sequences $G_0, \ldots, G_k$ with $G_{i-1} \Rightarrow_S G_i$ for all $0 < i \leq k$: $k \leq n$.

In the framework that we are going to develop in chapter 5 this condition will hold.

3.5 Graph Rewriting with Application Conditions

In the previous sections, I have described a general form of graph rewriting. The condition for a rule to be applied to a graph G (application condition) in this form of graph rewriting, is that there exists a homomorphism from the left-hand side of the rule to G. This application condition is positive: it requires the existence of a homomorphism.

Although this form of graph rewriting (in the sequel I will call it normal graph rewriting if there is a chance of ambiguity) is very powerful, it is sometimes not very convenient. In section 5.4, for example, a lot of rules would be required to establish the desired effect. The reason for this is, that we want to define rules that can only be applied if a certain homomorphism does *not* exist. In other words, we need negative application conditions as well as positive ones. Therefore, in section 5.4, we will use an extended form of graph rewriting, in which the application conditions are explicitly defined. To each of the graph rewrite rules a number of additional constraints, which are either positive or negative, will be associated. These constraints must all be satisfied for the rule to be applied. In this section I will briefly discuss this form of graph rewriting, which I will call graph rewriting with application conditions, or conditional graph rewriting — although, strictly, normal graph rewriting has an (implicit) application condition, also —, along the lines of [HHT96] (but converted to hypergraphs).

The techniques of this section are only used in section 5.4. Perhaps a good reading strategy is, therefore, to read chapter 4 and sections 5.1–5.3 first.

3.5.1 Application Conditions

In graph rewriting with application conditions, to each of the graph rewrite rules a condition is added. Such a condition consists of two sets of constraints, a set of positive constraints and a set of negative constraints. Constraints can be seen as ‘contexts’, of which the presence is either required (in the case of positive constraints) or forbidden. Naturally, the left-hand side of a rule is a part of the context.

Definition 3.5.1 (Application Conditions). Let $r = \langle L, R, \alpha \rangle$ be a graph rewrite rule.

(i) A *constraint* for r is a tuple $\langle \lambda, \hat{L} \rangle$, where $\hat{L}$ is a graph, and λ is a homomorphism from L to $\hat{L}$.

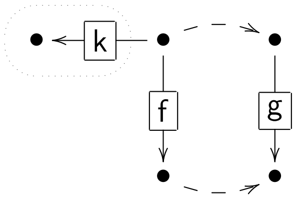


Figure 3.4: A rewrite rule with one positive constraint.

(ii) An *application condition* A for r is a tuple $\langle A^+, A^- \rangle$, where A^+ and A^- (the set of positive and the set of negative constraints, respectively) are two sets of constraints for r.

(iii) Let $l = \langle \lambda, \hat{L} \rangle$ be a constraint for r, and φ a homomorphism from L to a graph G. Then φ *satisfies* l iff there exists a homomorphism ψ from $\hat{L}$ to G such that $\psi \circ \lambda = \varphi$.

(iv) A homomorphism φ satisfies an application condition $A = \langle A^+, A^- \rangle$ iff φ satisfies all constraints $a \in A^+$ and φ satisfies *no* constraint $b \in A^-$.

Definition 3.5.2 (Conditional Graph Rewrite Rule). A *conditional graph rewrite rule* is a tuple $\hat{r} = \langle r, A \rangle$, where r is a (normal) graph rewrite rule, and A is an application condition for r. We will call r the underlying graph rewrite rule of $\hat{r}$.

When graphically representing a graph rewrite rule with an application condition, we will use dotted regions to denote the constraints. An example of a rewrite rule with one (positive) constraint can be found in figure 3.4. Let $\hat{r} = \langle r, A \rangle$ be this rule, with $r = \langle L, R, \alpha \rangle$ and $A = \langle \{\langle \lambda, \hat{L} \rangle\}, \emptyset \rangle$. Then, L is the part on the left, outside the region, $\hat{L}$ is the left-hand side *and* the part in the region, and λ is the obvious homomorphism from L to $\hat{L}$. Note that the rule $\hat{r}$ is almost the same as the rule r from figure 3.2 (which is its underlying graph rewrite rule), only the constraint has been added. Negative constraints are represented by a dotted region with a line through it, as can be seen in figure 3.5. Again, the underlying rewrite rule of this graph is r, only this time $A = \langle \emptyset, \{\langle \lambda, \hat{L} \rangle\} \rangle$.

3.5.2 Conditional Graph Rewriting

The idea of conditional graph rewriting, is that we pose additional conditions for a rule to be applicable to a graph. So, intuitively, a conditional graph rewrite rule $\hat{r}$ can be applied to a graph if the underlying graph rewrite rule can be applied to the graph, *and* the constraints of the application condition

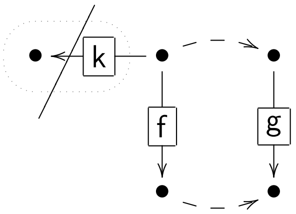


Figure 3.5: A rewrite rule with one negative constraint.

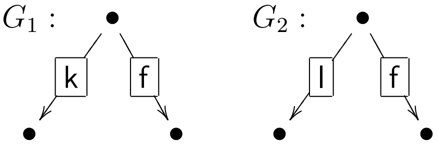
are satisfied. If so, the result of the rewrite is the same as the result of applying the underlying rewrite rule r of $\hat{r}$ to the graph. Formally:

Definition 3.5.3 (Conditional Graph Rewrite Step). Let $\hat{r} = \langle r, A \rangle$ be a graph rewrite rule with an application condition, and G a graph. Then, if there is a homomorphism φ from L_r to G such that φ satisfies A, there exists a graph rewrite step $G \rightarrow^{\mathsf{cond}}_{\hat{r},\varphi} H$, where $G \rightarrow_{r,\varphi} H$.

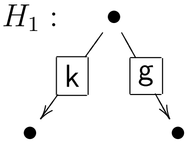
I leave it to the reader to extend the notion of graph rewrite system and algorithm 3.4.1 on page 30 to work with conditional graph rewrite rules also.

3.5.3 Examples and Remarks

Let $\hat{r}$ be the conditional graph rewrite rule of figure 3.4 and $\hat{s}$ the one of figure 3.5. Now let's see what happens if we try to apply these rules to the graphs G_1 and G_2:

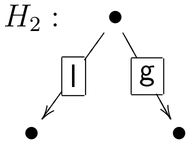


First, we try to apply $\hat{r}$ to G_1, via the obvious homomorphism. We see that this homomorphism satisfies the application condition of $\hat{r}$, so we may apply the rule and get the following graph:

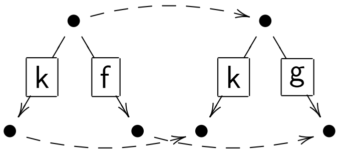


Now, let's try to apply $\hat{s}$ to G_1. Since its application condition is the opposite of the application condition of $\hat{r}$, of course it is not satisfied. Thus, $\hat{s}$ cannot be applied to G_1, although its underlying graph rewrite rule can.

Next, we try to apply $\hat{r}$ to G_2. The application condition is not satisfied (there is no edge labelled l from the top node), so $\hat{r}$ cannot be applied to G_2 (again, its underlying graph rewrite rule r can). The application condition $\hat{s}$, on the other hand, *is* satisfied, and so that rule can be applied on G_2, with the following result:



Note that it is the negative constraints which actually make graph rewriting with application conditions such a useful extension to normal graph rewriting. As Habel et al. note in [HHT96], positive constraints are easily simulated in normal graph rewriting by 'lifting' the context of the rule. For example, the following normal graph rewrite rule has the same effect as r from figure 3.4.



For this reason, in this thesis, I will only use negative constraints.

3.6 Summary

In this chapter we have discussed graph rewriting. In graph rewriting graphs are transformed into other graphs by using graph rewrite rules.

Graph rewrite rules consist of two graphs (a left-hand side and a right-hand side) and an identification function. When we want to apply a rule to a graph, we search for an occurrence of the left-hand side in the graph, and replace with the right-hand side of the rule. The identification function specifies which nodes are replaced by which.

Optionally, we may use graph rewriting with application conditions. In this form of graph rewriting, additional constraints are specified, which may be negative. All constraints must be satisfied before the rule may be applied.

Chapter 4

Representing Discourse by Hypergraphs

4.1 Introduction

In this chapter we will do Dynamic Semantics by making use of techniques from graph rewriting. As the basis, we will take the language of CML. Our strategy will be, to represent the information state of the discourse by a hypergraph, and the discourse itself by a sequence of graph rewrite rules. Just as the CML-symbols update the information state, our graph rewrite rules update the graph that models such an information state. The graph rewrite rules are applied to the initial graph in the correct order. When all the rules have been applied, the result is a graph that can be easily converted into a DRS.

4.2 Stacks in Graph Rewriting

As is clear from section 2.5, in CML, an information state consists of stacks. In our graph representations, we will require stacks too. Therefore, in this section we will discuss handling stacks with graph rewriting.

Stacks are represented by a sequence of nodes connected by edges. The edges in the stack are labelled `next`. In this section, the examples have only one stack, but there can be multiple stacks within one graph. To discriminate between all the different stacks, we give them a unique name (in this section we have only one stack, so we will just call it 'stack'). We add a node, and a binary edge, labelled with the name of the stack, from it to the top node of the stack.

To determine the number of elements in a stack, we need to count the

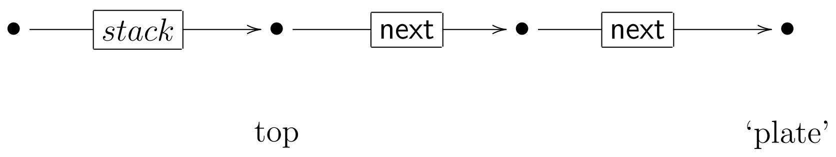


Figure 4.1: A stack with two elements.

edges labelled `next`, not the nodes. The reason for this is that the last node serves as a 'bottom' of the stack: it is the plate all the stack elements rest upon. This is necessary, because otherwise we cannot define graph rewrite rules that can operate on an empty stack.

As an example, consider a stack with two elements in it. Such a stack would be represented as in figure 4.1. This stack as such seems quite useless, since all the information the stack elements contain is which position within the stack they have. However, such stacks can be quite useful when we want to represent natural numbers.

In this paper, however, we require the stack to contain more information. This can be done by sharing the nodes in the stack. There are many ways in which we can add information. The simplest would of course be to connect the informative edges directly at the stack element. In this paper, however, we have chosen an indirect approach: we use a 'pointer' edge, labelled ⇒, that points to the actual content of the stack element, as in figure 4.2.

As we have chosen the representation of the stack, it is now time to model the two elementary stack operations: `push` and `pop`. As we all know, `push` creates a new top element, and moves the rest of the elements down, while `pop` removes the top element, and makes the second element in the row the new top element. We will design two graph rewrite rules, that, when applied to a graph containing a stack (named 'stack'), have the effect of pushing or popping an element to or from the stack.

Let's start with `push`. The left-hand side of the rule should match any

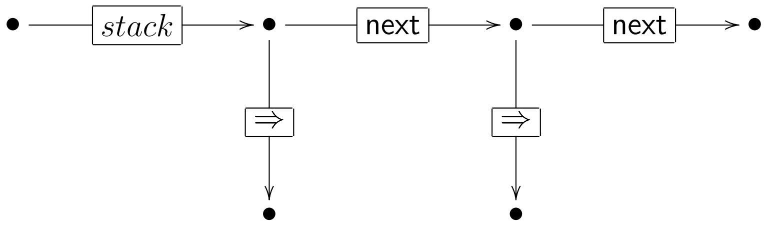


Figure 4.2: A stack with two elements and content.

stack. As we said above, all stacks contain at least one node, the bottom node, so we match a stack containing a single node. In the right hand side contains an extra node. The top node of the left-hand side is identified with the second node of the right-hand side. The complete push rule is in figure 4.3.

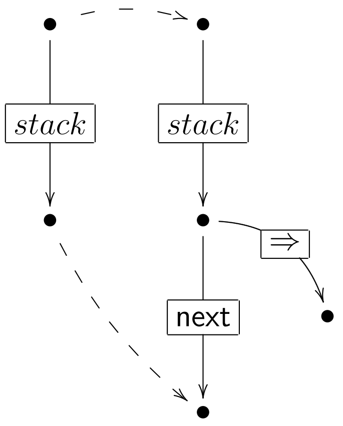


Figure 4.3: The push operation.

The pop rule is the inverse of the push rule. The left-hand side matches a stack with at least one stack element in it (so there are at least two nodes in the stack). The right-hand side has one node less. The top node of the left-hand side is discarded, and the second node is identified with the top node of the right-hand side, as in figure 4.4.

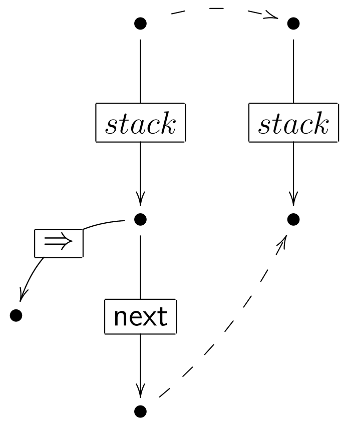


Figure 4.4: The pop operation.

Note that by applying graph rewrite rules, nodes are never removed.

Thus, both the popped stack element and its content node are maintained, they are only disconnected from the stack structure (but, they may still be connected to other parts of the graph, as we will see in the following sections). The graph rewrite rule does have the effect of destroying some edges, though.

4.3 Context Graphs

In this section we turn to the question of how to use hypergraphs (and the stack representations presented in the previous section) to represent discourse. In CML, the information states model the knowledge about the world as a set of assignments for the discourse referents to the objects in the domain. Here, we use a more DRS-like representation.

The graph representations that we are going to use consist of two parts, the *model graph* and the *stack graph*.

The Model Graph. The model graph represents a model of the knowledge of the 'real world' that was encountered in the discourse so far. It is, really, just a formalization of the intuitive graphs we draw when we want to represent relations in a graphical manner, as I described in the introduction.

Remember that in CML we did not use constants. I will adopt this approach here, too. If I talk about a model in this chapter, I will mean a DRT-model for a DRT-signature without constants.

Definition 4.3.1 (Model Graph). Let $\mathcal{M} = \langle D, I \rangle$ be a model. Then the model graph $M = \langle V, E, \mathsf{lab}, \mathsf{att} \rangle$ for $\mathcal{M}$ is given by: $V = D$, $E = \{\langle P, \vec{x} \rangle \mid \vec{x} \in I(P)\}$, $\mathsf{lab}(\langle P, \vec{x} \rangle) = P$, and $\mathsf{att}(\langle P, \vec{x} \rangle) = \vec{x}$. We will call $\mathcal{M}$ the underlying model of M.

In our set-up, we use a model graph to represent a state of knowledge about the world, not the world itself. The nodes in the model graph do not represent the objects themselves, they represent referents. However, of course we can also use a model graph to represent a real model in a graph like way. We will use this fact later.

In this chapter, the labels we are using in the model graphs are '*man*', '*dog*', '*sees*' and '*barks*', but obviously in real-life applications we need many more labels.

The Stack Graph. The stack graph represents the current state of the discourse. In CML, this is represented by stacks, hence the name and the structure. A stack graph consists of a number of stack representations in the style of section 4.2. The top node will be shared by all the stack representations. In this chapter this is not really necessary, but in chapter 5 it will be useful to keep the stacks connected to each other.

Definition 4.3.2 (Stack Graph). Let N be the set of stack names. A graph $S = \langle V, E, \mathsf{lab}, \mathsf{att}\rangle$ is a stack graph for N if there exists a unique root $v \in V$ such that:

- $S|_v = S$;
- there is a set $E_N \subseteq E$ of edges, such that:
  - for all $e \in E_N$, $\mathsf{att}(e) = vw$, for some $w \in V$, and $\mathsf{lab}(e) \in N$;
  - for all $n \in N$ there is at most one $e \in E_N$ such that $\mathsf{lab}(e) = n$;
  - there are no $e \in E - E_N$ such that $\mathsf{lab}(e) = vw$ for some $w \in V$;
- for all $u \in V - \{v\}$ there is at most one $d \in E$ with $\mathsf{att}(d) = xw$ for some $w \in V - \{v\}$, and if this d exists, then $\mathsf{lab}(d) = \mathsf{next}$;
- for all $u \in V - \{v\}$ there is at most one $d \in E$ with $\mathsf{att}(d) = wu$ for some $w \in V - \{v\}$.
- there is no $e \in E$ with $\mathsf{att}(e) = ww$ for some $w \in V$.

Note that the definition of the stack graph does not forbid us to use only a subset of the set of stack names. In the rest of this chapter, the set of stack names is given by $N = \{\$_{\mathrm{man}}, \$_{\mathrm{he}}, \$_{\mathrm{dog}}, \$_{\mathrm{it}}, \$_{\mathsf{SUB}}, \$_{\mathsf{OB}}, \$_{\mathsf{OB}}\}$. All the stack names are of the form $\$_x$, in order to easily distinguish them from the other labels. Of course, the above set too small for real-life situations, but it will be enough in our example.

As said, our graph representation of the context (which we will call a *context graph* in the sequel), consists of a model graph and a stack graph. There are also reference edges (labelled '⇒' that go from the stack elements of the stack graph to the nodes of the model graph. Or, formally:

Definition 4.3.3 (Context Graph). A graph $C = \langle V_C, E_C, \mathsf{lab}_C, \mathsf{att}_C\rangle$ is a context graph if there is a model $\mathcal{M}$ with a model graph $M = \langle V_M, E_M, \mathsf{lab}_M, \mathsf{att}_M\rangle$ and a stack graph $S = \langle V_S, E_S, \mathsf{lab}_S, \mathsf{att}_S\rangle$ such that

- $V_C = V_M \oplus V_S$
- $E_C \subseteq E_M \oplus E_S$

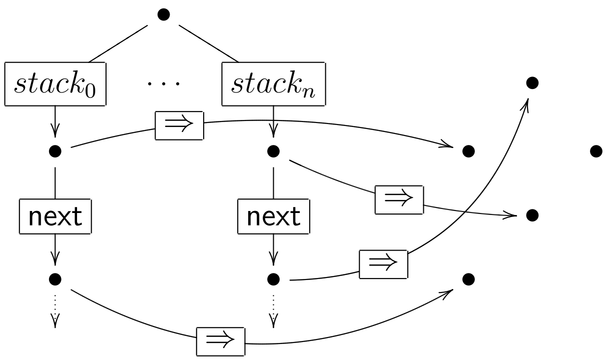


Figure 4.5: Illustration of a context graph.

- For all $v \in V_S$ that has an $e \in E_S$ with $\mathsf{lab}_S(e) = \mathsf{next}$ and $\mathsf{att}_S(e) = vw$ for some $w \in V_S$, there is a $d \in E_C$ with $\mathsf{lab}_C(d) =$ '$\Rightarrow$' and $\mathsf{att}_C(d) = vu$ for some $u \in V_M$.

Note that a stack graph S is a model graph, too. Let N be the set of stack names of S. Then we can define a model over a signature $\langle N \cup \mathsf{next}, \mathsf{arity}\rangle$ (where arity assigns 2 to everything), of which the model graph is S. However, we will assume that the label spaces of the stack graph and model graph are disjoint, so that we can distinguish between the two.

An illustration of a context graph can be seen in figure 4.5. Note the two parts that can be easily distinguished: on the left is the stack graph, and on the right the model graph.

In the following, we will refer to the nodes of a context graph as *referent nodes* if they come from the model graph, and as *control nodes* if they come from the stack graph.

4.4 Using Rules to Represent Discourse

As I briefly stated above, our strategy will be to associate with each symbol of the CML-string a (unique) graph rewrite rule, and apply these rules to the context graph in sequence. Then, when all the rules have been applied, the resulting context graph represents the context of the discourse. In the rest of this thesis we will call this framework Graph Modification Logic (GML), and to graph rewrite rules used for modifying context graphs I will refer as GML-rules.

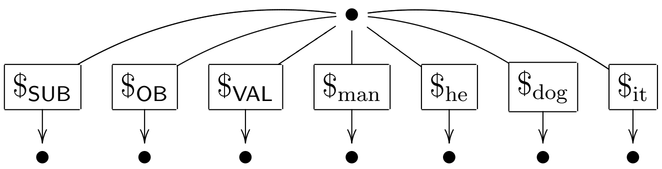


Figure 4.6: The initial context graph.

Definition 4.4.1 (GML-reduction). Let $\varphi = A_1 \dots A_n$ be a string of CML-symbols, and $r_1, \dots, r_n$ the GML-rules that are associated with these symbols (r_i is the rule for the symbol A_i). Then a GML-reduction of φ is a sequence of context graphs $G_0 \dots G_n$ such that for all $0 < i \leq n$: $G_{i-1} \rightarrow_{r_i} G_i$.

Usually, the starting point of the reduction sequence, G_0, will be the initial context graph, i.e. the context graph that consists of only empty stacks and no discourse referents. The initial context graph for the set of stack names N introduced above is presented in figure 4.6.

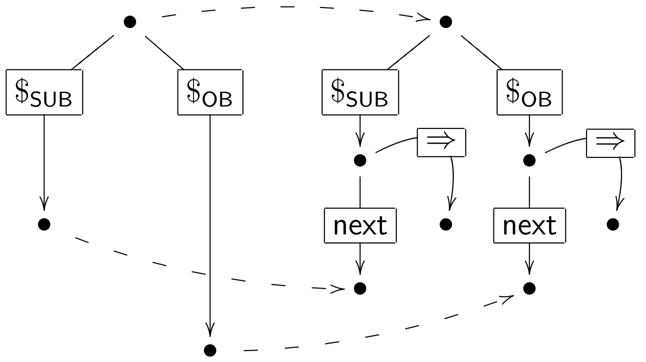
In the rest of this section, I will describe how we can build a GML-rule for CML-symbols from a number of linguistic classes, that has the desired effect on the context graph. The rules will use N as the set of stack names, but it is easy to extend the rules to use a larger set.

4.4.1 Relabellings

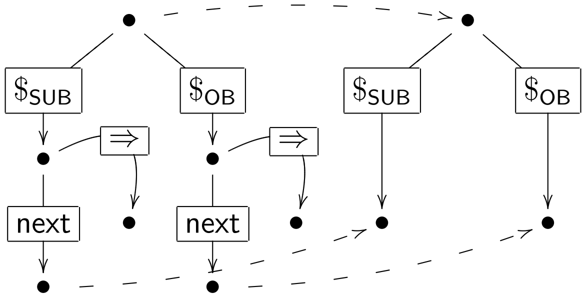
In CML, the relabelling symbols '[' and '⟨' do not *create* referents, they only relabel all the existing referents in the correspong argument stacks, in such a way that the label for the top element falls free. In other words, the symbols create a space in which a new referent can be stored later. The indefinite NP's create the referents, and the definite NP's add labels to existing referents to 'pull' them up to the top of the stack.

Here, I have chosen a different approach. The '[' and '⟨' symbols do not merely create space, they also create the referents. Now, the indefinite NP's just obtain these newly created referents, and the definite NP's identify them with an existing referent. The result is the same in both approaches.

Below is the GML-rule for '['. This rule pushes new referents to all the sentence level stacks (in our case $\$_{\mathsf{sub}}$ and $\$_{\mathsf{ob}}$). The reader will no doubt easily derive the rule for '⟨', which pushes new referents to the term level stacks (in our case only $\$_{\mathsf{val}}$).



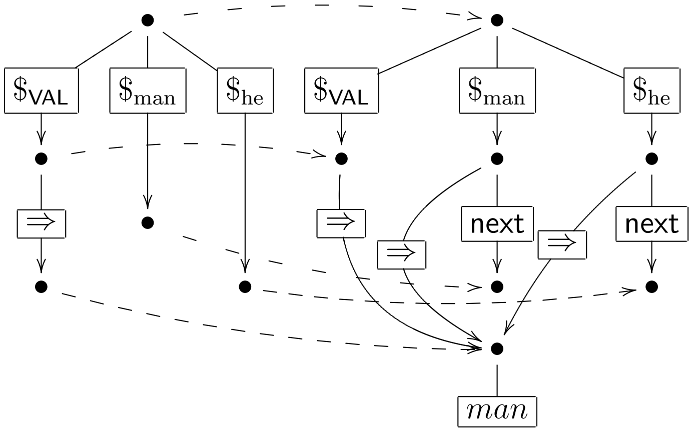
The symbols ']' and '⟩' clear up the stacks that were modified by '[' and '⟨'. They pop the top referent from the term level and the predicate level stacks, respectively. Note that both symbols do *not* destroy the discourse referents, they only disconnect them from the stacks, so that they are no longer 'visible' from the stack graph of the context graph. Below is the GML-rule for ']'. Again, the user will derive the rule for '⟩' easily.



4.4.2 Indefinite NP's

Natural language NP's have both a static and a dynamic component. In CML a definite or indefinite NP is a molecular symbol, that consists of relabelling symbols (the dynamic component), and a predicate symbol (the static component). In GML, we don't need to use molecular symbols: we can define a single GML-rule that handles both the dynamic and the static. I have chosen to construct such rules rather than to construct a separate rule for each of the atomic CML-symbols, because, of course, the ultimate goal is to create a graph-based semantics for natural language, rather than for a formal logic like CML. Note, however, that it is easy to specify GML-rules for the atomic CML-symbols also.

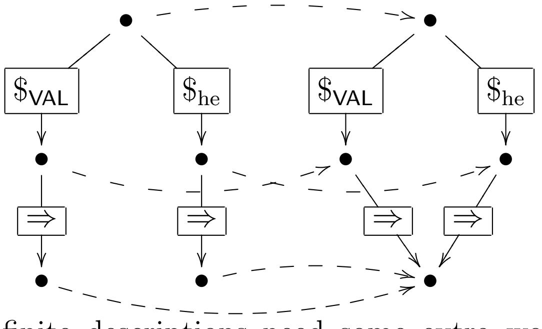
In our approach, indefinite NP's do not introduce new discourse referents as in CML. Instead, they push the discourse referent that was introduced by the term level '⟨' symbol onto one or more stacks (it depends on the actual indefinite to which stack(s) it is pushed). Also, an edge is attached to the referent, which tells us what kind of object it refers to. The GML-rule for the CML-symbol 'a-man' can be seen below. Note the two `push` rules (see figure 4.3) that are integrated into this rule. They do the job that $\lceil_{\texttt{man}}$ and $\lceil_{\texttt{he}}$ did in CML.



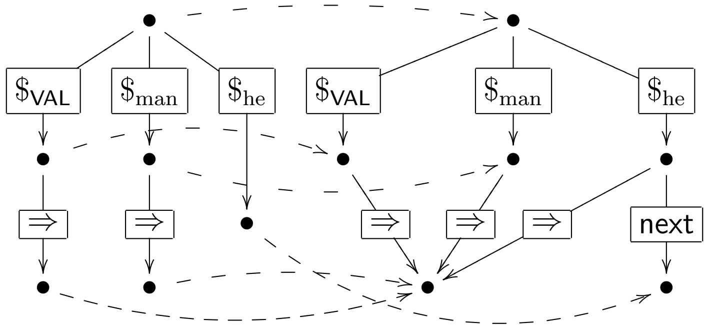
Of course, the rule for 'a-dog' will not push the value of the current term level to the $\$_{\text{man}}$ and $\$_{\text{he}}$ stacks, but to the $\$_{\text{dog}}$ and $\$_{\text{it}}$ stacks. It is also possible that the referent is pushed to more (or less) than two stacks.

4.4.3 Definite NP's

The simplest definite descriptions, which are mostly pronouns, identify two or more discourse referents, viz. the top referents of the $\$_{\mathsf{VAL}}$ and one or more other stacks, depending on the actual definite. For example, consider the GML-rule for 'he':



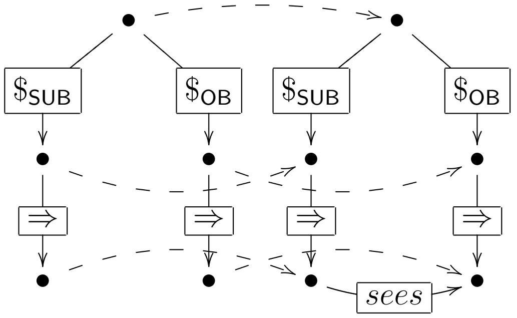
However, most definite descriptions need some extra work. For example, after uttering "the man" we make the man we talked about salient, and can refer back to him with "he". To model this phenomenon, we push the referent that results from identifying the top referent of the $\$_{\mathsf{VAL}}$ stack and the top referent of the $\$_{\text{man}}$ stack onto the $\$_{\text{he}}$ stack, as in the following GML-rule:



4.4.4 Verbs and Adjectives

Verbs and adjectives are handled alike: they only add information to the existing discourse referents, and do not create, push, pop or identify any referent. The only difference between verbs and adjectives is that they operate on a different level: verbs operate on the predicate level, and adjectives operate on the term level.

Below is the GML-rule for the verb 'sees'. The reader will be able to derive rules for adjectives and other verbs.



4.5 Example

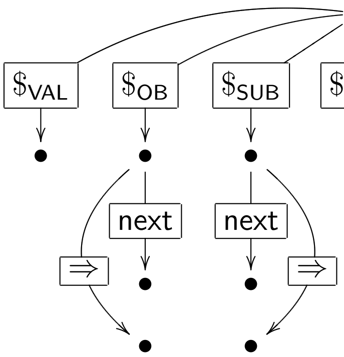
Now, let's work out an example. Let's take the discourse 'A man sees a dog that barks.' In CML$_1$, this discourse is represented as follows:

$$[\langle \text{a-man } \mathit{sub} \rangle\ \mathit{sees}\ \langle \text{a-dog } \mathit{ob}\ \rangle]$$

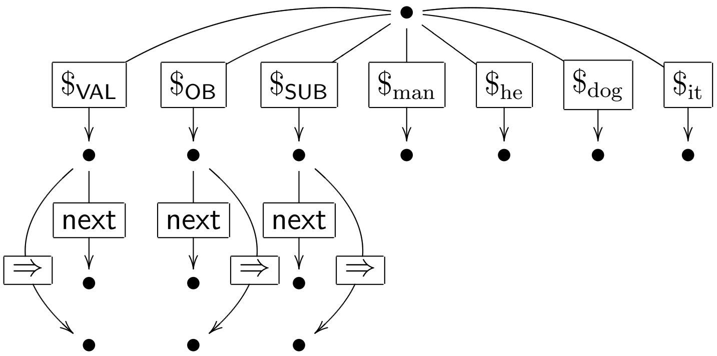
The graph rewrite rules of all symbols can be found in the previous section, or can be easily derived.

I have chosen to put the model graph in the lower part of the context graph rather than on the right side (as I did before), in order to keep the reference edges, labelled '⇒', as short as possible.

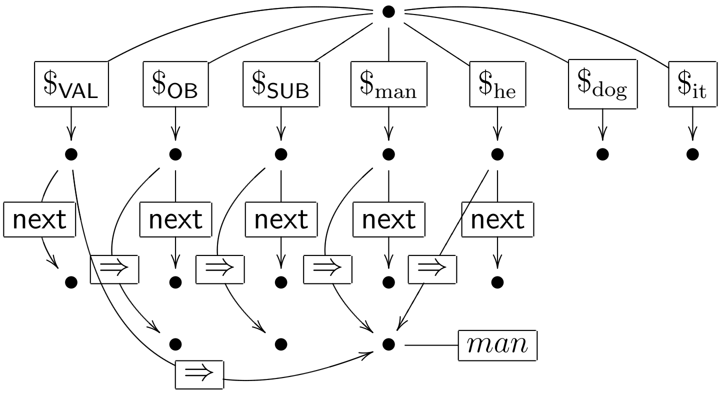
We begin by applying the rule for the CML-symbol '[' to the initial context graph, and we get:



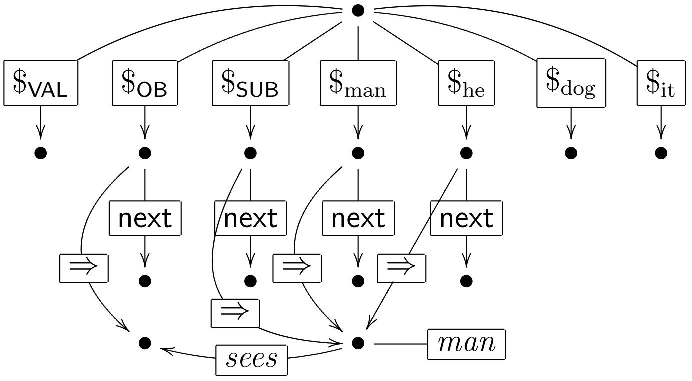
Next we apply the rule for '⟨' to this graph, which yields the following result:



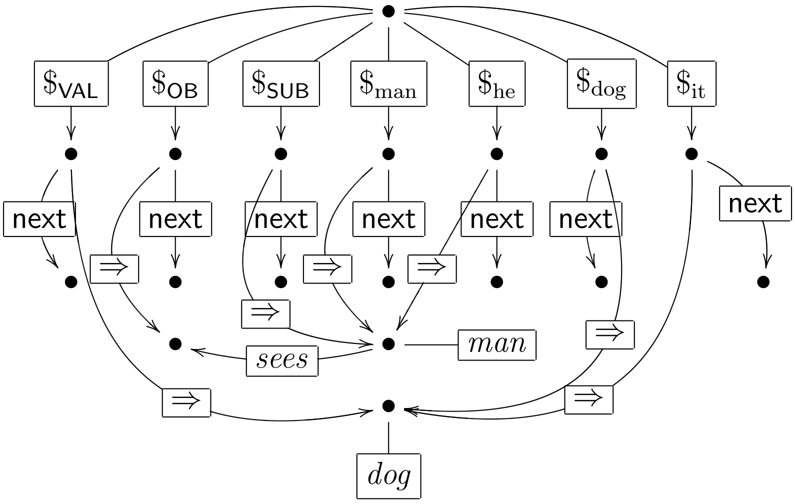
The next graph rewrite rule is the one for 'a-man', which pushes the referent at the top of the $\$_{\mathsf{VAL}}$ stack to the $\$_{\mathrm{man}}$ and $\$_{\mathrm{he}}$ stacks:



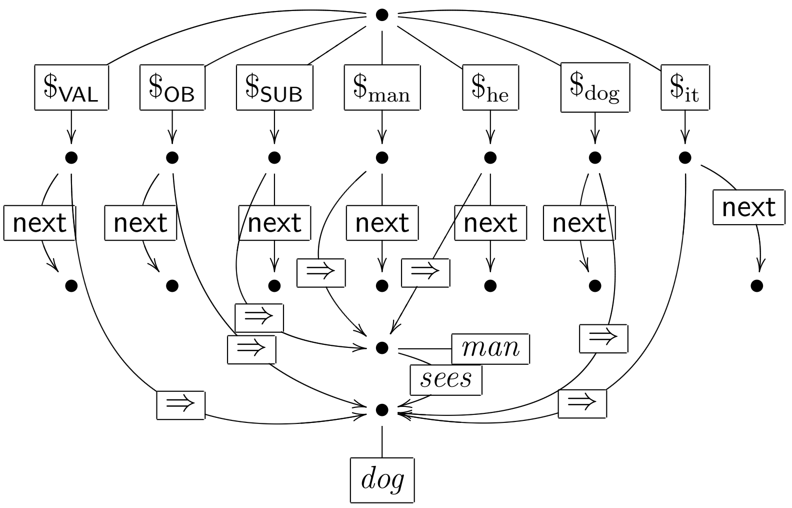
We apply the rule associated to '*sub*', which unifies two referents, then we apply the rule associated to '⟩', which pops the top referent from the $\$_{\mathsf{val}}$ stack, and then we apply the rule associated with '*sees*'. We now have the following context graph (note that I do not display the isolated nodes):



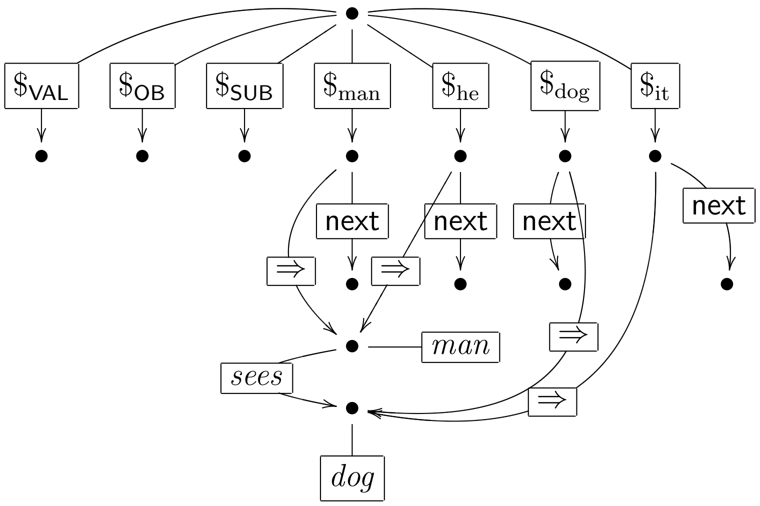
Again, we skip a few a steps, and apply the rules associated to '⟨' and 'a-dog', respectively, which results in the following context graph:



The rule associated to '*ob*' identifies the top referents of the $\$_{\mathsf{VAL}}$ and $\$_{\mathsf{OB}}$ stacks.



Finally, the last two rules, associated to '⟩' and ']', respectively, clear up the term-level and predicate-level stacks.



In the lower part of this context graph you can see the model graph, which is exactly what we draw intuitively to represent the discourse.

4.6 Building a DRS

When we have applied all the graph rewrite rules, the result is a context graph that represents the discourse. Context graphs can be trivially converted into DRS's. Since all the information about the world is in the model graph, we only use that part of the context graph and ignore the stack graph.

Definition 4.6.1 (DRS of a context graph). Let $C = \langle V_C, E_C, \mathsf{lab}_C, \mathsf{att}_C \rangle$ be a context graph that consists of a model graph $M = \langle V_M, E_M, \mathsf{lab}_M, \mathsf{att}_M \rangle$ and a stack graph S. Then the DRS $\mathsf{drs}(C) = \langle R, \mathcal{C} \rangle$ for C is given by:

- $R = V_M$
- $\mathcal{C} = \{P(x_1, \ldots, x_n) \mid \exists e \in E_M : \mathsf{lab}(e) = P \wedge \mathsf{att}(e) = x_1 \ldots x_n\}$

For example, if we apply this technique to the final context graph of the example above (we assume that the top referent of the model graph is called x, and the other referent y), we get the desired DRS:

$$\begin{array}{|l|}\hline x,y \\ \hline man(x) \\ dog(y) \\ sees(x,y) \\ \hline \end{array}$$

We can determine the truth of the context graph C in a model by checking if the DRS $\mathsf{drs}(C)$ is true in that model. This, however, is a detour. We will define truth directly in graph terminology, in the following way:

Definition 4.6.2 (Truth). Let C be a context graph, consisting of a model graph M and a stack graph S. Then C is true in a model $\mathcal{M}$ iff there exists a homomorphism from M to the model graph for $\mathcal{M}$.

The homomorphism plays the role of an assignment from the referents to the domain.

4.7 Uninitialized Referents

The system presented in this chapter fails to produce a correct DRS of the discourse “He walks.” (CML itself has a comparable problem). In CML this discourse is represented as follows:

$$[\langle \text{he } \mathit{sub} \rangle \ \mathit{walks}]$$

When we start with the initial context graph, by the time we want to apply the GML-rule associated with ‘he’, we will see that there is no referent on the $\$_{\text{he}}$ stack, so the rule cannot be applied. We understand the discourse, however, so it should have a DRS. Of course, the system does not forbid to begin with a context graph that contains a pre-initialized $\$_{\text{he}}$ stack. But still, the resulting DRS will then be:

x
$walks(x)$

which is incorrect, because the referent x is introduced in the DRS, but in the original discourse no referent is introduced at all. The correct DRS would be:

$walks(x)$

Below I will briefly sketch a possible solution to this problem, along the lines of Albert Visser, in an unfinished successor of [Vis01].

The general idea is to apply the rules to the smallest context graph to which the rules can be successfully applied in sequence. The referents that exist in this initial context graph are marked. We apply the GML-rules as always, but when constructing the DRS, we only put the unmarked (new) referents in the top box. We are now going to look at this in some more detail.

Definition 4.7.1 (Sufficient Context Graph). Let φ be a string of CML-symbols.

(i) A context graph C is sufficient for φ, iff there exists a GML-reduction $G_0 \dots G_n$ for φ such that $G_0 = C$.

(ii) A *minimal sufficient context graph* for φ is a context graph C which is sufficient for φ, such that there is no context graph $D \neq C$ which is sufficient for φ and is homomorphic to C.

Theorem 1. *For every string φ of CML-symbols, there exists a sufficient context graph C.*

I will not give a formal proof of the above theorem. It is, however, not difficult to see it's true intuitively. If you look at the GML-rules, you notice that the only reason that a rule cannot be applied, is that there are not enough referents on a specific stack. So, if a CML-string φ consists of n CML-symbols, a sufficient context graph for φ would be the one that has n referents on every stack (this is in most cases *not* a minimal sufficient context graph for φ).

Now we define a marking operator, $(\cdot)^{\#}$, that adds unary edges to the model part of the context graph, labelled '#'. We define for an arbitrary context graph $C = \langle V_C, E_C, \mathsf{lab}_C, \mathsf{att}_C \rangle$, which is built up from a stack graph S and a model graph $M = \langle V_M, E_M, \mathsf{lab}_M, \mathsf{att}_M \rangle$, where $V_M = \{v_1, \dots, v_n\}$:

$C^{\#} = \langle V_{C\#}, E_{C\#}, \mathsf{lab}_{C\#}, \mathsf{att}_{C\#} \rangle$, where:

- $V_{C\#} = V_C$
- $E_{C\#} = E_C \oplus \{e_1, \ldots, e_n\}$
- $\mathsf{lab}_{C\#} = \mathsf{lab}_C \oplus \{\langle e_i, \# \rangle \mid 0 < i \leq n\}$
- $\mathsf{att}_{C\#} = \mathsf{att}_C \oplus \{\langle e_i, v_i \rangle \mid 0 < i \leq n\}$

Note that since the '#'-edges do not appear in the left-hand side of any GML-rule, these edges are never removed. So, once a referent is marked, it stays marked. When two unmarked referents are identified, the result is still unmarked; when an unmarked ('new') referent is identified with a marked ('old') referent, the result is marked (note is this is the desired effect), and when two marked referents are identified, the result is marked (actually, in the last case, there are two or more '#'-edges attached to the same node, but that is not a problem).

Now, in the construction of a DRS from a context graph, we omit the marked referents in the top box of the DRS. The conditions are created in the same way as before (only we do not create conditions of the '#'-edges, of course). To summarize, creating a DRS of a CML-string φ is now done as follows:

1. Find a minimum context graph C for φ.

2. Find the GML-reduction $G_0 \ldots G_n$ for φ, such that $G_0 = C^{\#}$.

3. Transform G_n into a DRS as stated above.

In the following chapter we will not use the technique from this section, but, in order to keep things simple, we stick to original version of GML.

4.8 Summary

In this chapter we have developed GML, a framework for doing dynamic semantics by using techniques from graph rewriting. The framework works as follows.

First we have defined context graphs, which are used to represent information states in the form of a graph. A context graph consists of two parts. The model graph models the information about the world that we have so far. Nodes in the model graph represent the referents. The stack graph models the stacks. The nodes in the stack graph are analogous to the labels of CML. The empty, or initial, context graph contains a number of empty stacks and no referents (i.e. the model graph is empty).

To each possible CML-symbol a GML-rule is associated, which are applied to the initial context graph in sequence. In the end, we have a context graph which represent the entire discourse. We determine the truth of this context graph in a model, by finding a homomorphism from the model graph of the context graph to the graph representation of this model. Also, we can easily convert a context graph to a DRS by taking the nodes as the referents of the DRS, and the edges as the conditions.

Chapter 5

The Implementation in Graph Rewriting

5.1 Introduction

In the previous chapter we have developed a method of representing discourse with hypergraphs. Although our method works quite well, it is hardly graph rewriting: *we*, and not the mechanisms built into graph rewriting, choose the order in which the rules are applied.

In the first few sections of this chapter we will design a graph rewrite system, S_{GML}, that will reduce an (appropriate representation of a) CML-string to a context graph without our help. This graph rewrite system can be seen as an implementation of the mechanisms developed in the previous chapter. In fact, actually, we 'cheat' by hard-coding the order in which the rules are applied into the initial graph, by adding tags.

Finally, in section 5.4 we will extend the graph rewrite system in an important way, that was not possible with GML. We will create a system that does not require us to create all the stacks in the beginning. Instead, the system will automatically add a stack when this stack is required by a rewrite rule, but does not yet exist.

5.2 The representation of a CML-string.

As stated above, we are going to develop a graph rewrite system that is going to do the work that we did by hand in the previous chapter. So, the system must reduce a CML-string to a context graph. In order to produce such a system, of course, we have to find a way of representing a CML string by means of a hypergraph.

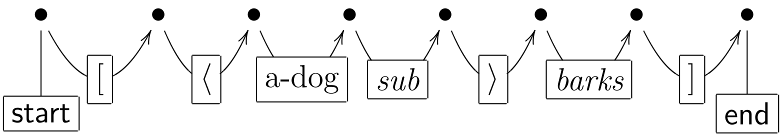


Figure 5.1: The graph representation of a CML-string.

The most intuitive way to do that, is to represent the string as a series of consecutive nodes, connected by edges that bear the symbols as their label. In order to quickly locate the beginning and the end of the string, two unary edges, labelled start and end, are connected to the first and the last node of the sequence, respectively. Consider, for example, the following CML-string, which is represented as in figure 5.1.

$$[\langle \text{a-dog}\ \textit{sub}\rangle\ \textit{barks}]$$

We are going to develop a graph rewrite system, S_{GML}, which will reduce 'string' graphs in this style to context graphs. Note that the styles of the input and output graphs of the system are different: the 'input' graph is a string graph, while the 'output' graph (i.e. the normal form) is a context graph. We need, therefore, an intermediary form that has properties of both string graphs and context graphs. There is a rule that converts the input string graph in the intermediate form, and there is a rule that converts the intermediary form into a context graph at the end of the reduction sequence.

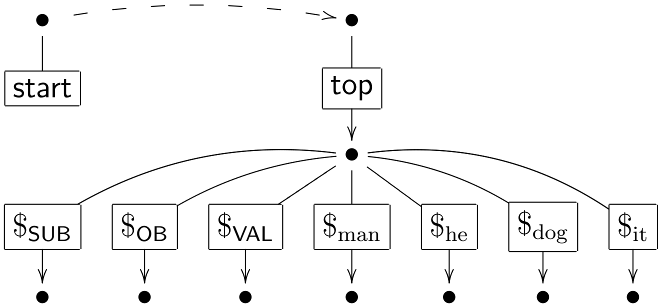
The intermediary form that we are going to use consists of the part of the string which still needs to be processed, and the current state of the context graph which is being built. The two parts are connected by a binary top node. An illustration of the intermediary form is in figure 5.2. As is clear, we have extended the graph signature from the previous chapter, the reader will easily specify the signature herself.

5.3 Rules for String Processing

In this section we are doing the real work of this chapter: designing the graph rewrite rules of S_{GML}.

5.3.1 The Start Rule

First we need a graph rewrite rule that converts a string graph in the intermediary representation. We desire, of course, that this rule can only be applied to a string graph, not to the intermediary representation. So, it matches the start edge, replaces it with the binary top edge and adds an empty context representation. The following rule, which we will call the *start rule* has that effect.



Note that, provided that the input graph is a well-formed string graph (i.e. it has only one edge labelled 'start'), this rule can be applied only once in a reduction sequence.

5.3.2 More Rules

Now we are going to build the graph rewrite rules that do the real work. These are analogous to the GML-rules of chapter 4. In fact, it will seem, we can translate any GML-rule to a rule that does the same job in the framework of this chapter (the converse does not hold, because, among other things, there is no GML counterpart of the start rule).

But before we turn to the systematic way of translating GML-rules into their S_{GML} counterparts, we are going to look what these new rules look like. To do that, we design the rule that handles the CML-symbol 'a-man'.

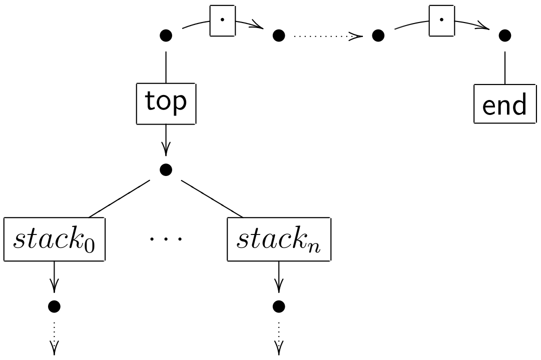
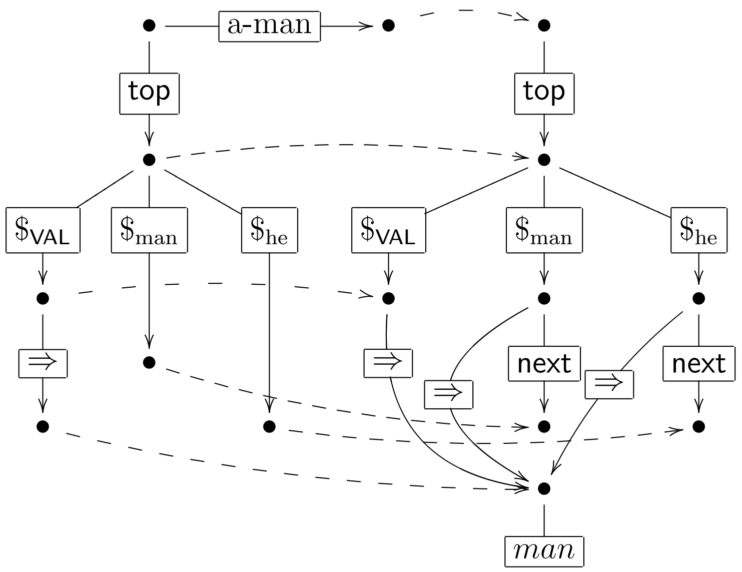


Figure 5.2: Illustration of the intermediary representation.

What must this rule do? First, it may only be applied if the currently processed CML-symbol is 'a-man', i.e. the source node of the edge labelled 'top' is also the source node of an edge labelled 'a-man'. We need to make modifications to the part of the graph representing the current state of the context, so we match the relevant parts of that also. In the right-hand side of the graph the following things will be changed: the currently selected CML-symbol is the following symbol; and the top referent of the $\$_{\mathsf{VAL}}$ stack is pushed to the $\$_{\mathrm{man}}$ and $\$_{\mathrm{he}}$ stacks as well. The result is the following graph rewrite rule:



When we compare the above rule with the GML-rule for 'a-man', on page 43, we see that they are the same except for the bit at the top. It is not surprising, therefore, that we can easily translate any GML-rule to a rule of S_{GML}. We add two new nodes to the left side of the rule, say a and b, and we add an edge labelled 'top' from a to the root of the stack graph of the left side, and an edge labelled with the CML-symbol in question from a to b. The right side is modified by adding one new node, $\{b\}$ (this corresponds to the coding we assumed in section 3.4.3), and an edge labelled 'top' from $\{b\}$ to the root node. The identification function is updated in the obvious way. Before I show the formal treatment of the above, for purists among you I have to say that we need to add a second node to the right-hand side, $\{a\}$ (an isolated node, which is therefore not in the graphical representation).

Algorithm 5.3.1. Let φ be a CML-symbol, let $r = \langle L_r, R_r, \alpha_r\rangle$ be the GML-rule that is associated with it, and let v be the root node of L_r and w the root node of R_r. Then we can calculate the rule $s = \langle L_s, R_s, \alpha_s\rangle$ of S_{GML} as follows:

(i) a. $V_{L_s} \leftarrow V_{L_r} \oplus \{a, b\}$.

b. $E_{L_s} \leftarrow E_{L_r} \oplus \{e_L, c\}$.

c. $\mathsf{lab}_{L_s} \leftarrow \mathsf{lab}_{L_r} \cup \{\langle e_L, \mathsf{top}\rangle, \langle c, \varphi\rangle\}$

d. $\mathsf{att}_{L_s} \leftarrow \mathsf{att}_{L_r} \cup \{\langle e_L, av\rangle, \langle c, ab\rangle\}$.

(ii) a. $V_{R_s} \leftarrow V_{R_r} \oplus \{\{a\}, \{b\}\}$.

b. $E_{R_s} \leftarrow E_{R_r} \oplus \{e_R\}$.

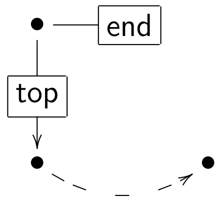
c. $\mathsf{lab}_{R_s} \leftarrow \mathsf{lab}_{R_r} \cup \{\langle e_R, \mathsf{top}\rangle\}$

d. $\mathsf{att}_{R_s} \leftarrow \mathsf{att}_{R_r} \cup \{\langle e_R, \{b\}w\rangle\}$.

(iii) $\alpha_s \leftarrow \alpha_r \cup \{\langle a, \{a\}\rangle, \langle b, \{b\}\rangle\}$.

5.3.3 The End Rule

When there are no more CML-symbols in the string part of the intermediary representation, we do not yet have a context graph: there is still an edge labelled '`top`' and an edge labelled '`end`', both of which may not be part of a context graph. The end rule has no other purpose than to remove these two edges, and produce a real context graph. The following rules does just that:



When the end rule has been applied, we have a context graph, which can be transformed into a DRS in the way described in section 4.6.

5.3.4 An Example

I will not work out an example in this chapter. Instead, I refer to Thor [Bru01], a Java program which does graph rewriting. In the standard distribution package of Thor, you will find the file `S_GML.grs`, which contains all the rules of this chapter. The graphs `discourse`N`.g`, in which N is an integer from 1 to 3 are string graphs of some typical examples you may try.

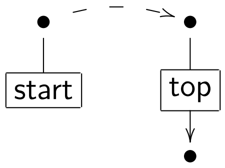
5.4 Adding Flexibility

In chapter 4, all the stacks that could possibly be used in the discourse had to be present from the start, and in sections 5.2 and 5.3, all the stacks were created in one time by the start rule. In both cases, there was no way to create new stacks in the process of analyzing the discourse.

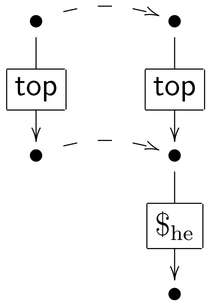
In the examples we conveniently created just those stacks that we needed to represent the discourse. In reality, however, there is no way to find out in advance what stacks are going to be needed to represent a starting discourse. So, in a general system, we will have to create all possible stacks in the beginning, and most of those we're not even going to use. What a waste. In this section we are going to develop a graph rewrite system that automatically creates a stack, when it is attempted to push an element to a stack that does not already exist.

5.4.1 The first attempt

If the stacks are going to be created on-the-fly, we need a different start rule, of course. The new start rule creates no stacks, it just changes the unary start edge into a binary top edge, as follows:



For each stack name we need a rule that create a stack of that name. This is not too difficult. A rule that creates a new $\$_{\text{he}}$ stack looks like:



The above rule creates a $\$_{\text{he}}$ stack. However, there is no way to control when the rule is applied: any rule that can be applied may be chosen to be applied. This introduces two problems. Firstly, the rule may be applied when the $\$_{\text{he}}$ stack is not required, so we would still have a lot of useless stacks. And secondly, the rule may also be applied if a $\$_{\text{he}}$ stack does already exist; in that case there would be two of the same stacks, which is of course a problem.

The second problem is a greater problem, so we are going to solve it first. We will do that by using graph rewriting with application conditions, as introduced in section 3.5. Then, solving the problem is almost trivial. Then, we are going to look into the first problem.

5.4.2 Ruling out Duplicate Stacks

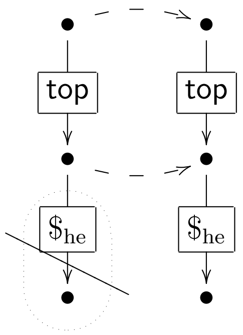
What we want, is to modify the graph rewrite rule above so that it can only be applied if a stack of the specified name does not yet exist. If possible, we want to maintain the current representation of the stacks. I will first give the reasons why we cannot do that with normal graph rewriting, and then I will formulate the graph rewrite rule with application condition that does the job with ease.

Since a normal graph rewrite rule can only be applied to a graph if there exists a homomorphism from the left-hand side of the rule to that graph, it is impossible to define a single rule that can only be applied if an edge of the left-hand side has a *different* label than the corresponding edge in the graph. But, because we have fixed the set of possible labels in advance, we can define a *set* of graph rewrite rules that has the same effect (we just enumerate all the different labels).

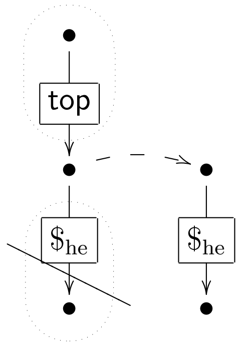
However, in our present representation of the stacks, it is not possible to create a set of rules, of which one can be applied if a *stack* with a specific name does not exist: if there are two or more stacks with different names, there is *always* a stack that has a different name than the name whish is searched: if we have a set of graph rewrite rules of which one can be applied if a $\$_{\text{he}}$ stack does not exists, and there exists a $\$_{\text{he}}$ stack, we just find a homomorphism that points to one of the other stacks with a different name, and apply one of the rules in the set after all. So, if we want to maintain the present representation, we have to use the graph rewrite rules with application conditions from section 3.5[1].

With this form of graph rewriting, it is not difficult anymore to solve the problem: we just have to find an appropriate (negative) application condition. This application condition, of course, states that a stack of the specified name does not already exist. For the $\$_{\text{man}}$ stack, we get the following graph rewrite rule with one negative constraint:

[1] We could also modify the representation and stick to general graph rewriting. However, not only would the representation get too complex, we would also need a lot of rules. Therefore, I prefer keeping our old representation and extending the notion of graph rewriting.



Note that we also could have used a positive constraint to formalize the fact that the top edge is required. We would have used the following rule (which is equivalent to the above rule):

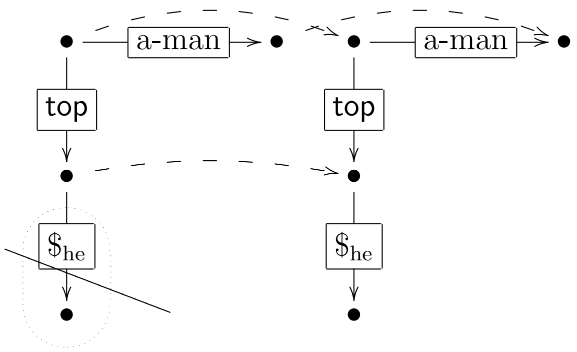


In the following, I will use the former format (without positive constraints), to be more consistent with the previous part of this chapter, in which I used normal graph rewriting.

5.4.3 Ruling out Useless Stacks

Now we are going to work on the first problem I raised. How are we going to make sure that stacks are not created when they are not needed? In other words: how can we see whether a stack is required or not? I find this problem of a lesser magnitude than the previous one: when two stacks with the same name have been created, wrong results may be produced because there is no way to discover which stack was created most recently. So, the top referent of the wrong stack may be selected as if it were the most recent referent. However, when a stack is created that is not required, we only have too many stacks: we do too much work. But the final DRS will be the same.

Still, the solution to this problem is quite simple: we know exactly which CML-symbols push referents to which stacks. So, we define multiple rules that create, for example, a $\$_{he}$ stack; one for each CML-symbol that pushes a referent to that stack. One of the rules is:



And we define the same rule with the symbols ‘the-man’, ‘a-boy’, etc. in the top part.

5.5 Summary

In this chapter we have built a graph rewrite system that does dynamic semantics exactly like GML. The difference is, that now we do not apply the rules to the context graph as we go along (‘at run time’), but we build the order of the rules into the graph ‘at design time’. We do this by a string of CML-symbols that uniquely encode the order in which the rules must be applied. Now, the normal forms of the graph rewrite system are context graphs that can be converted into DRS as in the previous chapter.

In the second half of the chapter we have extended this system to automatically create stacks when needed. For this, we have use graph rewriting with application conditions. Especially, the possibility to define negative constraints proved very useful.

Chapter 6

Conclusion and Final Remarks

As became clear in chapters 4 and 5, it is possible to calculate a DRS for a discourse represented in CML, by using techniques from graph rewriting. In this thesis, I have presented two frameworks that do the job.

The first framework, which I called GML, associates to each (atomic or molecular) CML-symbol a single graph rewrite rule. Information states are modelled in a context graph. The rules are then applied to an initial context graph in sequence, resulting in a context graph that represents the entire discourse.

The second framework, which a called S_{GML}, is a graph rewrite system which takes a graph representation of a CML-string as input, and reduces it to a normal form in the form of a context graph. The rules of the graph rewrite system are derived from the GML-rules. Finally, I gave an extention to this graph rewrite system, that automatically created stacks when required.

In both frameworks, the resulting context graph can be trivially transformed into a DRS (in a way, context graphs are DRS's in a slightly different notation).

Although both do the job, they have different advantages and disadvantages. I give a quick comparison of GML (chapter 4) and S_{GML} (chapter 5).

- In the present form of GML, it is impossible to define rules that flexibly create stacks when needed, as we did with relative ease in S_{GML}. So, all stack must be created in advance.

- S_{GML} can only be used when the discourse is finished. So, we cannot calculate a DRS of a discourse when it still going on.

In other words, neither of the systems is 100% perfect. Still, the results of my project show that it is possible to use graph rewriting for dynamic semantics.

Because of the intuitive nature of hypergraphs, in my opinion (extensions to) the graph-based systems developed in this thesis are a good alternative to encoding (representations of) meaning directly in set theory.

Still, some things need to be addressed in the future. Here are a few topics that I couldn't look into, but are, in my opinion, quite interesting:

Negation and Implication I have only paid attention to positive discourses, i.e. discourses without negation, implication and universal quantification. Of course, these topics are very important, and must be addressed in the future.

Determining the truth at the same time In this thesis, we create a DRS(-like graph) with graph rewriting. If we have done that, we can see if it is true in a model, by finding an homomorphism from it to the (model graph of a) model. It would be very interesting to see how we can calculate the truth of a discourse during the creation of the DRS. I think that the parallel nature of graph rewriting will truly come to its right in this way.

Composition of Context Graphs Context graphs are of the same nature as positive DRS's. It would be interesting to know whether they share other properties as well. In particular: is it possible to calculate the composition of two context graphs by using graph rewriting, like we can calculate the composition of two DRS's.

I am confident, that extensions to GML and S_{GML} can be developed that solve the above problems.

Appendix A

Overview of Notations

General

iff	if and only if
ϵ	the empty sequence
A^*	the set of sequences of A
A^+	$A^* - \{\epsilon\}$
$A \oplus B$	the disjoint union of A and B
$\emptyset$	the empty set, empty labelled set, empty function, etc.
$f \circ g$	composition of functions, $(f \circ g)(x) = f(g(x))$

Dynamic Semantics

$g[x_0, \ldots, x_n]h$	assignment g differs from assignment h at most in the values it assigns to $x_0, \ldots, x_n$
$\models_{\mathcal{M},g} \varphi$	condition φ is true in model $\mathcal{M}$ w.r.t. assignment g
$h \models_{\mathcal{M},g} \Phi$	assignment h is a verifying embedding for DRS Φ in model $\mathcal{M}$ w.r.t. assignment g
ℓ, Σ-set	labelled set based on label space Σ
$\nu + \mu$	for multisets ν and μ: $(\nu + \mu)(x) = \nu(x) + \mu(x)$ for all x
$\nu \leq \mu$	for multisets ν and μ: $\nu \leq \mu$ iff $\nu(x) \leq \mu(x)$ for all x
$\xi \star E$	labelled set ξ is relabelled by E

Graph Rewriting

$f[S]$	$\{f(x) \mid x \in S\}$

$\mathrm{Dom}(f)$	the domain of f
$\mathrm{Rng}(f)$	the range of f, $\mathrm{Rng}(f) = f[\mathrm{Dom}(f)]$
$\bigcup A$	$\{x \mid \exists B \in A : x \in B\}$
$G \rightarrow_{r,\varphi} H$	H is the result of applying rule r to graph G via a homomorphism φ
$G \rightarrow^{\mathtt{cond}}_{\hat{r},\varphi} H$	same as $G \rightarrow_{\hat{r},\varphi} H$, with graph rewr. with app. cond.
$G \Rightarrow_S H$	$G \rightarrow_{r,\varphi} H$ for some $r \in S$ and some homomorphism φ
$\Rightarrow^*_S$	reflexive and transitive closure of $\Rightarrow_S$.

Bibliography

[BCK01] Paolo Baldan, Andrea Corradini, and Barbara König. A static analysis technique for graph transformation systems. Dipartimento di Informatica, Univerità di Pisa, Italia, 2001.

[Bru01] H. J. Sander Bruggink. Thor online user's manual. Electronical publication, 2001. `http://thorhome.tripod.com`.

[Can93] Ronnie Cann. *Formal Semantics*. Cambridge Textbooks in Linguistics. Cambridge University Press, 1993.

[Gam91] L. T. F. Gamut. *Logic, Language and Meaning*, volume II. The University of Chicago Press, 1991.

[GS91] Jeroen Groenendijk and Martin Stokhof. Dynamic predicate logic. *Linguistics and Philosophy*, 14:39–100, 1991.

[GS98] Jeroen Groenendijk and Marting Stokhof. Betekenis in beweging. *Algemeen Nederlands Tijdschrift voor Wijsbegeerte, 90–1*, pages 26–53, 1998.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996.

[KKSdV93] J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries. An introduction to term graph rewriting. In M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors, *Term Graph Rewriting, Theory and Practice*, pages 1–12. John Wiley and Sons, 1993.

[KR93] Hans Kamp and Uwe Reyle. *From Discourse to Logic: Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*. Studies in Linguistics and Philosophy. Kluwer Academic Publishers, 1993.

[Plu98] D. Plump. *Term Graph Rewriting*, chapter I. Universität Bremen, Fachbereich Mathematik und Informatik, 1998.

[Rus96] Bertrand Russel. Descriptions. In *The Philosophy of Language, Third Edition*, pages 208–214. Oxford University Press, 1996.

[SS94] J. Scheerder and J. Springintveld. Proof graphs. Master's thesis, University of Utrecht, Department of Philosophy, 1994.

[Vis01] Albert Visser. Context modification in action, lecture notes, 8 January 2001. Draft.

Index