When a Sentence falls apart...
Using heuristically guided dead end detection in sentence realization

Bachelor's Thesis

submitted by
Maximilian Schwenger

Reviewers
Prof. Dr. Jörg Hoffmann
Dr. Vera Demberg

Saarland University
Department of Computer Science
Foundations of Artificial Intelligence

September 2015

Acknowledgement

Introductory I want to thank my advisors Dr. Álvaro Torralba and David Howcroft who were helping and supporting me whenever needed. Furthermore thank you to my supervisors Prof. Dr. Jörg Hoffman and Dr. Vera Demberg who made this work possible as well as the Foundation of Artificial Intelligence group for general support and Julia Wichlacz for proof reading.

Declaration

I certify that all material in this thesis which is not my own work has been identified with appropriate acknowledgment and referencing and I also certify that no material is included for which a degree has previously been conferred upon me.

Saarbrücken, September 2015 Maximilian Schwenger

Abstract

In this thesis we approach the problem of automatically generating naturally sounding sentences. We discover the similarities between a search based realization process and searches in the field of artificial intelligence. These allow us to compile the problem of sentence realization into a representation on which we can use well established techniques from the automatic planning community. We introduce a polynomially space and time bound algorithm and proof its correctness. After various experiments we gathered empirical data, which allows an analysis regarding its practical relevance. Conclusively, we propose ways to further improve the process in the future.

Contents

Chapter 1

Introduction

Since the 1950s, natural language processing has been an active field of research as sub-area of artificial intelligence. The overall goal is to make machines able to communicate with humans, which means understanding as well as generating natural language. This seems to be a simple task because for humans processing language is a crucial part of the everyday life since their early childhood and they do this without spending a lot of thoughts on it. However, it quickly turned out to be not that simple. Early projects included primitive language processing by pattern matching, as seen in ELIZA [14], a program pretending to be a psychiatrist. Even though in 1966, when it was first presented, people were astonished by its realism, for today's standards, ELIZA is easily recognizable as a machine. It is overshadowed by tools like the Google's search engine now, which - in its beginnings - was merely searching for keywords found in a sequence of words and is now capable of answering natural questions in text as well as spoken form. Nowadays, tools are developed with the purpose of allowing a discourse between human and computer in a natural way.

In recent years, mobile devices became more and more ubiquitous with the intend to assisting the user though their whole day. So highly domain specific approaches like ELIZA's became less interesting; the need for broad coverage tools grew. Moreover, dialog systems should be able to react according to the user's situation, i.e. refrain from complex syntax when the user is already in a stress-situation and generate utterances that are easy to understand and rather contain redundant information than risking a misunderstanding. This level of complexity in terms of conveyed information is called information density. This information density should be flexible and adaptive while the process of obtaining an utterance has to be fast enough to simulate a discourse that feels natural to the user.

In this thesis we try to approach this problem with techniques from the field of automatic planning, another sub-area of artificial intelligence, in which a generic problem is solved automatically by performing a search for a sequence of steps leading to the desired solution. We do this by adapting the problem of *sentence realization*, i.e. *how* we say something, to a problem representation on which planning techniques can be used. A realization framework called OpenCCG uses a search based realization process which is thus naturally similar to planning problems. So we want to make advanced research results from the planning community usable for search based realization processes allowing to find solutions faster and aim for more complex results like generating utterances featuring a specific information density.

For this, we introduce the necessary background knowledge in Chapter 2 allowing us to present a compilation technique transforming the OpenCCG problem into a planning problem in Chapter 3. However, this is a theoretical approach, so for practical use some further adaptations are required which are discussed in Section 3.6. We also present and discuss experiments using said transformation and their results in Chapter 4. Lastly, a perspective on future work taking the experiment's results into account is presented in Chapter 5.

Chapter 2

Background

In this thesis we combine the concept of *automatic planning* with the problem of realizing sentences using the grammar formalism *CCG*. To understand this we will give a quick overview of what CCG is and how we can use it on the example of *OpenCCG*, a state of the art tool to parse and realize sentences based on a CCG lexicon. Afterwards we introduce the basic definitions of a planning task as well as heuristic functions, especially the h^{max} heuristic.

2.1 CCG

CCG (Combinator Categorial Grammar) is a grammar formalism, which - in a nutshell - assigns *syntactic categories* (or short *categories*) to words or sequences thereof and provides a set of rules to combine these. We call the combination of a word and its category a *lexical entry*. Categories can be either atomic (e.g. n) or complex, i.e. an atomic category has been combined with another category using the forward concatenation (/) or backward concatenation (\) operator. The rules are defined as follows where each rule exists in a forward (left) and a backward fashion (right):

Application rules	$\dfrac{A/B \quad B}{A}{>}$	$\dfrac{B \quad A\backslash B}{A}{<}$
Composition rules	$\dfrac{A/B \quad B/C}{A/C}{>}\mathbf{B}$	$\dfrac{A\backslash C \quad B\backslash A}{A\backslash C}{<}\mathbf{B}$
Type raise rules	$\dfrac{A}{T/(T\backslash A)}{>}\mathbf{T}$	$\dfrac{A}{T\backslash(T/A)}{<}\mathbf{T}$

On the example of the forward application rule, we can combine a category A/B with a category B to get A as a result.

When trying to parse a sentence, we use a *lexicon* which yields a lexical entry for a given word. We then try to combine these lexical entries until we get the syntactic category s, for *sentence*. Other common categories are n for *noun*, np for an *noun phrase*, pp for *prepositional phase*, and $punc$ for punctuation. The process is depicted in a parse tree.

Example 2.1.1.
Consider the sentence "I love the cup that Germany won". Its parse tree is illustrated

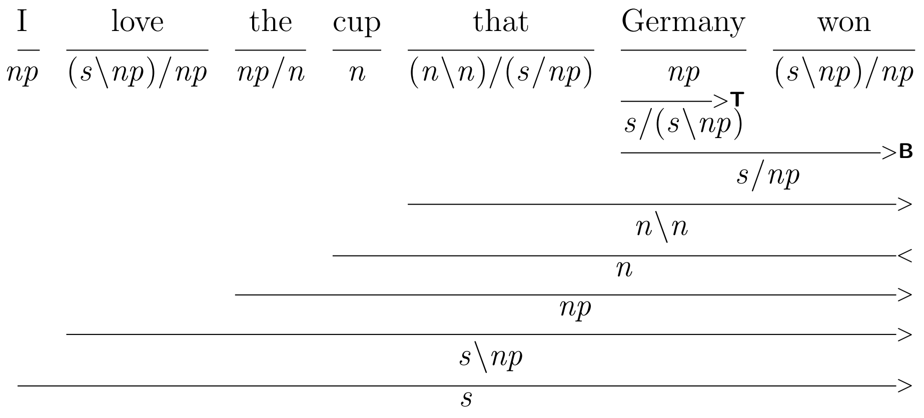

Figure 2.1: Parse tree for "I love the cup that Germany won"

in Fig. 2.1 where the topmost row is the sentence itself. Below each word there is its syntactic category given by the lexicon. First we raise Germany's category by type *s*. The result thereof is combined with "won"'s category by forward composition to acquire the category *s/np*. Combining this with the category of "that" via forward composition completes parsing the relative clause. One backward application followed by three forward applications completes the parsing process.

Note that the concatenation operators are left associative, so we can equivalently write *A/B/C* instead of *(A/B)/C*. Nonetheless, the following composition is possible:

$$\frac{A/B \quad B/C/D}{A/C/D} >\mathbf{B}$$

One advantage of CCG is the simplicity of its rules which is provided by moving complexity into the lexicon itself, i.e. into finding fitting categories for the words.

2.2 OpenCCG

In 1987, Mark Steedman proposed to use CCG to parse sentences with a chart-based algorithm [10]. Chart parsing is a dynamic programming technique introduced by Martin Kay [8] avoiding the need of backtracking when resolving the grammar's ambiguities. This technique is commonly used in computer linguistics, e.g. in the Early parser [2] and in *OpenCCG* [15].
The process of realizing a sentence can roughly be divided into three phases as depicted in Fig. 2.2.

Document Planning In this phase we figure out what we want to express, how to structure the information, and how to involve world knowledge.

Micro Planning The micro planner decides how information can be aggregated, how references can be used, and performs the lexicalization, i.e. decides what words can be used to express the semantics.

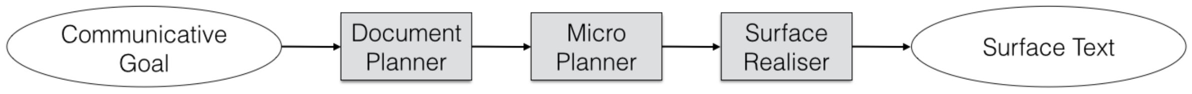

Figure 2.2: Schematic illustration of a realization process

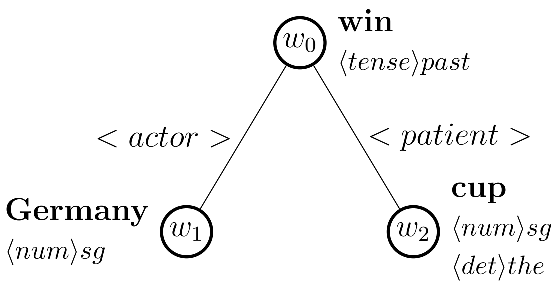

Figure 2.3: Graph representation of 2.1

Surface Realization Here, the process is finalized to form a grammatically valid sentence expressing the output of the document planner entirely. This can then be used as output.

There is no clear line between the last two phases, e.g. when the usage of a reference is not fixed in the micro planning phase, the surface realizer has to come up with a solution.

In this thesis we will solely focus on the third phase where we use OpenCCG to realize the input logical formula (LF), which is an HLDS (hybrid logic dependency semantics) term capturing everything we want to convey. To better understand this, we will run an example.

Consider we want to express, that “Germany won the cup” after the last championship match. An appropriate HLDS term can look like the following:

$$\begin{aligned} @_{w_0}(\mathbf{win} \wedge \langle \mathit{tense}\rangle \mathit{past} \\ &\wedge \langle \mathit{actor}\rangle(w_1 \wedge \mathbf{Germany} \wedge \langle \mathit{num}\rangle \mathit{sg}) \\ &\wedge \langle \mathit{patient}\rangle(w_2 \wedge \mathbf{cup} \wedge \langle \mathit{num}\rangle \mathit{sg} \wedge \langle \mathit{det}\rangle \mathit{the})) \end{aligned} \tag{2.1}$$

This is isomorphic to a more illustrative tree representation shown in Fig 2.2. Here we can see that the verb “win” has to have the linguistic feature ⟨*tense*⟩*past* and is depending on two arguments, its *actor* and *patient*. These again have linguistic features, namely both have to be singular and “cup” has to be accompanied by the determiner “the”.

At the beginning of the realization, we want to allow easier handling of the term, which in this case means faster comparisons in terms of equality and overlapping. For this, the term becomes *flattened*. This process leads to the LF being a conjunction of elementary predications (EPs), each representing a semantic item that has to be expressed. Flattening 2.1 will result in:

$$\begin{aligned} @_{w_0}\mathbf{win} \wedge @_{w_0}\langle \mathit{tense}\rangle \mathit{past} \wedge @_{w_0}\langle \mathit{actor}\rangle w_1 \wedge @_{w_0}\langle \mathit{patient}\rangle w_2 \\ &\wedge @_{w_1}\mathbf{germany} \wedge @_{w_1}\langle \mathit{num}\rangle \mathit{sg} \\ &\wedge @_{w_2}\mathbf{cup} \wedge @_{w_2}\langle \mathit{num}\rangle \mathit{sg} \wedge @_{w_2}\langle \mathit{det}\rangle \mathit{the} \end{aligned} \tag{2.2}$$

The EPs in the first line capture all properties of the verb, the second line captures the actor and the last line captures the patient.

Next comes the *lexical look-up* in which we look for lexical entries covering parts of the LF making them potentially useful for the realization. An entry in OpenCCG consists of three parts: the word or word sequence, its syntactic categories and the covered EPs. The result of the look-up phase can look like the following, depending on the lexicon in use:

(a) Germany $\vdash np : @_x\mathbf{Germany} \wedge @_x\langle num\rangle sg$

(b) the $\vdash np/n : @_x\langle det\rangle the$

(c) cup $\vdash n : @_x\mathbf{cup} \wedge @_x\langle num\rangle sg$

(d) won $\vdash s\backslash np : @_x\mathbf{win} \wedge @_x\langle tense\rangle past \wedge @_x\langle actor\rangle y \wedge @_x\langle patient\rangle z$

(e) won $\vdash (s\backslash np)/np : @_x\mathbf{win} \wedge @_x\langle tense\rangle past \wedge @_x\langle actor\rangle y \wedge @_x\langle patient\rangle z$

(f) won $\vdash ((s\backslash np)/np)/np : @_x\mathbf{win} \wedge @_x\langle tense\rangle past \wedge @_x\langle actor\rangle y \wedge @_x\langle patient\rangle z$

(g) wins $\vdash (s\backslash np)/np : @_x\mathbf{win} \wedge @_x\langle tense\rangle pres \wedge @_x\langle actor\rangle y \wedge @_x\langle patient\rangle z$

(h) win $\vdash (s\backslash np)/np : @_x\mathbf{win} \wedge @_x \wedge @_x\langle actor\rangle y \wedge @_x\langle patient\rangle z$

(i) did $\vdash (s\backslash np)/np/((s\backslash np)/np) : @_x\langle tense\rangle past$

The entries (a) - (c) are not surprising. (d) - (f) cover the same semantics but have different syntactic categories to fit their role as *intransitive, transitive,* and *ditransitive* verbs.

The transitivity of a verb states the number of arguments a verb needs to form a valid sentence. For example consider the intransitive verb “to rush”, which requires only a subject to form a sentence like “We rush.”. When adding an additional argument we get e.g. “Mike rushes Steve” which is not valid (the sentence is *incoherent*). Conversely, when providing only a subject to the transitive verb “bury”, the sentence is *incomplete* and therefore invalid (e.g. “Sammy buries.”). This syntactic property is reflected in the verb’s category: To get an s out of $(s\backslash np)/np$ we need to provide an np to the left and one to the right, hence this is a transitive verb. Some verbs, like “win”, can be any subset of intransitive, transitive and ditransitive.

(g) represents the present tense version of “win” and (h) the infinite form. Moreover, by (i) we have an entry for “did” which provides only the EP for the past tense. Its category is canonical for a *modifier*: It takes a syntactic category and results in the very same category, which renders the modifier syntactically optional.

In addition to these lexical entries we have words that are *semantically null*, like the infinitive “to”. They do not convey any semantics and are treated specially as they are not entries themselves.

The entries we now have will then be filtered. To do so, we check the semantics for contradictions like in g. It provides a $\langle tense\rangle pres$ EP and is therefore contradictory to the $\langle tense\rangle past$ EP in (2.2). (h) is not filtered since it does not specify a tense and hence does not contradict anything in this regard.

We can now transform the lexical entries in *edges*, a data type used in OpenCCG containing (among others) the category, word sequence, and a bit vector. This vector’s length is equal to the number of EPs in 2.2 and each bit represents whether or not a specific EP is expressed by the edge. The index of each bit is the unique id of the EP it represents. The edge containing the information of the lexical entry for "cup" has the bit vector $\langle 000000110 \rangle$ since index 7 represents $@_{w_2} cup$ and index 8 represents $@_{w_2}\langle num \rangle sg$. These bit vectors allow near-constant time checks for overlapping semantics. We will see why this is needed when looking at the next phase of OpenCCG's realization: The combination phase. For this an *agenda* and a *chart* are used. The former one is initialized with all *initial edges*, i.e. the edges representing the lexical entries we found. The latter is initialized empty.

Now, the elements in the agenda are consecutively removed and each element e is attempted to be combined with elements in the chart. When this is possible, the result is added to the agenda. Afterwards, e is added to the chart independently of whether we were able to combine it with any other edge or not.[1]. To combine two edges they have to meet two criteria. Firstly, the syntactic categories have to be combinable by the rules of CCG. Secondly, the covered EPs have to be disjoint, i.e. the bitwise conjunction has no bit set. When a combination is possible, the resulting edge's semantics is the bitwise disjunction of the original edges' semantics. It's category is the result of combining the edges' categories according to the rules of CCG. The new edge represents the word sequence which is acquired by concatenating the word sequence of both edges. Additionally, a unary rule can be applied to obtain a new edge. When doing so, there is no change in the covered EPs.

We can see that (a) cannot be combined with (b) since we cannot apply any CCG rule on *np/n* and *np*. Similarly we cannot combine (d) with (h) for their overlapping bit vectors. However, we can combine (b) with (c) to get

$$\text{the cup} \vdash np : @_x \mathbf{cup} \wedge @_x\langle num \rangle sg \wedge @_x\langle det \rangle the$$

In addition to that, OpenCCG allows more rules than the before-mentioned to be specified. As an example consider the uncountable noun "fruit" which act as ordinary noun in "I eat the sweet fruit" but does not require a determiner (here: "the") as in "I eat fruit" where it is a noun phrase by itself. So we can specify the rule $n_{mass} \Rightarrow np_{mass}$. Moreover, unary rules can contribute to the covered semantics [5], i.e. after applying the rule, the resulting edge covers more EPs.

OpenCCG's realization algorithm runs in an anytime fashion, i.e. when a time limit is reached, it is possible to check the chart for the best edge and use this as output even though there are still elements in the agenda. This might lead to invalid or sub-optimal results but this is in practice often good enough or rather better than no result at all. Two kinds of time limits are used, both determined statically: First of all, an *overall time limit*. When this one is reached, the search stops and the currently best edge is the result. Secondly, the *next best time limit* states how much time is spent to find a better edge after a potential result is found, i.e. an edge conveying all semantics with a syntactically valid sentence. As soon as such an edge is found, the search continues for at max x milliseconds, where x is the next best time limit. After this time the currently best edge is considered the result. The algorithm is illustrated in Fig. 2.4.

In this thesis we try to improve this process by approximating whether an edge has the potential to contribute to a solution. If not, the edge is not added to the agenda. The

[1]Note that we skip over a lot of details here. These can be found in [15]

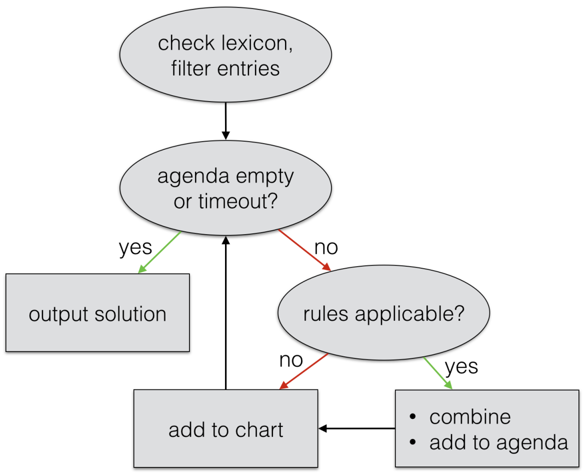

Figure 2.4: Schematic illustration of a OpenCCG's search algorithm

details are explained in Section 3 after the groundwork is laid, but we will show the basic idea on our running example.

To cover the whole semantics of the sentence "Germany won the cup" we need to cover the EPs $@_{w_0}\langle actor\rangle w_1$ and $@_{w_0}\langle patient\rangle w_2$ for the sentence's subject and object. For this to be possible, we need the category $(s\backslash np)/np$ representing the transitive version of "win". This renders the intransitive useless because there is no way to combine it to form a valid sentence and cover all EPs. The sentence would metaphorically fall apart.

For this we want to use well established planning techniques, which is why we introduce some basic concepts and definitions in the following section.

2.3 Planning

A planning task is a (formal) description for a problem, e.g. the towers of hanoi, for which we want to find a solution, i.e. an ordered sequence of actions leading to the problem being solved. This search for a solution is performed by a *planner*. There are two main ways to formalize a planning task: Using the STRIPS (STanford Research Institute Problem Solver) [3] formalism or FDR (Finite Domain Representation)[4]. We will first introduce STRIPS, which is used for easier illustration of the key aspect used in section 2.4 and the FDR which is what most planners today are based on.

Definition 2.3.1 (STRIPS Planning Task)**.** A STRIPS planning task is denoted by a quintuple $\Pi = (P, A, c, I, G)$. Its constituents are defined as follows:

- P is a finite set of binary facts which can either be fulfilled or not.
- A is a finite set of actions. Each action consists of three sets representing its preconditions, add list, and delete list, all of which being subsets of P. All preconditions pre_a of an action a have to be true so that we can apply a. The add list add_a contains all facts that become true after applying the action, the delete list del_a, accordingly, contains all facts that become false after the application.

- $c : A \to \mathbb{R}^+$ is a cost function mapping an action to its cost. Unless otherwise stated, we assume $\forall a \in A : c(a) = 1$.
- $I \subseteq P$ is the initial state. Each fact contained in I is initially true.
- $G \subseteq P$ is the goal state. When all elements of G are fulfilled, the task is complete.

A STRIPS task Π is a way to formalize a problem. It induces a *search space* in which the planner searches for a solution.

Definition 2.3.2 (STRIPS State Space)**.** Π induces a labeled transition system $\Theta_\Pi = (S, A, T, I, S^G)$ with

- $S = 2^P$ is the power set of P. Each element of S represents a *state*.
- A is the set of actions we get by Π.
- $T = \{(s, a, s')|s, s' \in S \land a \in A \land pre_a \subseteq s \land s' = (s \cup add_a)\backslash del_a\}$. We denote $(s, a, s') \in T$ equivalently by $s \xrightarrow{a} s'$ and call it a *transition*.
- I is the initial state we get by Π.
- $S^G = \{s \in S|G \subseteq s\}$ is the set of all goal states.

Example 2.3.1. Consider the following example where the task is to say a sentence consisting of exactly one subject, object and verb.

$$\begin{aligned}
P =& \{has\_subj, has\_no\_subj, has\_obj, has\_no\_obj, has\_verb, has\_no\_verb, said\} \\
I =& \{has\_no\_subj, has\_no\_obj, has\_no\_verb\} \\
G =& \{said\} \\
A =& \{add\_subj, add\_obj, add\_verb, say\}
\end{aligned}$$

With

$$
\begin{aligned}
add\_subj &:\\
&pre = \{has\_no\_subj\}\\
&add = \{has\_subj\}\\
&del = \{has\_no\_subj\}\\
add\_obj &:\\
&pre = \{has\_no\_obj\}\\
&add = \{has\_obj\}\\
&del = \{has\_no\_obj\}\\
add\_verb &:\\
&pre = \{has\_no\_verb\}\\
&add = \{has\_verb\}\\
&del = \{has\_no\_verb\}\\
say &:\\
&pre = \{has\_subj, has\_obj, has\_verb\}\\
&add = \{has\_no\_subj, has\_no\_obj, has\_no\_verb, said\}\\
&del = \{has\_subj, has\_obj, has\_verb\}
\end{aligned}
$$

We have two facts per constituent, one for the absence and one for the presence. This is necessary, because the STRIPS formalism does not allow to check for negative facts. We will now introduce the FDR formalism, where several STRIPS facts can be collapsed into a single variable which allows us to reduce the number of facts as well as check for negative facts since a variable - as opposed to a fact - cannot be absent.

Definition 2.3.3 (FDR Planning Task)**.** An FDR (finite-domain representation) planning task Π is a quintuple (V, I, c, A, G) where

- V is the finite set of variables v which domains D_v are finite. We call $\{v := x \in D_v\}$ an assignment for v and a (partial) assignment for V. An assignment is complete when there is a variable assignment for each variable in V.
- I is a complete variable assignment denoting the initial values for all variables.
- $c : A \to \mathbb{R}_0^+$ is a function mapping an action to its cost. Unless otherwise stated, we assume $\forall a \in A : c(a) = 1$.
- A is the set of actions. Each action a consists of a (partial) assignment pre_a denoting the preconditions which have to be met so that the action can be applied, and a (partial) assignment eff_a denoting the values of variables that might be alternated by the application of this action.
- G is a (partial) variable assignment denoting the goal state.

Similarly to the STRIPS formalization, we can define an (induced) search space.

Definition 2.3.4 (FDR State Space)**.** An FDR state space Θ_Π is labeled transition system defined by a quintuple (S, A, T, I, S^G) with

- S being the set of all complete variable assignments for V.
- A being the set of actions we get by Π.
- $T \subseteq S \times A \times S$ is a (labeled) transition relation. We denote $(s, a, s') \in T$ equivalently by $s \xrightarrow{a} s'$ and define T as

$$T = \{(s, a, s') | \forall v \in V : (\{v := x\} \in \mathit{pre}_a \Rightarrow \{v := x\} \in s)$$
$$\wedge \{v := \begin{cases} x & \text{if } \{v := x\} \in \mathit{eff}_a \\ y & \text{otherwise with} \{v := y\} \in s \end{cases} \} \in s'\}$$

  That means, we need the preconditions to be met and keep all assignments from s unless they are overwritten by the effect of a.
- I is the initial state we get by Π.
- $S^G \subseteq S$ is the set of complete assignments in which the goal criterion is met, i.e. $S^G = \{s \in S | G \subseteq s\}$.

Note that due to the agreement of c being a constant function, we leave the specification thereof as optional. We will now transform example 2.3.1 into an FDR task with four binary variables where 0 indicates that we have not yet reached the respective fact.

$$\begin{aligned} V =& \{\mathit{subj}, \mathit{obj}, \mathit{verb}, \mathit{said}\} \\ I =& \{\mathit{subj} := 0, \mathit{obj} := 0, \mathit{verb} := 0, \mathit{said} := 0\} \\ G =& \{\mathit{said} := 1\} \\ A =& \{\mathit{add\_subj}, \mathit{add\_obj}, \mathit{add\_verb}, \mathit{say}\} \end{aligned}$$

With

$$\begin{aligned} \mathit{add\_subj} &: \\ \mathit{pre} =& \{\mathit{subj} := 0\} \\ \mathit{eff} =& \{\mathit{subj} := 1\} \\ \mathit{add\_obj} &: \\ \mathit{pre} =& \{\mathit{obj} := 0\} \\ \mathit{eff} =& \{\mathit{obj} := 1\} \\ \mathit{add\_verb} &: \\ \mathit{pre} =& \{\mathit{verb} := 0\} \\ \mathit{eff} =& \{\mathit{verb} := 1\} \\ \mathit{say} &: \\ \mathit{pre} =& \{\mathit{subj} := 1, \mathit{obj} := 1, \mathit{verb} := 1\} \\ \mathit{eff} =& \{\mathit{subj} := 0, \mathit{obj} := 0, \mathit{verb} := 0, \mathit{said} := 1\} \end{aligned}$$

In the following we use the STRIPS and the FDR representation interchangeably. The former one is used for explanation purposes, the latter one is used in practice, so the compilate, i.e. the result of compiling the CCG task into a planning task, will be in FDR.

Definition 2.3.5 (search terms)**.** For any search space $\Theta = (S, A, T, I, S^G)$, we say a sequence of actions $\langle a_0, \ldots, a_{n-1} \rangle$ is applicable to a state s iff

$$\exists s_1, \ldots, s_n \in S : s \xrightarrow{a_0} s_0 \xrightarrow{a_1} \ldots \xrightarrow{a_n} s_n.$$

We denote this by $s \circ \langle a_0, \ldots, a_{n-1} \rangle = s_n$ and, in case there are no ambiguities or the action sequence is irrelevant, $s \rightarrow^n s_n$. If the action sequence was not sequentially applicable, we say $s \circ \langle a_0, \ldots, a_{n-1} \rangle = \textit{undefined}$. A state s' is *reachable* from a state s when there is a sequence of actions resulting in s' i.e.

$$\exists n \in \mathbb{N} : s \rightarrow^n s'.$$

A state is reachable in general if it is reachable from the initial state. The planning task Π is *solvable* if any goal is reachable, i.e.

$$\exists n \in \mathbb{N}, \exists g \in S^G : s \rightarrow^n g.$$

States from which no goal is reachable are called *dead ends*. A sequence of actions $\langle a_0, \ldots, a_n \rangle$ leading from the initial state to the goal is called a *plan*, with the cost being the sum of each action's cost. A *search* works on *(search) nodes*. These are states with additional information, e.g. the cost of a plan leading to this node or references to their parents. The *search algorithm* tries to find a path to a goal state. These algorithms exist in various flavors, some of which can be found in Russel, Norvig (2010) [12].

In example 2.3.1 a plan is

$$\langle \textit{add\_subj}, \textit{add\_obj}, \textit{add\_verb}, \textit{say} \rangle$$

as well as

$$\langle \textit{add\_subj}, \textit{add\_obj}, \textit{add\_verb}, \textit{say}, \textit{add\_verb}, \textit{add\_obj}, \textit{add\_subj}, \textit{say} \rangle$$

Obviously, all plans for the STRIPS version are also valid in the FDR version and vice versa.

We later have to deal with conditional effect, so we define them here.

Definition 2.3.6. A conditional effect in a STRIPS planning task is an action's effect that only occurs when specific preconditions are met. For example, a speeding driver will only get a ticket if the police is near. We denote this the following way for $p, p' \in P \cup \{\top\}$ where p is the condition and p' is the conditional effect:

$$\{p \Rightarrow p'\} \in \textit{eff}_a$$

If $p = \top$ the conditional effect always happens.

Example 2.3.2.

$$\begin{aligned}
&A = \{\textit{speed}\} \text{ with} \\
&\textit{pre}_{\textit{speed}} = \{\textit{have\_car}\} \text{ and} \\
&\textit{eff}_{\textit{speed}} = \{\top \Rightarrow \textit{arrive\_in\_time},\ \textit{police\_around} \Rightarrow \textit{ticket}\}
\end{aligned}$$

Neither STRIPS, nor FDR support *conditional effects*, but we have to deal with them later on.

2.4 Heuristic functions

When trying to find a goal, one approach is searching through the transition system blindly, which is called *blind search* but this is quite inefficient when the transition system grows large. As an improvement, we want to estimate, how *good* a state is and prioritize the according nodes in the search, i.e. expanding them earlier and ignoring *worse* states. This kind of estimation is done by a heuristic function taking a state and approximating, how far the distance to a goal is [1].

Definition 2.4.1. A heuristic function is a function $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$. Its result is called "heuristic value".

In a perfect world, a heuristic function would give us the exact distance to a goal, but computing this value is as hard as solving the problem in the first place. For this reason, we use approximations that are computable in a reasonable amount of time. One way to achieve this is by *relaxing* the problem, i.e. making it easier to solve and compute a goal distance on the relaxed task. We will present and use the *delete-relaxed critical path heuristic h^{max}* [6].

Definition 2.4.2 (Delete Relaxation)**.** Let $\Pi = (P, A, I, G)$ be a STRIPS planning task. We call Π^+ the *delete-relaxed planning task* and obtain it by removing the delete list of all actions.

In a nutshell we can say, that in a delete-relaxed world, everything that once was true, will always be true. If we had \$1000 and gave it to a friend, both will have the money. The greatest improvement we get by delete-relaxing is that the problem of deciding whether there is a solution for a given delete-relaxed problem is possible in polynomial time. This is exactly what we need later on and h^{max} does this for us. To define h^{max} , the last thing we need is regression:

Definition 2.4.3 (Regression)**.** $regr : A \times S \to S$ is a function allowing us to *regress* over a state, i.e. $regr(s, a) = s'$ expresses we can use action a in state s' to reach s.

$$regr(s,a) = \begin{cases} (s \setminus add_a) \cup pre_a & \text{if } add_a \cap s \neq \emptyset \wedge del_a \cap s = \emptyset \\ undefined & \text{otherwise} \end{cases}$$

$regr(s, a) = undefined$ expresses that we cannot reach s using the action a since a's delete list contains a fact which is present in s or that a does not contribute in reaching s, so s and add_a are disjoint.

Definition 2.4.4 (Critical Path Heuristic h^{max})**.** We define h^{max} as the point-wise greatest function satisfying

$$h^{max}(s,g) = \begin{cases} 0 & \text{if } g \subseteq s \\ min_{a \in A} c(a) + h^{max}(s, regr(g,a)) & \text{if } |g| = 1 \wedge regr(g,a) \neq undefined \\ max_{g' \in g} h^{max}(s, \{g'\}) & |g| \geq 1 \end{cases}$$

For any state $s \in S$ we say $h^{max}(s) := h^{max}(s, G)$.

h^{max} can be computed very efficiently, as described by Hoffmann, Nebel in 2001 [6]. Lastly, at some point we need an *abstraction* which allows to collapse several search states into one. There are two ways to apply an abstraction: We can define a heuristic which internally uses an abstraction or we can use the abstraction on the search space we have and let the heuristic work on the result. We chose the second approach.

Definition 2.4.5. Let $\alpha : S \to S^\alpha$ be a surjective function mapping one or more states of a search space onto one state.

Every other definition can be left as it was just that the abstraction has to be applied on each component. We do not spell this out in detail because in Chapter 3.3 we will see that we merely borrow the concept of abstraction and adopt it to fit to our problem, i.e. we will use it to collapse an infinite amount of different categories into one category.

Definition 2.4.6 (safe)**.** A heuristic h is safe if the following equivalence holds:

$$h(s) = \infty \iff s \text{ is a dead end.}$$

Furthermore, a compilation ϕ mapping a search space onto another one is safe if the existence of a plan in the original search space implies the existence of a plan in the compiled one, i.e. $\forall \Theta, \Theta'$ with $\Theta = (S, A, T, I, S^G)$, $\Theta' = (S', A', T', I', (S^G)')$ and $\phi(\Theta) = \Theta'$:

$$\begin{aligned} &\forall \langle a_0, \dots, a_i \rangle \text{ with } I \circ \langle a_0, \dots, a_i \rangle = g \in S^G \\ &\implies \exists \langle a'_0, \dots, a'_j \rangle \text{ with } I' \circ \langle a'_0, \dots, a'_j \rangle = g' \in (S^G)' \end{aligned}$$

Chapter 3

Compilation

To be able to use the h^{max} heuristic in the context of OpenCCG, we need to build a foundation on which h^{max} can work. That means, we have to compile the rules of CCG into a set of facts and actions in an intelligent way such that we can proof desirable properties, in this case *safety*. In addition we want the compilation to be *tractable*, i.e. the runtime is polynomially bound on the CCG task's size.

As a first step we are going to define a CCG task Θ^{CCG} in a formal way as well as what is a dead end therein. On this base we can then define a compilation into an *intermediate planning task* Θ^1 which is infinite in size and acts as a kind of interface between CCG and planning. We improve this compilation by mapping Θ^1 onto another planning task Θ^{α_k} which is finite, but grows exponentially in the size of Θ^{CCG} and is only able to detect *some* dead ends, as opposed to Θ^1. Another refinement finally yields Θ^{Σ}, which fulfills our requirements of safety and tractability. Lastly, we adapt this problem minorly such that we can use this result to classify edges as potentially useful or not which allows us to exclude edges classified as not useful for the search.

In the following we use the convention that all variables defined in a CCG context are Greek letters while variables in the planning context are denoted by Latin letters.

3.1 CCG Task

For defining the CCG task we use terms from the OpenCCG context, in particular the term "edge" for the compound of a syntactic category and its coverage of the LF. We intentionally skip over the word sequence associated with an edge because we do not need any knowledge about the actual words for determining whether or not a problem is solvable.

Definition 3.1.1 (CCG Task)**.** A CCG task is a quintuple Θ^{CCG} as follows:

$$\Theta^{CCG} = (\Delta, \Gamma_0, \Lambda, \Lambda^I, \lambda^G)$$

where

- Γ_0 is the set of atomic syntactic categories. We denote the set of all possible syntactic categories by Γ.
- $\Lambda = \Gamma \times \Sigma$ is the set of all edges with $\Sigma = \{\top, \bot\}^n$ where n is the size of the LF.

- $\Delta = (\Delta^B, \Delta^U)$ denotes the transition relation with
  - $\Delta^B \subseteq \Lambda \times \Lambda \times \Lambda$ representing the binary combination rules
  - $\Delta^U \subseteq \Lambda \times \Gamma \times \Lambda$ representing the unary combination rule
- $\Lambda^I \subseteq \Lambda$ defines the (finite) set of all edges on which we can work in the beginning of the task. These edges represent the lexical entries we have in the lexicon initially.
- $\lambda^G \in \Lambda$ represents the goal edge.

To sum this up, λ represents an edge consisting of its category γ and semantics σ. δ represents an action consisting of either three edges in case of a binary combination, where the first element of δ represents the left operand, the second element the right operand, and the third element the resulting edge. For unary actions the first component is the original edge, the second one is the category by which we raise and the third component is the resulting edge. To ease working with Θ^{CCG} we introduce some notational sugar.

Tuple access:

$\forall\lambda = (\gamma, \sigma) \in \Lambda : \lambda.\gamma = \gamma \wedge \lambda.\sigma = \sigma$
$\forall\delta = (\lambda_1, \lambda_2, \lambda_3) \in \Delta^B : \delta.\lambda_1 = \lambda_1 \wedge \delta.\lambda_2 = \lambda_2 \wedge \delta.\lambda_3 = \lambda_3$
$\forall\delta = (\lambda_b, \gamma, \lambda_r) \in \Delta^U : \delta.\lambda_b = \lambda_b \wedge \delta.\gamma = \gamma \wedge \delta.\lambda_r = \lambda_r$
This means that $\lambda.\gamma$ gives us the γ component of λ.

Bit vector access:

$\forall\sigma \in \Sigma. \quad \forall i \in \{0, \ldots, n-1\} : \sigma = (b_0, \ldots, b_{n-1}) \wedge \sigma[i] = b_i$

Bit vector con- and disjunction:

$\forall\sigma_1, \sigma_2 \in \Sigma. \quad \forall\circ \in \{\wedge, \vee\} : \sigma_1 \circ \sigma_2 = (\sigma_1[0] \circ \sigma_2[0], \ldots, \sigma_1[n-1] \circ \sigma_2[n-1])$

Furthermore, unless otherwise defined, the $\circ$-operator denotes a slash operator in CCG, i.e. $\circ \in \{\text{'/'}, \text{'\textbackslash'}\}$ with

$$\bar{\circ} = \begin{cases} \text{'\textbackslash'} & \text{if } \circ = \text{'/'} \\ \text{'/'} & \text{otherwise} \end{cases}$$

Defining the constituents for a given CCG task is straight-forward. Note that we require $s \in \Gamma_0$ as the goal category.

- $\lambda^G = (s, \top^n)$ such that we have a syntactically valid sentence which suffices the LF entirely.
- Λ^I contains all lexical entries upon which we can build the sentence.

For gamma we need the union over categories consisting of i slashes for all $i \in \mathbb{N}$. We have to make sure to include categories like *np/(n/n)*.

$$\Gamma_i = \bigcup_{\gamma\in\Gamma_0, \gamma'\in\Gamma_{i-1}} \{\gamma \circ (\gamma'), \gamma \circ \gamma', \gamma\bar{\circ}(\gamma'), \gamma\bar{\circ}\gamma', \gamma' \circ \gamma, \gamma'\bar{\circ}\gamma\} \quad \forall i > 0$$

$$\Gamma = \bigcup_{i=0}^{\infty} \{\Gamma_i\}$$

Note that this also includes categories like *(np/n)/(np/n)* because the first set of parenthesis is superfluous here, so we can equivalently write *np/n/(np/n)* with $np \in \Gamma_0$ and $n/(np/n) \in \Gamma_3$.

We can now define the transition relation Δ which consists of 3 parts, one for each binary and one for the unary rule. We compose all transitions resulting from the application rule in Δ^{app}, transitions from the composition rule in Δ^{comp} and unary rules in Δ^U.

$$\Delta = (\Delta^{app} \cup \Delta^{comp}, \Delta^U)$$

For the application rule in terms of the categories we only require the right constellation on the right and on the left hand side of each operand's slash. We do not distinguish between the direction of the slash operator because of before-mentioned irrelevance of the actual word sequence. Regarding the semantics, we require the coverage vectors to be disjoint and the resulting vector to be the disjunction of each operand's coverage. We use the following predicates to determine whether two categories can be combined:

$$\mathit{left}(\gamma) = \begin{cases} \mathit{left}(\gamma') & \text{if } \gamma = \gamma' \circ \gamma'' \wedge \mathit{left}(\gamma') \neq \bot \\ \gamma' & \text{if } \gamma = \gamma' \circ \gamma'' \wedge \mathit{left}(\gamma') = \bot \\ \bot & \text{if } \gamma \in \Gamma_0 \end{cases} \tag{3.1}$$

$$\mathit{right}(\gamma) = \begin{cases} \gamma'' & \text{if } \gamma = \gamma' \circ \gamma'' \\ \bot & \text{if } \gamma \in \Gamma_0 \end{cases} \tag{3.2}$$

$$\mathit{fwd}(\gamma) \equiv (\gamma = \gamma' \circ \gamma'') \wedge (\circ = \text{`/'}) \tag{3.3}$$

$$\mathit{appl}(\gamma_0, \gamma_1) \equiv (\gamma_0 = \gamma'_0 \circ \gamma''_0) \wedge (\gamma''_0 = \gamma_1) \tag{3.4}$$

$$\begin{aligned} \mathit{comp}(\gamma_0, \gamma_1) \equiv\ & \mathit{right}(\gamma_0) \neq \bot \wedge \mathit{right}(\gamma_1) \neq \bot \wedge (\mathit{fwd}(\gamma_0) = \mathit{fwd}(\gamma_1)) \\ & \wedge \begin{cases} \mathit{right}(\gamma_0) = \mathit{left}(\gamma_1) & \text{if } \mathit{fwd}(\gamma_0) \\ \mathit{right}(\gamma_1) = \mathit{left}(\gamma_0) & \text{otherwise} \end{cases} \end{aligned} \tag{3.5}$$

We defined *left* recursively to make sure categories like γ_0/γ_1 and $\gamma_1/\gamma_2/\gamma_3$ can be composed.

$$\begin{aligned} \Delta^{app} =& \{(\lambda_1, \lambda_2, \lambda_3) | \mathit{appl}(\lambda_1.\gamma, \lambda_2.\gamma) \wedge (\lambda_1.\gamma = \gamma_L \circ \gamma_R) \wedge (\lambda_3.\gamma = \gamma_L) \\ & \wedge (\lambda_3.\sigma = \lambda_1.\sigma \vee \lambda_2.\sigma) \wedge (\lambda_1.\sigma \wedge \lambda_2.\sigma = \bot^n)\} \end{aligned} \tag{3.6}$$

$$\begin{aligned} \Delta^{comp} = \{(\lambda_1, \lambda_2, \lambda_3) | & \mathit{comp}(\lambda_1.\gamma, \lambda_2.\gamma) \wedge (\lambda_3.\sigma = \lambda_1.\sigma \vee \lambda_2.\sigma) \wedge (\lambda_1.\sigma \wedge \lambda_2.\sigma = \bot^n) \\ & \wedge \lambda_1.\gamma = \gamma'_1 \circ \gamma''_1 \wedge \lambda_2.\gamma = \gamma'_2 \circ \gamma''_2 \\ & \wedge \lambda_3.\gamma = \begin{cases} \gamma'_1 \circ \gamma''_2 & \text{if } \mathit{fwd}(\lambda_1.\gamma) \\ \gamma'_2 \circ \gamma''_1 & \text{otherwise} \end{cases} \} \end{aligned}$$

For the unary rule, we have no change in the edge's semantics, hence the vector of the result is equal to the one we have before applying the rule and the resulting category has to have the right format.

$$\Delta^U = \{(\lambda_b, \gamma_t, \lambda_r) | (\lambda_r.\sigma = \lambda_b.\sigma) \wedge (\lambda_r.\gamma = \gamma_t \circ (\gamma_t \bar{\circ} \lambda_b.\gamma)\}$$

We can now define the terms *search state*, *transition*, *reachable* and *solvable* on any CCG Task $\Theta^{CCG} = (\Delta, \Gamma_0, \Lambda, \Lambda^I, \lambda^G)$ in a straight-forward way:

Definition 3.1.2 (Search State)**.** A search state $s_{CCG} \subseteq \Lambda$ defines a set of edges. The initial state is denoted by Λ^I.

Definition 3.1.3 (Transition)**.** A state s_{CCG} allows a transition $\delta = (\lambda_1, \lambda_2, \lambda_{res}) \in \Delta^B$ to be made if and only if $\lambda_1 \in s_{CCG} \wedge \lambda_2 \in s_{CCG}$. Transitions $\delta = (\lambda_1, \gamma, \lambda_{res}) \in \Delta^U$ can be made if and only if $\lambda_1 \in s_{CCG}$. The resulting state in both cases is $s'_{CCG} = s_{CCG} \cup \lambda_{res}$. We denote an applicable transition by $s_{CCG} \xrightarrow{\delta} s'_{CCG}$. When there is no ambiguity we omit δ. By $\longrightarrow^*$ we denote the reflexive-transitive closure of the transition relation for both Δ^B and Δ^U while $s \longrightarrow^n s'$ means that we need n transitions to reach s' starting at s.

Definition 3.1.4 (Reachability)**.** A state s'_{CCG} is reachable from any state s_{CCG} if and only if $s_{CCG} \longrightarrow^* s'_{CCG}$. A state s_{CCG} is reachable for a CCG task if and only if $\Lambda^I \longrightarrow^* s_{CCG}$.

Definition 3.1.5 (Solvability)**.** A CCG task is solvable, if and only if the goal is reachable, i.e. $\exists \Lambda^G \subseteq \Lambda$ with $\lambda^G \in \Lambda^G \wedge \Lambda^I \longrightarrow^* \Lambda^G$.

Definition 3.1.6. A state s_{CCG} is a *dead end* if no goal state is reachable.

Furthermore we can see that the following property holds:

Lemma 3.1.1 (Monotonicity)**.** *We will never lose what we once reached.*

$$\forall \Lambda', \Lambda'' \subseteq \Lambda, \forall \delta \in \Delta : \Lambda' \xrightarrow{\delta} \Lambda'' \implies (\forall \lambda \in \Lambda' : \lambda \in \Lambda'')$$

Proof. By definition. □

In the following we will present a compilation which will be refined a number of times until all desired properties are met. In proofs we omit a case distinction between transitions based on a unary or binary rule for simplicity whenever the omitted case is obvious, e.g. because it is analogous to the other case. We then silently assume the action to be binary, i.e. we cover the more interesting case.

3.2 Intermediate Planning Task

Definition 3.2.1 (Intermediate Planning Task)**.** An intermediate planning task Π_{inter} is a quadruple (V, A, I, G) (cf. definition of *planning task* (2.3.3)). It is induced by a CCG task $\Theta^{CCG} = (\Delta, \Gamma_0, \Lambda, \Lambda^I, \lambda^G)$ but does not necessarily fulfill all requirements of a planning task.

We define a first induced planning task Θ^1 as follows while ignoring the finiteness aspect and tractability constraint of the compilation:

Each fact has a binary domain. For each edge, we have a fact (s_λ) stating whether we are allowed to use this edge at this point in time. This is a very simple compilation and its primary function is to be the base for further refinements.

$$V_1 := \bigcup_{\lambda \in \Lambda} \{s_\lambda\}$$

So there is one variable for each potential edge.
The definitions for the initial and the goal state follow directly from the definition of a fact: All initial edges are reached at the beginning and we want to reach the goal edge at the end.

$$\begin{aligned} I_1 &:= \bigcup_{\lambda \in \Lambda} \{s_\lambda := (\lambda \in \Lambda^I)\} \\ G_1 &:= \{s_{\lambda^G} := \top\} \end{aligned}$$

The actions are divided into unary and binary actions with appropriate preconditions and effects.

$$\begin{aligned} A_1 &:= A_1^B \cup A_1^U \\ A_1^B &:= \bigcup_{\delta \in \Delta^B} \{a_\delta^B\} \\ pre_{a_\delta^B} &:= \{s_{\delta.\lambda_1} := \top, s_{\delta.\lambda_2} := \top\} \\ \mathit{eff}_{a_\delta^B} &:= \{s_{\lambda_3} := \top\} \\ A_1^U &:= \bigcup_{\delta \in \Delta^U} \{a_\delta^U\} \\ pre_{a_\delta^U} &:= \{s_{\delta.\lambda_b} := \top\} \\ \mathit{eff}_{a_\delta^U} &:= \{s_{\delta.\lambda_r} := \top\} \end{aligned}$$

We denote states in any planning task (intermediate or not) by s_{Plan}.

Θ^1 is infinite in size because we can easily construct an infinite chain of actions resulting in pairwise different states. We proof this claim in Theorem 3.3.1. However, we can see that the following property holds:

Lemma 3.2.1 (Monotonicity). *The intermediate task is a delete relaxed one, i.e. when a variable becomes $\top$, it never turns back to $\bot$.*

$$\forall s_{Plan}, \forall s \in V_1, \forall a \in A_1 : \{s := \top\} \subseteq s_{Plan} \implies \{s := \top\} \subseteq (s_{Plan} \circ \langle a \rangle)$$

Proof. By construction we know

$$\forall a \in A_1, \forall s \in S : \{s := \bot\} \notin \mathit{eff}_a$$

□

To proof that the construction of Θ^1 is safe, we need the following lemma[2]:

Lemma 3.2.2 (Mutual Consistency). *For all transition sequences in the CCG task we can find an action sequence in Θ^1 resulting in a state where the facts assigned to $\top$ reflect the edges we have reached in the CCG task after taking these transitions.*

$$\forall \langle \delta_0, \ldots, \delta_{k-1} \rangle : \Lambda^I \xrightarrow{\delta_0} \ldots \xrightarrow{\delta_{k-1}} \Lambda' : \forall \lambda \in \Lambda : (\lambda \in \Lambda' \iff \{s_\lambda := \top\} \subseteq I_1 \circ \langle a_{\delta_0}, \ldots, a_{\delta_{k-1}} \rangle)$$

[2]Please keep in mind that we use Greek letters for CCG related variables and Latin ones for planning related variables

Proof. Induction on k.
Base $k = 0$:

$$\forall \lambda \in \Lambda : (\lambda \in \Lambda^I \iff \{s_\lambda := \top\} \subseteq I_1) \text{ by construction}$$

Step $k \to k + 1$:
We know $\Lambda^I \xrightarrow{\delta_0} \dots \xrightarrow{\delta_{k-1}} \Lambda' \xrightarrow{\delta_k} \Lambda''$ and

$$\begin{aligned}
&\forall \lambda \in \Lambda : (\lambda \in \Lambda'' \overset{(*)}{\iff} (\lambda \in \Lambda' \vee \delta_k.\lambda_3 = \lambda)) \\
\overset{\text{IH}}{\iff} &\forall \lambda \in \Lambda : (\lambda \in \Lambda'' \iff (\{s_\lambda := \top\} \subseteq I_1 \circ \langle a_{\delta_0}, \dots, a_{\delta_{k-1}} \rangle \vee \delta_k.\lambda_3 = \lambda)) \\
\overset{(**)}{\iff} &\forall \lambda \in \Lambda : (\lambda \in \Lambda'' \iff (\{s_\lambda := \top\} \subseteq I_1 \circ \langle a_{\delta_0}, \dots, a_{\delta_{k-1}} \rangle \vee \mathit{eff}_{a_{\delta_k}} = \{s_\lambda := \top\})) \\
\overset{(***)}{\iff} &\forall \lambda \in \Lambda : (\lambda \in \Lambda'' \iff (\{s_\lambda := \top\} \subseteq I_1 \circ \langle a_{\delta_0}, \dots, a_{\delta_{k-1}}, a_{\delta_k} \rangle))
\end{aligned}$$

$(*)$	by 3.1.1 (Monotonicity) and definition of transitions
$(**)$	by construction
$(***)$	by applying the IH twice resolving the universal quantifier with $\delta_k.\lambda_1$ and $\delta_k.\lambda_2$ to meet δ_k's preconditions and 3.2.1 (Monotonicity)

□

Theorem 3.2.3. *A transition sequence $\langle \delta_0, \dots, \delta_{k-1} \rangle$ leads to a goal state if and only if the action sequence $\langle a_{\delta_0}, \dots, a_{\delta_{k-1}} \rangle$ leads to a goal, i.e.*

$$\Lambda^I \xrightarrow{\delta_0} \dots \xrightarrow{\delta_{k-1}} \Lambda^G \ni \lambda^G \iff I_1 \circ \langle a_{\delta_0}, \dots, a_{\delta_k - 1} \rangle \supseteq G_1$$

Proof.

$$\begin{aligned}
\Lambda^I \xrightarrow{\delta_0} \dots \xrightarrow{\delta_{k-1}} \Lambda^G \ni \lambda^G &\iff I_1 \circ \langle a_{\delta_0}, \dots, a_{\delta_k - 1} \rangle \supseteq \{s_{\lambda^G} := \top\} && \text{by lemma 3.2.2} \\
&\iff I_1 \circ \langle a_{\delta_0}, \dots, a_{\delta_k - 1} \rangle \supseteq G_1 && \text{by construction}
\end{aligned}$$

□

Corollary 3.2.3.1. Θ^{CCG} *solvable* $\iff (V_1, A_1, I_1, G_1)$ *solvable*

So the compilate Θ^1 is equivalent since each transition in the CCG task can be simulated by an action in the intermediate planning task and vice versa. Hence there is a bijection between any transition sequence and the according plan resulting in equivalent states regarding the goal status. Nonetheless, one can easily see that the compilate is not finite and therefore not particularly useful when trying to run a heuristic on it.

3.3 Restricting the Number of Categories

To tackle the problem of infiniteness we introduce a way to stop infinite sequences of rule applications resulting in pairwise different states. So Δ, Γ and Λ become finite. The consequence is that the set of reachable edges and the set of syntactic categories also become limited in size so that we can find a compilation that suffices the criteria for a planning task.

Firstly, we define $\# : \Gamma \to \mathbb{N}$ as a function giving us the number of slashes in a category.

$$\#(\gamma) = \begin{cases} 1 + \#(\gamma') + \#(\gamma'') & \text{if } \gamma = \gamma' \circ \gamma'' \\ 0 & \text{otherwise} \end{cases}$$

The problem of infiniteness does not only occur in rare cases but rather in all non-trivial CCG task, which allows us to proof the following theorem:

Theorem 3.3.1. *All non-trivial CCG tasks are infinite.*

$$\forall \Theta^{CCG} = (\Delta, \Gamma_0, \Lambda, \Lambda^I, \lambda^G) : \Lambda^I \neq \emptyset \implies |\Lambda| = \infty \wedge |\Gamma| = \infty \wedge |\Delta| = \infty$$

Proof. It suffices to show that we can find an infinite sequence of transitions leading to pairwise different states, i.e. there is an infinite, acyclic path through the transition system. More formally:

Lemma 3.3.2 (Infinite Transition Sequences)**.**

$$\exists \langle \delta_0, \delta_1, ... \rangle : \Lambda^I \xrightarrow{\delta_0} \Lambda' \xrightarrow{\delta_1} \dots$$

Proof. We construct a sequence of unary rules inductively:
For $\lambda_0 \in \Lambda^I$ we define $\delta_0 := (\lambda_0, \lambda_0.\gamma, \lambda_1)$ with $\lambda_1 = (\lambda_0.\gamma \circ (\lambda_0.\gamma \bar{\circ} \lambda_0.\gamma), \lambda_0.\sigma)$
Obviously, $\lambda_1.\gamma \in \Gamma$ and $\lambda_1 \in \Lambda$ due to the definition of the type raising rule.
Now for the induction step we define

$$\delta_i := (\delta_{i-1}.\lambda_r, \lambda_0.\gamma, \lambda_i) \text{ with } \lambda_i = (\lambda_0.\gamma \circ (\lambda_0.\gamma \bar{\circ} \lambda_{i-1}.\gamma), \lambda_{i-1}.\sigma)$$

This way, we have an infinite sequence of applicable transitions starting from the non-empty initial state. □

Using this construction we can proof its acyclicity.

Lemma 3.3.3 (Acyclicity)**.**

$$\forall i, j \in \mathbb{N} : i \neq j \implies \delta_i.\lambda_r \neq \delta_j.\lambda_r$$

Proof. W.l.o.g. $i < j$: We know by definition: $\forall m \in \mathbb{N} : \#(\delta_m.\lambda_b) < \#(\delta_m.\lambda_r)$
With $\delta_i.\lambda_r \longrightarrow^{j-i-1} \delta_j.\lambda_b$ we know $\#(\delta_i.\lambda_r) \leq \#(\delta_j.\lambda_b) < \#(\delta_j.\lambda_r)$. □

Hence, there is an infinite (3.3.2), acyclic (3.3.3) sequence of transitions applicable from Λ^I, so Γ, Δ and Λ have to be infinite. □

We can see that the function value of $\#$ increases after each type raising. The composition rule can have a similar property, e.g./ when composing *n/n/np* with *np/pp/s* to get *n/n/pp/s*. In contrast, the application rule poses no problem: The number of slashes decreases after each application.

We now want to limit the number of elements in Γ while keeping safety in mind. That means, we still want to be able to proof that a dead end in the new intermediate planning instance is also a dead end in the CCG task. To do so, we introduce an abstraction on the syntactic categories collapsing an infinite amount of different categories into a single one. We then allow combinations in an unrestrictive manner to preserve safety.

The idea is to define a maximal number of slashes k of which a category can consist. Everything after the k-th slash will then be discarded. Discarding everything on the left hand side would result in a loss of information about what the category results in. As a result we would have to consider every category to result in an s which renders the problem unnecessarily easy. For this reason we discard everything on the right hand side.

We adapt the composition and application rules in a way that all formerly possible combinations are still possible; the loss of information leads to some invalid combinations, though.

Definition 3.3.1. Let $\alpha_k : \Gamma \to \Gamma^*$ be a parametrized abstraction with $k > 1$ mapping a category onto either the identity or a *star category* γ^*, which get a special treatment for combinations. We will see this in Chapter 3.3.2.

$$\alpha_k(\gamma) = \begin{cases} \gamma & \text{if } \#(\gamma) \leq k \\ \alpha'_k(\gamma) & \text{otherwise} \end{cases}$$

$$\alpha'_k(\gamma) = \begin{cases} \gamma^* & \text{if } \#(\gamma) = k \\ \alpha'_k(\gamma_L) & \text{if } \gamma = \gamma_L \circ \gamma_R \wedge \#(\gamma_L) \geq k \\ \gamma_L \circ \alpha'_{k-\#(\gamma_L)-1}(\gamma_R) & \text{if } \gamma = \gamma_L \circ \gamma_R \wedge \#(\gamma_L) < k \end{cases}$$

Example 3.3.1.

$$\begin{aligned} &\alpha_2(s/np) = s/np \\ &\alpha_2(s/np\backslash s/np) = s/np\backslash s^* \\ &\alpha_2(s/(s\backslash np/pp)) = s/(s\backslash np^*) \\ &\alpha_2(s/(s\backslash np/pp)/np) = s/(s\backslash np^*) \end{aligned}$$

We say $\Gamma^* = \alpha_k(\Gamma)$ and define for convenience for all sets S and $\lambda \in \Lambda$:

$$\alpha_k(S) = \bigcup_{s \in S} \{\alpha_k(s)\}$$

$$\alpha_k(\lambda) = (\alpha_k(\lambda.\gamma), \lambda.\sigma) \in \Lambda^* = \Gamma^* \times \Sigma$$

Furthermore we find $(\gamma_L \circ \gamma_R)^* = \gamma_L \circ \gamma_R^*$ to be valid since we never remove slashes from the left hand side of a slash while preserving the right hand side.
To see how star categories are handled, consider the following examples:

$$\dfrac{(np)^* \quad pp}{(np)^*}>$$

This way we cover the possibility that $(np)^*$ was formally, i.e. before applying the abstraction, np/pp, as well as $np/\gamma_1/pp$ for any γ_1.

$$\dfrac{a/b^* \quad b}{a}>$$

So we allow the star category to become a normal category. This is necessary e.g. for a type raised goal category to become $\lambda^G.\gamma$ instead of $\lambda^G.\gamma^*$ so that the task can be completed.
Not using parenthesis around categories introduces no ambiguities, hence $(\gamma)^* = \gamma^*$.

3.3.1 Finiteness Considerations

We can see that the proof for theorem 3.3.1 (None-trivial CCG tasks are infinite) breaks now because lemma 3.3.3's proof relies on the fact that each type raising will yield a larger category regarding the number of slashes. By limiting the number of slashes any category can have, we will naturally restrict Γ to be finite under one assumption: the number of atomic categories[3] is finite. This assumption is perfectly valid since we retrieve the initial edges from a finite lexicon and the only way to introduce an arbitrary new atomic category γ_{new} is raising by type γ_{new}. But there is no use in doing so because there is no partner for combinations except results from other raises by γ_{new}. However, in these cases we could have raised by any other already present type in the first place with equivalent results so it makes no sense to raise by γ_{new}.

3.3.2 Second Compilation Approach

We now extend the set of transitions by taking star categories into account. To do so, we say $\Lambda^* = \Gamma^* \times \Sigma$ and revise the predicates determining which categories can be combined. Due to the star categories, we no longer require strict equality, it suffices to have a *match* for which definition we also need to revise *right* to be able to return $*$ as wildcard to be matched with anything. We leave *left* as it is since a star can never occur on the left hand side of a slash.

$$\mathit{isStar}(\gamma) \equiv \begin{cases} \gamma = (\gamma_G)^* & \text{if } \gamma \neq \gamma_L \circ \gamma_R \\ \mathit{isStar}(\gamma_R) & \text{otherwise } \gamma = \gamma_L \circ \gamma_R \end{cases} \tag{3.7}$$

$$\begin{aligned} \mathit{match}(\gamma_0, \gamma_1) \equiv\ & (\gamma_0 = \gamma_1) \lor \gamma_0 = (\gamma_G)^* \lor \gamma_1 = (\gamma_G)^* \\ & \lor ((\gamma_0 = \gamma_0' \circ \gamma_0'') \land (\gamma_1 = \gamma_1' \circ \gamma_1'') \\ & \land \mathit{match}(\gamma_0'', \gamma_1'') \land \mathit{match}'(\gamma_0', \gamma_1')) \end{aligned} \tag{3.8}$$

$$\mathit{match}'(\gamma_0, \gamma_1) \equiv (\gamma_0 = \gamma_1) \lor ((\gamma_1 = \gamma_1' \circ \gamma_1'') \land \mathit{match}'(\gamma_0, \gamma_1')) \tag{3.9}$$

So *apply* and *comp* become

$$\mathit{appl}^*(\gamma_0, \gamma_1) = (\gamma_0 = \gamma_0' \circ \gamma_0'') \land \mathit{match}(\gamma_0'', \gamma_1) \tag{3.10}$$

$$\mathit{comp}^*(\gamma_0, \gamma_1) = (\gamma_0 = \gamma_0' \circ \gamma_0'') \land (\gamma_1 = \gamma_1' \circ \gamma_1'') \tag{3.11}$$

$$\land ((\mathit{match}(\gamma_0'', \gamma_1') \land (\circ = \text{‘}/\text{’})) \lor (\mathit{match}(\gamma_0', \gamma_1'') \land (\circ = \text{‘}\backslash\text{’}))) \tag{3.12}$$

The definition of *match* makes sure that $\mathit{match}(a/(b/c^*), a/(b/c/d))$ holds. This is necessary to allow compositions like:

$$\frac{s/(a/(b/c^*)) \quad a/(b/c/d)/np}{s/np} {>}\mathbf{B}$$

[3]reminder: γ atomic $\Leftrightarrow \#(\gamma) = 0$

Using this, we define the set Δ^{B*} including transitions with star categories.

$$\Delta^{B*} = \Delta^{B1} \cup \Delta^{B2} \cup \Delta^{B3} \cup \Delta^{B4}$$

$$\Delta^{B1} = \bigcup_{\lambda,\lambda' \in \Lambda^*} \{(\lambda, \lambda', (\lambda.\gamma, \lambda.\sigma \vee \lambda'.\sigma) | isStar(\lambda.\gamma) \wedge (\lambda.\sigma \wedge \lambda'.\sigma = \bot^n)\} \tag{3.13}$$

$$\Delta^{B2} = \bigcup_{\lambda,\lambda' \in \Lambda^*} \{(\lambda, \lambda', (\gamma_L, \lambda.\sigma \vee \lambda'.\sigma)) | appl^*(\lambda.\gamma, \lambda'.\gamma \wedge (\lambda.\sigma \wedge \lambda'.\sigma = \bot^n)\} \tag{3.14}$$

$$\Delta^{B3} = \bigcup_{\lambda,\lambda' \in \Lambda^*} \{(\lambda, \lambda', (\gamma_{res}, \lambda.\sigma \vee \lambda'.\sigma) | \lambda.\gamma = \gamma_L \circ \gamma_R \wedge \lambda'.\gamma = \gamma'_L \circ \gamma'_R$$
$$\wedge\, comp^*(\lambda.\gamma, \lambda'.\gamma) \wedge (\lambda.\sigma \wedge \lambda'.\sigma = \bot^n) \wedge \gamma_{res} = \begin{cases} \gamma_L \circ \gamma'_R & \text{if } \circ = \text{‘/’} \\ \gamma'_L \circ \gamma_R & \text{otherwise} \end{cases} \} \tag{3.15}$$

$$\Delta^{B4} = \bigcup_{\delta \in \Delta^B} \{\delta | \delta.\lambda_1, \delta.\lambda_2 \in (\Gamma^* \cap \Gamma)\} \tag{3.16}$$

By 3.13 we allow star-categories to be applied to or composed with other categories without losing the star-category status. To allow termination, i.e. becoming a non-star-category, we have rule 3.14 for the application and 3.15 for the composition rule. Lastly, 3.16 allows to keep all transitions where both operands are not star categories.

Definition 3.3.2. We retrieve another intermediate planning task $\Theta^{\alpha_k} = (V_2, A_2, I_2, G_2)$ by defining V_2, I_2, G_2, and A_2 very similar to their corresponding sets in Θ^1. We now use a fact $s_{\alpha_k(\lambda)}$ representing whether or not we have reached $\alpha_k(\lambda)$ instead of λ. This way, the number of facts is depending on $\Gamma^* \times \Sigma$ instead of $\Gamma \times \Sigma$. Furthermore we use Δ^{B*} instead of Δ^B to make sure all newly acquired combinations are covered as well:

$$V_2 := \bigcup_{\lambda \in \Lambda} \{s_{\alpha_k(\lambda)}\} \tag{3.17}$$

$$I_2 := \bigcup_{\lambda \in \Lambda} \{s_{\alpha_k(\lambda)} := (\lambda \in \Lambda^I)\} \tag{3.18}$$

$$G_2 := \{s_{\alpha_k(\lambda^G)} := \top\} \tag{3.19}$$

$$A_2 := A_2^B \cup A_2^U \tag{3.20}$$

$$A_2^B := \bigcup_{\delta \in \Delta^{B*}} \{a_\delta^B\} \tag{3.21}$$

$$pre_{a_\delta^B} := \{s_{\delta.\lambda_1} := \top, s_{\delta.\lambda_2} := \top\} \tag{3.22}$$

$$eff_{a_\delta^B} := \{s_{\alpha_k(\delta.\lambda_3)} := \top\} \tag{3.23}$$

$$A_2^U := \bigcup_{\delta \in \Delta^U} \{a_\delta^U\} \tag{3.24}$$

$$pre_{a_\delta^U} := \{s_{\alpha_k(\delta.\lambda_b)} := \top\} \tag{3.25}$$

$$eff_{a_\delta^U} := \{s_{\alpha_k(\delta.\lambda_r)} := \top\} \tag{3.26}$$

In the next section we will proof the finiteness of our construction and afterwards show that Θ^{α_k} simulates the Θ^1 in terms of solvability. Unlike with Θ^{CCG} and Θ^1 we will see that this is no bisimulation[4].

[4]Note that we omitted the proof for bisimulation since we only need the weaker proposition of equivalent goal statuses.

Algorithm 1 Computing Γ^*

```
1: $\Gamma := \Gamma_0$
2: for $i$ in $1 \dots k+1$ do
3:     $\Gamma_i := \emptyset$
4:     for $\gamma$ in $\Gamma_{i-1}$ do
5:         for $\gamma'$ in $\Gamma_0$ do
6:             $\Gamma_i := \Gamma_i \cup \{\gamma/\gamma', \gamma/(\gamma'), \gamma\backslash(\gamma'), \gamma\backslash\gamma', \gamma'/\gamma, \gamma'\backslash\gamma\}$
7:         end for
8:     end for
9:     $\Gamma := \Gamma \cup \Gamma_i$
10: end for
11: for $\gamma \in \Gamma$ with $\#(\gamma) = k+1$ do
12:     replace $\gamma$ by $\alpha_k(\gamma)$
13: end for
```

3.3.3 Space and Time Analysis

We now want to proof that the size of Θ^{α_k} is space bound, i.e. not infinite. To do so, we first show some lemmas and assemble them afterwards.
In regards of a later proof of tractability we already introduce an algorithm to compute Γ and proof it time bound. Since we know $P \subseteq PSPACE$, we can conclude the component to be space bound as well. We denote $|\Gamma_0|$ by x.

Γ lays the ground for all further definitions, hence we will compute this first.

Lemma 3.3.4. *Finding $\alpha_k(\Gamma)$ is polynomially time bound by x and exponentially bound by k.*

Proof. With Algorithm 1 we have an algorithm computing Γ^* which first computes

$$\bigcup_{i \leq k+1} \{\Gamma_i\}$$

in line 2 - 10. To obtain Γ_i, we successively add elements of Γ_0 to element of Γ_{i-1} (line 6). This is then extended by categories with star status by replacing categories with too many slashes with their according star category (line 12), which all have exactly k slashes[5]. This coincides with the rules we defined in 3.13 - 3.16. The runtime r_Γ is bound by

$$r_\Gamma(x,k) \leq 2 * \sum_{i=0}^{k+1} x^i \in \mathcal{O}(x^k)$$

for building the cross product over Γ_i with Γ_0 for $i \in \{0, \dots, k+1\}$ and iterating over the result once afterwards. □

Lemma 3.3.5. *V_2 is finite.*

Proof. Since acquiring Γ^* is time bound by Lemma 3.3.4, Γ^* itself is space bound and thereby finite. We know that Σ is finite, so the cross product $\Lambda^* = \Gamma^* \times \Sigma$ is finite as well. As a trivial result, V_2 is finite. □

[5]This algorithm has an illustrative purpose and is thus not very efficient.

Lemma 3.3.6. *A_2 is finite.*

Proof. We show this property in two steps. First we proof Δ^{B*} and thereby A_2^B finite. For $\Delta^{B1} - \Delta^{B3}$ we need to iterate over the cross product of Λ^* with itself and compute simple predicates whose structural recursion obviously terminates always. Λ^*'s size is the multiplication of Γ^* and Σ. Hence the runtime of each is bound by

$$r_{\Delta^{B1-B3}}(x,k,n) \leq 3 * (r_\Gamma(x,k) * |\Sigma|)^2 = 3 * (r_\Gamma(x,k) * 2^n)^2 \in \mathcal{O}((r_\Gamma(x,k) * 2^n)^2).$$

We can see that this set's size grows exponentially in the number of semantic items. We will have to compensate this for a tractable compilation later on in Section 3.4.
Obtaining Δ^{B4} is a little more involved. Obviously we cannot filter the infinite set Δ by any predicate, neither can we build the intersection $\Gamma^* \cap \Gamma$ in a straight forward way. However, the latter can be acquired by removing all star categories from Γ^* in linear runtime.
Instead of filtering Δ^B, we use the contrary approach of building possible combinations based on the set of categories we already have. The result is not necessarily in Γ^*, we ensure this membership when building up A_2^B by applying the abstraction on the result (cf. 3.23). For this, we use oracle functions determining whether two categories can be combined and how the result looks like. We have already spelled out the details on this in definitions 3.4 and 3.5, so we omit explicitly defining these functions. We use *composable_fwd* and *composable_bkwd* to check whether two categories can be composed with the forward and backward rules respectively. The same holds for *applicable_fwd* and *applicable_bkwd*. The results are given by *compose_fwd*, *compose_bkwd*, *apply_fwd*, and *apply_bkwd*. Putting all this together gives us Algorithm 2 where we iterate over the categories contained in the earlier mentioned intersection of Γ^* and Γ and add combination results to Δ^{B4}. We can see that the runtime is bound by $|\Lambda^*|^2$ so $r_{\Delta^{B4}}(x,k) \in \mathcal{O}((r_{\Gamma^*}(x,k) * 2^n)^2)$.
Computing A_2^B involves iterating over Δ^{B*} once, which takes linear time, so

$$r_{A_2^B}(x,k) \in \mathcal{O}(r_{\Gamma^*}(x,k)^2),$$

which completes the first part of the proof.

For the second part it suffices to show that A_2^U is finite.
For this we need to iterate over Δ^U which is not possible. Alternatively, we can simply take all elements of Λ^*, apply the type raising rule, and the abstraction on the result. Since Λ^* contains all possible categories with a limited number of slashes, this method yields the same result as iterating over Δ^U. In Algorithm 3 we can see a way to compute Δ^{U*}: We iterate over the set of possible edges and raise each element by every possible type in a forward and a backward fashion. We can use this result instead of Δ^U in 3.24. The runtime is bound by $|\Lambda^*|$ and r_{Γ^*} so

$$r_{\Delta^{U*}}(x,k,n) \in \mathcal{O}(r_{\Gamma^*}(x,k) * |\Sigma| * r_{\Gamma^*}(x,k) * 2) = \mathcal{O}(x^k * 2^n * x^k * 2) = \mathcal{O}(x^{2k} * 2^n).$$

As a result we can conclude that A_2 is finite. $\square$

Now we can finally proof the theorem:

Theorem 3.3.7. *$\Theta^{\alpha_k} = (V_2, A_2, I_2, G_2)$ is space bound.*

Algorithm 2 Computing Δ^{B4}

```
$\Gamma_B := (\Gamma^* \cap \Gamma)$
$\Delta^{B4} := \emptyset$
for $\gamma, \gamma' \in \Gamma_B$ do
    if $composable\_fwd(\gamma, \gamma') \wedge compose\_fwd(\gamma, \gamma') \notin \Gamma_B$ then
        $\Delta^{B4} := \Delta^{B4} \cup \{(\gamma, \gamma', compose\_fwd(\gamma, \gamma'))\}$
    end if
    if $composable\_bkwd(\gamma, \gamma') \wedge compose\_bkwd(\gamma, \gamma') \notin \Gamma_B$ then
        $\Delta^{B4} := \Delta^{B4} \cup \{(\gamma, \gamma', compose\_bkwd(\gamma, \gamma'))\}$
    end if
    if $applicable\_fwd(\gamma, \gamma') \wedge apply\_fwd(\gamma, \gamma') \notin \Gamma_B$ then
        $\Delta^{B4} := \Delta^{B4} \cup \{(\gamma, \gamma', apply\_fwd(\gamma, \gamma'))\}$
    end if
    if $applicable\_bkwd(\gamma, \gamma') \wedge apply\_bkwd(\gamma, \gamma') \notin \Gamma_B$ then
        $\Delta^{B4} := \Delta^{B4} \cup \{(\gamma, \gamma', apply\_bkwd(\gamma, \gamma'))\}$
    end if
end for
```

Algorithm 3 Computing Δ^{U*}

```
$\Delta^{U*} := \emptyset$
for $\lambda \in \Lambda^*$ do
    for $\gamma \in \Gamma^*$ do
        for $\circ \in \{‘/’, ‘\backslash’\}$ do
            $\Delta^{U*} = \Delta^{U*} \cup \{(\lambda, \gamma, \gamma \circ (\gamma \bar{\circ} \lambda.\gamma))\}$
        end for
    end for
end for
```

Proof.

- V_2 is finite by lemma 3.3.5.
- A_2 is finite by lemma 3.3.6.
- I_2 is bound by V_2 and hence finite.
- $|G_2| = 1 \leq \infty$.

□

So with Θ^{α_k} we have a valid planning task which grows polynomially in the number of atomic categories and exponentially in the number of semantic items and k.

3.3.4 Simulation

Now we want to proof that Θ^{α_k} simulates Θ^1.
We first find and proof several auxiliary lemmas, then we construct a simulation relation which we can proof very easily using the lemmas.

Lemma 3.3.8.

$$\forall \gamma, \gamma' \in \Gamma : \mathit{appl}(\gamma, \gamma') \implies \mathit{appl}^*(\gamma, \gamma')$$

Proof.

$$\begin{aligned}
\mathit{appl}(\gamma, \gamma') &\iff \gamma = \gamma_L \circ \gamma_R \wedge \gamma_R = \gamma' \\
&\implies \gamma = \gamma_L \circ \gamma_R \wedge \mathit{match}(\gamma_R, \gamma') \\
&\iff \mathit{appl}^*(\gamma, \gamma')
\end{aligned}$$

□

Lemma 3.3.9.

$$\forall \gamma, \gamma' \in \Gamma : \mathit{comp}(\gamma, \gamma') \implies \mathit{comp}^*(\gamma, \gamma')$$

Proof.

$$\begin{aligned}
\mathit{comp}(\gamma, \gamma') \iff & \mathit{right}(\gamma) \neq \bot \wedge \mathit{right}(\gamma') \neq \bot \wedge \mathit{fwd}(\gamma) = \mathit{fwd}(\gamma') \\
& \wedge \begin{cases} \mathit{right}(\gamma) = \mathit{left}(\gamma') & \text{if } \mathit{fwd}(\gamma) \\ \mathit{right}(\gamma') = \mathit{left}(\gamma) & \text{otherwise} \end{cases} \\
\iff & \gamma = \gamma_L \circ \gamma_R \wedge \gamma' = \gamma'_L \circ \gamma'_R \wedge \begin{cases} \mathit{right}(\gamma) = \mathit{left}(\gamma') & \text{if } \mathit{fwd}(\gamma) \\ \mathit{right}(\gamma') = \mathit{left}(\gamma) & \text{otherwise} \end{cases} \\
\implies & \gamma = \gamma_L \circ \gamma_R \wedge \gamma' = \gamma'_L \circ \gamma'_R \wedge (\mathit{right}(\gamma) = \mathit{left}(\gamma') \vee \mathit{right}(\gamma') = \mathit{left}(\gamma)) \\
\implies & \gamma = \gamma_L \circ \gamma_R \wedge \gamma' = \gamma'_L \circ \gamma'_R \\
& \wedge (\mathit{match}(\mathit{right}(\gamma), \mathit{left}(\gamma')) \vee \mathit{match}(\mathit{right}(\gamma'), \mathit{left}(\gamma))) \\
\iff & \mathit{comp}^*(\gamma, \gamma')
\end{aligned}$$

□

Lemma 3.3.10. *Given the left argument has less slashes, match recognizes equality under α, i.e.*

$$\gamma = \gamma' \implies \forall\ 0 < i \leq k : \mathit{match}(\alpha_i(\gamma), \alpha_k(\gamma'))$$

Proof.
We have three cases:
Case 1: $\alpha_i(\gamma) = \gamma \wedge \alpha_k(\gamma') = \gamma'$.
This case is trivial since $\mathit{match}(\gamma, \gamma)$ holds for all γ.
Case 2: $\alpha_i(\gamma) \neq \gamma \wedge \alpha_k(\gamma') = \gamma'$
This case cannot occur since $i \leq k$.
Case 3: $\alpha_k(\gamma') \neq \gamma'$
For $0 < i \leq k$, we know:

$$\alpha_i(\gamma) = \gamma_{\alpha L} \circ \gamma_{\alpha R} \wedge \alpha_k(\gamma') = \gamma'_{\alpha L} \circ' \gamma'_{\alpha R} \text{ with } \circ' \in \{`/\text{'}, `\backslash\text{'}\}$$

And by definition of *match*:

$$\begin{aligned}
\mathit{match}(\gamma_0, \gamma_1) \equiv\ & \gamma_0 = \gamma_1 \vee \gamma_0 = \gamma_G^* \vee \gamma_1 = \gamma_G^* \\
& \vee (\gamma_0 = \gamma'_0 \circ \gamma''_0 \wedge \gamma_1 = \gamma'_1 \circ \gamma''_1 \\
& \wedge \mathit{match}(\gamma''_0, \gamma''_1) \wedge \mathit{match}'(\gamma'_0, \gamma'_1)) \\
\mathit{match}(\alpha_i(\gamma), \alpha_k(\gamma')) \equiv\ & \alpha_i(\gamma) = \alpha_k(\gamma') \vee \alpha_i(\gamma) = (\gamma_G)^* \vee \alpha_k(\gamma') = (\gamma_G)^* \\
& \vee (\alpha_i(\gamma) = \gamma_{\alpha L} \circ \gamma_{\alpha R} \wedge \alpha_k(\gamma') = \gamma'_{\alpha L} \circ' \gamma'_{\alpha R} \\
& \wedge \mathit{match}(\gamma_{\alpha R}, \gamma'_{\alpha R}) \wedge \mathit{match}'(\gamma_{\alpha L}, \gamma'_{\alpha L}))
\end{aligned}$$

We know that $match'(\gamma_{\alpha L}, \gamma'_{\alpha L})$ holds since $\gamma = \gamma'$ and $i \leq k$: α leaves everything left of the k-th/i-th slash as it was and $match'$'s recursion is on the left hand side of the topmost slash. So at some point, both arguments have to be equal, since we successively remove slashes from the right argument, which has k slashes, i.e. a greater or equal amount than the left argument.
Moreover, we know that $match(\gamma_{\alpha R}, \gamma'_{\alpha R})$ holds since after $i - \#(\gamma_{\alpha L})$ recursive calls, the left argument has to be a star category. This number of recursive steps is possible because

$$\#(\gamma'_{\alpha R}) = k - \#(\gamma'_{\alpha L}) \geq i - \#(\gamma'_{\alpha L}) \geq i - \#(\gamma_{\alpha L}),$$

where the last step can be justified by $match'(\gamma_{\alpha L}, \gamma'_{\alpha L})$. □

First we notice that every category γ with $\#(\gamma) \geq 1$ has the following form before and after the application of α_k:

$$\gamma = \gamma_0 \circ_0 \ldots \circ_i \gamma_i$$
$$\alpha_k(\gamma) = \gamma_0 \circ_0 \ldots \circ_{j-2} \gamma_{j-1} \circ_{j-1} \gamma_\alpha$$

where again two cases can occur:

a) $\gamma_\alpha = \gamma_j^*$

b) $\gamma_\alpha = \alpha_{k-\eta}(\gamma_j)$ with $\eta = 1 + \sum_{x=0}^{j-1} \#(\gamma_x)$

We now say $\gamma_L = \gamma_0 \circ_0 \ldots \circ_{j-2} \gamma_{j-1}$ and observe further, that by applying α, we lose information about $\gamma_{j+1} \ldots \gamma_i$. This is compensated by allowing combinations with all categories and choosing the unchanged left operand as result (see definition of Δ^{B1} 3.13). This way, no combination gets lost. For $i = j$, this obviously has no impact at all. Now, to combine $\alpha_k(\gamma) = \gamma_L \circ \gamma_\alpha$, only the last category is relevant. Using this observation, we can proof the following two lemmas:

Lemma 3.3.11.

$$appl(\gamma, \gamma') \implies appl^*(\alpha_k(\gamma), \alpha_k(\gamma')) \tag{3.27}$$

Proof.

$$appl(\gamma, \gamma') \iff appl(\gamma_L \circ \gamma_j, \gamma')$$
$$\overset{def}{\iff} \gamma_j = \gamma'$$

In the before-mentioned case a) we know

$$\begin{aligned} \gamma_j = \gamma' &\implies match(\gamma_j, \gamma') \\ &\implies match(\gamma_j^*, \gamma') \\ &\implies appl^*(\gamma_L \circ \gamma_j^*, \gamma') \\ &\overset{(*)}{\iff} appl^*(\alpha_k(\gamma), \alpha_k(\gamma')) \end{aligned}$$

$(*)$ We know $\#(\gamma_j) \leq k$ and thus $\#(\gamma') \leq k$
Hence

$$appl(\gamma_L \circ \gamma_j, \gamma') \implies appl^*(\alpha_k(\gamma), \alpha_k(\gamma')$$

For case b) we know $\gamma_\alpha = \alpha_{k-\eta}(\gamma_j)$ and $k - \eta \leq k$, hence

$$\begin{aligned}\gamma_j = \gamma' &\implies \mathit{match}(\gamma_j, \gamma') \\ &\overset{3.3.10}{\implies} \mathit{match}(\alpha_{k-\eta}(\gamma_j), \alpha_k(\gamma')) \\ &\implies \mathit{appl}^*(\gamma_L \circ \gamma_\alpha, \alpha_k(\gamma')) \\ &\iff \mathit{appl}^*(\alpha_k(\gamma), \alpha_k(\gamma')\end{aligned}$$

□

Lemma 3.3.12.

$$\mathit{comp}(\gamma, \gamma') \implies \mathit{comp}^*(\alpha_k(\gamma), \alpha_k(\gamma'))$$

Proof. We can see that it actually makes no difference whether we have a forward or backward composition, so we consider the forward application only.

$$\begin{aligned}\mathit{comp}(\gamma, \gamma') &\iff \mathit{comp}(\gamma_L \circ \gamma_j, \gamma') \\ &\iff \gamma' = \gamma'_L \circ \gamma'_R \wedge \gamma_j = \gamma'_L\end{aligned}$$

However, we distinguish between the following cases.
Case 1: $\alpha_k(\gamma'_L) = \gamma'_L$

$$\begin{aligned}\gamma' = \gamma'_L \circ \gamma'_R \wedge \gamma_j = \gamma'_L &\iff \alpha_k(\gamma') = \gamma'_L \circ \gamma'_R \wedge \gamma_j = \gamma'_L \\ &\implies \alpha_k(\gamma') = \gamma'_L \circ \gamma'_R \wedge \mathit{match}(\gamma_j, \gamma'_L)\end{aligned}$$

In case a) we get

$$\begin{aligned}\alpha_k(\gamma') = \gamma'_L \circ \gamma'_R \wedge \mathit{match}(\gamma_j, \gamma'_L) &\iff \alpha_k(\gamma') = \gamma'_L \circ \gamma'_R \wedge \mathit{match}(\gamma_j^*, \gamma'_L) \\ &\implies \mathit{comp}^*(\gamma_L \circ \gamma_j^*, \alpha_k(\gamma')) \\ &\implies \mathit{comp}^*(\alpha_k(\gamma), \alpha_k(\gamma'))\end{aligned}$$

In case b) we can conclude

$$\begin{aligned}\alpha_k(\gamma') = \gamma'_L \circ \gamma'_R \wedge \mathit{match}(\gamma_j, \gamma'_L) &\overset{3.3.10}{\implies} \alpha_k(\gamma') = \gamma'_L \circ \gamma'_R \wedge \mathit{match}(\alpha_{k-\eta}(\gamma_j), \alpha_k(\gamma'_L)) \\ &\implies \mathit{comp}^*(\gamma_L \circ \alpha_{k-\eta}(\gamma_j), \alpha_k(\gamma')) \\ &\implies \mathit{comp}^*(\alpha_k(\gamma), \alpha_k(\gamma'))\end{aligned}$$

Case 2: $\alpha_k(\gamma'_L) \neq \gamma'_L$ (*).

$$\begin{aligned}\gamma' = \gamma'_L \circ \gamma'_R \wedge \gamma_j = \gamma'_L &\overset{k>0}{\implies} \alpha_k(\gamma') = \gamma'_{\alpha L} \circ \gamma'_{\alpha R} \wedge \gamma_j = \gamma'_L \\ &\implies \alpha_k(\gamma') = \gamma'_{\alpha L} \circ \gamma'_{\alpha R} \wedge \mathit{match}(\gamma_j, \gamma'_L)\end{aligned}$$

Case a) cannot occur because after (*) we know that $\#(\gamma'_L) \geq k$ and thus $\#(\gamma_j) \geq k$. Since $\#(\gamma_L) > 0$, we can conclude $\#(\gamma_L \circ \gamma_j^*) \neq k$ which is a contradiction to $\alpha_k(\gamma) = \gamma_L \circ \gamma_j^*$.

Case b):

$$\begin{aligned}\alpha_k(\gamma') = \gamma'_{\alpha L} \circ \gamma'_{\alpha R} \wedge \mathit{match}(\gamma_j, \gamma'_L) &\overset{3.3.10}{\implies} \alpha_k(\gamma') = \gamma'_{\alpha L} \circ \gamma'_{\alpha R} \wedge \mathit{match}(\alpha_{k-\eta}(\gamma_j), \alpha_k(\gamma'_L)) \\ &\overset{(*)}{\implies} \alpha_k(\gamma') = \gamma'_{\alpha L} \circ \gamma'_{\alpha R} \wedge \mathit{match}(\alpha_{k-\eta}(\gamma_j), \alpha_k(\gamma')) \\ &\iff \mathit{comp}^*(\gamma_L \circ \alpha_k(\gamma), \alpha_k(\gamma')) \\ &\iff \mathit{comp}^*(\alpha_k(\gamma), \alpha_k(\gamma'))\end{aligned}$$

□

Theorem 3.3.13 (Simulation Relation)**.** *Let $\sim \subseteq S_1 \times S_2$ be a relation defined as*

$$s \sim s' \iff \forall \{s_\lambda := \top\} \in s : \{s_{\alpha_k(\lambda)} := \top\} \in s'.$$

$\sim$ is a simulation relation, i.e.

$$\begin{aligned}&\forall s_1, s_1' \in S_1, s_2 \in S_2, a_{\delta_i} \in A_1\\ &s_1 \sim s_2 \wedge s_1 \xrightarrow{a_{\delta_i}} s_1' \implies \exists s_2' \in S_2 . \exists a \in A_2 : \; s_2 \xrightarrow{a} s_1' \wedge s_1' \sim s_2'\end{aligned}$$

Proof. For actions in A_2 there are the same constraints on semantics as for actions in A_1, i.e. we require disjointness for an action to be existent and the result is the disjunction. Hence, we do not need to consider this during the proof. Furthermore, the desired property holds for unary actions by definition. Considering actions where both operands remain unchanged by α_k, we trivially have an element in Δ^{B4} giving us an action in A_2 with the desired property.
We now distinct two cases:
Case 1:

$$\begin{aligned}\delta_i \in \Delta^{app} &\iff appl(\delta_i.\lambda_1.\gamma, \delta_i.\lambda_2.\gamma)\\ &\overset{3.3.11}{\implies} appl^*(\alpha_k(\delta_i.\lambda_1.\gamma), \alpha_k(\delta_i.\lambda_2.\gamma))\\ &\overset{(*)}{\implies} \exists a \in A_2 : pre_a = \{s_{\alpha_k(\delta.\lambda_1.\gamma)} := \top, s_{\alpha_k(\delta.\lambda_2.\gamma)} := \top\}\\ &\qquad \wedge \mathit{eff}_a = \{s_{\alpha_k(\delta.\lambda_3.\gamma)} := \top\} && (3.28)\\ &\implies \forall s \in S_2 : \{s_{\alpha_k(\delta.\lambda_3.\gamma)} := \top\} \in s \circ \langle a \rangle && (3.29)\end{aligned}$$

(*) Here, two cases can occur. Firstly, $\alpha_k(\lambda_3.\gamma) = \alpha_k(\lambda_1.\gamma)$. In this case we know that a is the result of an element of Δ^{B1}. Otherwise, i.e. $\alpha_k(\lambda_3.\gamma) \neq \alpha_k(\lambda_1.\gamma)$, there is a respective element in Δ^{B2} because $appl^*$ holds.

We can now instantiate the universal quantifier properly and use the definition of the simulation relation to get the desired property.

$$\begin{aligned}&s_1 \sim s_2 \wedge \delta_i \in \Delta^{app} \wedge s_1 \xrightarrow{a_{\delta_i}} s_1'\\ &\overset{a_{\delta_i}}{\implies} s_1 \sim s_2 \wedge \delta_i \in \Delta^{app} \wedge s_1 \xrightarrow{a_{\delta_i}} s_1' \wedge \{s_{\delta_i.\lambda_1.\gamma} := \top, s_{\delta_i.\lambda_2.\gamma} := \top\} \subseteq s_1\\ &\overset{3.28}{\implies} s_1 \sim s_2 \wedge \delta_i \in \Delta^{app} \wedge s_1 \xrightarrow{a_{\delta_i}} s_1'\\ &\qquad \wedge \exists a' \in A_2 : s_2 \circ \langle a' \rangle \neq \mathit{undefined} \wedge \mathit{eff}_{a'} = \{s_{\alpha_k(\delta.\lambda_3.\gamma)} := \top\}\\ &\overset{3.29}{\implies} s_1 \sim s_2 \wedge \delta_i \in \Delta^{app} \wedge s_1 \xrightarrow{a_{\delta_i}} s_1' \wedge \exists a' \in A_2 : \{s_{\alpha_k(\delta.\lambda_3.\gamma)} := \top\} \in s_2 \circ \langle a' \rangle\\ &\implies \exists a' \in A_2 : s_2 \xrightarrow{a'} s_2' \wedge s_1' \sim s_2'\end{aligned}$$

Case 2 works very similar because all significant changes in handling these kinds of actions are already dealt with in the preceding lemmas:

$$\begin{aligned}
\delta_i \in \Delta^{comp} &\iff \mathit{comp}(\delta_i.\lambda_1.\gamma, \delta_i.\lambda_2.\gamma) \\
&\overset{3.3.12}{\implies} \mathit{comp}^*(\alpha_k(\delta_i.\lambda_1.\gamma), \alpha_k(\delta_i.\lambda_2.\gamma)) \\
&\overset{(*)}{\implies} \exists a \in A_2 : \mathit{pre}_a = \{s_{\alpha_k(\delta.\lambda_1.\gamma)} := \top, s_{\alpha_k(\delta.\lambda_2.\gamma)} := \top\} \\
&\qquad \wedge \mathit{eff}_a = \{s_{\alpha_k(\delta.\lambda_3.\gamma)} := \top\} &(3.30)\\
&\implies \forall s \in S_2 : \{s_{\alpha_k(\delta.\lambda_3.\gamma)} := \top\} \in s \circ \langle a \rangle &(3.31)
\end{aligned}$$

(*) Confer the remark in case 1.
This allows us to conclude

$$\begin{aligned}
&s_1 \sim s_2 \wedge \delta_i \in \Delta^{comp} \wedge s_1 \xrightarrow{a_{\delta_i}} s_1' \\
&\overset{a_{\delta_i}}{\implies} s_1 \sim s_2 \wedge \delta_i \in \Delta^{comp} \wedge s_1 \xrightarrow{a_{\delta_i}} s_1' \wedge \{s_{\delta_i.\lambda_1.\gamma} := \top, s_{\delta_i.\lambda_2.\gamma} := \top\} \subseteq s_1 \\
&\overset{3.30}{\implies} s_1 \sim s_2 \wedge \delta_i \in \Delta^{comp} \wedge s_1 \xrightarrow{a_{\delta_i}} s_1' \\
&\qquad \wedge \exists a' \in A_2 : s_2 \circ \langle a' \rangle \neq \mathit{undefined} \wedge \mathit{eff}_{a'} = \{s_{\alpha_k(\delta.\lambda_3.\gamma)} := \top\} \\
&\overset{3.31}{\implies} s_1 \sim s_2 \wedge \delta_i \in \Delta^{comp} \wedge s_1 \xrightarrow{a_{\delta_i}} s_1' \wedge \exists a' \in A_2 : \{s_{\alpha_k(\delta.\lambda_3.\gamma)} := \top\} \in s_2 \circ \langle a' \rangle \\
&\implies \exists a' \in A_2 : s_2 \xrightarrow{a'} s_2' \wedge s_1' \sim s_2'
\end{aligned}$$

So we have a simulation relation by $\sim$ granting us the following corollary: □

Corollary 3.3.13.1.

$$\begin{aligned}
&\forall \langle a_{\delta_0}, \ldots, a_{\delta_k - 1} \rangle \in A_1 \times \ldots \times A_1 . \exists \langle a_0, \ldots, a_{k-1} \rangle \in A_2 \times \ldots \times A_2 : \\
&I_1 \circ \langle a_{\delta_0}, \ldots, a_{\delta_k - 1} \rangle \supseteq G_1 \iff I_2 \circ \langle a_0, \ldots, a_{k-1} \rangle \supseteq G_2
\end{aligned}$$

So this compilate is finite and safe, but it is not tractable because the size of the planning task grows exponential in the number of semantic items. We now introduce a new compilation compensating this problem.

3.4 Compensating the semantics

We can see that in Θ^{α_k} the problem is that we work on Λ^*, which is by definition exponential in size, due to its semantics component $\Sigma = \{\bot, \top\}^n$. The new approach is to split the handling of syntax, i.e. the syntactic categories and semantics, apart. We then keep track of the semantic items we already reached for each category using binary facts and compose edges with the same category by building the disjunction of each semantics to preserve safety. So for each former fact s_λ we have one fact $s_{\lambda.\gamma}$ representing its category and n facts $s^{\lambda.\gamma}_{\sigma[i]}$ representing its semantics. Initially, we build the disjunction over all edges with the same category and for each set bit, the respective fact is $\top$. The goal is to reach the goal category and all its semantic items have to be $\top$.

Definition 3.4.1. Let $\Pi^\Sigma = (V_3, A_3, I_3, G_3)$ be a planning task. Let

$$\sigma_\gamma = \bigwedge_{s_\lambda \in V_2 : \lambda.\gamma = \gamma} \lambda.\sigma$$

for $\gamma \in \Gamma^*$. So we can define:

$$V_3 := \bigcup_{s_\lambda \in V_2} (\{s_{\lambda.\gamma}\} \cup \bigcup_{i=0}^{n-1} \{s^{\lambda.\gamma}_{\sigma[i]}\}) \tag{3.32}$$

$$I_3 := \bigcup_{s_\lambda \in V_2} (\{s_{\lambda.\gamma} := (\{s_\lambda := \top\} \in I_2)\} \cup \bigcup_{i=0}^{n-1} \{s^{\lambda.\gamma}_{\sigma[i]} := \sigma_{\lambda.\gamma}[i]\}) \tag{3.33}$$

$$G_3 := \{s_{\lambda^G.\gamma} := \top\} \cup \bigcup_{i=0}^{n-1} \{s^{\lambda^G.\gamma}_{\sigma[i]} := \top\} \text{ with } G_2 = \{s_{(\lambda^G.\gamma, \top^n)} := \top\} \tag{3.34}$$

Regarding the actions, we cannot have one action per possible semantics vector and category to avoid exponential grow. We also cannot know the semantics a given category fulfills at all points during the search simply because in the initial state we already collapse several initial edges and merge their semantics disjunctively. Moreover, figuring out for a specific category what semantic items it can potentially fulfill, is as hard as solving the problem in the first place (consider the category being the goal category and the semantics vector being $\top^n$). As a result, we also do not require disjointness in semantics because we do not exactly know where each set bit came from. To clarify this, consider the following example:

Example 3.4.1. We have the initial edges $np : \langle 110\rangle, np : \langle 011\rangle, s\backslash np : \langle 100\rangle$, so the resulting initial state looks like:

$$I_3 \supseteq \{s_{np} := \top, s_{s\backslash np} := \top, s^{np}_{\sigma[0]} := \top, s^{np}_{\sigma[1]} := \top, s^{np}_{\sigma[2]} := \top,$$
$$s^{s\backslash np}_{\sigma[0]} := \top, s^{s\backslash np}_{\sigma[1]} := \bot, s^{s\backslash np}_{\sigma[2]} := \bot\}$$

When requiring $\sigma[0]$ to be $\bot$ to combine $s\backslash np$ with np, we cannot apply any action. As a result, we have to allow potential overlapping semantics.

The next challenge are *conditional effects*. When combining two edges, having a semantic item reached in an edge should result in it being set in the resulting edge as well, otherwise not. This is called conditional effect. Nebel discussed a way to compile these into FDR in [11] and we will use his technique allowing a polynomial time compilation. Instead of having one action distinguishing between the already reached semantics via having exponentially many different actions or using conditional effects, we introduce several actions simulating this behavior. Concretely, for each action a_i we have one action a_i^c in which effect we mark the resulting category to be reached. Additionally, for each semantic item with index j, we have two actions $a^{\sigma[j]}_{i,L}$ and $a^{\sigma[i]}_{i,R}$ for the left operand and the right operand respectively. The preconditions are that the semantic item is reached and the action's effect allows to reach it as well. So applying one action in the original task requires the application of several actions in the compilate. As a result, the length of a plan in the compilate increases significantly but we do not need to compensate this in any way: In the end we only care about having a dead end or not, i.e. we distinguish between values greater than 0 or not. The actual value makes no difference.

Example 3.4.2. Consider again example 3.4.1. We now get the following actions for combining the np with $s\backslash np$:

- a_0^c : $pre_a = \{s_{np} := \top, s_{s\backslash np} := \top\}$, $\mathit{eff}_a = \{s_s := \top\}$
- $a_{0,L}^{\sigma[0]}$: $pre_a = \{s_{np} := \top, s_{s\backslash np} := \top, s_{\sigma[0]}^{np} := \top\}$, $\mathit{eff}_a = \{s_{\sigma[0]}^{s} := \top\}$
- $a_{0,L}^{\sigma[1]}$: $pre_a = \{s_{np} := \top, s_{s\backslash np} := \top, s_{\sigma[1]}^{np} := \top\}$, $\mathit{eff}_a = \{s_{\sigma[1]}^{s} := \top\}$
- $a_{0,L}^{\sigma[2]}$: $pre_a = \{s_{np} := \top, s_{s\backslash np} := \top, s_{\sigma[2]}^{np} := \top\}$, $\mathit{eff}_a = \{s_{\sigma[2]}^{s} := \top\}$
- a_1^c : $pre_a = \{s_{np} := \top, s_{s\backslash np} := \top, s_{\sigma[0]}^{s\backslash np} := \top\}$, $\mathit{eff}_a = \{s_{\sigma[0]}^{s} := \top\}$
- $a_{1,R}^{\sigma[0]}$: $pre_a = \{s_{np} := \top, s_{s\backslash np} := \top, s_{\sigma[0]}^{s\backslash np} := \top\}$, $\mathit{eff}_a = \{s_{\sigma[0]}^{s} := \top\}$
- $a_{1,R}^{\sigma[1]}$: $pre_a = \{s_{np} := \top, s_{s\backslash np} := \top, s_{\sigma[1]}^{s\backslash np} := \top\}$, $\mathit{eff}_a = \{s_{\sigma[1]}^{s} := \top\}$
- $a_{1,R}^{\sigma[2]}$: $pre_a = \{s_{np} := \top, s_{s\backslash np} := \top, s_{\sigma[2]}^{s\backslash np} := \top\}$, $\mathit{eff}_a = \{s_{\sigma[2]}^{s} := \top\}$

Now we can define A_3 accordingly. For this let $A_2^B = \{a_0, \ldots, a_{m-1}\}$ with $pre_{a_i} = \{s_{\lambda_i} := \top, s_{\lambda_i'} := \top\}$ and $\mathit{eff}_{a_i} = \{s_{\lambda_i''} := \top\}$.

$$A_3 = A_3^B \cup A_3^U$$

$$A_3^B = \bigcup_{i=0}^{m} (\{a_i^c\} \cup \bigcup_{j=0}^{n-1} \{a_{i,L}^{\sigma[j]}, a_{i,R}^{\sigma[j]}\}) \tag{3.35}$$

$$pre_{a_i^c} = \{s_{\lambda_i.\gamma} := \top, s_{\lambda_i'.\gamma} := \top\} \tag{3.36}$$

$$\mathit{eff}_{a_i^c} = \{s_{\lambda_i''.\gamma} := \top\} \tag{3.37}$$

$$pre_{a_{i,L}^{\sigma[j]}} = \{s_{\lambda_i.\gamma} := \top, s_{\lambda_i'.\gamma} := \top, s_{\sigma[j]}^{\lambda_i.\gamma} := \top\} \tag{3.38}$$

$$\mathit{eff}_{a_{i,L}^{\sigma[j]}} = \{s_{\sigma[j]}^{\lambda_i''.\gamma} := \top\} \tag{3.39}$$

$$pre_{a_{i,R}^{\sigma[j]}} = \{s_{\lambda_i.\gamma} := \top, s_{\lambda_i'.\gamma} := \top, s_{\sigma[j]}^{\lambda_i'.\gamma} := \top\} \tag{3.40}$$

$$\mathit{eff}_{a_{i,R}^{\sigma[j]}} = \{s_{\sigma[j]}^{\lambda_i''.\gamma} := \top\} \tag{3.41}$$

Now, let analogously $A_2^U = \{a_0, \ldots a_{m-1}\}$ with $pre_{a_i} = \{s_{\lambda_i} := \top\}$ and $\mathit{eff}_{a_i} = \{s_{\lambda_i'} := \top\}$.

$$A_3^U = \bigcup_{i=0}^{m} (\{a_i^c\} \cup \bigcup_{j=0}^{n-1} \{a_{i,L}^{\sigma[j]}, a^{\sigma[j]}\}) \tag{3.42}$$

$$pre_{a_i^c} = \{s_{\lambda_i.\gamma} := \top\} \tag{3.43}$$

$$\mathit{eff}_{a_i^c} = \{s_{\lambda_i'.\gamma} := \top\} \tag{3.44}$$

$$pre_{a_i^{\sigma[j]}} = \{s_{\lambda_i.\gamma} := \top, s_{\sigma[j]}^{\lambda_i.\gamma} := \top\} \tag{3.45}$$

$$\mathit{eff}_{a_i^{\sigma[j]}} = \{s_{\sigma[j]}^{\lambda_i'.\gamma} := \top\} \tag{3.46}$$

$$\tag{3.47}$$

3.4.1 Space and Time Analysis

We now want to show that the compilation is polynomially time bound. Note that k can be chosen freely so even though the compilation is exponentially bound by k, it is polynomially bound by the CCG task. We refrain from mentioning this in the following.

Lemma 3.4.1. *Computing V_3 is polynomially time bound by the size of the CCG task.*

Proof. One can easily see that - due to set semantics - we can rewrite the definition of V_3 as

$$V_3 = \bigcup_{\gamma \in \Gamma^*} (\{s_\gamma\} \cup \bigcup_{i=0}^{n-1} \{s^{\gamma}_{\sigma[i]}\})$$

In lemma 3.3.4 we showed that $|\Gamma^*|$ is bound by $\mathcal{O}(x^k)$, so $r_{V_3}(x, k, n) \in \mathcal{O}(x^k * n)$ is an appropriate time and space bound. □

Lemma 3.4.2. *Computing A_3 is polynomially time bound.*

Proof. For binary actions we know that each category in Γ^* can maximally be combined with each other, so the number of actions is at most

$$\binom{|\Gamma|}{2} = \frac{|\Gamma|^2}{2} \in \mathcal{O}((x^k)^2). \tag{3.48}$$

Checking whether two categories can be combined happens in a negligible amount of time, so by 3.48 we have also found a time bound. For unary actions, we have one action per category in Γ^*, that means they are asymptotically irrelevant. □

Theorem 3.4.3. *Computing Π^Σ is polynomially time bound.*

Proof. We can compute V_3 and A_3 in polynomial time (lemma 3.4.1 and 3.4.2 resp.), so it is sufficient to show that the same holds for I_3 and G_3. By definition of I_2 we can say

$$\{s_\gamma := \top\} \in I_3 \iff \exists \lambda \in \Lambda^I : \lambda.\gamma = \gamma.$$

Regarding the facts representing the semantics of each category, we can initialize all of them with $\bot$ except the ones where there is an according element in Λ^I. Those have to be set in respect to the (initially known) semantics. This process is spelled out in algorithm 4. We can see that for each category in Γ we loop over Λ^I and $i \in \{0 \dots n\}$, so a runtime bound r_{I_3} can be constructed as follows:

$$r_{I_3}(x, k, n) \leq x^k * (|\Lambda^I| + n) \leq x^k * (x^k + n) \in \mathcal{O}((x^k)^2 + x^k * n).$$

To obtain G_3, we only need linear runtime bound by n, which completes the proof. □

Algorithm 4 Computing I_3

```
1: $I_3 := \emptyset$
2: for $\gamma \in \Gamma$ do
3:     $\lambda :=$ None
4:     for $\lambda^I.\gamma \in \Lambda^I$ do
5:         if $\lambda^I.\gamma = \gamma$ then
6:             $\lambda := \lambda^I$
7:             break
8:         end if
9:     end for
10:    if $\lambda =$ None then
11:        $I_3 := I_3 \cup \{s_\gamma := \bot\}$
12:        $\lambda := (\text{None}, \bot^n)$
13:    else
14:        $I_3 := I_3 \cup \{s_\gamma := \top\}$
15:    end if
16:    for $i := 0 \ldots (n-1)$ do
17:        $I_3 := I_3 \cup \{s^\gamma_{\sigma[i]} := \lambda.\sigma[i]\}$
18:    end for
19: end for
```

3.4.2 Simulation

The last step is to proof that the new construction simulates the last one. The proof is very simple, so we present only a proof sketch. For each action $a \in A_2$ representing a binary rule application there are two preconditions. These preconditions make sure that we already reached the necessary categories and their semantics do not overlap. By construction, there is a set of actions $A_a \subseteq A_3$ which preconditions only require the categories to be reached. We keep track of this by dedicated facts representing the category rather than the respective edges. So if a is applicable, all actions in A_a will be applicable, too. For each semantic item there is an action in A_a allowing us to carry over the semantic of each operand. Hence, we can apply up to $2 * n + 1$ (one action for the category and n actions per operand for the semantics) actions to reach a state simulating the respective state after applying a. For unary actions we use the same technique to keep track of the preconditions and the effect, especially allowing to carry over semantics of the category that is raised.

Carrying over semantics is not always necessary, because we collapse edges with the same category and use the disjunction of their semantics, but this way no formerly reached semantic item can get lost. Hence a plan leading to a goal state in Π^{α_k} can be simulated by a (potentially longer) plan in Π^{Σ}.

We have now shown that Π^{Σ} is a compilation for Θ^{CCG} which can be acquired in a polynomially bound runtime and is safe in terms of dead ends.

3.5 Combining CCG and planning

Our results so far are still lacking a crucial feature: We wanted to be able to classify edges as potentially useful or not. But the construction can only tell whether or not there could be a solution for the CCG task. That means, we have to adapt the problem one last time.

Definition 3.5.1. We call the edge λ^I the *edge in question*. We say the edge is *potentially useful* if there is a plan combining it with other edges to get a valid output. Otherwise, it is *useless*.

We encode this into the planning task Π^Σ by handling the edge in question strictly separate. We introduce facts representing only this edge, one fact s^C for its category and n facts $s^C_{\sigma[i]}$ for its semantics. During the search we call its category the *main category* which (as opposed to the other category-facts) specifies the category itself instead of its status as reached or not reached.

Example 3.5.1. Let the initial edges be $s/np : \langle 01\rangle, np : \langle 10\rangle$ with the latter one as the edge in question. As a result, we will get an initial state I containing the following information:

$$I \supset \{s_{s/np} := \top, s^{s/np}_{\sigma[0]} := \bot, s^{s/np}_{\sigma[1]} := \top, s^C := np, s^C_{\sigma[0]} := \top, s^C_{\sigma[1]} := \bot\}$$

The need of using the edge in question will be encoded in the goal state, where we require the main category to be $\lambda^G.\gamma$ and all its semantics facts to be $\top$. Regarding the actions, for each binary action we have to include two more actions, where the edge in question acts as left or right operand, and for each unary actions we have to add an action allowing the main category to be raised.

Definition 3.5.2. We define the compiled planning task $\Pi = (V, A, I, G)$ as follows:

$$V := V_3 \cup \{s^C\} \cup \bigcup_{i=0}^{n-1} \{s^C_{\sigma[i]}\} \tag{3.49}$$

$$I := I_3 \cup \{s^C := \lambda^I.\gamma\} \cup \bigcup_{i=0}^{n-1} \{s^C_{\sigma[i]} := \lambda^I.\sigma[i]\} \tag{3.50}$$

$$G := \{s^C := \lambda^G.\gamma\} \cup \bigcup_{i=0}^{n-1} \{s^C_{\sigma[i]:=\top}\} \tag{3.51}$$

We obtain A by extending A_3. Assume we have the following actions in A_3 representing one action in A_2 (cf. definitions 3.36 - 3.41):

$$\begin{aligned}
pre_{a_i^c} &= \{s_{\lambda_i.\gamma} := \top, s_{\lambda'_i.\gamma} := \top\}\\
\mathit{eff}_{a_i^c} &= \{s_{\lambda''_i.\gamma} := \top\}\\
pre_{a_{i,L}^{\sigma[j]}} &= \{s_{\lambda_i.\gamma} := \top, s_{\lambda'_i.\gamma} := \top, s^{\lambda_i.\gamma}_{\sigma[j]} := \top\}\\
\mathit{eff}_{a_{i,L}^{\sigma[j]}} &= \{s^{\lambda''_i.\gamma}_{\sigma[j]} := \top\}\\
pre_{a_{i,R}^{\sigma[j]}} &= \{s_{\lambda_i.\gamma} := \top, s_{\lambda'_i.\gamma} := \top, s^{\lambda'_i.\gamma}_{\sigma[j]} := \top\}\\
\mathit{eff}_{a_{i,R}^{\sigma[j]}} &= \{s^{\lambda''_i.\gamma}_{\sigma[j]} := \top\}
\end{aligned}$$

We include four actions to simulate this behavior for the main category as well. Hereby $\bar{a}_i^c$ and $\bar{\bar{a}}_i^c$ allow combinations where the main category is the left and right operand respectively while $\bar{a}_{i,L}^{\sigma[j]}$ and $\bar{a}_{i,R}^{\sigma[j]}$ allow to carry over the semantics of the left and right part respectively. We do not need actions carrying over the main categories semantics because the category itself changes, so there is no need to carry anything over.

$$\begin{aligned}
pre_{\bar{a}_i^c} &= \{s^C := \lambda_i.\gamma, s_{\lambda'_i.\gamma} := \top\} \\
\mathit{eff}_{\bar{a}_i^c} &= \{s^C := \lambda''_i.\gamma\} \\
pre_{\bar{\bar{a}}_i^c} &= \{s^C := \lambda'_i.\gamma, s_{\lambda_i.\gamma} := \top\} \\
\mathit{eff}_{\bar{\bar{a}}_i^c} &= \{s^C := \lambda''_i.\gamma\} \\
pre_{\bar{a}_{i,L}^{\sigma[j]}} &= \{s_{\lambda_i.\gamma} := \top, s^C := \lambda'_i.\gamma, s_{\sigma[j]}^{\lambda_i.\gamma} := \top\} \\
\mathit{eff}_{\bar{a}_{i,L}^{\sigma[j]}} &= \{s_{\sigma[j]}^C := \top\} \\
pre_{\bar{a}_{i,R}^{\sigma[j]}} &= \{s^C := \lambda_i.\gamma, s_{\lambda'_i.\gamma} := \top, s_{\sigma[j]}^{\lambda'_i.\gamma} := \top\} \\
\mathit{eff}_{\bar{a}_{i,R}^{\sigma[j]}} &= \{s_{\sigma[j]}^C := \top\}
\end{aligned}$$

The correctness of the construction is obvious: We can still simulate each CCG rule as before when the main category is not involved. Otherwise there is a respective action for the main category being the left or right operand. The only thing that changes is the goal: We only reach the goal when the main category is involved. Assume there is a rule sequence including the main category that leads to the goal. Then we can simulate each rule's application in the same way we did in Π^Σ. And whenever the main category or one of its combinations is involved, we can use the new actions to simulate that. This way it is assured, that the main category is the goal category in the end. All semantics can be carried over as before, so this criterion is fulfilled, too.

Regarding the runtime we can see that it is strictly bound by the size of Π^Σ, so the final compilation fulfills all our criteria. Nonetheless, theory and practice oftentimes diverge in several points, so we will discuss some of these aspects in the next section. Also the construction is polynomially time bound, and we will see in chapter 4 whether this is sufficient to be useful in practice or not.

3.6 Implementation

When realizing a sentence with OpenCCG, we do not want to invoke a planner using only its heuristic because we potentially do this hundreds or even thousands of times and invoking another program inside of a program is always accompanied by some computational overhead. This sums up during the process and slows the realization down unnecessarily. So it is faster to include the heuristic directly in the OpenCCG project such that we do not have to call another program but merely another method. That means, we do no longer compile the problem into a planning task but *adapt* it.

So there are two points in which we have to make changes to OpenCCG. Firstly, when we have gathered all initial edges, we use an *adapter* transforming these edges into a *planning instance* using the techniques presented in the preceding chapter. However, when building up the set of categories, we use a different, significantly more efficient

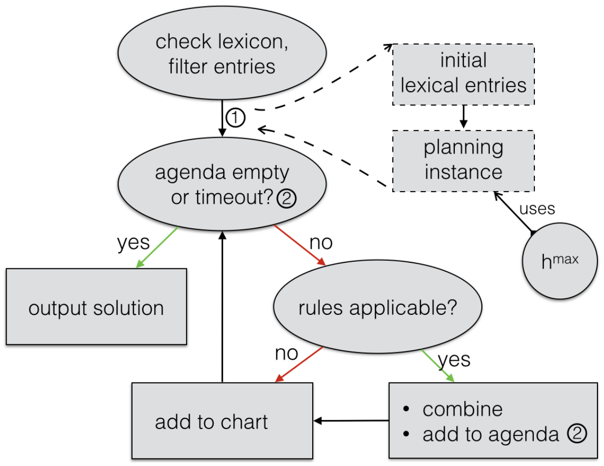

Figure 3.1: OpenCCG algorithm annotated with the changes

algorithm. This algorithm starts with the initial edges' categories and builds the cross product over them. Whenever two of them are combinable, we add the result to the set of already reached categories. Doing this successively will at some point yield a fix point (i.e. we do not discover new categories) in the worst case after discovering all of them. During this process we already gather all necessary information, that means which categories can be combined, to obtain the set of actions quickly. In practice this allows to prune various unreachable categories and due to the abstraction we have a guarantee to terminate. The abstraction itself is another point where theory and practice diverge: An permissive handling of star categories results in the problem, that merely any edge can be considered a dead end. As a result, we handle them restrictively, i.e. a star category cannot be combined with any other category. We discuss this decision more intensively in the Section 3.6.1.

The planning instance acts as base for the h^{max} implementation: it provides all necessary information h^{max} needs to work properly. This especially means that other heuristics can be plugged in without further changes. A quick overview about the potential behind this is given in Section 5. The implementation of h^{max} itself is completely generic and can potentially be enhanced with domain specific knowledge.

After creating the planning instance, OpenCCG continues with its initialization until the actual search starts, i.e. the agenda is initialized with the edges we used to build up Γ. But instead of adding them to the agenda, we first invoke the heuristic to potentially filter some of them out. This process is illustrated in Fig. 3.1 where in ① we do the precomputation for the heuristic and in ② we actually invoke the heuristic before adding edges to the agenda, especially not only when initializing, but also when an edge is about to be added to the agenda after a combination.

Moreover, we use CCG's type raise rule as the only unary rule unless the lexicon specifies otherwise. In practice, allowing any category to be raised by any type is often not necessary, so we can specify type raises that are only applicable on *np*s and *pp*s as proposed e.g. by Lewis and Steedman (2014) [9].

3.6.1 Restrictive versus Permissive

The adaptation we presented used a permissive handling for star categories. However, this only makes sense when the value for k is large enough, i.e. categories that occur very frequently are not affected, because otherwise it is easy to combine potential dead end edges with other edges such that the category becomes a star category. As a result, we can cover all EPs by combining with any edges, even though they are not syntactically compatible, because we have to allow such combinations due to the loss of information. Lastly, we can combine the star category to get the goal category relatively easy, because the underlaying non-star category is small in size.

Unfortunately, choosing k large enough is impractical, because of the memory consumption (as we will see in the next chapter). Hence, we have to work with a small value for k. To compensate the before-mentioned problem, we handle the star categories restrictively by disallowing all combinations, thus rendering edges with large categories dead ends right away.

Chapter 4

Experiments

The following experiments have been run on a machine with a 2.4 GHz Intel Core i5 processor and a 1600 MHz DDR3 RAM. The JVM had a memory limit of 4 GB.

We wanted to find answers to the following questions:

- What are the limits to k, i.e. how high can we chose k without running out of memory?
- Can we save time using the heuristic by speeding up the process of realization?
- How many nodes can we prune using the heuristic?
- Does it make a difference when invoking the heuristic only on initial edges or on all edges?

To find an answer to the last question, we have to keep the time limit in mind: When realizing larger sentences or paragraphs, the time limit might be exceeded and thus the search cut off. As a result, less edges are created, so comparing the number of created edges in a search with and without the heuristic might not be sufficient. Instead we introduced the concept of *hypothetical* edges. Intuitively, these are edges that would have been pruned if we used the heuristic but are kept for the sake of discovering the impact thereof. We acquire these edges the following way: Whenever an edge is created, we run the heuristic on it. If it reports a dead end status for this edge, we do add it to the agenda, but mark it as hypothetical. When a combination or unary rule application takes place, the resulting edge is marked hypothetical if either one of the parents is hypothetical or the edge itself is marked as a dead end. The process is illustrated in Fig. 4.1 where DE:1 means the edge is a dead end, DE:0 means the opposite. The same holds for H:1 and H:0 for hypothetical and non-hypothetical edges respectively.

We wanted to create realistic conditions, so we ran OpenCCG with a next best time limit of 2 seconds the 3-best pruning strategy, i.e. a set of edges sharing the same syntactic category is built and everything worse than the three best edges, in terms of their n-gram score[6], is discarded.

For most of the tests we used the SPaRKy (Sentence Planning with Rhetorical Knowledge) [13] restaurant corpus as lexicon. OpenCCG comes with several lexica but in comparison to SPaRKy, these are small and allow less extensive test runs, which rendered the

[6]An n-gram is a sequence of n words. The n-gram score is the probability of occurrence relative to a given corpus, e.g. a set of text passages which we use to extract these probabilities.

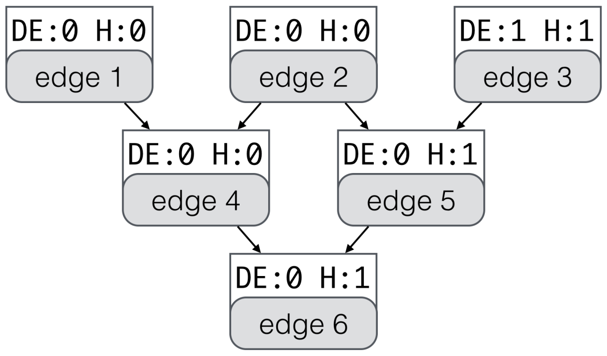

Figure 4.1: Propagation of hypothetical edges

results rather uninteresting. For this reason we will only present one example run in the *world cup* domain. This is a grammar where the lexicon consists mostly of words that are used in the context of a football world cup.

Using SPaRKy we realized 15 different logical formulas in several settings: Firstly, we realized these with k set to 3 and 5. For each of these, we conducted one run for counting the number of dead ends and one for the hypothetical edges.

Moreover we used a less restrictive version of the heuristic, in which we classify edges with more than k slashes as potentially useful instead of dead ends. To answer the question whether it is better to use the heuristic during the whole search or only on the initial edges, we ran these tests for both settings. Lastly, we ran OpenCCG without the heuristic to obtain data for comparisons.

4.1 SPaRKy

The SPaRKy grammar allows to realize sentences as well as paragraphs. For this it introduces punctuation having its own syntactic category and semantics. As an example consider a period (.) which is placed in-between two sentences. Hence its syntactic category is $s/s\backslash s$, so the compound of both sentences is treated as sentence itself allowing further combinations with potentially following sentences.

When realizing the logical formulas, the first thing we noticed is that it makes no difference when using the heuristic on all edges instead of only on the initial ones; we were not able to find additional dead ends. However, the only difference is the significantly higher run time caused by the additional calls to h^{max} . We do not rule out the possibility of finding additional dead ends during the search because of the following example:

Example 4.1.1. We want to realize the sentence "The Polish, successful, well-dressed athlete won the race". Consider we only have one edge per word, hence there is no useless initial edge. Nonetheless, an edge representing "The Polish athlete won the race" is a dead end because we are not able to add any further adjectives describing the athlete and hence we cannot convey the complete semantics. However, in the tested domain we were able to add additional adjectives simply be adding another sentence, so we could obtain "The Polish athlete won the race. The athlete was successful and well-dressed". As a result, we are able to find a solution, hence dead ends during the search occur rarely.

When measuring the number of edges, due to the time limit in use, the heuristic's run time reduces the number of edges created. So as a reference we use the number of edges created when running the heuristic but ignoring the result.

The results of our tests are gathered in Table 4.1. The values are averages over all runs, where one value is annotated as $a\langle b/c/d\rangle$ with a being the average and b, c and d being the 25th, 50th and 75th quantile respectively. All time values are in Milliseconds.

The first row shows the results when running OpenCCG without using the heuristic for dead end detection. In the second row, we set k to 3, in the third row k is 5. Out of the 15 sample LFs, we ran out of memory in two cases when setting k to 5, because these two cases were significantly larger than the others. For this reason we excluded them from the results and present them in more detail afterwards.

The first column states the number of initial edges. The second column contains the absolute number of dead ends found beneath the initial edges and their relative occurrence ratio compared to the number of initial edges. Next is the n-gram score for the resulting utterance, followed by the number of created edges in the edge-column. The number of hypothetical edges, i.e. how many edges we can prune, annotated with the relative ratio compared to the overall number of edges is listed in the next column. In the last column we report the time needed for realization.

Note that we omitted the percentiles for the n-gram score, because for $k = 3$ as well as $k = 5$, all percentiles were 0.

In Table 4.1 we can see that the run time needed for the realization was significantly higher when using the heuristic even though we created less edges. The reason for this is that the adaptation step as well as the computation of h^{max} take way too long to be of practical relevance. In the 13 sample sentences, the adaptation took 2063 $\langle 709/1701/2169\rangle$ ms and on average each call to h^{max} took 153.6 $\langle 92/131/157\rangle$ ms [7]. Taking into account that the overall realization without heuristic takes 990 ms, this overhead is completely impractical. The adaptation creates a planning instance with 433339 $\langle 270862/469706/552035\rangle$ actions and 19238 $\langle 14148/21624/23088\rangle$ facts, where most of the actions are completely useless. In the next chapter we present an example and elaborate in a little more detail how we can improve this. However, we can see that the n-gram score decreases when using the heuristic, especially for $k = 3$. The reason for this is that we disallow several words that would lead to an increased score, so the realization has to fall back on less natural and hence less frequent formulations resulting in a decreased score. This thesis is supported by the increased score for $k = 5$ and the even higher score when being less restrictive as seen in Table 4.2.

With $k = 3$ we can prune about 40.6% of the state space whereas $k = 5$ prunes only 25.9% at a cost of an about 9 times higher run time and for the gain of better resulting sentences. These results, however, are still significantly worse than without the heuristic.

We can see that the less restrictive variant performs better in terms of score, but the number of dead ends among the initial edges drops by about 33%, so we have to explore more edges.

An astonishing result is that there is no difference between $k = 3$ and $k = 5$ anymore when using the less restrictive variant. So the detected dead ends with $k = 3$ are either dead ends regardless of the restriction of category length or we would need an even higher value for k for these edges to become useful.

[7]Note that we obtained these numbers by computing the average of the h^{max} run time per call for each sentence and computed the average and percentiles of the results.

k	initial	dead ends(%)	n-gram	edges	hypothetical(%)	realization time
–	75.5 ⟨69/75/79⟩	-	0.0040	953.4 ⟨395/430/528⟩	-	990.7 ⟨130/175/215⟩
3	75.5 ⟨69/75/79⟩	21.3(28.2%) ⟨21/23/23⟩	0	477.1 ⟨318/350/398⟩	193.6(40.6%)⟨103/116/158⟩	10028.5 ⟨5300/10522/12380⟩
5	75.5 ⟨69/75/79⟩	14.8(19.7%) ⟨14/15/17⟩	0.0006	932.8 ⟨357/430/482⟩	241.7(25.9%) ⟨125/137/221⟩	87326.5 ⟨30413/103877/126423⟩

Table 4.1: Results of the experiments.

k	dead ends(%)	dead ends(%)[lr]	n-gram	n-gram[lr]	edges	edges[lr]
–	-	-	0.0040	0.0040	953.4 ⟨395/430/528⟩	-
3	21.3(28.2%) ⟨21/23/23⟩	14.1(18.6%) ⟨12/15/16⟩	0	0.0011	477.1 ⟨318/350/398⟩	654.8 ⟨209/221/240⟩
5	14.8(19.7%) ⟨14/15/17⟩	14.1(18.6%) ⟨12/15/16⟩	0.0006	0.0011	932.8 ⟨357/430/482⟩	654.8 ⟨209/221/240⟩

Table 4.2: Results of the experiments. Key: [lr] - less restrictive

k	run	initial	de(%)	n-gram	edges	hypo(%)	real time
–	1	144	-	0.0021	27072	-	6182
–	2	141	-	0.0021	26771	-	4475
3	1	144	40(27.8%)	0	6381	4290(67.2%)	55637
3	2	141	39(27.7%)	0	9364	6712(71.7%)	58547

Table 4.3: Results of the experiments for larger instances. Key: de - Dead End, hypo - hypothetical edges

heuristic	edges	time	score
no	53	69	1.0
yes	40	241	1.0

Table 4.4: Results for “Brazil will not win the cup”. Key: de - dead ends

In Table 4.3 we can see the results for the larger instance where a run with $k = 5$ ran out of memory. For $k = 3$, however, we can see that about 70% of the edges became hypothetical. So we had significantly less edges to explore but in terms of run time the ratio of a run with and without the heuristic remains roughly at 9, similar to the smaller instances.

4.2 World Cup

In this domain we realized the sentence “Brazil will not win the cup”. In all tested sentences there were no significant differences in the result of all measured parameters, hence we leave this as the only sample sentence. Due to the small size of the problem, we chose k as 20, so that no cut-off will take place. Here, we used OpenCCG to parse the sentence and extract the semantics, which we then used as input for the realization.

The result is collected in Table 4.4. We can see that the results are similar to the ones from SPaRKy only in a smaller scale, except for the score which is 1 in both cases since we were able to perform an exhaustive search and hence did not miss the “perfect” result.

To put this in a nutshell, we have seen that there is potential for pruning dead ends using the heuristic. However, the score decreases significantly when using a small value for k but can be improved by using the less restrictive variant. In all cases, the computational overhead is by far greater than the run time gain by pruning parts of the search tree.

Chapter 5

Future Work

In the preceding chapter we have seen that the computational overhead is not worth the gain of the pruning. So we want to present ways to potentially improve this behavior by either improving the adaptation or moving some of the computations in an *off-line* phase, i.e. computing some results before starting the application in which we need them. This has the benefit that the on-line phase can use these results immediately instead of having to compute them first. So the run time is reduced.

5.1 Improving the Adaptation

h^{max} is considered an extremely fast heuristic but during the adaptation we have to generate a myriad of actions: for each category combination we need one action per semantic item and many of these actions are actually unnecessary because there are parts of the LF that will never be covered by an edge with this specific category. Consider for example a set of initial lexical entries where one of them is a period covering the first of the 100 EP and nothing else. The period's category is *punc*. Now we create 100 actions[8] for carrying over the semantics but only one of those actions will ever be of use: The action carrying over the information that the first EP is covered.

So finding a way to efficiently identify unnecessary actions and preventing the creation of those can improve the runtime of the adaptation as well as h^{max} significantly.

We already mentioned that the implementation of h^{max} is generic, so some adaptations might make it possible to remove the necessity of generating many very similar actions and instead provide "generic" actions on which base h^{max} can reconstruct their preconditions and effects.

Lastly, one could omit the last adaptation phase, i.e. the usage of a "main category", and instead remove the edge in question. If the resulting task would be a dead end, we can consider the edge in question a *landmark edge* [7], which is crucial for the task to be solvable.

5.2 Off-line Computations

A different approach is to move parts of the adaptation into an off-line precomputation phase. That means instead of compiling only the initial edges, we compile the whole lexi-

[8]in the presented adaptation, 300 actions are created of which 3 can be of use

con. This would lead to an even greater amount of actions. But these can then be filtered on-line according to the initial edges we have. Under the assumption that the filtering can be performed efficiently, this potentially reduces the run time significantly. Additionally, in an off-line adaptation, we can chose k greater than in an on-line adaptation, hence we have a more informed search. So there are two open questions here: Firstly, is there an efficient filtering possible, and secondly, is a precomputation for large lexica with sufficiently high values of k tractable. Even though the computation is off-line, the run time grows exponentially in k and the size of *broad-coverage lexica* can exceed 100,000 entries, so the tractability is not necessarily given.

5.3 Classifying Combinations

During the OpenCCG search, all edges from the agenda are attempted to be combined with all elements in the chart but most of these attempts fail due to overlapping semantics or syntactic incompatibility. Instead of removing edges from the agenda - resulting in less edges in the chart as well - one can classify the edges and combinations in a way that incompatible combinations are spotted earlier on, so there is no need to try a rule application on them. For this classification, approximate results can be used.

Chapter 6

Conclusion

To sum this thesis up, we have tried to improve the process of realizing a sentence by using planning techniques. For the realization process, OpenCCG creates a set of edges representing sequences of words, a category marking their syntactic role and the conveyed semantics. These edges are then combined to form new edges. After discovering all initial edges, we compiled this problem into a planning task on which we used h^{max} to filter out useless edges. In Chapter 4 we have seen that there are in fact dead ends that we can recognize, but the high computational overhead makes this process irrelevant for practice. However, using an unsafe abstraction allows us to shorten the search in terms of generated edges, so making the process more efficient has the potential to improve the search.

Furthermore, in practice, time limits are used because an exhaustive search is too time consuming. As a result, the pruning does not necessarily only make it possible to find a result faster, but also to improve the result's quality because we can explore a larger part of the original search tree by pruning less promising parts. However, we discussed that the question how to improve the process has no linear answer, it is necessary to consider different approaches and evaluate how well they work in terms of potential pruning and run time gain.

Bibliography

[1] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33, 2001.

[2] J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[3] Richard E. Fikes and Nils Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[4] Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173:503–535, 2009.

[5] J. Hockenmaier and M. Steedman. Acquiring compact lexicalized grammars from a cleaner treebank. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC-2002)*, Las Palmas, Canary Islands - Spain, May 2002. European Language Resources Association (ELRA). ACL Anthology Identifier: L02-1263.

[6] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[7] Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

[8] M Kay. *Readings in Natural Language Processing.* Morgan Kaufmann Publishers Inc., 1986.

[9] Mike Lewis and Mark Steedman. A* ccg parsing with a supertag-factored model. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 990–1000, Doha, Qatar, October 2014. Association for Computational Linguistics.

[10] R. Pareschi M. Steedman. A lazy way to chart-parse with categorial grammars, stanford ca. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, pages 81–88, 1987.

[11] Bernhard Nebel. On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research*, 12:271–315, 2000.

[12] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (Third Edition).* Prentice-Hall, Englewood Cliffs, NJ, 2010.

[13] Marilyn Walker, Amanda Stent, François Mairesse, and Rashmi Prasad. Individual and domain adaptation in sentence planning for dialogue. *J. Artif. Int. Res.*, 30(1):413–456, November 2007.

[14] Joseph Weizenbaum. Eliza&mdash;a computer program for the study of natural language communication between man and machine. *Commun. ACM*, 9(1):36–45, January 1966.

[15] M. White. Efficient realization of coordinate structures in combinatory categorial grammar. *Research on Language and Computation*, 4(1):39–75, 2006.