Parsing Combinatory Categorial Grammar via Planning in Answer Set Programming

Yuliya Lierler[1] and Peter Schüller[2]

[1] Department of Computer Science, University of Kentucky
yulia@cs.uky.edu
[2] Institut für Informationssysteme, Technische Universität Wien
ps@kr.tuwien.ac.at

Abstract. Combinatory categorial grammar (CCG) is a grammar formalism used for natural language parsing. CCG assigns structured lexical categories to words and uses combinatory rules to combine these categories to parse a sentence. In this work we propose and implement a new approach to CCG parsing that relies on a prominent knowledge representation formalism, answer set programming (ASP) — a declarative programming paradigm. We formulate the task of CCG parsing as a planning problem and use an ASP computational tool to compute solutions that correspond to valid parses. Compared to other approaches, there is no need to implement a specific parsing algorithm using such a declarative method. Our approach aims at producing all semantically distinct parse trees for a given sentence. From this goal, normalization and efficiency issues arise, and we deal with them by combining and extending existing strategies. We have implemented a CCG parsing tool kit — ASPCCGTK— that uses ASP as its main computational means. The C&C supertagger can be used as a preprocessor within ASPCCGTK, which allows us to achieve wide-coverage natural language parsing.

1 Introduction

The task of parsing, i.e., recovering the internal structure of sentences, is an important task in natural language processing. Combinatory categorial grammar (CCG) is a popular grammar formalism used for this task. It assigns basic and complex lexical categories to words in a sentence and uses a set of combinatory rules to combine these categories to parse the sentence. In this work we propose and implement a new approach to CCG parsing that relies on a prominent knowledge representation formalism, answer set programming (ASP) — a declarative programming paradigm. Our aim is to create a wide-coverage[3] parser which returns all semantically distinct parse trees for a given sentence.

One major challenge of natural language processing is ambiguity of natural language. For instance, many sentences have more than one plausible internal structure, which often provide different semantics to the same sentence. Consider a sentence

John saw the astronomer with the telescope.

It can denote that John used a telescope to see the astronomer, or that John saw an astronomer who had a telescope. It is not obvious which meaning is the correct one without additional context. Natural language ambiguity inspires our goal to return *all semantically distinct* parse trees for a given sentence.

CCG-based systems OPENCCG [29] and TCCG [1, 3] (implemented in the LKB toolkit) can provide multiple parse trees for a given sentence. Both use chart parsing algorithms with CCG extensions such as modalities or hierarchies of categories. While OPENCCG is primarily geared towards generating sentences from logical forms, TCCG targets parsing. However, both implementations require lexicons[4] with specialized categories. Generally, crafting a CCG lexicon is a time–consuming task. An alternative method to using a hand-crafted lexicon has been implemented in a wide-coverage CCG parser — C&C [6, 7]. C&C relies on machine learning techniques for tagging an input sentence with CCG categories as well as for creating parse trees with a chart algorithm. As training data, C&C uses CCGbank— a corpus of

[3] The goal of wide-coverage parsing is to parse sentences that are not within a controlled fragment of natural language, e.g., sentences from newspaper articles.

[4] A CCG lexicon is a mapping from each word that can occur in the input to one or more CCG categories.

CCG derivations and dependency structures [20] based on the translation of the Penn Treebank[5] using CCG. It pairs syntactic derivations with sets of word-word dependencies which approximate the underlying predicate-argument structure. The parsing algorithm of C&C returns a *single* most probable parse tree for a given sentence[6].

In the approach that we describe in this paper we formulate the task of CCG parsing as a planning problem. Then we solve it using answer set programming [23, 25]. ASP is a declarative programming formalism based on the answer set semantics of logic programs [18]. The idea of ASP is to represent a given computational problem by a program whose answer sets correspond to solutions, and then use an answer set solver to generate answer sets for this program. Utilizing ASP for CCG parsing allows us to control the parsing process with declarative descriptions of constraints on combinatory rule applications and parse trees. Moreover, there is no need to implement a specific parsing algorithm, as an answer set solver is used as a computational vehicle of the method. In our ASP approach to CCG parsing we formulate a problem in such a way that multiple parse trees are computed.

An important issue inherent to CCG parsing are spurious parse trees: a given sentence may have many distinct parse trees which yield the same semantics. Various methods for eliminating such spurious parse trees have been proposed [6, 12, 30]. We adopt some of these syntactic methods in this work.

We implemented our approach in an ASPCCGTK toolkit. The toolkit equips a user with two possibilities for assigning plausible categories to words in a sentence: it can either use a given (hand-crafted) CCG lexicon or it can take advantage of the C&C supertagger [7] for this task. The second possibility provides us with wide-coverage CCG parsing capabilities. The ASPCCGTK toolkit computes best-effort parses in cases where no full parse can be achieved with CCG, resulting in parse trees for as many phrases of a sentence as possible. This behavior is more robust than completely failing in producing a parse tree. It is also useful for development, debugging, and experimenting with rule sets and normalizations. In addition to producing parse trees, ASPCCGTK contains a module for visualizing CCG derivations. The following table compares ASPCCGTK to CCG parsers discussed earlier.

Properties	OPENCCG	TCCG	C&C	ASPCCGTK
multiple parses	✓	✓		✓
wide-coverage			✓	✓

A number of theoretical characterizations of CCG parsing exists. They differ in their use of specialized categories, their sets of combinatory rules, or specific conditions on applicability of rules. We see an ASP approach to CCG parsing implemented in ASPCCGTK as a basis of a generic tool for encoding different CCG category and rule sets in a declarative and straightforward manner. Such a tool provides a test-bed for experimenting with different theoretical CCG frameworks without the need to craft specific parsing algorithms.

The structure of this paper is as follows: we start by reviewing planning, ASP, and CCG. We describe our new approach to CCG parsing by formulating this task as a planning problem in Section 3. The implementation and framework for realizing this approach using ASP technology is the topic of Section 4. We conclude with a discussion of future work directions and challenges.

2 Preliminaries

2.1 Planning

Automated planning [5] is a widely studied area in Artificial Intelligence. In *planning*, given knowledge about

(a) available actions, their executability, and effects,
(b) an initial state, and
(c) a goal state,

[5] `http://www.cis.upenn.edu/~treebank/`.

[6] Parser C&C defines "most probable" based on categories co-occurrence statistics derived from CCGbank corpus.

the task is to find a sequence of actions that leads from the initial state to the goal state. A number of special purpose planners have been developed in this sub-area of Artificial Intelligence. Answer set programming provides a viable alternative to special-purpose planning tools [13, 22, 25].

2.2 Answer Set Programming (for Planning)

Answer set programming (ASP) [23, 25] is a declarative programming formalism based on the answer set semantics of logic programs [18, 19]. The idea of ASP is to represent a given computational problem by a program whose answer sets correspond to solutions, and then use an answer set solver to generate answer sets for this program. In this work we use the CLASP[7] system with its front-end (grounder) GRINGO [16], which is currently one of the most widely used answer set solvers.

A common methodology to solve a problem in ASP is to design GENERATE, DEFINE, and TEST [22] parts of a program. The GENERATE part defines a large collection of answer sets that could be seen as potential solutions. The TEST part consists of rules that eliminate the answer sets of the GENERATE part that do not correspond to solutions. The DEFINE section expresses additional concepts and connects the GENERATE and TEST parts.

A typical logic programming rule has the form of a Prolog rule. For instance, program

$$p.$$
$$q \leftarrow p,\ not\ r.$$

is composed of such rules. This program has one answer set $\{p, q\}$. In addition to Prolog rules, GRINGO also accepts rules of other kinds — "choice rules" and "constraints". For example, rule

$$\{p, q, r\}.$$

is a choice rule. Answer sets of this one-rule program are arbitrary subsets of the atoms $p,\ q,\ r$. Choice rules are typically the main members of the GENERATE part of the program. Constraints often form the TEST section of a program. Syntactically, a constraint is the rule with an empty head. It encodes the conditions on the answer sets that have to be met. For instance, the constraint

$$\leftarrow p,\ not\ q.$$

eliminates the answer sets of a program that include p and do not include q.

System GRINGO allows the user to specify large programs in a compact way, using rules with schematic variables and other abbreviations. A detailed description of its input language can be found in the online manual [16]. Grounder GRINGO takes a program "with abbreviations" as an input and produces its propositional counterpart that is then processed by CLASP. Unlike Prolog systems, the inference mechanism of CLASP is related to that of Propositional Satisfiability (SAT) solvers [17].

The GENERATE-DEFINE-TEST methodology is suitable for modeling planning problems. To illustrate how ASP programs can be used to solve such problems, we present a simplified part of the encoding of a classic toy planning domain *blocks world* given in [22]. In this domain, blocks are moved by a robot. There are a number of restrictions including the fact that a block cannot be moved unless it is clear.

Lifschitz [22] models the blocks world domain by means of five predicates: *time/1*, *block/1*, *location/1*, *move/3*, *on/3*; a location is a *block* or the *table*. The constant *maxsteps* is an upper bound on the length of a plan. States of the domain are modeled by the ground atoms of the form *on(b,l,t)* stating that block b is at location l at time t. Actions are modeled by ground atoms *move(b,l,t)* stating that block b is moved to location l at time t.

The GENERATE section of a program consists of a single rule

$$\{move(B, L, T)\} \leftarrow block(B),\ location(L),\ time(T),\ T < maxsteps.$$

that defines a potential solution to be an arbitrary set of *move* actions executed before *maxsteps*.

[7] `http://potassco.sourceforge.net/`.

The fact that moving a block to a position at time T forces a block to be at this position at time $T{+}1$ is encoded in DEFINE part of the program by the rule

$$on(B, L, T{+}1) \leftarrow move(B, L, T),\ block(B),\ location(L),\ time(T),\ T{<}maxsteps.$$

The rule below specifies the commonsense law of inertia for a predicate *on* stating that unless we know that the block is no longer at the same position it remains where it was:

$$on(B, L, T{+}1) \leftarrow on(B, L, T),\ not\ \neg on(B, L, T{+}1),\ block(B),\ location(L),\\ time(T),\ T < maxsteps.$$

The following constraint in TEST encodes the restriction that a block cannot be moved unless it is clear

$$\leftarrow move(B, L, T),\ on(B1, B, T),\ block(B),\ block(B1),\\ location(L),\ time(T),\ T < maxsteps.$$

Given the rest of the encoding and the description of an initial state and of the goal state, answer sets of the resulting program represent plans. The ground atoms of the form *move(b,l,t)* present in an answer set form the list of actions of a corresponding plan.

2.3 Combinatory Categorial Grammar

Combinatory Categorial Grammar (CCG) [27] is a linguistic grammar formalism. CCG uses a small set of combinatory rules – combinators – to combine rich lexical categories of words.

Categories in CCG are either atomic or complex. For instance, noun N, noun phrase NP, and sentence S are atomic categories. Complex categories are functors that specify the type and direction of the arguments and the type of the result. A complex category

$$S\backslash NP$$

is a category for English intransitive verbs (such as *walk*, *hug*), which states that a noun phrase is required to the left, resulting in a sentence. A category

$$(S\backslash NP)/NP$$

for English transitive verbs (such as *like* and *bite*) specifies that a noun phrase is required to the right and yields the category of an English intransitive verb, which (as before) requires a noun phrase to the left to form a sentence.

Given a sentence and a lexicon containing a set of word-category pairs, we can replace words in the sentence by appropriate categories. For example, for a sentence

$$\textit{The dog bit John} \tag{1}$$

and a lexicon containing pairs

$$The\ \text{-}\ NP/N;\ dog\ \text{-}\ N;\ bit\ \text{-}\ (S\backslash NP)/NP;\ John\ \text{-}\ NP \tag{2}$$

we obtain

$$\frac{The}{NP/N} \qquad \frac{dog}{N} \qquad \frac{bit}{(S\backslash NP)/NP} \qquad \frac{John}{NP}.$$

Words may have multiple categories, e.g., "bit" is also an intransitive verb and a noun. To simplify the presentation of parsing in this paper we limit out attention to the case when there is a unique category corresponding to each word. Nevertheless, our framework is able to handle multiple categories by considering all combinations of word categories.

To parse English sentences a number of combinators are required [27]: forward and backward application ($>$ and $<$, respectively), forward and backward *composition* ($>\mathbf{B}$ and $<\mathbf{B}$), forward and backward

type raising ($>\mathbf{T}$ and $<\mathbf{T}$), backward *cross composition*, backward *cross substitution*, and *coordination*. Specifications of some of these combinators follow:

$$\frac{A/B \quad B}{A} > \qquad \frac{A/B \quad B/C}{A/C} >\mathbf{B} \qquad \frac{A}{B/(B\backslash A)} >\mathbf{T}$$
$$\frac{B \quad A\backslash B}{A} < \qquad \frac{B\backslash C \quad A\backslash B}{A\backslash C} <\mathbf{B} \qquad \frac{A}{B\backslash(B/A)} <\mathbf{T}$$

where A, B, C are variables that can be substituted by CCG categories such as N or $S\backslash NP$. An *instance* of a CCG combinator is obtained by substituting CCG categories for variables. For example,

$$\frac{NP/N \quad N}{NP} > \tag{3}$$

is an instance of the forward application combinator ($>$). A CCG combinatory rule combines one or more adjacent categories and yields exactly one output category. To parse a sentence is to apply instances of CCG combinators so that the final category S is derived at the end. A sample CCG derivation for sentence (1) follows

$$\begin{array}{lll} & \overline{\textit{The}} \quad \overline{\textit{dog}} \quad \overline{\textit{bit}} \quad \overline{\textit{John}} & (4) \\ \text{initial state (time 0)} & \overline{NP/N} \quad \overline{N} \quad \overline{(S\backslash NP)/NP} \quad \overline{NP} & \\ & \xrightarrow{\qquad} \quad \xrightarrow{\qquad} & \text{two } > \text{ actions at time 0} \\ \text{state (time 1)} & NP \qquad S\backslash NP & \\ & \xrightarrow{\qquad\qquad\qquad} & \text{one } < \text{ action at time 1} \\ \text{goal state (time 2)} & S & \end{array}$$

On the left and right side of the derivation we give an intuition about how we translate the CCG parsing task into action planning. Section 3.1 gives a formal definition of this translation.

Type Raising and Spurious Parses: CCG restricted to application combinators generates the same language as CCG restricted to application, composition, and type raising rules [10,26]. One of the motivations for type raising are non-constituent coordination constructions[8] that can only be parsed with the use of raising [2, Example (2)] and the additional coordination rule shown below (coordinating words such as "and" receive category $CONJ$).

$$\frac{A \;\; CONJ \;\; A}{A} \;\mathbf{\Phi}$$

Unrestricted applications of composition and type raising combinators often create spurious parse trees which are semantically equivalent to parse trees derived using application rules only. Eisner [12, Example (3)] presents a sample sentence with 12 words and 252 parses but only 2 distinct meanings. An example of a spurious parse for sentence (1) is the following derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\textit{The}} \quad \overline{\textit{dog}}}{\overline{NP/N} \quad \overline{N}}>}{\overline{S/(S\backslash NP)}}>\mathbf{T} \quad \dfrac{\overline{\textit{bit}}}{\overline{(S\backslash NP)/NP}}}{\overline{S/NP}}>\mathbf{B} \quad \dfrac{\overline{\textit{John}}}{\overline{NP}}}{S}> \tag{5}$$

which utilizes application, type raising, and composition combinators. Both derivations (4) and (5) have the same semantic value (in a sense, the difference between (4) and (5) is not essential for subsequent semantic analysis).

[8] E.g, in the sentence "We gave Jan a record and Jo a book", neither "Jan a record" nor "Jo a book" is a linguistic constituent of the sentence. With raising we can produce meaningful categories for these non-constituents and subsequently coordinate them using "and".

In this work we aim at the generation of parse trees that have different semantic values so that they reflect a real ambiguity of natural language, and not a spurious ambiguity that arises from the underlying CCG formalism. Various methods for dealing with spurious parses have been proposed such as limiting type raising only to certain categories [6], normalizing branching direction of consecutive composition rules by means of predictive combinators [30] or restrictions on parse tree shape [12]. We combine and extend these ideas to pose restrictions on generated parse trees within our framework. Details about normalizations and type raising limits that we implement are discussed in Section 3.3.

3 CCG Parsing via Planning

3.1 Problem Statement

We start by defining precisely the task of *CCG parsing*. We then state how this task can be seen as a planning problem.

A *sentence* is a sequence of words. An *abstract sentence representation* (ASR) is a sequence of categories annotated by a unique id. Recall that given a lexicon, we can replace words in the sentence by appropriate categories. As a result we can turn any sentence into ASR using a lexicon. For instance, for sentence (1) and lexicon (2) a sequence

$$[NP/N^1,\ \ N^2,\ \ (S\backslash NP)/NP^3,\ \ NP^4]. \tag{6}$$

is an ASR of (1). We refer to categories annotated by id's as *annotated categories*. Members of (6) are annotated categories.

Recall that an instance of a CCG combinator C has a general form

$$\frac{X_1, \ldots, X_n}{Y}\ C.$$

We say that the sequence $[X_1, \ldots, X_n]$ is a *precondition* sequence of C, whereas Y is an *effect* of applying C. The precondition sequence and the effect of instance (3) of the combinator $>$ are $[NP/N, N]$ and NP, respectively.

Given an instance C of a CCG combinator we may annotate it by

- assigning a distinct id to each member of its precondition sequence, and
- assigning the id of the left most annotated category in the precondition sequence to its effect.

We call such an instance an *annotated (combinator) instance*. For example,

$$\frac{NP/N^1 \quad N^2}{NP^1}\ > \tag{7}$$

is an annotated instance w.r.t. (3).

An annotated instance C is applied to an ASR sequence A by replacing the substring of A corresponding to the precondition sequence of C by its effect. For example, applying (7) to (6) yields ASR $[NP^1, (S\backslash NP)/NP^3, NP^4]$. In the following we will often say annotated combinator in place of annotated instance.

To view CCG parsing as a planning problem we need to specify states and actions of this domain. In CCG planning, states are ASRs and actions are annotated combinators. So the task is given the initial ASR, e.g., $[X_1^1, \ldots, X_n^n]$, to find a sequence of annotated combinators that leads to the goal ASR — $[S^1]$.

Let $\mathcal{C}_1$ denote annotated combinator (7), $\mathcal{C}_2$ denote

$$\frac{(S\backslash NP)/NP^3 \quad NP^4}{S\backslash NP^3}\ >,$$

and $\mathcal{C}_3$ denote

$$\frac{NP^1 \quad S\backslash NP^3}{S^1}\ >.$$

Given ASR (6) a sequence of actions $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ forms a plan:

$$\begin{array}{ll} \text{Time 0:} & [NP/N^1,\ \ N^2,\ \ (S\backslash NP)/NP^3,\ \ NP^4] \\ \quad \text{action: } \mathcal{C}_1 & \\ \text{Time 1:} & [NP^1,\ \ (S\backslash NP)/NP^3,\ \ NP^4], \\ \quad \text{action: } \mathcal{C}_2 & \\ \text{Time 2:} & [NP^1,\ \ S\backslash NP^3], \\ \quad \text{action: } \mathcal{C}_3 & \\ \text{Time 3:} & [S^1]. \end{array} \tag{8}$$

This plan corresponds to parse tree (4) for sentence (1). On the other hand, a plan formed by a sequence of actions $\mathcal{C}_2$, $\mathcal{C}_1$, and $\mathcal{C}_3$ also corresponds to (4).

In planning the notion of *serializability* is important. Often given a plan, applying several consecutive actions in the plan in any order or in parallel does not change the effect of their application. Such plans are called *serializable*. Consequently, by allowing parallel execution of actions one may represent a class of plans by a single one. This is a well-known optimization in planning. For example, plan

$$\begin{array}{ll} \text{Time 0:} & [NP/N^1,\ \ N^2,\ \ (S\backslash NP)/NP^3,\ \ NP^4] \\ \quad \text{actions: } \mathcal{C}_1, \mathcal{C}_2 & \\ \text{Time 1:} & [NP^1,\ \ S\backslash NP^3], \\ \quad \text{action: } \mathcal{C}_3 & \\ \text{Time 2:} & [S^1] \end{array}$$

may be seen as an abbreviation for a group of plans, i.e., itself, plan (8), and a plan formed by a sequence $\mathcal{C}_2$, $\mathcal{C}_1$, and $\mathcal{C}_3$. In CCG parsing as a planning problem, we are interested in finding plans of this kind, i.e., plans with concurrent actions.

Next we present the ASP encoding of the planning problem. In order to enforce normalizations that limit spurious parses, in the encoding the planning problem presented so far is extended further to eliminate some redundant plans.

3.2 ASP Encoding

In an ASP approach to CCG parsing, the goal is to encode the planning problem as a logic program so that its answer sets correspond to plans. As a result answer sets of this program will contain the sequence of annotated combinators (actions, possibly concurrent) such that the application of this sequence leads from a given ASR to the ASR composed of a single category S. We present a part of the encoding `ccg.asp`[9] in the GRINGO language that solves a CCG parsing problem by means of ideas presented in Section 2.2.

First, we need to decide how we represent states — ASRs — by sets of ground atoms. To this end, we introduce symbols called "positions" that encode annotations of ASR members. In `ccg.asp`, relation $posCat(p, c, t)$ states that a category c is annotated with (position) p at time t. Relation

$$posAdjacent(p_L, p_R, t)$$

states that a position p_L is adjacent to a position p_R at time t. In other words, a category annotated by p_L immediately precedes a category annotated by p_R in an ASR that corresponds to a state at time t (intuitively, L and R denote left and right, respectively.) These relations allow us to encode states of a CCG planning domain. For example, given an ASR (6) as the initial state, we can encode this state by the following set of facts

$$\begin{array}{l} posCat(1, rfunc(\text{“}NP\text{”}, \text{“}N\text{”}), 0).\ \ posCat(2, \text{“}N\text{”}, 0). \\ posCat(3, rfunc(lfunc(\text{“}S\text{”}, \text{“}NP\text{”}), \text{“}NP\text{”}), 0).\ \ posCat(4, \text{“}NP\text{”}, 0). \\ posAdjacent(1, 2, 0).\ \ posAdjacent(2, 3, 0).\ \ posAdjacent(3, 4, 0). \end{array} \tag{9}$$

[9] The complete listing of `ccg.asp` is available at
`http://www.kr.tuwien.ac.at/staff/ps/aspccgtk/ccg.asp`

Next we need to choose how we encode actions. The combinators mentioned in Section 2.3 are of two kinds: the ones whose precondition sequence consists of a single element (i.e., $>\mathbf{T}$ and $<\mathbf{T}$) and of two elements (e.g., $>$ and $<$). Coordination combinator is of a third type, i.e., its precondition sequence contains three elements. We simplify the presentation of the encoding by omitting the details of this case. We call the combinators from Section 2.3 *unary* and *binary* respectively. Reification of actions is a technique used in planning that allows us to talk about common properties of actions in a compact way. To utilize this idea, we first introduce relations $unary(a)$ and $binary(a)$ for every unary and binary combinator a respectively. For a unary combinator a, a relation $occurs(a, p, c, t)$ states that a type raising action a occurring at time t raises a category identified with position p (at time t) to category c. For a binary combinator a a relation $occurs(a, p, t)$ states that an action a applied to positions p (and the position adjacent to p to the right) occurs at time t. For instance, given the initial state (9)

- $occurs(ruleFwdTypeR, 4, (S\backslash NP)/NP, 0)$ represents an application of the annotated combinator

$$\frac{NP^4}{(S\backslash NP)/NP^4}\ >\mathbf{T}$$

  to (9) at time 0,
- $occurs(ruleFwdAppl, 1, 0)$ represents an application of (7) to (9) at time 0.

Given an atom $occurs(a, p, c, t)$ or $occurs(a, p, t)$ we often say that action a *modifies* position p at time t.

Recall that solutions of this formalization correspond to parse trees so that each application of a combinator forms an edge (or a set of edges) in a tree. We introduce an auxiliary action named

$$placeEdgeTag(p, t, tag),$$

which states that there is an edge placed at position p at time t tagged by a tag $unary$ or $binary$. Intuitively occurrence of an atom $placeEdgeTag(p, t, unary)$ (or $placeEdgeTag(p, t, binary)$) in a solution guarantees that an atom of the form $occurs(a, p, c, t)$ (or $occurs(a, p, t)$) is also present in the solution. In other words, some unary (or binary) action *modifies* a position p at time t. Introducing this auxiliary relation allows us to state some constraints on solutions by referring only to the type of combinators (i.e., $unary$ or $binary$) rather than their kind (i.e., $ruleFwdTypeR$ or $ruleFwdAppl$).

The GENERATE section of `ccg.asp` contains a choice rule

$$0\{placeEdgeTag(P, T, TAG) : type(TAG)\}1 \leftarrow posASR(P, T), time(T),\ T < maxsteps.$$

where $posASR$ is an auxiliary relation specifying that a position p is part of an ASR encoded by a state at time t. This rule states that for "ASR" positions it is possible to place a single edge either of type unary or binary. Another sample GENERATE rule

$$1\{occurs(A, P, T) : binary(A)\}1 \leftarrow placeEdgeTag(P, T, binary).$$

specifies that if a binary edge is placed at position P then one of the binary actions must occur at this time modifying P. Such choice rules describe a potential solution to the planning problem as an arbitrary set of actions executed before *maxsteps*.

In order to state effects of actions and executability conditions the DEFINE part of a program introduces an auxiliary relation $precCat$ for each action corresponding to a CCG combinator. For example,

$$precCat(ruleFwdAppl, L, T, A) \leftarrow posAdjacent(L, R, T),\ time(T),\\ posCat(L, rfunc(A, B), T),\ posCat(R, B, T).$$

states that if there are two adjacent positions such that the category of the left and right positions are A/B and B respectively then preconditions of binary action $ruleFwdAppl$ are satisfied and the resulting category of applying $ruleFwdAppl$ to this position is A.

A rule that models effects of actions in the CCG parsing domain using the $precCat$ relation follows

$$posCat(P, C, T{+}1) \leftarrow precCat(A, P, T, C),\ occurs(A, P, T),\ time(T).$$

It states that an application of a combinator A at time T causes a category annotated by P to be C at the next time point. Note that this rule takes advantage of reification and provides means for compact encoding of common effects of all binary actions. On the other hand, following rules

$$prec(A, L, T) \leftarrow precCat(A, L, T, C).$$
$$\leftarrow occurs(A, P, T),\ not\ prec(A, P, T).$$

formulate executability conditions by forbidding a combinator A to occur modifying position P at time T unless its preconditions are satisfied.

The following rule characterizes another effect of combinators and defines the $posAffected$ concept which is useful in stating several normalization conditions described in Section 3.3:

$$posAffected(P, T{+}1) \leftarrow 1\{placeEdgeTag(P, T, TAG) : type(TAG)\}\ time(T),\ T < maxsteps.$$

Relation $posAffected(P, T{+}1)$ holds if the element annotated by P in the ASR was modified by a combinator at time T. Note that this rule takes advantage of auxiliary relation $placeEdgeTag$ that provides means for compact encoding of common effects of all unary, binary (and ternary) actions. Furthermore, $posAffected$ is used to state the law of inertia for the predicate $posCat$

$$posCat(P, C, T{+}1) \leftarrow posCat(P, C, T),\ not\ posAffected(P, T{+}1),\ time(T),\ T < maxsteps.$$

stating that a category of a position stays the same unless it is affected.

In the TEST section of the program we encode such restrictions as no two combinators may modify the same position simultaneously and the fact that the goal has to be reached. We allow two possibilities for specifying a goal. In one case, the goal is to reach an ASR composed of a single category S by $maxsteps$. In another case, the goal is to reach the shortest possible ASR sequence by $maxsteps$.

The TEST section also includes a set of constraints modeling conditions when it is impossible for an action a to modify position p at time t. These rules form the main mechanism by which normalization techniques are encoded in `ccg.asp`. For instance, a rule

$$\leftarrow occurs(ruleFwdAppl, P, T),\ occurs(ruleFwdRaise, P, X, TLast{-}1),\ posLastAffected(P, TLast, T),\ time(TLast), time(T),\ T < maxsteps.$$

states that a forward application modifying position P may not *occur* at time T if the last action modifying P was forward type raising ($posLastAffected$ is an auxiliary predicate that helps to identify the last action modifying an element of the ASR). This corresponds to one of the normalization rules discussed in [12] and reviewed in the following subsection.

We pose additional restrictions, which ensure that only a single plan is produced when multiple serializable plans correspond to the same parse tree.

Finally, we devised punctuation specific combinators which have been described in [8, Appendix A] and are based on Sections 02-21 of CCGbank.

Given `ccg.asp` and the set of facts describing the initial state (ASR representation of a sentence) and the goal state (ASR containing a single category S), answer sets of the resulting program encode plans corresponding to parse trees. The ground atoms of the form $occurs(a, p, t)$ and $occurs(a, p, c, t)$ present in an answer set form the list of actions of a matching plan.

3.3 Normalizations

Currently, `ccg.asp` implements a number of normalization techniques and strategies for improving efficiency and eliminating spurious parses:

• One of the techniques used in C&C to improve its efficiency is to limit type raising to certain categories *based on the most commonly used type raising rule instantiations in Sections 2-21 of CCGbank* [6]. We adopt this idea by limiting type raising to be applicable only to noun phrases, NP, so that NP can be raised using categories S, $S\backslash NP$, or $(S\backslash NP)/NP$. This technique reduces the size of the ground program for `ccg.asp` and subsequently the performance of `ccg.asp` considerably. We plan to extend limiting type raising to the full set of categories used in C&C that proved to be suitable for wide-coverage parsing.

• We normalize branching direction of subsequent functional composition operations [12]. This is realized by disallowing functional forward composition to apply to a category on the left side that has been created by functional forward composition. (And similar for backward composition.)
• We disallow some combinations of rule applications if the same result can be achieved by other rule applications as shown in the following

$$\dfrac{\dfrac{X/Y \;\; Y/Z}{X/Z}{>}\mathbf{B} \;\; Z}{X}{>} \quad \Rightarrow_{\text{normalize}} \quad \dfrac{X/Y \;\; \dfrac{Y/Z \;\; Z}{Y}{>}}{X}{>} \qquad \dfrac{\dfrac{X}{Y/(Y\backslash X)}{>}\mathbf{T} \;\; Y\backslash X}{Y}{>} \quad \Rightarrow_{\text{normalize}} \quad \dfrac{X \;\; Y\backslash X}{Y}{<}$$

where the left-hand side is the spurious parse and the right-hand side the normalized parse. These two normalizations (plus analogous normalizations for backward composition and backward type raising) eliminate spurious parses like (5) and have been discussed in [3, 12].

4 ASPCCG Toolkit

We have implemented ASPCCGTK— a python[10] framework for using `ccg.asp`. The framework is available online[11], including documentation and examples.

Figure 1 shows a block diagram of ASPCCGTK. We use GRINGO and CLASP for ASP solving and control these solvers from python using a modified version of the BioASP library [14]. BioASP is used for calling ASP solvers as subtasks, parsing answer sets, and writing these answer sets to temporary files as facts.

Input for parsing can be

- a natural language sentence given as a string, or
- a sequence of words and a dictionary providing possible categories for each word, both given as ASP facts.

In the first case, the framework uses C&C supertagger[12] [7] to tokenize and tag this sentence. The result of supertagging is a sequence of words of the sentence, where each word is assigned a set of likely CCG categories. From the C&C supertagger output, ASPCCGTK creates a set of ASP facts representing the sequence of words and a corresponding set of likely CCG categories. This set of facts is passed to `ccg.asp` as the initial state. In the second case the input can be processed directly by `ccg.asp`. The maximum parse tree depth (i.e., the maximum plan length – *maxsteps*) currently has to be specified by the user. Auto detection of useful depth values is subject of future work.

ASPCCGTK first attempts to find a "strict" parse which requires that the resulting parse tree yields a category S (by $maxsteps$). If this is impossible, we do "best-effort" parsing using CLASP optimization features to minimize the number of categories left by the time $maxsteps$. For instance, consider a lexicon that provides a single category for "bit", namely $(S\backslash NP)/NP$, then the following derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\textit{The}}{NP/N} \;\; \dfrac{\textit{dog}}{N}}{NP}{>} \;\; \dfrac{\textit{bit}}{(S\backslash NP)/NP}}{S/(S\backslash NP)}{>}\mathbf{T}}{S/NP}{>}\mathbf{B} \tag{10}$$

corresponds to a best-effort parse.

Answer sets resulting from `ccg.asp` represent parse trees. ASPCCGTK passes them to a visualization component, which invokes GRINGO+CLASP on another ASP encoding `ccg2idpdraw.asp`.[13] The

[10] `http://www.python.org/`
[11] `http://www.kr.tuwien.ac.at/staff/ps/aspccgtk/`
[12] `http://svn.ask.it.usyd.edu.au/trac/candc`
[13] This visualization component could be put directly into `ccg.asp`. However, for performance reasons it has proved crucial to separate the parsing calculation from the drawing calculations.

resulting answer sets of `ccg2idpdraw.asp` contain drawing instructions for the IDPDraw tool [31], which is used to produce a two-dimensional image for each parse tree. Figure 2 demonstrates an image generated by IDPDraw for parse tree (4) of sentence (1). If multiple parse trees exist, IDPDraw allows to switch between them.

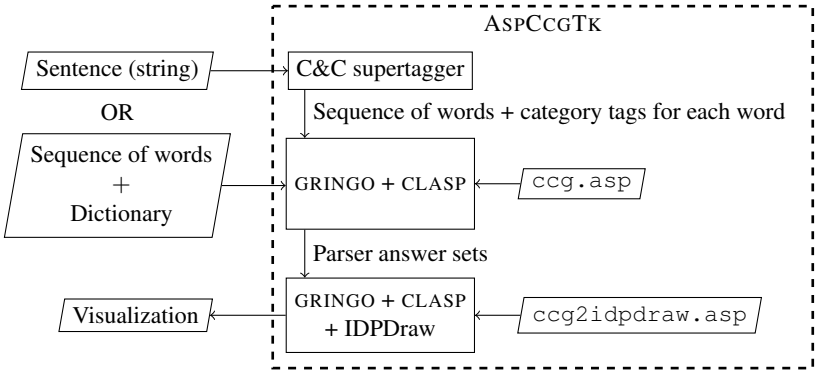

Fig. 1. Block diagram of the ASPCCG framework. (Arrows indicate data flow.)

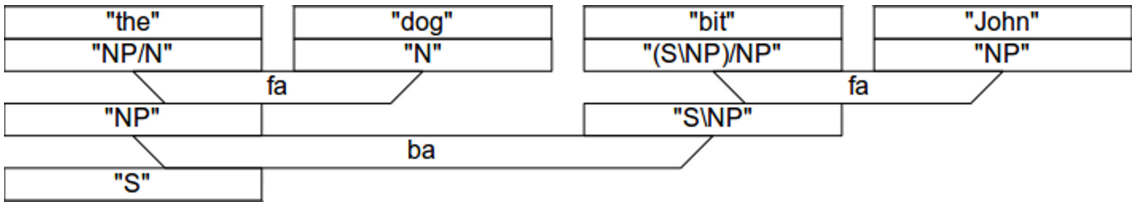

Fig. 2. Visualization of parse tree (4) for sentence (1) using IDPDraw.

5 Experimental Evaluation

We evaluated the efficiency of `ccg.asp` on Section 00 of CCGbank [20] (about 2000 sentences). CCGbank contains a gold-standard parse tree for each sentence where words are annotated by unique categories. We extracted these categories and used them as input for `ccg.asp`. This method for evaluating parsing performance is used in [8, 9] as well. Figure 3 presents the experimental results that summarize the efficiency of `ccg.asp`. The experiments were run on Xeon X5355 @ 2.66GHz. The runtimes presented account for solving time of CLASP v 2.0.2 on `ccg.asp`. The timeout was set to 1200 seconds. Furthermore, in the experiments we only considered the task of finding a single parse tree. We present the results by splitting the sentences from Section 00 of CCGbank into 8 groups depending on the number of words occurring in them (any punctuation symbol is also treated as a word). The second line in Figure 3 specifies how many sentences each of the groups contains. The third line presents average number of words in the sentences of the corresponding group. The fourth line accounts for average *maxsteps* parameter of a plan to be searched for. We used the height of gold-standard parse tree from CCGbank as *maxsteps* for a corresponding instance. The last four lines present average runtime and number of timeouts for two configurations of CLASP that we denote CLASP^d and CLASP^t. The configuration CLASP^d is the default call to the system, whereas CLASP^t stands for a commandline

```
clasp -q --heu=VSIDS --sat-pre=20,25,120 --trans-ext=dynamic
```

The portfolio toolkit BORG[14] was used to evaluate 25 different configurations of CLASP in order to select the best performing one, i.e., CLASP-t. We considered the same configurations for evaluations as the ones used in portfolio answer set solver CLASPFOLIO v 1.0.1 [15].

Groups: Number of Words		1-10	11-15	16-20	21-25	25-30	31-35	36-40	41+
Number of Sentences		191	282	342	319	326	217	114	118
Average Number of Words		7.3	13.2	18.1	22.9	27.5	33.0	37.8	47.3
Average Maxsteps		6.6	10.0	11.6	13.7	15.1	16.2	17.8	19.3
CLASP^d	Average Time	0.13	2.78	24.41	134.87	336.04	529.82	609.14	655.86
	Number of Timeouts	0	0	0	4	60	123	99	111
CLASP^t	Average Time	0.13	1.45	8.25	41.39	127.69	319.40	494.72	581.36
	Number of Timeouts	0	0	0	0	2	15	41	86

Fig. 3. Experimental results on CLASP using `ccg.asp` on Section 00 of CCGbank.

These experiments demonstrate that for sentences of length 20 and less, the presented approach to CCG parsing is viable.

6 Discussion and Future Work

To increase parsing efficiency of ASPCCGTK we consider to reformulate the CCG parsing problem as a "configuration" problem. This might improve performance. At the same time the framework would keep its beneficial declarative nature. Investigating applicability of incremental ASP [15] to enhance system's performance is another direction of future research. Furthermore, deciding whether a sentence is in the language of a given CCG grammar can be done in polynomial time [28] (if there is unique category for each word in the sentence). In [28] the authors described a recognition algorithm for CCG grammar based on the Cocke-Younger-Kasami (CYK) chart parsing algorithm for Context Free grammars. In the future, we would like to mimic the algorithm in [28] by means of ASP so that the task of enumerating CCG parse trees would rely on its result. We expect substantial performance gains by adopting this approach. Both OPENCCG and C&C rely on variants of CYK algorithm. Also Drescher and Walsh [11] described ASP-based formulation of CYK algorithm for Context Free grammar. Extending the recognition algorithm in [28] to multiple categories makes the problem computable in nondeterministic polynomial time.

It might seem tempting to realize the planning task described in this work in a planning language such as PDDL [24] and use specialized planning tools to compute all parse trees. However CCG parsing requires objects with inner structure, i.e., nested forward and backward slashes in fluent constants, that is cumbersome to encode in planning languages. Furthermore, it is unclear how executability conditions on actions enforcing normalizations maybe stated using standard planing languages.

Preliminary experiments on using the C&C supertagger as a front-end of ASPCCGTK yielded promising results for achieving wide-coverage parsing. The supertagger of C&C not only provides a set of likely category assignments for the words in a given sentence but also includes probability values for assigned categories. C&C uses a dynamic tagging strategy for parsing. First only very likely categories from the tagger are used for parsing. If this yields no result then less likely categories are also taken into account. In the future, we will implement a similar approach in ASPCCGTK.

Creating semantic representations for sentences is an important task in natural language processing. Boxer [4] is a tool which accomplishes this task, given a CCG parse tree from C&C. To take advantage of

[14] `http://nn.cs.utexas.edu/pages/research/borg/`.
[15] `http://potassco.sourceforge.net/`.

this advanced computational semantics tool, we aim at creating an output format for ASPCCGTK that is compatible with Boxer.

As our framework is a generic parsing framework, we can easily compare different CCG rule sets with respect to their efficiency and normalization behavior. We also suspect that improving scalability of `ccg.asp` is possible using an alternative combinatory rule set in place of the one currently implemented in `ccg.asp`. Type raising is a core source of nondeterminism in CCG parsing and is one of the reasons for spurious parse trees and long parsing times. In the future we would like to evaluate an approach that partially eliminates type raising by pushing it into all non-type-raising combinators. A similar strategy has been proposed for composition combinators by Wittenburg [30].[16] Combining CCG rules this way creates more combinators, however these rules contain fewer nondeterministic guesses about raising categories. The reduced nondeterminism should improve solving efficiency without losing any CCG derivations.

Acknowledgments. We would like to thank John Beavers and Vladimir Lifschitz for valuable detailed comments on the workshop paper that presented the preliminary results on this work [21]. We are especially grateful to Bryan Silverthorn for sharing with us the experimental results presented in Figure 3. We are indebted to Jason Baldridge, Marcello Balduccini, Johan Bos, Esra Erdem, Michael Fink, Michael Gelfond, Joohyung Lee, and Miroslaw Truszczynski for useful discussions and comments related to the topic of this work. Yuliya Lierler was supported by a CRA/NSF 2010 Computing Innovation Fellowship. Peter Schüller was supported by the Vienna Science and Technology Fund (WWTF) project ICT08-020.

References

1. Beavers, J.: Documentation: A CCG implementation for the LKB. Tech. rep., Stanford University, Center for the Study of Language and Information (2003)
2. Beavers, J., Sag, I.: Coordinate ellipsis and apparent non-constituent coordination. In: International Conference on Head-Driven Phrase Structure Grammar (HPSG'04). pp. 48–69 (2004)
3. Beavers, J.: Type-inheritance combinatory categorial grammar. In: International Conference on Computational Linguistics (COLING'04) (2004)
4. Bos, J.: Wide-coverage semantic analysis with boxer. In: Bos, J., Delmonte, R. (eds.) Semantics in Text Processing. STEP 2008 Conference Proceedings. pp. 277–286. Research in Computational Semantics, College Publications (2008)
5. Cimatti, A., Pistore, M., Traverso, P.: Automated planning. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation. Elsevier (2008)
6. Clark, S., Curran, J.R.: Log-linear models for wide-coverage CCG parsing. In: SIGDAT Conference on Empirical Methods in Natural Language Processing (EMNLP-03) (2003)
7. Clark, S., Curran, J.R.: Parsing the WSJ using CCG and log-linear models. In: Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL'04). pp. 104–111. Barcelona, Spain (2004)
8. Clark, S., Curran, J.R.: Wide-coverage efficient statistical parsing with CCG and log-linear models. Computational Linguistics 33(4), 493–552 (2007)
9. Djordjevic, B., Curran, J.R.: Efficient combinatory categorial grammar parsing. In: Proceedings of the 2006 Australasian Language Technology Workshop (ALTW). pp. 3–10 (2006)
10. Dowty, D.: Type raising, functional composition, and non-constituent conjunction. In: Oehrle, R.T., Bach, E., Wheeler, D. (eds.) Categorial grammars and natural language structures, vol. 32, pp. 153–197. Dordrecht, Reidel (1988)
11. Drescher, C., Walsh, T.: Modelling grammar constraints with answer set programming. In: Gallagher, J.P., Gelfond, M. (eds.) Technical Communications of the 27th International Conference on Logic Programming, ICLP 2011. vol. 11, pp. 28–39 (2011)
12. Eisner, J.: Efficient normal-form parsing for combinatory categorial grammar. In: Proceedings of the 34th annual meeting on Association for Computational Linguistics (ACL'96). pp. 79–86 (1996)
13. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning: Semantics and complexity. ACM Trans. Comput. Logic 5, 206–263 (April 2004)
14. Gebser, M., König, A., Schaub, T., Thiele, S., Veber, P.: The BioASP library: ASP solutions for systems biology. In: 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI'10). vol. 1, pp. 383–389 (2010)

[16] Wittenburg introduced a new set of combinatory rules by combining the functional composition combinators with other combinators. By omitting the original functional composition combinators, certain spurious parse trees can no longer be derived.

15. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental asp solver. In: Proceedings of International Logic Programming Conference and Symposium (ICLP'08) (2008)
16. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to gringo, clasp, clingo, and iclingo. (2010), http://sourceforge.net/projects/potassco/files/potassco_guide/2010-10-04/guide.pdf
17. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Proceedings of 20th International Joint Conference on Artificial Intelligence (IJCAI'07). pp. 386–392. MIT Press (2007)
18. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Proceedings of International Logic Programming Conference and Symposium (ICLP'88). pp. 1070–1080. MIT Press (1988)
19. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
20. Hockenmaier, J., Steedman, M.: CCGbank: A corpus of CCG derivations and dependency structures extracted from the Penn Treebank. Comput. Linguist. 33, 355–396 (2007)
21. Lierler, Y., Schüller, P.: Parsing combinatory categorial grammar with answer set programming: Preliminary report. In: Workshop on Logic programming (WLP) (2011)
22. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence 138, 39–54 (2002)
23. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer Verlag (1999)
24. McDermott, D., et al.: PDDL — the Planning Domain Definition Language. Tech. rep., Yale Center for Computational Vision and Control (1998), CVC TR-98-003/DCS TR-1165
25. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25, 241–273 (1999)
26. Partee, B., Rooth, M.: Generalized conjunction and type ambiguity. In: Baeuerle, R., Schwarze, C., von Stechov, A. (eds.) Meaning, Use, and Interpretation. pp. 361–383 (1983)
27. Steedman, M.: The syntactic process. MIT Press, London (2000)
28. Vijay-Shanker, K., Weir, D.J.: Polynomial time parsing of combinatory categorial grammars. In: Proceedings of the 28th annual meeting on Association for Computational Linguistics. pp. 1–8. ACL '90 (1990)
29. White, M., Baldridge, J.: Adapting chart realization to CCG. In: European Workshop on Natural Language Generation (EWNLG'03) (2003)
30. Wittenburg, K.: Predictive combinators: a method for efficient processing of combinatory categorial grammars. In: 25th Annual Meeting of the Association for Computational Linguistics (ACL'87). pp. 73–80 (1987)
31. Wittocx, J.: IDPDraw (2009), Katholieke Universiteit Leuven, http://dtai.cs.kuleuven.be/krr/software/download