2011

Parsing Combinatory Categorial Grammar with Answer Set Programming: Preliminary Report

Yuliya Lierler
University of Nebraska at Omaha, ylierler@unomaha.edu

Peter Schüller
Sabancı University

Parsing Combinatory Categorial Grammar with Answer Set Programming: Preliminary Report

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 a small set of 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 [23] 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 developed and 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

---

[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.

of CCG derivations and dependency structures [17]. The parsing algorithm of C&C returns a *single* most probable parse tree for a given sentence.

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 [19, 20]. ASP is a declarative programming formalism based on the answer set semantics of logic programs [15]. 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. Similarly to OPENCCG and TCCG, 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, 9, 24]. 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 comparable to C&C. 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.

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. An ASP approach to CCG parsing implemented in ASPCCGTK can be seen 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,

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 [10, 18, 20].

2.2 Answer Set Programming (for Planning)

Answer set programming (ASP) [19, 20] is a declarative programming formalism based on the answer set semantics of logic programs [15, 16]. 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[5] system with its front-end (grounder) GRINGO [13], 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 [18] 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 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 a 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 [13]. 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 [14].

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 [18]. 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 [18] 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*.

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) [22] is a linguistic grammar formalism. Compared to other grammar formalisms, CCG uses a comparatively small set of combinatory rules – combinators – to combine comparatively rich lexical categories of words.

Categories in CCG are either atomic or complex. For instance, noun N, noun phrase NP, propositional phrase PP, 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. For presentation of parsing we limit each word to one category. 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 [22]: forward and backward application ($>$ and $<$, respectively), forward and backward *composition* ($>$**B** and $<$**B**), forward and backward *type raising* ($>$**T** and $<$**T**), backward *cross composition*, backward *cross substitution*, and *coordination*. Specifications of some of these combinators follow:

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

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,

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

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

$$\cfrac{\cfrac{\cfrac{The}{NP/N} \quad \cfrac{dog}{N}}{NP} > \quad \cfrac{\cfrac{bit}{(S\backslash NP)/NP} \quad \cfrac{John}{NP}}{S\backslash NP} >}{S} <. \tag{4}$$

Section 3.1 gives a formal definition of the CCG parsing task.

Type Raising and Spurious Parses: CCG restricted to application combinators generates the same language as CCG restricted to application, composition, and type raising rules [8, 21]. One of the motivations for type raising are non-constituent coordination constructions[6] that can only be parsed with the use of raising [2, Example (2)].

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 [9, 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

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

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).

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 [24] or restrictions on parse tree shape [9]. 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

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

---

[6] 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".

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 (i) assigning a distinct id to each member of its precondition sequence, and (ii) assigning the id of the left most annotated category in the precondition sequence to its effect. We say that such an instance is 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).

We say that an annotated instance C of a CCG combinator is *relevant* to an ASR sequence A if the precondition sequence of C is a substring of A. 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, annotated instance (7) is relevant to ASR (6). 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:

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

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

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

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.

We note that the planning problem that we solve is somewhat different from the one we just described as we would like to eliminate ("ban") some of the plans corresponding to spurious parses by enforcing normalizations.

3.2 ASP Encoding

In an ASP approach to CCG parsing, the goal is to encode the planning problem presented above 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`[7] 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

$$posCat(1, rfunc(\text{``}NP\text{''}, \text{``}N\text{''}), 0). \ posCat(2, \text{``}N\text{''}, 0).$$
$$posCat(3, rfunc(lfunc(\text{``}S\text{''}, \text{``}NP\text{''}), \text{``}NP\text{''}), 0). \ posCat(4, \text{``}NP\text{''}, 0). \quad (9)$$
$$posAdjacent(1, 2, 0). \ posAdjacent(2, 3, 0). \ posAdjacent(3, 4, 0).$$

Next we need to choose how we encode actions by ground atoms. 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 $<$)[8]. We call these combinators *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 $occurs(a, p_L, p_R, t)$ states that an action a applied to positions p_L and p_R 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,
– $occurs(ruleFwdAppl, 1, 2, 0)$ represents an application of (7) to (9) at time 0.

Given an atom $occurs(A, P, X, T)$ we sometimes say that an action A *modifies* a position P at time T. The GENERATE section of `ccg.asp` contains the rules of the kind

$$\{occurs(ruleFwdAppl, L, R, T)\} \leftarrow posCat(L, rfunc(A, B), T), \ posCat(R, B, T),$$
$$posAdjacent(L, R, T),$$
$$not \ ban(ruleFwdAppl, L, T),$$
$$time(T), \ T < maxsteps.$$

---

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

[8] In fact, coordination combinator is of the third type, i.e., its precondition sequence contains three elements. Presenting the details of its encoding is out of the scope of this paper.

for each combinator. Such choice rules describe a potential solution to the planning problem as an arbitrary set of actions executed before *maxsteps*. These rules also captures some of the executability conditions of the corresponding actions. For example, $posCat(L, rfunc(A, B), T)$ states that the left member of the precondition sequence of the forward application combinator $ruleFwdAppl$ is of the form A/B. At the same time, $posAdjacent(L, R, T)$ states that $ruleFwdAppl$ may be applied only to adjacent positions. A relation $ban(a, p, t)$ specifies when it is impossible for an action a to modify position p at time t. Often there are several rules defining this relation for a combinator. These rules form the main mechanism by which normalization techniques are encoded in `ccg.asp`. For instance, a rule defining ban relation

$$ban(ruleFwdAppl, L, T) \leftarrow occurs(ruleBwdRaise, L, X, TLast-1),$$
$$posLastAffected(L, TLast, T),\ time(TLast),$$
$$time(T),\ T < maxsteps.$$

states that a forward application modifying a position L may not *occur* at time T if the last action modifying L was backward 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 [9].

There are a number of rules that specify effects of actions in the CCG parsing domain. One such rule

$$posCat(L, A, T+1) \leftarrow occurs(ruleFwdAppl, L, R, T),$$
$$posCat(L, rfunc(A, B), T),\ time(T),\ T < maxsteps.$$

states that an application of a forward application combinator at time T causes a category annotated by L to be X at time $T+1$.

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

$$posAffected(L, T+1) \leftarrow occurs(Act, L, R, T),\ binary(Act),$$
$$time(T),\ T < maxsteps.$$

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

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

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 $maxsteps$.

Finally we pose additional restrictions, which ensure that only a single plan is produced when multiple serializable plans correspond to the same parse tree. Note that applying a CCG rule r at a time t creates a new category required for subsequent application of another rule r' at a time $t'>t$. We request that r' is applied at $t'=t+1$. Furthermore, in `ccg.asp` we enforce the condition that combinators are applied as early as possible: by requesting that a rule applied at time t uses at least one position that was modified at time $t-1$.

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 $occurs(a, p, c, t)$ 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 propositional (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 [9]. This is realized by disallowing functional forward composition to apply to a category on the left side, which has been created by functional forward composition. (And similar for backward composition.)
- We disallow certain combinations of rule applications if the same result can be achieved by other rule applications as shown in the following

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

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 similar form in [3, 9].

4 ASPCCG Toolkit

We have implemented ASPCCGTK— a python[9] framework for using `ccg.asp`. The framework is available online[10], 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 [11]. 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) a natural language sentence given as a string, or (b) 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[11] [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 (b) 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 not possible, we do "best-effort" parsing using CLASP optimization features to minimize the number of categories left at the end. For instance, consider a lexicon that provides a single category for "bit", namely $(S\backslash NP)/NP$, then the following derivation

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

corresponds to a best-effort parse.

---

[9] `http://www.python.org/`

[10] `http://www.kr.tuwien.ac.at/staff/ps/aspccgtk/`

[11] `http://svn.ask.it.usyd.edu.au/trac/candc`

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`.[12] The resulting answer sets of `ccg2idpdraw.asp` contain drawing instructions for the IDPDraw tool [25], 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.
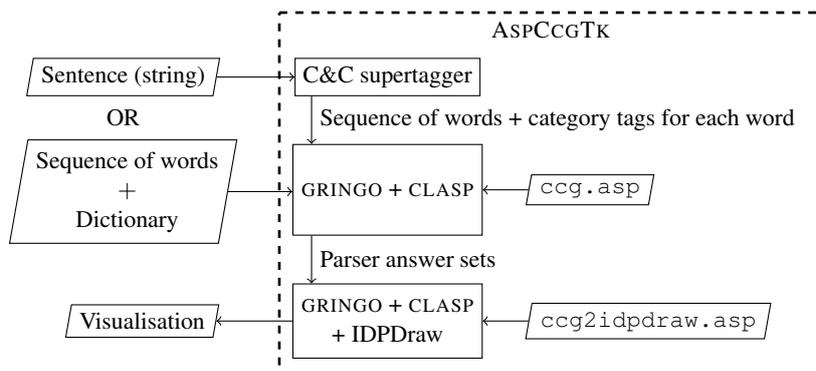
Fig. 1. Block diagram of the ASPCCG framework. (Arrows indicate data flow.)
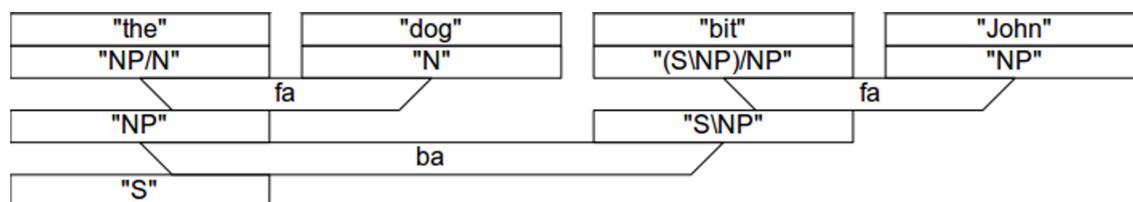

Fig. 2. Visualization of parse tree (4) for sentence (1) using IDPDraw.

5 Discussion and Future Work

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.

We have evaluated the efficiency of AspCcgTk on a small selection of examples from CCGbank [17]. In the future we will evaluate our parser against a larger corpus of CCGbank, considering both parsing efficiency and quality of results as evaluation criteria. Experiments done so far are encouraging and we are convinced that wide-coverage CCG parsing using ASP technology is feasible.

---

[12] 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.

To increase parsing efficiency 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 [12] to enhance system's performance is another direction of future research.

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 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 next discuss an idea for improving scalability of `ccg.asp` that is based on an alternative combinatory rule set to the one currently implemented in `ccg.asp`. Type raising is a core source of nondeterminism in CCG parsing and is one of the main 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 [24].[13] 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.

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. 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)
9. 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)
10. 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)
11. 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)

---

[13] 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.

12. 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)
13. 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`
14. 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)
15. 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)
16. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
17. Hockenmaier, J., Steedman, M.: CCGbank: A corpus of CCG derivations and dependency structures extracted from the Penn Treebank. Comput. Linguist. 33, 355–396 (2007)
18. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence 138, 39–54 (2002)
19. 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)
20. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25, 241–273 (1999)
21. 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)
22. Steedman, M.: The syntactic process. MIT Press, London (2000)
23. White, M., Baldridge, J.: Adapting chart realization to CCG. In: European Workshop on Natural Language Generation (EWNLG'03) (2003)
24. 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)
25. Wittocx, J.: IDPDraw (2009), Katholieke Universiteit Leuven, `http://dtai.cs.kuleuven.be/krr/software/download`