

Review: Realization with CCG

Alex Rudnick (alexr@cs.indiana.edu)

May 2010

Abstract

Here I give an overview of recent work on natural language realization with Combinatory Categorical Grammar, done by Michael White and his colleagues, with some more specific descriptions of the algorithms used, where they were unclear to me. In particular, I focus on the work presented in his 2007 paper [5], in which White et al describe a process for extracting a grammar from the CCGBank and using it to generate text based on semantic descriptions given in HLDS, the Hybrid Logic Dependency Semantics. I also give some background from his earlier work, a description of some background ideas.

1 Overview

In their 2007 paper, Towards Broad Coverage Surface Realization with CCG [5], White, Rajkumar and Martin present their progress towards developing a system capable of producing text from semantic representations of a wide variety of sentences, such as those found in news text. The sentences actually used in this effort are from the Penn Treebank, by way of CCGbank, a corpus of Penn Treebank sentences coupled with CCG parses due to Hockenmaier and Steedman [4].

The extensions presented by White et al broaden the realization coverage and robustness of OpenCCG [10], a freely available system for NLP with the Combinatory Categorical Grammar, in several novel ways. First, they make use of a large, automatically extracted grammar, the creation of which took some engineering work, described in the paper. Additionally, to cope with the large search space of possible sentences licensed by a purely symbolic grammar, they use a statistical language model to guide the search for high-quality output text. They also add the ability to generate sentences not explicitly licensed by the grammar, when a semantic form can't be completely expressed. Rather than simply giving up, the system always produces some text for a given semantic representation, although the text produced may not be completely grammatical.

While there have been previous realizers using OpenCCG for more restricted domains with hand-crafted grammars, the system presented by White et al aims to demonstrate how to build realization systems that handle a wide range of semantic inputs with relative ease, working acceptably for open-domain applications. In the long run, such a broad-coverage realizer could be used for machine translation based on semantic transfer [5].

Overall, the system seems to work well, both in light of BLEU scores calculated between the generated text (the scores are far above those typical for machine translation systems) and the sensibility of the sentences generated, although deciding on an appropriate metric for success in this task is something of an open problem.

In the following sections, I describe the system’s generation process, including the chart realizer, its statistical language model and the process for generating sentences not licensed by the grammar; how the grammar is extracted from CCGbank; and issues with evaluating the system. I also give a brief overview of BLEU and HLDS, the Hybrid Logic Dependency Semantics. Text in `typewriter` refers to functions and class names from the OpenCCG software [10].

2 Realization process

The realization process follows a “generate-and-select” paradigm; the generative aspect is a symbolic process based on production rules, in the form of a chart realizer. The selection happens through a statistical language model, which the system uses to rank proposed solutions, and when applicable, guide a beam search. The extracted grammar itself is not weighted or probabilistic; all ranking of derivations is done by the language model. This approach contrasts with some familiar techniques for broad-coverage grammars, such as PCFGs. It may be useful to think of this system as similar to the decoder of a machine translation system, if we imagine the extracted CCGbank grammar as a synchronous grammar (as described by Chiang [3]) between logical forms and syntactic derivations.

The realizer has two modes, which have been explored at different phases of the research. In the 2007 paper, White et al make use of the “anytime best-first search” mode, which enables users to spend more time or computing power for higher-quality results. The goal of anytime mode is to reliably produce acceptably good realizations quickly, without worrying about finding all possible realizations [8], which is the practical case for a task like a dialog system or machine translation, where only one output realization is required. The other mode supported by the CCG realizer is a two-stage algorithm with packing and unpacking phases, which will find all licensed derivations. The two-stage algorithm is described in detail in the 2006 paper [6].

2.1 Chart realization

In their 2003 paper, White and Baldridge describe the basic operations of the OpenCCG chart realizer [8], which forms the basis of the broad-coverage realization system used in the more recent work [5]. Here I describe the process from the 2003 paper, which does not account for semantic representation that contain disjunctions. In White’s 2006 paper, he describes elaborations to the process that make it possible to generate text from logical forms that contain alternatives where one out of several must be chosen, such as synonyms or alternative phrasings. As far as I can tell, while this capability was mentioned in the 2007 paper, it was not used in the experiments.

While there are many possible approaches to realization, White et al choose a chart realizer because it can use the same grammar as a chart parser [6], which makes it easier to

ensure that all derivations that could be produced with the realizer are also parsable, and vice versa. A chart-based realizer also allows optimizations developed for the chart parser to be adapted for use in the realizer, and makes generation agnostic regarding the order of terms in the logical form (LF) input. This last desideratum is useful for the OpenCCG approach; the realization algorithm relies on being able to reorder parts of a logical form.

In OpenCCG, the chart realizer is a bottom-up process that produces a derivation of the input logical form, and thus the text of the sentence, covering every subpart of the input semantic representation. A nonobvious high-level insight is that the derivations being searched for are not syntactic derivations, but semantic. This is an important distinction: the realization process is a search through the space of ways to combine the *semantic* components of lexical entries. Once that derivation has been found, as a happy side effect, it has also produced a syntactic representation of the sentence and thus the output text.

The OpenCCG chart realizer works in several steps. First, it flattens the input LF into a list of elementary propositions, or EPs, each of which describes one very small part of the semantic content of a sentence. These EPs will form the basis of the chart over which the realizer will try to form a derivation, analogous to individual words that need to be accounted for by a parser, although they do not have a linear order.

Once all of the EPs have been listed, the realizer looks up lexical entries that have semantic entries matching those of the EPs that we want to cover, records both their semantic and syntactic components, and puts these entries into a list of components that may be used to construct the desired meaning. In the 2003 paper, there is a note that some of the retrieved lexical entries may contain semantic features that are not called for by the input LF, which would result in an output LF (thus output text) that contains semantic features not specified by the input [8]. This is interesting and possibly worrisome, but it's not discussed how often this is necessary, or whether it helps produce desirable output.

To account for the possibility that a given predicate in the LF may not have a corresponding lexical entry in the grammar in use, the system also has a non-probabilistic idea of “smoothing” that makes sure there are suitable lexical entries available. For each word in the test sentences, out-of-vocabulary nouns, proper names, numbers, adjectives, verbs and adverbs are all given lexical categories that were seen for their respective part-of-speech, but not for those particular words. Verbs are all ascribed the five most common lexical entries for verbs, and adverbs the three most common for adverbs, presumably with the predicate in their LF changed. Other words are given all categories that occur more than 50 times in the training corpus [5].

From this point, the realizer tries to find a derivation that combines the semantic forms of the lexical entries into the given LF. This is done with three types of data structures. The first is an *edge*, which consists of a CCG sign (derived from a lexical entry, initially), coupled with an encoding of which parts of the target LF this edge covers, and which indices (nominals) can be referenced from this sign. For efficiency reasons, the latter two are stored as bit vectors, which enable quick checks for disjointness by taking the bitwise AND of two edges and checking for a result of zero. Edges initially correspond to lexical entries, but the result of combining two edges during the search procedure is also an edge, so throughout the semantic derivation, they correspond with chunks of semantic and syntactic material larger than a word.

Edges can reside either in the *agenda*, or in the *chart*. The agenda is a priority queue of

edges that should be added to the chart; the realizer can pluggably use different strategies for sorting the agenda, causing different realizations to be reached first. The chart is similar to the chart of a chart parser, in that it keeps track of partial solutions to the problem at hand, but different in two ways: it keeps track of which elementary propositions an edge covers instead of which words a syntactic constituent covers, and since there’s no linear order for EPs, partial solutions are not kept in a two-dimensional array, as would be typical with chart parsing. The edges (partial solutions) are simply kept in a list and scanned linearly; the algorithm, described succinctly in Figure 1 in the 2003 paper [8], is in fact implemented very simply with Java ArrayLists, in the class `opennlp.ccg.realize.Chart` [10].

The chart parser proceeds as follows:

- While there are remaining edges in the agenda, get the first edge in the agenda and call this the “current edge”. We now attempt to add the current edge into the chart, but will skip it if there’s already an edge that covers the same set of EPs.
- For each edge already in the chart, see if we can combine the current edge with this edge. This can be done when the current edge and the edge in question have disjoint LF coverage but they have overlapping references in the LF (i.e., they can be combined semantically)¹, and there is a syntactically valid way to combine the two edges. Any successful combination of the current edge with edges already in the chart creates a new edge, which is added to the agenda.
- Apply any unary (e.g., CCG type-raising) rules that apply to the current edge and combine with semantically null words, such as syntactic particles.
- Add the current edge to the chart and continue processing through the agenda.

Each edge in the chart knows how much of the LF in question it covers, and the two modes differ in what they do having found an edge that with complete coverage. The pack-unpack mode continues searching until the agenda is empty (i.e., all possible derivations have been added to the chart, in a packed representation), and then can expand them into their constituent parts on demand (unpacking). The any-time search will terminate having found an edge that covers the entire LF, if the allotted search time is up, but continue looking for more solutions if there’s more time available.

Next, we’ll describe the language model that guides the search for derivations.

2.2 Language model

As mentioned earlier, the realizer system needs some way to efficiently search the space of derivations and select the best outputs, given that many may be possible. The grammar itself has no notion of which syntactic productions are the most likely. So in order to rank solutions, both partial and complete, White et al tried several different versions of trigram models, combining scores based on transitions over words, part-of-speech tags, and supertags.

¹This is not strictly true in the case of generating coordinations; see the 2003 paper [8] for the complete account.

The ngram-based language models assign a probability to a word, given the context of the two previous words, and to a part-of-speech tag, given the two previous tags. The probability of a supertag is estimated not using the two previous supertags, but the two previous tags. These models are built using the SRILM toolkit², and trained using sections 2 to 21 of the CCGbank, which are referred to as the standard training sections [5].

Given that the language models assign probabilities to individual words in a particular context, one might wonder how these scores are combined to give a score for an entire edge, which will typically include many words. Multiplying probabilities together, as is described in Equation 1 in the 2007 paper [5], would mean that scores get monotonically lower, since probabilities are always in $[0, 1]$. This seems to suggest that larger (more complete) edges would tend to be filtered out by the beam search.

The paper doesn't explicitly answer the question of how this problem is avoided, but taking a look at the code in `FactoredNgramModelFamily` and `NgramScorer:logProb`, it looks as though the logarithms of probabilities are just being added, which is equivalent to the probabilities being multiplied. I haven't yet understood how this can work such that the edges that contain more words aren't bumped from the chart at pruning time. But since realizations are in fact being produced, I must have missed something.

White et al report that the most effective approach to scoring that they found was to multiply the scores from the POS model with the supertag model, and then take a weighted average between that number and the word-based trigram scores, giving 75% of the weight to the word-based ngram model. The weights here weren't tuned, but seem effective enough. One wonders what would have happened if different weights were used, or if these are parameters that could be tuned in a principled way.

2.3 Putting it together: beam search

White et al use a beam search to find the most promising realizations, as described in the 2007 paper [5]. Not much is said about it, merely that there's a beam width (which they call "n-best pruning value") of 5. What this means in practice, according to `Chart:addEdgeToChart` and `NBestPruningStrategy`, is that at any given point during the search, there may be up to five edges in the chart at any point. These edges may be partial or complete solutions, but they are kept in order by score (given by the language model), and every time a new edge is added to the chart, edges after the best five are dropped.

There is also a configurable time limit on search. In the paper, White et al use a limit of 15 seconds; presumably longer times could be used to get better results, although in the paper the time limit was held constant. A longer time limit might be helpful for a larger grammar, however; when testing the realizer with a grammar derived from all available training data, the realizer runs out of time on 68% of the test sentences.

2.4 Graceful degradation

To handle the case where no complete realization for the desired LF can be found, White et al add an extension to the chart realizer that combines partial solutions into some output

²Available here: <http://www-speech.sri.com/projects/srilm/>

text, even when this is not licensed by the grammar. They say in the 2007 paper that it’s the most significant extension to the realizer itself implemented so far [5].

The extension is implemented in about thirty lines of Java in `Chart:joinBestFragments` [10]; it works by finding the most complete realization (the one that covers the largest fraction of the desired elementary predications), with ties broken by highest score from the language model. This is placed in a list of edges that will be concatenated together. After the biggest chunk is found, then the system searches the chart and agenda, greedily, for edges that cover disjoint parts of the LF, adding the ones with the greatest (disjoint) coverage to the list first, until no further nodes can be found in the chart or the agenda.

At this point, the target LF may or may not be covered, but there is a list of fragments that can be joined together. Then for each fragment in the list, it is added to a running concatenation of the edges, either on the left or right, depending on which side yields a higher score from the language model. This process makes just one pass through the list of fragments, and may not yield grammatical or sensible output, but it does result in text of some sort.

3 Extracting the grammar

In the 2007 paper, White et al present the novel use of a large, automatically extracted grammar for generating text with OpenCCG. They derive the grammar for the realizer from the CCGbank corpus, in a process that takes several steps. The first step is to translate the sentences in the CCGbank into an XML format, so that they can be operated on by XSLT, a programming language for performing transformations on XML data. As they’re distributed, the sentences in the CCGbank corpus look something like the original Penn Treebank format ³, and are thus not compatible with XSLT.

After the sentences are converted to XML, the XSLT processing changes the derivations of the sentences “to reflect the desired syntactic derivations” [5]; the differences between the CCGbank analyses of sentences and analyses that would work well with OpenCCG have to do with coordinaton and the handling of punctuation. Additional changes must be done “to support semantic dependencies rather than surface syntactic ones”. I would like to understand and explain this process more deeply, but in the interest of time and space (and since I’m unfamiliar with XSLT), I will leave it as an exercise to the interested reader to look at the relevant source in the OpenCCG repository in `ccgbank/extract`.

After syntactic transforms have been performed, the system extracts a grammar from the derivations in the corpus. During this process, HLDS logical forms must be produced for the lexical entries; this is done by matching derivations against a series of “generalized templates” for HLDS entries; then syntactic parts of derivations are matched to semantic roles. At this point, due to the lexicalized nature of CCG, the work of extracting the grammar is done: all the information about a lexical entry and how it behaves with respect to the CCG combinatory rules is contained in the lexical entry itself.

White et al additionally do some pruning on the grammar for the experiments presented in the 2007 paper. Particularly, they exclude lexical categories (words with a given syntactic

³Example CCGbank sentences are available on Hockenmaier’s site here:
<http://www.cs.uiuc.edu/~juliahr/CCGbankDemo/>

shape) and unary type-changing rules that occur less than a constant number of times ($k = 3, 5, \text{ or } 10$, depending on the experiment being performed), and only allow categories and rules that match the generalized templates.

Right after the grammar extraction process, but possibly before the pruning takes place (the paper is unclear on this point, in section 3), sentences from the development and test sections that can be successfully parsed according to their gold-standard derivations have their logical forms saved for later testing of the realizer. The great majority, roughly 95%, of sentences in the development and testing sets are assigned LFs in this way.

4 Experiments

For the 2007 paper, White et al run two experiments, “non-blind” and “blind”. For the non-blind experiment, they extract a grammar from the development section of the CCGbank, then for each LF that was successfully extracted from the development section, they run the realizer in any-time search mode with a beam-width of five and a cutoff time of 15 seconds. For this development run, they prune out lexical categories that occurred fewer than five times, and unary rules that occurred fewer than three times, where these parameters were empirically tuned during development. During the non-blind experimentation, they try several different language models for comparison, including various combinations of a word-based trigram model (with different kinds of lexical smoothing), a POS tag trigram model, and the supertag trigram model, explained earlier. They find that the scoring method that includes all three language models is the most effective on the development set.

For the blind portion of the testing, they use the traditional training/test split, and extract a larger grammar from the training portion of the CCGbank, sections 2 through 21. Lexical categories and unary rules that occurred fewer than ten times were pruned. Having extracted the larger grammar, they go through all of the LFs taken from the development and test sections and run the any-time search procedure (again with a 15-second time limit) for realizations, using the combined trigram language model (word-based, tag-based, and supertag-based). For comparison, they also repeat the experiment with just the word-based trigram model. Again, they find that the combined trigram language model has the best performance, but with the larger grammar, the search for complete realizations is much less successful.

5 Results

White et al report BLEU scores for the two experiments in Tables 1 and 2 of the 2007 paper [5]. They find that, with the best scoring model, they are able to get a BLEU score of 0.6615 on the development data (with the grammar based on the development data), and on the blind test, they are able to get a BLEU score of 0.5578 over the development sentences, and 0.5768 over the test sentences. These scores are all much better than the scores that were produced using just the word-based trigram model: the blind test score for that model is 0.5178, for an increase of almost 6 BLEU points. So it’s fairly clear that the more detailed language model helps realization in this case.

Additionally, they give a few examples of sentences that produced by the realizer; these two were generated from the well-known first two sentences of the Penn Treebank, using the grammar derived from the development set. Generating the first sentence apparently required the use of the fragment concatenation algorithm.

- “61 years old Pierre Vinken will join the board as a nonexecutive director Nov. 29”
- “Mr. Vinken is chairman of Elsevier N.V. , the publishing Dutch group.”

These two sentences contain exactly the same words as those in the Penn Treebank sentences, just in slightly different orders: the first sentence has “61 years old” moved before “Pierre Vinken”, whereas in the original it occurs afterwards. The second realization presented simply switches the order of “Dutch” and “publishing”.

White et al also report on the fraction of the time that complete realizations were achieved only using the grammar, as opposed to the fragment-adjoining algorithm. Apparently in the development tests, only 55% of the sentences are realized completely from the grammar, with the other 45% having their results composed by fragment adjoining. The results are worse when the larger grammar was used; in that case, the system only found complete realizations for 22% of the development sentences, and ran out of time during search in 68% of the cases. They write in section 4.2 of the 2007 paper that this is due to search errors; the larger grammar licenses too many derivations for the beam search to completely explore in the time allotted, and the trigram language model only helps so much.

To confirm that search errors were a problem, they run further experiments, including one in which they are able to generate complete realizations for 50% of the sentences in one file from the development set using the large grammar by making use of an “oracle” language model that gives better scores for higher BLEU scores. This is certainly cheating for machine-learning purposes, but useful for debugging. We know that the gold standard derivation is in fact licensed by the extracted grammar, otherwise the sentence would not have been included in the development set. So the failures to find a complete derivation in that case must be due to search errors; perhaps the beam is too narrow.

5.1 About BLEU

BLEU is a standard metric typically used to judge the performance of machine translation systems. It estimates the goodness of the translation of an entire corpus by taking the “modified precision” of the output text with respect to example translations, which were produced by human translators [9]. Typically, measuring the precision of a translation would be done by taking the fraction of words in the proposed output text that occur in one of the reference translations; this is performed on a sentence-by-sentence basis. This approach sounds reasonable, but has the problem that a candidate translation may contain only one of the words in the reference translation and still have perfect recall. For example, in “the the the the the the”, every word in the candidate translation is present in the reference “the cat is on the mat”, so it is given a precision of 7/7 [9]. Modified precision changes this by only giving points for the first n occurrences of a word in the candidate translation, when it occurs n times in a reference. Thus, the series of seven “the”s gets a modified precision of 2/7, since “the” occurs twice in the reference.

Analogous modified precision scores are computed, sentence by sentence, for all of the ngrams in the candidate sentences, up to a length of four by default. Then these modified precisions are combined together in a geometric average and given a “brevity penalty” (to discourage translations like “the”, which still has a modified precision of 1/1). The exact equation used to calculate scores is given in section 2.3 of the BLEU paper [9]. In experiments run by the developers of BLEU, they find that it correlates well with human judgement of translation quality. But in footnote 4, they also note the importance of having several different human translators produce reference text, because there can be many valid ways to express the same idea [9].

In the long run, BLEU scores as a metric for this realization task may not be the best choice, for two reasons; the first being that there should be a number of ways to express the same idea (corresponding to a semantic representation), but there is only one reference realization used to compute BLEU scores in the 2007 White paper – just the original source text. Secondly, the use of HLDS predicates that correspond so closely to individual words seems like it would almost guarantee that the words in the source sentence will be nearly identical to the ones in the output sentence. Taking a look at Figure 1 in [5], if we look at the predicates used and interpret them as words instead of abstract semantic entities, we see: *design, based on, be, collection, series, Funny Day, and Villeroy and Boch*; the presence of these words constricts which possible sentences could be generated greatly.

White et al seem to have similar concerns, writing in the 2007 paper: “We suspect that BLEU scores may no longer be useful in measuring progress, given ... our goal of producing desirable variation. As such, we expect that targeted human evaluations will become essential.”

6 Conclusions and questions

White et al describe a system that extracts a broad-coverage grammar from a large corpus of CCG derivations and then uses it, apparently successfully, to generate text based on semantic representations that they extract from sentences in the same corpus.

Quite a lot of interesting future work remains in applications of this technique. Which domains will this system work well for? Will the grammar extracted from the CCGbank, based on news text as it is, generalize well to other domains? Which ones?

6.1 Possible applications

The application of this system to a given domain depends on being able to produce the semantic representations in HLDS. It may not be theoretically difficult to produce HLDS propositions given entries in a database or some other logical formalism, but surely it would require some software engineering for each new domain. Worse, many database rows correspond to boring sentences: “Inventory item number 1072 is a *T-SHIRT* that has size *LARGE* and color *RED*”.

Using the OpenCCG realizer for machine translation would require designing an appropriate interlingua, such that HLDS propositions could be mapped from one language into another. HLDS isn’t a very abstract notion of meaning, in that it contains literal words

from the source language; but words don't translate between languages in a clean one-to-one way. It's a very messy many-to-many relationship, and each language divides up the space of possible meanings quite differently. This problem isn't specific to OpenCCG or HLDS, of course, but is more general to machine translation based on semantic transfer.

6.2 Evaluation

A more immediate issue, brought up in the 2007 paper, is experimental design for evaluating this task with human subjects, considering that BLEU scores may be less applicable to this task than to machine translation. In machine translation, evaluation with human subjects can be done by asking bilingual speakers of the source and target languages for their judgments as to whether a source sentence and a proposed translation mean the same thing, or which translation they like the best, among many. But finding human subjects familiar with Hybrid Logic seems difficult unless one has access to a large number of logicians.

One possible evaluation experiment would be to generate a number of different realizations, mix in the sentence from which the input LF was extracted, and have human subjects (native speakers of the output language) judge whether all of the sentences listed mean the same thing, without an indication of which of sentences were generated by computer. A good evaluation metric might be the likelihood that a human judge will decide that the input and output sentences mean the same thing; this metric would express the accuracy of the translation from input text to LF, and back to text. Another axis that should be measured is how natural the output text seems to human readers, and this could be tested by having human judges rank candidate outputs.

6.3 Questions unanswered in the papers

In reading the papers and my cursory exploration of the OpenCCG code, I came across a few questions that I wasn't readily able to answer. The most worrisome one, which I wasn't able to address even after some delving into the source code (although I am likely just missing the answer), is how the longer edges manage to avoid being pruned from the chart. It seems as though there should be some normalization to keep from penalizing longer strings of text, and possibly even a bonus to help the beam search not drop complete solutions. It's clear why phrases and complete sentences should be assigned lower probabilities than words, but I don't understand how this makes a good scoring metric if low-probability edges get pruned.

Another unanswered question is how often the realizer ends up using the rule that it may include lexical items that contain semantic properties not mentioned in the input LF (this is mentioned in Section 4.2 of [8]); is it needed often? If so, do the sentences that it generates come out well? Or perhaps this capability isn't even used in the experiments run for the 2007 paper?

Regarding the any-time search procedure, while presumably many complete derivations corresponding to a given LF could have been licensed by the broad-coverage grammar, a beam search doesn't completely cover the space, and White et al discuss search errors in Section 4.2 of [5]; it would be interesting to know, for a sampling of the LFs, how many complete derivations could have been found given a complete search. When a complete solution was turned up, how good was it compared to the best one possible? Is it ever

the case that the grammar (even with lexical smoothing) simply can't produce a complete derivation for the target LF, or if that were the case, would the sentence have been excluded from testing?

On the topic of the “graceful failure” greedy-concatenation mode, the 2007 paper doesn't give very many examples of its output text, or say how sensible the text it produced was in general. One could imagine that it would tend to produce text with a high BLEU score relative to the input text, but this may be attributable to individual words corresponding with particular predicates in the LF; if the input text contains the word “dog”, this will correspond with an EP that calls for *Dog*; there's no danger that words like “canine” or “puppy” will show up in the output text. This seems like a more fundamental issue for the approach of generating from HLDS inputs without more systemic knowledge of the relationships between words. But handling this problem properly would require a lot of ontology-building – essentially an English-English interlingua.

A About HLDS

Hybrid logic dependency semantics (HLDS) is the formalism used for semantic representations in OpenCCG. To properly understand the term, it helps to get the right bracketing: HLDS is semantics based on *dependencies* and *hybrid logic*, not a hybrid between logic and dependencies. Hybrid logic is a logical formalism that extends modal logic and has the useful property that its propositions are first-class objects themselves, and may be referenced [1]. This is not possible in some other logical formalisms. Hybrid logic is described in detail by Blackburn in his 2000 paper [2].

In HLDS, variables that reference other propositions are called *nominals*, and they are referenced with the @ operator. These nominals are particularly important in OpenCCG, because they are what allow us to flatten out and reorder the semantic representations into a large number of EPs, greatly simplifying the implementation of the chart generator while letting us keep track of which EPs hang together by co-reference. The modal aspect of HLDS allows us to put labels on relationships in an LF; OpenCCG uses modal logic's modes (what would typically be used to indicate, say, degrees of certainty) to indicate relationships between parts of the logical form, say the ARG1/ARG2 or ACTOR/PATIENT of a verb. For more detail, see the 2002 Baldridge and Kruijff paper [1].

References

- [1] Jason Baldridge and Geert-Jan M. Kruijff. 2002. Coupling CCG and Hybrid Logic Dependency Semantics. In Proc. of the 40th Annual Meeting of the ACL.
- [2] Patrick Blackburn. Representation, Reasoning, and Relational Structures: a Hybrid Logic Manifesto. Logic Journal of the IGPL, 8(3), 339-625, 2000.
- [3] David Chiang. An introduction to synchronous grammars. 2006. Tutorial given at ACL 2006.

<http://www.isi.edu/~chiang/papers/synchtut.pdf>

- [4] Julia Hockenmaier and Mark Steedman. CCGbank: a corpus of CCG derivations and dependency structures extracted from the Penn Treebank. 2007. In *Computational Linguistics* 33(3), pp 355-396, MIT press.
- [5] Michael White, Rajakrishnan Rajkumar and Scott Martin. 2007. Towards Broad Coverage Surface Realization with CCG. In *Proc. of the 2007 Workshop on Using Corpora for NLG: Language Generation and Machine Translation (UCNLG+MT)*.
- [6] Michael White. 2006. CCG Chart Realization from Disjunctive Inputs. In *Proc. of the 4th International Conference on Natural Language Generation (INLG-06)*.
- [7] Michael White. 2004. Reining in CCG Chart Realization. In *Proc. of the 3rd International Conference on Natural Language Generation (INLG-04)*.
- [8] Michael White and Jason Baldridge. 2003. Adapting Chart Realization to CCG. In *Proc. of the 9th European Workshop on Natural Language Generation*.
- [9] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a Method for Automatic Evaluation of Machine Translation. *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, Philadelphia, July 2002, pp. 311-318.
- [10] OpenCCG natural language processing library.
<http://openccg.sourceforge.net/>