Using Higher-Order Logic Programming for Semantic Interpretation of Coordinate Constructs

Seth Kulick
University of Pennsylvania
Computer and Information Science
200 South 33rd Street
Philadelphia, PA 19104-6389 USA
skulick@linc.cis.upenn.edu

Abstract
Many theories of semantic interpretation use \( \lambda \)-term manipulation to compositionally compute the meaning of a sentence. These theories are usually implemented in a language such as Prolog that can simulate \( \lambda \)-term operations with first-order unification. However, for some interesting cases, such as a Combinatory Categorial Grammar account of coordination constructs, this can only be done by obscuring the underlying linguistic theory with the "tricks" needed for implementation. This paper shows how the use of abstract syntax permitted by higher-order logic programming allows an elegant implementation of the semantics of Combinatory Categorial Grammar, including its handling of coordination constructs.

1 Introduction
Many theories of semantic interpretation use \( \lambda \)-term manipulation to compositionally compute the meaning of a sentence. These theories are usually implemented in a language such as Prolog that can simulate \( \lambda \)-term operations with first-order unification. However, there are cases in which this can only be done by obscuring the underlying linguistic theory with the "tricks" needed for implementation. For example, Combinatory Categorial Grammar (CCG) (Steedman, 1990) is a theory of syntax and semantic interpretation that has the attractive characteristic of handling many coordination constructs that other theories cannot. While many aspects of CCG semantics can be reasonably simulated in first-order unification, the simulation breaks down on some of the most interesting cases that CCG can theoretically handle. The problem in general, and for CCG in particular, is that the implementation language does not have sufficient expressive power to allow a more direct encoding. The solution given in this paper is to show how advances in logic programming allow the implementation of semantic theories in a very direct and natural way, using CCG as a case study.

We begin by briefly illustrating why first-order unification is inadequate for some coordination constructs, and then review two proposed solutions. The sentence in (1a) usually has the logical form (LF) in (1b).

(1a) John and Bill run.
(1b) (and (run John) (run Bill))

CCG is one of several theories in which (1b) gets derived by raising John to be the LF \( \lambda P.(P \text{ John}) \), where \( P \) is a predicate that takes a NP as an argument to return a sentence. Likewise, Bill gets the LF \( \lambda P.(P \text{ Bill}) \), and coordination results in the following LF for John and Bill:

(2) \( \lambda P.(\text{and} (P \text{ John}) (P \text{ Bill})) \)

When (2) is applied to the predicate, (1b) will result after \( \beta \)-reduction. However, under first-order unification, this needs to simulated by having the variable \( z \) in \( \lambda z.\text{run}(z) \) unify both with Bill and John, and this is not possible. See (Jowsey, 1990) and (Moore, 1989) for a thorough discussion.

(Moore, 1989) suggests that the way to overcome this problem is to use explicit \( \lambda \)-terms and encode \( \beta \)-reduction to perform the needed reduction. For example, the logical form in (3) would be produced, where \( \backslash x.\text{run}(x) \) is the representation of \( \lambda z.\text{run}(z) \).

(3) and(apply(\backslash x.\text{run}(x), \text{John}),
apply(\backslash x.\text{run}(x), \text{Bill}))

This would then be reduced by the clauses for apply to result in (1b). For this small example, writing such an apply predicate is not difficult. However, as the semantic terms become more complex, it is no trivial matter to write \( \beta \)-reduction that will correctly handle variable capture. Also, if at some point it was desired to determine if the semantic forms of two different sentences were the same, a predicate would be needed to compare two lambda forms for \( \alpha \)-equivalence, which again is not a simple task. Essentially, the logic variable \( X \) is meant to be interpreted as a bound variable, which requires an additional layer of programming.
(Park, 1992) proposes a solution within first-order unification that can handle not only sentence (1a), but also more complex examples with determiners. The method used is to introduce spurious bindings that subsequently get removed. For example, the semantics of (4a) would be (4b), which would then get simplified to (4c).

(4a) A farmer and every senator talk

(4b) $\exists(x_1, \text{farmer}(x_1)) \land (\exists(x_2, (x_2 = x_1) \land \text{talk}(x_2))) \\
\forall(x_3, \text{senator}(x_3)) \Rightarrow (\exists(x_2, (x_2 = x_3) \land \text{talk}(x_2)))$

(4c) $\exists(x_1, \text{farmer}(x_1)) \land \text{talk}(x_1) \\
\forall(x_3, \text{senator}(x_3) \Rightarrow \text{talk}(x_3))$

While this pushes first-order unification beyond what it had been previously shown capable of, there are two disadvantages to this technique: (1) For every possible category that can be conjoined, a separate lexical entry for and is required, and (2) As the conjoinable categories become more complex, the and entries become correspondingly more complex and greatly obscure the theoretical background of the grammar formalism.

The fundamental problem in both cases is that the concept of free and bound occurrences of variables is not supported by Prolog, but instead needs to be implemented by additional programming. While theoretically possible, it becomes quite problematic to actually implement. The solution given in this paper is to use a higher-order logic programming language, $\lambda$Prolog, that already implements these concepts, called "abstract syntax" in (Miller, 1991) and "higher-order abstract syntax" in (Pfenning and Elliot, 1988). This allows a natural and elegant implementation of the grammatical theory, with only one lexical entry for and. This paper is meant to be viewed as furthering the exploration of the utility of higher-order logic programming for computational linguistics - see, for example, (Miller & Nadathur, 1986), (Pareschi, 1989), and (Pereira, 1990).

2 CCG

CCG is a grammatical formalism in which there is a one-to-one correspondence between the rules of composition at the level of syntax and logical form. Each word is (perhaps ambiguously) assigned a category and LF, and when the syntactical operations assign a new category to a constituent, the corresponding semantic operations produce a new LF for that constituent as well. The CCG rules shown in Figure 1 are implemented in the system described in this paper.

As an illustration of how the semantic rules can be simulated in first-order unification, consider the derivation of the constituent harry found, where harry has the category np with LF harry' and found is a transitive verb of category (s\np)/np with LF

$\lambda x.((\text{found'} x) \text{subject object})$

In the CCG formalism, the derivation is as follows: harry gets raised with the $> T$ rule, and then forward composed by the $> B$ rule with found, and the result is a category of type s/rip with LF $\lambda x.((\text{found'} harry' z))$. In section 3 it will be seen how the use of abstract syntax allows this to be expressed directly. In first-order unification, it is simulated as shown in Figure 2.

The final CCG rule to be considered is the coordination rule that specifies that only like categories can coordinate:

In the general sense, not specifically the CCG rule for function composition.

Footnotes:
1. In the general sense, not specifically the CCG rule for function composition.
2. The type-raising rules shown are actually a simplification of what has been implemented. In order to handle determiners, a system similar to NP-complement categories as discussed in (Dowty, 1988) is used. Although a worthwhile further demonstration of the use of abstract syntax, it has been left out of this paper for space reasons.
3. The \ for a backward-looking category should not be confused with the \ for $\lambda$-abstraction.
4. Example adapted from (Steedman, 1990, p. 220).
This is actually a schema for a family of rules, collectively called “generalized coordination”, since the semantic rule is different for each case. For example, if X is a unary function, then the semantic rule is (7a), and if the functions have two arguments, then the rule is (7b).

(7a) $\Phi FGH = \lambda x. F(Gx)(Hz)$

(7b) $\Phi^2 FGH = \lambda x. \lambda y. F(Gxy)(Hxy)$

For example, when processing (1a), rule (7a) would be used with:

- $F = \lambda x. \lambda y. (\text{and'} x y)$
- $G = \lambda P. (P \text{ john'})$
- $H = \lambda P. (P \text{ bill'})$

with the result

$$\phi FGH = \lambda x. (\text{and'} (x \text{ john'}) (x \text{ bill'}))$$

which is $\alpha$-equivalent to (2).

3 λPROLOG and Abstract Syntax

λProlog is a logic programming language based on higher-order hereditary Harrop formulae (Miller et al., 1991). It differs from Prolog in that first-order terms and unification are replaced with simply-typed λ-terms and higher-order unification, respectively. It also permits universal quantification and implication in the goals of clauses. The crucial aspect for this paper is that together these features permit the usage of abstract syntax to express the logical forms terms computed by CCG. The built-in λ-term manipulation is used as a “meta-language” in which the “object-language” of CCG logical forms is expressed, and variables in the object-language are mapped to variables in the meta-language.

The λProlog code fragment shown in Figure 3 declares how the CCG logical forms are represented. Each CCG LF is represented as an untyped λ-term, namely type $tm$. abs represents object-level abstraction $\lambda x. M$ by the meta-level expression $\text{(abs x)}$.

![Figure 3: Declarations for λProlog representation of CCG logical forms](image)

where $N$ is a meta-level function of type $tm \rightarrow tm$. A meta-level $\lambda$-abstraction $\lambda y. P$ is written $y \backslash P$. Thus, if $\text{walked'}$ has type $tm \rightarrow tm$, then $y \backslash (\text{walked'} y)$ is a λProlog (meta-level) function with type $tm \rightarrow tm$, and $(\text{abs y} \backslash (\text{walked'} y))$ is the object-level representation, with type $tm$. The LF for $\text{found}$ shown in (5) would be represented as $(\text{abs obj} \backslash (\text{abs sub} \backslash (\text{found'} \text{ sub obj})))$. app encodes application, and so in the derivation of $\text{harry found}$, the type-raised $\text{harry}$ has the λProlog value $(\text{abs p} \backslash (\text{app p} \text{ harry}'))$.

The second part of Figure 3 shows declares how quantifiers are represented, which are required since the sentences to be processed may have determiners. $\forall$ and $\exists$ are encoded similarly to abstraction, in that they take a functional argument and so object-level binding of variables by quantifiers is handled by meta-level $\lambda$-abstraction. $\rightarrow$ and $\&$ are simple constructors for implication and conjunction, to be used with $\forall$ and $\exists$ respectively, in the typical manner (Pereira and Shieber, 1987). For example, the sentence $\text{every man found a bone}$ has as a possible LF (8a), with the λProlog representation (8b):

---

5It is not established if this schema should actually produce an unbounded family of rules. See (Weir, 1988) and (Weir and Joshi, 1988) for a discussion of the implications for automata-theoretic power of generalized coordination and composition, and (Gazdar, 1988) for linguistic arguments that languages like Dutch may require this power, and (Steedman, 1990) for some further discussion of the issue. In this paper we use the generalized rule to illustrate the elegance of the representation, but it is an easy change to implement a bounded coordination rule.

6The $\Phi$ notation is used because of the combinator logic background of CCG. See (Steedman, 1990) for details.

7Defined as the unification of simply typed λ-terms, modulo $\beta\eta$ conversion.

---

215
type apply \( \text{tm} \rightarrow \text{tm} \rightarrow \text{tm} \rightarrow o \).

type compose \( \text{tm} \rightarrow \text{tm} \rightarrow \text{tm} \rightarrow o \).

type raise \( \text{tm} \rightarrow \text{tm} \rightarrow \text{tm} \rightarrow o \).

apply \((\text{abs } R) \ S \ (R \ S)\).

compose \((\text{abs } F) \ (\text{abs } G) \ (\text{abs } x)(F \ (G \ x))\).

raise \( T \ (\text{abs } P(\text{app } P \ Tm))\).

Figure 4: λProlog implementation of CCG logical form operations

\[(8a) \exists x.(\text{bone' } x) \land \forall y.(\text{man' } y) \rightarrow (\text{found' } y \ x))\]

(8b) (\exists x ((\text{bone' } x) \ & \n
\forall \ x1 ((\text{man' } x1) \ \Rightarrow (\text{found' } x1 \ x))))

Figure 4 illustrates how directly the CCG operations can be encoded. \( o \) is the type of a meta-level proposition, and so the intended usage of apply is to take three arguments of type \( \text{tm} \), where the first should be an object-level λ-abstraction, and set the third equal to the application of the first to the second. Thus, for the query

\?\,- apply \((\text{abs sub}(\text{walked' sub})) \ h a r r y' \ M\).

\( R \) unifies with the \( \text{tm} \rightarrow \text{tm} \) function \( \text{sub}(\text{walked' sub}) \), \( S \) with \( \text{harry' } \) and \( M \) with \((\text{R S})\), the meta-level application of \( R \) to \( S \), which by the built-in β-reduction is \((\text{walked' harry')}.\) In other words, object-level function application is handled simply by the meta-level function application.

Function composition is similar. Consider again the derivation of \( \text{harry found} \) by type-raising and forward composition. \( \text{harry} \) would get type-raised by the raise clause to produce \((\text{abs } p)(\text{app } p \ h a r r y')\), and then composed with \( \text{found} \), with the result shown in the following query:

\?\,- compose \((\text{abs } p)(\text{app } p \ h a r r y'))\)

\((\text{abs } \text{obj})\)

\((\text{abs } \text{sub})\)

\((\text{found' } \text{sub } \text{obj}))\)

\( N \).

\( M = (\text{abs } x)\)

\((\text{app})\)

\((\text{abs } \text{sub}(\text{found' } \text{sub } x))\)

\( \text{harry')}.\)

derivation fails to yield all available quantifier scopings. We do not address here the further question of how the remaining scoped readings are derived. Alternatives that appear compatible with the present approach are quantifier movement (Hobbs & Shieber, 1987), type-raising at LF (Partee & Rooth, 1983), or the use of disambiguated quantifiers in the derivation itself (Park, 1995).

\[11\]There are other clauses, not shown here, that determine the direction of the CCG rule. For either direction, however, the semantics are the same and both directional rules call these clauses for the semantic computation.

At this point a further β-reduction is needed. Note however this is not at all the same problem of writing a β-reducer in Prolog. Instead it is a simple matter of using the meta-level β-reduction to eliminate β-redexes to produce the final result \( (\text{abs } x)(\text{found' harry } x) \). We won’t show the complete declaration of the β-reducer, but the key clause is simply:

\text{red} \ (\text{app} \ (\text{abs } N) \ N) \ (N \ N).

Thus, using the abstract syntax capabilities of λProlog, we can have a direct implementation of the underlying linguistic formalism, in stark contrast to the first-order simulation shown in Figure 2.

4 Implementation of Coordination

A primary goal of abstract-syntax is to support recursion through abstractions with bound variables. This leads to the interpretation of a bound variable as a “scoped constant” - it acts like a constant that is not visible from the top of the term, but which becomes visible during the descent through the abstraction. See (Miller, 1991) for a discussion of how this may be used for evaluation of functional programs by “pushing” the evaluation through abstractions to reduce redexes that are not at the top-level. This technique is also used in the β-reducer briefly mentioned at the end of the previous section, and a similar technique will be used here to implement coordination by recursively descending through the two arguments to be coordinated.

Before describing the implementation of coordination, it is first necessary to mention how CCG categories are represented in the λProlog code. As shown in Figure 5, \text{cat} is declared to be a primitive type, and \text{np}, \text{s}, \text{conj}, \text{noun} are the categories used in this implementation. \text{fs} and \text{bs} are declared
Figure 6: Implementation of coordination

type coord
  cat -> tm -> tm -> tm -> o.

coord (fs A B) (abs R) (abs S) (abs T) :-
  pi x\ (coord B (R x) (S x) (T x)).

coord (bs A B) (abs R) (abs S) (abs T) :-
  pi x\ (coord B (R x) (S x) (T x)).

coord B & S (and' R S) :- atomic-type B.

The code for coordination is shown in Figure 6. The four arguments to coord are a category and three terms that are the object-level LF representations of constituents of that category. The last argument will result from the coordination of the second and third arguments. Consider again the earlier problematic example (1a) of coordination. Recall that after john is type-raised, its LF will be (abs p\(app p john')) and similarly for bill.

This will match with the first clause for coord, with

- A instantiated to (bs np s)
- B to s
- R to (p\(app p john'))
- S to (p\(app p bill'))

and T a logic variable waiting instantiation.

Then, after the meta-level β-reduction using the new scoped constant c, the following goal is called:

?- coord s (app c john') (app c bill') M.

where M = (T c). Since s is an atomic type, the third coord clause matches with

- B instantiated to s
- R to (app c john')
- S to (app c bill')
- M to (and' (app c john') (app c bill'))

Since M = (T c), higher-order unification is used by λProlog to instantiate T by extracting c from M with the result

\[ T = x\(and' (app x john') (app x bill')) \]

and so M from the original query is

\[ (abs x\(and' (app x john') (app x bill'))) \]

Note that since c is a scoped constant arising from the proof of an universal quantification, the instantiation

\[ T = x\(and' (app x john') (app x bill')) \]

is prohibited, along with the other extractions that do not remove c from the body of the abstraction.

This use of universal quantification to extract out c from a term containing c in this case gives the same result as a direct implementation of the rule for coordination of unary functions (7a) would. However, this same process of recursive descent via scoped constants will work for any member of the conj rule family. For example, the following query

?- coord
  (fs np (bs np s))
  (abs obj\(abs sub\(like' sub obj)))
  (abs obj\(abs sub\(hate' sub obj)))
  M.

M = (abs x\(and' (like' x1 x)
  (hate' x1 x)))).

corresponds to rule (7b). Note also that the use of the same bound variable names obj and sub causes no difficulty since the use of scoped-constants, meta-level β-reduction, and higher-order unification is used to access and manipulate the inner terms. Also, whereas (Park, 1992) requires careful consideration of handling of determiners with coordination, here such sentences are handled just like any others. For example, the sentence Mary gave every dog a bone and some policeman a flower results in the LF

\[ 12 \]

\[ )\]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]

\[ 12 \]
Thus, "generalized coordination", instead of being a family of separate rules, can be expressed as a single rule on recursive descent through logical forms. (Steedman, 1990) also discusses "generalized composition", and it may well be that a similar implementation is possible for that family of rules as well.

5 Conclusion

We have shown how higher-order logic programming can be used to elegantly implement the semantic theory of CCG, including the previously difficult case of its handling of coordination constructs. The techniques used here should allow similar advantages for a variety of such theories.

An argument can be made that the approach taken here relies on a formalism that entails implementation issues that are more difficult than for the other solutions and inherently not as efficient. However, the implementation issues, although more complex, are also well-understood and it can be expected that future work will bring further improvements. For example, it is a straightforward matter to transform the AProlog code into a logic called L₁ (Miller, 1990) which requires only a restricted form of unification that is decidable in linear time and space. Also, the declarative nature of AProlog programs opens up the possibility for applications of program transformations such as partial evaluation.

6 Acknowledgments

This work is supported by ARO grant DAAL03-89-0031, DARPA grant N00014-90-J-1863, and ARO grant DAIIH04-94-G-0426. I would like to thank Aravind Joshi, Dale Miller, Jong Park, and Mark Steedman for valuable discussions and comments on earlier drafts.

References

David Dowty. 1988. Type raising, functional composition, and non-constituent conjunction. In Richard T. Oehrle, Emmon Bach, and Deirdre Wheeler, editors, Categorial Grammars and Natural Language Structures. Reidel, Dordrecht, pages 153-198.

Jerry R. Hobbs and Stuart M. Shieber. 1987. An algorithm for generating quantifier scopings. Computational Linguistics, 13:47-63.

Einar Jownes. 1990. Constraining Montague Grammar for Computational Applications. PhD thesis, University of Edinburgh.

Dale Miller. 1990. A logic programming language with lambda abstraction, function variables and simple unification. In P. Schroeder-Heister, editor, Extensions of Logic Programming, Lecture Notes in Artificial Intelligence, Springer-Verlag, 1990.

Dale Miller. 1991. Abstract syntax and logic programming. In Proceedings of the Second Russian Conference on Logic Programming, September 1991.

Dale Miller and Gopalan Nadathur. 1986. Some uses of higher-order logic in computational linguistics. In 24th Annual Meeting of the Association for Computational Linguistics, pages 247-255.

Dale Miller, Gopalan Nadathur, Frank Pfening, Andre Scedrov. 1991. Uniform proofs as a foundation for logic programming. In Annals of Pure and Applied Logic, 51:125-157.

Robert C. Moore. 1989. Unification-based semantic interpretation. In 27th Annual Meeting of the Association for Computational Linguistics, pages 33-41.

Remo Pareschi. 1989. Type-Driven Natural Language Analysis. PhD thesis, University of Edinburgh.

Jong C. Park. 1992. A unification-based semantic interpretation for coordinate constructs. In 30th Annual Meeting of the Association for Computational Linguistics, pages 209-215.

Jong C. Park. 1995. Quantifier scope and constituency. In 33rd Annual Meeting of the Association for Computational Linguistics (this volume).

Barbara Partee and Mats Rooth. 1983. Generalized conjunction and type ambiguity. In Rainer Bauerle, Christoph Schwarze, and Arnim von Stechow, editors, Meaning, Use, and Interpretation of Language. W. de Gruyter, Berlin, pages 361-383.

Fernando C.N. Pereira. 1990. Semantic interpretation as higher-order deduction. In Jan van Eijck, editor, Logics in AI: European Workshop JELIA'90, Lecture Notes in Artificial Intelligence number 478, pages 78-96. Springer-Verlag, Berlin, Germany.

Fernando C.N. Pereira and Stuart M. Shieber. 1987. Prolog and Natural-Language Analysis. Number 10 in CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, California,
Frank Pfenning and Conal Elliot. 1988. Higher-order abstract syntax. In Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation, 1988.

Mark J. Steedman. 1990. Gapping as constituent coordination. In Linguistics and Philosophy 13, pages 207–263.

David Weir. 1988. Characterizing Mildly Context-sensitive Grammar Formalism. CIS-88-74, PhD thesis, University of Pennsylvania.

David Weir and Aravind Joshi. 1988. Combinatory categorial grammars: generative power and relation to linear CF rewriting systems. In 26th Annual Meeting of the Association for Computational Linguistics, pages 278–285.