INFERENCE OF SEMANTICS FROM EXAMPLES:

FROM CONTEXT-FREE TO DEFINITE-CLAUSE GRAMMARS

by

JUERGEN HAAS

A dissertation submitted to the
Faculty of the Graduate School of
State University of New York at Buffalo
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

September 1993

Acknowledgments

I thank my thesis advisor Professor Bharat Jayaraman for giving me the opportunity to pursue this interesting and exciting project. His suggestions and guidance helped to keep me on the right track and made the development of this dissertation a productive and enjoyable experience.

I am grateful to Professor Stuart C. Shapiro and Professor William J. Rapaport for the excellent education I received through them during my years at SUNY Buffalo and for their comments on my dissertation research.

I also thank Dr. Fernando C. N. Pereira and Dr. Dale A. Miller for interesting discussions and valuable comments.

This research was supported in part by grant CCR 9004357 from the National Science Foundation.

Abstract

This dissertation considers the problem of synthesizing mappings from syntactic structures to meaning representations given a grammar and sample sentence-meaning pairs. The motivation for this work stems from the potential use of such techniques in the development of natural-language front-ends, e.g., natural query languages. The central technical problem addressed in the dissertation is the mechanical transformation of a *context-free grammar* (CFG) into a *definite clause grammar* (DCG) using sample sentence-meaning pairs of the form $\langle s,\ m \rangle$, where s is a sentence belonging to the language defined by CFG and m is the semantic representation (meaning) of s. The resulting DCG would be such that it could be executed, by the interpreter of a logic programming language, to compute the semantic representation(s) for every sentence of the original CFG. Two important assumptions underlie the proposed approach: (i) the semantic representation language is the *simply typed lambda-calculus*, and (ii) the semantic representation of a sentence can be obtained from the semantic representations of its parts (*compositionality*).

The basic technique involves an enumeration of a representative finite set of sentences and formation of a corresponding set of equations over (typed) function variables. Each function variable represents the meaning of a particular grammar rule, and effectively serves to augment the original CFG in order to derive a higher-order DCG. The main research topics investigated in this disseration are: (i) formulation of a solution technique using a variant of Huet's unification procedure for the simply-typed lambda-calculus; (ii) efficient implementation of the solution using constraints from multiple examples, "macro" substitution rules that package common sequences of more basic substitutions, and a dependency-directed backtracking search; (iii) development of a provably-correct *partial execution* procedure to convert the constructed higher-order DCG into a first-order DCG, for more efficient execution; and (iv) the application of the entire methodology for developing a natural query language—a variant of the CHAT-80 query language—starting from a grammar and sample sentence-meaning pairs.

Contents

1. Introduction

1.1. The Problem and Its Significance

The two definitive properties of a *language* are its *syntax* and *semantics*. At the outset, I should clarify that I use the term language in a broad sense, and include the kinds of formal languages that one encounters in automata theory, conventional programming language constructs, and certain restricted subsets of natural languages. The term syntax refers to the structure of well-formed sentences of a language, whereas the term semantics refers to the meaning of well-formed sentences. The syntax of the kinds of languages considered in this dissertation is fairly well-understood, that is, it is relatively straightforward to define the set of well-formed sentences of such languages using context-free grammars[1] or Backus-Naur Form (BNF). However, the formal semantics of a language is harder to specify, and numerous approaches have been proposed in the literature, e.g., denotational, axiomatic, operational, etc. (Gordon 1988).

This dissertation is concerned with the problem of inferring semantics of a language from examples, assuming that we are already given its syntax. More precisely, I assume that the syntax is given using an *unambiguous context-free grammar*, although the proposed techniques also apply to certain attribute grammars, where the attributes specify context-sensitive features such as number or gender agreement, and can be extended to apply to ambiguous grammars. There are several semantic representation languages, *first-order logic* (FOL), *λ-calculus*, *semantic networks* (Brachman 1979), etc. (For brevity, I will use the term 'semantics' and 'semantic representations' as synonyms throughout this dissertation.) Given an unambiguous grammar, the problem of inferring its meaning from examples is one of finding the mapping (i.e., function) from sentences to their meanings on the basis of sample sentence-meaning pairs. More specifically, the goal of this research is to develop a system that takes as input an unambiguous *context-free grammar* (CFG) and a finite set of pairs $\langle s,\ m \rangle$, where s is a sentence belonging to the language defined by the context-free grammar and m is the semantics

[1] Context-free grammars are clearly insufficient to specify the complete syntax of a natural language or even a programming language. I therefore limit attention to restricted subsets of these languages. See section 1.3 for further discussion.

of s, and will produce as output a *definite clause grammar* (DCG) (Pereira and Warren 1980) that will compute the semantics of every sentence of the input grammar. A DCG is essentially a CFG wherein each nonterminal symbol has been enhanced with a parameter that provides semantic information. A DCG can be directly converted into a logic program which can then be executed to perform parsing or generation. This dissertation shows that in many practical cases such a mapping can be automatically inferred from a representative finite set of examples. The next subsection will clarify the precise sense in which this problem can be solved and the research issues it raises, but first I will discuss the significance of this problem:

(1) Why is it desirable to automatically generate a DCG from a CFG?

(2) What are the applications of such a system?

It is not easy to *manually* augment a CFG with semantic constructors to obtain a DCG because the task of building a correct and efficient DCG requires a fair amount of search, the process being tedious and error-prone. Even for the small grammars considered in later chapters of this dissertation, it is not obvious what the semantic constructors should be. However, it is easy to give sample sentence-meaning pairs, and often the semantic representation of a sentence is systematically composed from those of the phrases that constitute the sentence. Therefore it natural to seek a mechanical procedure that will compute (induce) the semantics of all sentences of a given CFG on the basis of a representative set of sentence-meaning pairs.

Another motivation for a *mechanical* transformation procedure from a CFG to a DCG is to accommodate changes quickly and correctly. For example, in the context of natural query languages, it will be a fairly common task to change an interface to another language or another dialect or jargon, but to keep the semantic representations the same. In general this requires a complete redesign of the grammar, a task that would be easily accomplished by the proposed system, whereas conventional methods would require a substantial amount of programming effort, the result of which will likely be error-prone and not as general.

The proposed system would facilitate *rapid prototyping* of natural language interfaces for database systems or customizing such interfaces for specific applications (Velardi 1989, Wallace 1984), since the interface could be obtained merely by defining the grammar and typical sentence-meaning pairs. Both the conversion of the natural language query into this representation and the conversion from this representation back

into natural language would be handled by the generated interface—the latter operation would be achieved by applying the definite-clause grammar in the reverse direction to the semantic representations. Reversible execution of DCGs is possible because they are essentially logic programs. Since the DCGs considered in this dissertation are mechanically generated, they have a more restricted form compared with arbitrary logic programs, or for that matter arbitrary DCGs. This in turn makes them more amenable to reversible execution than for arbitrary logic programs or DCGs.

The techniques developed in this dissertation may provide a new approach to machine learning and program synthesis from examples. Similar techniques have recently been explored by Hagiya (1990). For example, program synthesis from examples is related to our stated problem in the following way: the CFG is analogous to a program schema; the resulting DCG is analogous to the program to be synthesized; and, the sample sentence-meaning pairs are analogous to the sample input-output pairs of the program to be synthesized. However, program synthesis is the harder of the two problems because it also involves the determination of the right schema. This topic is discussed further in chapter 2.

1.2. Approach and Technical Results

An *arbitrary* transformation (i.e., an arbitrary infinite mapping) cannot be inferred from finitely many examples, and hence it is necessary to impose additional constraints on our problem. We make the following two assumptions in order to facilitate the mechanical transformation of a CFG to a DCG: (i) the semantic representation language is the *simply typed λ-calculus* (Church 1940); (ii) the semantic representation of a sentence is systematically constructed from those of its phrases (*compositionality*). These assumptions are not unusual, since such assumptions have been adopted, for example, by R. Montague for a proper treatment of quantification of English (Montague 1974, Dowty et al. 1981). To illustrate, consider the following CFG rule,

$$sentence \rightarrow nounphrase,\ verbphrase$$

which specifies that a sentence is composed of a noun phrase followed by a verb phrase (*sentence*, *nounphrase* and *verbphrase* are nonterminals). A key idea of my approach is to exploit the compositionality principle to enhance the rules as follows:

$$sentence((F\ X\ Y)) \rightarrow nounphrase(X),\ verbphrase(Y)$$

where uppercase letters are variables. That is, if variables X and Y represent respectively the meanings of the nonterminals *nounphrase* and *verbphrase*, then the meaning of

nonterminal *sentence* is obtained by applying some function F to X and Y. The function variable F is a term in the simply typed λ-calculus, and must be determined by the system based upon the finite set of input examples. The semantic representations of the terminal symbols of the CFG are the other unknowns to be determined by the system.

The compositionality principle effectively means that the grammatical structure constrains the allowable semantics. The choice of the simply-typed λ-calculus as the semantic representation language drastically reduces the search space of allowable solutions, as we shall later see. Under these two assumptions, it can be seen that the stated problem is recursively enumerable in that, if there exists a DCG satisfying the finitely many examples, it is possible to systematically find it; if there is no solution, the search may sometimes be nonterminating. The typed λ-calculus is particularly suitable for analyzing and synthesizing semantic representations. It effectively allows us to reduce the generalization problem to a unification problem over simply-typed terms. This unification problem is called *higher-order unification* because variables may range over functions. A semi-decidable solution procedure for this problem was first described by Huet (1975).

Briefly, my technique is to enumerate sentences in a certain order, query the user for the semantic representation of each of the generated sentences, formulate a set of equations over the unknown function variables, and solve these equations using a variant of Huet's unification procedure. The solutions for these function variables serve to augment the original CFG in order to derive the final DCG.

Referring to the grammar rule given above, if the solution for F was $\lambda A.\lambda B.(B\ A)$, then the grammar rule would become

$$sentence((\lambda A.\lambda B.(B\ A)\ X\ Y)) \rightarrow nounphrase(X),\ verbphrase(Y)$$

which is equivalent to

$$sentence((Y\ X)) \rightarrow nounphrase(X),\ verbphrase(Y).$$

It turns out that Huet's procedure cannot be directly used to solve the kinds of equations that arise in our context. The reason is that this procedure requires that the types for all terms are known in advance; however, in the synthesis scheme, in general only some of the types are known when the equations are set up. It therefore becomes necessary to augment his procedure with two important operations—*type inference* and *type enumeration*.

1.2.1. Efficient Synthesis

In order to achieve acceptable performance for realistic grammars, the search has to make effective use of the constraints from multiple examples. Thus the proposed system marks an interesting point of departure from λProlog (Nadathur and Miller 1988) both with respect to the unification procedure as well as the search regime. (This is the reason that the synthesis system was implemented from scratch rather than on top of λProlog.) Another difference from λProlog arises from the fact that the right-hand sides of all equations generated from the examples are ground (i.e., do not contain any free variables). Thus higher-order unification reduces to *higher-order matching* in this context. As the decidability of general higher-order matching still remains an open problem, in this dissertation I will adapt the (semi-decidable) *unification* procedure of Huet, exploiting where possible the fact that all right-hand sides are ground. To speed up the synthesis of common types of substitution terms, I also explore the use of "macro" substitution rules, which are certain combinations of Huet's substitution rules without any free variables. In this way the kinds of substitutions needed in the context of DCG synthesis can be enumerated more efficiently. The approach of simultaneously solving a set of higher-order equations also facilitates an effective scheme of dependency directed backtracking. If a substitution causes failure in a particular equation, backtracking can be restricted to substitutions that have influenced that equation.

Unlike first-order unification, the unification of simply-typed λ-terms can yield more than more one solution. However, these solutions do not necessarily result in DCGs that implement different sentence-meaning functions. But if the problem is underconstrained by providing too few examples, the resulting DCGs need not be equivalent. If more examples are provided than necessary, there may be no solution at all if the examples are inconsistent, or unnecessary computations may be performed when solving the equations.[2] Therefore one should use as few examples as are necessary to guarantee a unique solution (sentence-meaning function). I derived a set of criteria for determining whether a set of examples has this property. These criteria ensure that the grammar rules are exposed to as many variations of sentences as are necessary to enforce maximally general semantic rules. An important technique in this context is to change one word of a sentence at a time, so that it can be uniquely determined which words contribute

[2]This problem is analogous to the linear algebra problem of determining a plane by specifying a set of points in space; at least three points are necessary to fix a (two-dimensional) plane. Specifying less than three points allows many different planes, whereas specifying more than three points can make a solution impossible.

which subterms of the semantic representation. Further performance improvements can be achieved by presenting shorter training instances before longer ones. The equations corresponding to shorter training instances are easier to solve, and the constraints introduced by them reduce the search for substitutions of subsequent equations.

1.2.2. Efficient Execution

While higher-order logic is useful for reasoning about and synthesizing programs, it is not very amenable to efficient execution. To achieve acceptable performance for larger grammars, the constructed higher-order DCG should be converted into a first-order DCG where possible. A first-order DCG is also more amenable to reversible execution than a higher-order DCG. I have investigated a technique called *partial execution*, which effectively replaces λ-terms by first-order terms, and therefore replaces higher-order unification by (the more efficient) first-order unification. The use of first-order unification to simulate certain cases of β-reduction was first introduced by Colmerauer (1978), and the connection between partial execution of predicates and Colmerauer's method for doing semantic interpretation in a logic grammar was made explicit by Pereira & Shieber (1987). I have developed a specialized version of partial execution that automatically converts a higher-order DCG into a first-order DCG guided by the set of examples that were used to derive the higher-order DCG.

The partially executed DCG works in the forward direction (i.e., computing the semantic representation of a sentence) by using first-order matching instead of β-reduction, and works in the reverse direction (i.e., computing the sentence(s) for a given semantic representation) by using first-order unification instead of higher-order unification. A simple form of partial execution is possible for the class of DCGs where all application terms are reduced during execution and the bodies of semantic terms do not have multiple occurrences of variables. If these assumptions do not hold, tracing the execution of the training instances can be used to determine which application terms should be partially executed. Copy operations may be necessary if a variable occurs more than once in a semantic term.

Even though one can construct pathological grammars and semantic representations where this scheme of partial execution fails, it appears to be applicable for most practical applications, and I have shown its correctness in those cases. Higher-order DCGs for which I could not find a satisfactory solution are those where a particular application term is reduced for some sentences but not for others. However, usually such grammars

can be easily rewritten into a more natural form that avoids this problem. For cases where the partially executed DCG fails to compute the correct answer in the reverse direction, I have developed a simple enhancement that will restore correctness.

Efficient forward execution of DCGs has received considerable attention in the literature (e.g., see Matsumoto (1983) and references therein), but techniques for efficient reverse execution are also important. It turns out that a partially executed DCG is particularly well-suited for reverse execution, since first-order unification is more efficient than higher-order unification. I have implemented an interpreter for reverse execution of a partially executed DCG that uses a selection strategy that ensures that at each step constraints from the semantic representation are utilized, so that nondeterminism is minimized.

1.2.3. Application to Natural Query Languages

In order to demonstrate the viability of my approach for larger grammars, I applied my techniques to synthesize a variant of the CHAT-80 natural query language (Warren and Pereira 1980). The grammar defining the syntax of queries contained about 90 rules (excluding terminals). This exercise helped develop a methodology for generating large DCGs: I have found that the best way to synthesize a large DCG is to group the given syntactic rules into independent modules that can be "trained" individually, and to incrementally add new rules. Such an approach also ensures that the number of variables to be solved for at each incremental step is small, by taking advantage of the results from previous steps. This in turn helps keep the search space for solutions small. The implemented system also permits the semantics for grammar rules and terminals to be optionally specified along with the syntactic rules, if they are known. This incremental approach is also useful if the syntactic rules need to be modified to maintain compositionality. My schemes for partial execution and reverse execution of DCGs have proved to be effective for all test cases the system was applied to.

1.3. Scope and Outline of the Dissertation

The objective of this research differs from those of Berwick (1985), Ishizaka (1990) and others, who are concerned with inferring a grammar (syntax) from example sentences. Instead, given the grammar, my objective is to infer the semantics of sentences from examples. Natural languages are of interest in my work since they are good examples of languages whose semantics require the use quantified terms and hence the full use

of the typed λ-calculus. However, my work is not directly concerned with devising suitable semantics for natural languages; it is the user's responsibility to construct both the grammar as well as the semantic representation for typical sentences in the typed λ-calculus.

This dissertation is concerned with that subset of natural languages that can be adequately described with CFGs and the typed λ-calculus. For applications such as natural query languages, it seems feasible to describe the language with a context-free grammar and also to insist on sentences with unambiguous meanings. However, the techniques apply equally to CFGs that have been extended with additional arguments to control rule application, which makes them effectively context sensitive. It appears that issues related to pronoun resolution can be separated from those related to generalization of semantics. Therefore I restrict my attention to languages without pronouns or anaphora. In this dissertation I am not concerned with the issues related to resolution of ambiguities, but the present system can be extended to handle certain types of syntactic and semantic ambiguity. Suggestions on how that might be done are given in chapter 9.

The remainder of this dissertation is organized as follows: Chapter 2 surveys related research; chapters 3 and 4 review the typed λ-calculus and definite-clause grammars respectively (these chapters are provided in order to make the dissertation self-contained); chapter 5 describes the basic techniques underlying the synthesis of a higher-order DCG; chapter 6 describes how to improve the efficiency of the synthesis procedure; chapter 7 describes how to convert the higher-order DCG into a first-order DCG by partial execution, and shows the correctness of the partially executed DCG under appropriate conditions; chapter 8 describes the application of the foregoing techniques to the synthesis of natural query languages; and chapter 9 presents conclusions, contributions, possible extensions, and areas of further work.

2. Related Research

I am not aware of any published research that meets the stated objectives of the previous chapter, but there are several closely related research areas. I briefly survey these research areas below and mention their relationship to my work.

2.1. Automatic Synthesis of Semantics

In order to partially automate the synthesis of semantics, various formalisms have been developed within the logic grammar framework, e.g., *modifier structure grammars*, *restriction grammars*, *discontinuous grammars*, and *puzzle grammars* (Abramson and Dahl 1989). These grammars provide the user with means for specifying guidelines which a system could consult in order to construct the final representation. That is, the user specifies the desired type of semantic representation in some high-level language, and the system then translates these specifications into constructors which can be incorporated into the executable grammar rules. It appears that the only types of semantic representations that could be completely automated with these approaches are parse trees, since their representations follow exactly the history of rule application.

Hauptmann (1991) discusses the automatic acquisition of semantic interpretation rules which convert syntactic tree-structures (the output of an ATN syntactic parser) into tree structured frame representations (of the KL-ONE knowledge representation system). The basic ideas behind his approach are similar to those discussed in this dissertation, but his generalization and induction processes are much more heuristic and *ad hoc* compared with my proposed approach based upon higher-order unification. As a result, his system is more restrictive and probably harder to generalize beyond the specific types of transformations discussed in his thesis. For example, his approach assumes that the *lexical mapping rules* are already known; i.e., the semantic representations of all the individual words must be given to the system, whereas the system discussed in this dissertation can infer them from the examples. He uses various heuristics, for example, specific mapping rules are generalized by allowing a certain substitution because "all other critical parts of the rule are identical and the embedded concept that was substituted is sufficiently similar to the original one, based on a definition of similarity which exploits the frame hierarchy." His system also does not appear to be reversible.

The reason why no heuristics are needed in the system discussed in this dissertation is because the constraints are collected in a set of higher-order equations to be solved simultaneously, so that once a solution has been found there are no other constraints that could invalidate the solution. Whereas when one tries to generalize locally without taking all the other restrictions into account, similarity heuristics are needed to control the search.

2.2. Program Synthesis by Examples

A definite clause grammar can be viewed as a program that takes as input a sentence and computes its semantic representation (see section 4 for details). Therefore the augmentation process discussed in this dissertation can be considered a type of automatic program synthesis from examples (Summers 1977, Bauer 1979, Kodratoff 1979, Biermann *et al.* 1984). Programming by examples, on the other hand, is a special case of *inductive inference*, since the synthesis of a program generally involves the inference of an extended pattern of program behavior from the patterns discovered in the examples—computability theory has shown that it is possible to infer large useful classes of programs simply from examples of input/output behavior (Gold 1967, Blum 1975, Barzdin 1977). A common way to deal with the search problem in automatic program synthesis from examples is to use *program schemata* to constrain the way in which the control structures and data operators of the chosen programming language are used (Smith 1984). In this dissertation, the parsing grammar provided to the system can be considered such a program schema.

Until recently, research in program synthesis from examples has not considered the kind of input/output pairs with quantified terms and types. Hagiya (1991) extends the simply typed λ-calculus with inductive definitions, providing a formalism for solving both *inductive learning* ("programming by example") as well as *deductive learning* ("proving by example") problems. Both types of problem are formulated as equations in a typed λ-calculus. However, in order to avoid combinatorial explosion when solving such equations, appropriate program or recursion schemata must be provided. These schemata can also be formulated in the calculus introduced by Hagiya and could in principle be inferred through higher-order unification.

2.3. Machine Learning

The use of the typed λ-calculus and higher-order unification can be thought of as carrying out inductive learning (programming by examples). There is a strong similarity to deductive learning (proving by example) as well: if the semantics for the terminal symbols of the grammar are given, the correct substitutions for the remaining function variables can be considered a proof that a particular sentence has a particular semantic representation as specified by an example. Generalization takes place in that once a function substitution has been determined by a set of examples, the corresponding semantics is determined for *all* sentences that use the rule with that substitution during parsing. The problem of inferring these substitutions is made tractable in our case by the fact that syntactic structure is predefined, thus providing an appropriate program schema.

Proving by examples is known more commonly as *explanation-based learning* (EBL) (Shavlik 1990, Hagiya 1991). EBL has been investigated mainly in the area of theorem proving, although the same mechanism underlies much of the work in other fields such as skill acquisition and automatic programming. EBL can be considered as the generalization of a given proof, so that theorems that are "similar" to the one derived by that proof can be derived more efficiently. The initial proof that is used to guide the generalization in EBL corresponds to *semitraces* in the area of programming by examples (Smith 1984), and the general proof in EBL corresponds to the synthesized program. In our work, the analysis of a particular sentence and its representation corresponds to the initial proof and is used to guide the augmentation of the grammar.

Determining the association between terminal symbols (words) and their semantic representations using *anti-unification* can be considered a generalization of learning concepts from examples. Anti-unification is essentially the dual of unification; that is, U_A is an "anti-unifier" of two terms t_1 and t_2, if U_A can be unified with t_1 and with t_2. Instead of using an expressive description language to formulate possible generalizations, as in the version-space approach (Mitchell 1978), anti-unification generalizes only by turning constants or terms into variables, thus facilitating efficient implementation. Anti-unification is implicitly handled in my system by the higher-order unification procedure. By successively considering example sentences whose sentences differ in only one word from the sentence of the "main" example sentence, my system effectively incorporates the powerful concept of *near misses* (Winston 1975) in its generalization process.

2.4. Natural Language Learning

Lehnert (1987) discusses a system that uses limited syntactic knowledge expressed in a chart parser and relevant conceptual case frame representations as the basis to learn semantic representations of natural language sentences from examples. The system is illustrated by using an "approximation" of conceptual dependency as sample representation (Schank 1975). In order to determine the associations between the words of the sentence and the fragments of the semantic representation, Lehnert's system uses *lexical matches*; i.e., the word has to appear explicitly in the representation, otherwise the association has to be provided by the user in the form of a *conceptual definition*. The semantic representations that can be learned by Lehnert's system are also restricted by the fact that they must be non-recursive case-frame representations.

Other research projects in the area of natural language learning, e.g., (Anderson 1981) and (Selfridge 1986), combine the acquisition of syntactic and semantic knowledge. A major objective of those projects is to explain the characteristics of human language learning. Anderson (1977) discusses a system that infers augmented transition networks given pairs of sentences and structures representing their meanings. The type of meaning representation used by Anderson's system is a propositional semantic network. The augmented transition network inferred by the system can be used for both converting sentences into their semantic representations and vice versa; however, two separate interpreters are required for these two modes of operation, whereas in the case of definite clause grammars only one is needed due to the reversibility property. Overall, Anderson takes a very heuristic approach to language learning, in contrast to the systematic techniques discussed in this dissertation.

Selfridge (1986) discusses a program that acquires word meanings and language structure from examples of sentences and their corresponding meaning representations using conceptual dependency (Schank 1973). It is an attempt to model the development of language comprehension in a child. Word meanings are learned either by providing the association between a particular word and its meaning representation to the program directly, or by providing pairs of whole phrases and their meaning representations and then factoring out the parts already known to the program and associating the unknown parts of the phrases with the unknown parts of the representations.

The approach discussed in this dissertation is essentially a generalization of the second method (using complete sentences). By systematically varying sentences the word meanings can be inferred with maximum efficiency and precision, whereas Selfridge's program may temporarily undergeneralize or make wrong associations.

Since the sentences handled by Selfridge's program are quite simple and the vocabulary very limited, no formalism like ATN's or DCG's is required. It can learn how to fill the slots of the representations of certain actions using various heuristics, but does not handle more complex sentences involving quantification or recursive grammar structures, whereas the DCG formalism in conjunction with higher-order unification can handle such applications. Selfridge's program uses a separate procedure for language generation. It converts a meaning representation into a phrase (not necessarily a complete sentence) in a heuristic fashion, whereas the DCG's constructed by the proposed system are suitable for both efficient parsing and efficient generation.

Selfridge's system is restricted to a fixed set of frames of the conceptual dependency knowledge representation system, whereas the the proposed system can handle any consistent representation expressible in the typed λ-calculus.

3. Typed λ-Calculus: Equality and Unification

This chapter gives an overview of the *simply typed λ-calculus* and discusses two important operations, *reduction* and *unification*, following the notation and terminology from Huet (1975). Reduction of λ-terms is fairly well known, but (higher-order) unification is not as well understood. This chapter presents the unification procedure along with examples.

3.1. Typed λ-terms

Types

The typed λ-calculus is based on Church's simple theory of types (Church 1940). Each well-formed expression (term) of this language has an unambiguous *type* that indicates its position in a functional hierarchy.

Definition 3.1: Assuming T_0 is a finite set of *elementary* types (also called *primitive* types), the set T of types is defined as the smallest superset of T_0 closed under the binary operator '$\rightarrow$':

$$\alpha, \beta \in T \Rightarrow (\alpha \rightarrow \beta) \in T.$$

If A is a set of elements of type α, and B a set of elements of type β, then $\alpha \rightarrow \beta$ denotes the type of functions with domain A and range B. Types are designated by the Greek letters α, β, γ, etc. A colon is used to indicate the type of a term; e.g., $t : \alpha$ means that the term t has type α.

λ-Terms

There are basically four kinds of terms in the typed λ-calculus: *variables*, *constants*, *abstractions*, and *applications*. Variables and constants are also referred to as *atoms*. For every $\alpha \in T$ there is a denumerable set $\mathcal{V}_\alpha$ of variables of type α. The elements of the set $\mathcal{C}$ of constants have arbitrary given types. All sets $\mathcal{V}_\alpha$ and $\mathcal{C}$ are pairwise disjoint.

Definition 3.2: The set $\mathcal{A}$ of *atoms* is defined as:

$\mathcal{A} = \mathcal{C} \cup \mathcal{V}$, where $\mathcal{V} = \bigcup_{\alpha \in T} \mathcal{V}_\alpha$.

In this section, variables are written in lowercase letters x, y, ..., f, g, ..., constants are written in capitals A, B, ..., F, G, ..., and atoms written using the symbols @, @′, ...

Definition 3.3: If e_1 is a term of type $(\alpha \to \beta)$, and e_2 a term of type α, then the term $(e_1\ e_2)$ is an *application* of type β.

(The outermost parentheses of a term are often omitted; i.e., $(e_1\ e_2)$ is the same as $e_1\ e_2$. The application operator is left-associative; i.e., $((e_1\ e_2)\ e_3)$ is the same as $e_1\ e_2\ e_3$.)

Definition 3.4: If e is a term of type β, and $x \in \mathcal{V}_\alpha$, then the term $\lambda x.e$ is an *abstraction* of type $\alpha \to \beta$. The variable x is called a *binder variable*, or *prefix variable* of the term e.

Therefore, the set of terms is defined as the smallest superset of $\mathcal{A}$ closed by application and abstraction. Terms are denoted by e, e', ..., E, E', ..., which may have subscripts. The type of a term e is denoted by $\tau(e)$.

Definition 3.5: The relation *subterm of* is defined as the reflexive and transitive closure of:

$$\begin{cases} e_1 \text{ and } e_2 \text{ are subterms of } (e_1\ e_2), \\ e \text{ is a subterm of } \lambda x.e\ . \end{cases}$$

$\mathcal{E}[e]$ denotes a term that has a subterm e, and $\mathcal{E}[e']$ denotes the term obtained by replacing all occurrences of e by e' in $\mathcal{E}$ (if $\tau(e') = \tau(e)$).

Definition 3.6: Let $E = \mathcal{E}[\lambda x.e]$. An occurrence of x in $\lambda x.e$ is *bound in* E.

Definition 3.7: Let $E = \mathcal{E}[\lambda x.e]$. A non-bound occurrence of x in E is *free in* E.

$\mathcal{F}(E)$ denotes the set of variables having a free occurrence in E.

Assuming $\tau(e) = \tau(x)$, $\mathcal{S}^x_e(E)$ denotes the term obtained by substituting e for every free occurrence of x in E, taking care to rename variables of E as necessary to avoid "capture" of free (occurrences of) variables in e by prefix variables of E.

3.2. Equality between λ-terms

The following three conversion rules define equality between λ-terms.

Definition 3.8: α-*conversion*: $\mathcal{E}[\lambda y.\mathcal{S}_y^x(e)] = \mathcal{E}[\lambda x.e]$, for any $y \notin \mathcal{F}(e)$ such that $\tau(y) = \tau(x)$, assuming that the same subterm position is being referred to on both sides of the equality.

Definition 3.9: β-*conversion*: $\mathcal{E}[\mathcal{S}_e^x(e')] = \mathcal{E}[(\lambda x.e'\ e)]$ assuming that the same subterm position is being referred to on both sides of the equality.

Definition 3.10: η-conversion: $\mathcal{E}[\lambda x.(e\ x)] = \mathcal{E}[e]$, where $x \notin \mathcal{F}(e)$, assuming that the same subterm position is being referred to on both sides of the equality.

Definition 3.11: λ-*conversion* is the reflexive, symmetric, and transitive closure of α-, β-, and η-conversion.

Informally, α-conversion is simply variable renaming. That is, two terms are equivalent if they can be made identical by appropriately renaming variables. β-conversion can be used to simplify an application term $((\lambda V.E_1)\ E_2)$ by removing the left-most prefix variable V of the function, and the right-most argument E_2, and at the same time replacing all free occurrences of V in E_1 with E_2. This use of β-conversion is referred to as β-*reduction*. An abstraction $\lambda V.(E\ V)$ can be simplified through η-conversion by removing the left-most prefix variable and the right-most argument of the function E if they are identical and if V has no free occurrences in E.

Example 3.2.1:

Examples of α-conversion:

$\lambda x.x = \lambda y.y$

$\lambda x.\lambda y.((x\ y)\ a) = \lambda x.\lambda z.((x\ z)\ a)$

Example 3.2.2:

Examples of β-conversion (assuming $\tau(x) = \tau(a)$ and $\tau(y) = \tau(b)$):

$(\lambda x.\lambda y.(f\ (g\ x\ y))\ a) = \lambda y.(f\ (g\ a\ y))$

$(\lambda x.\lambda y.(f\ (g\ x\ y))\ a\ b) = (f\ (g\ a\ b))$

Note that $(g\ x\ y)$ is equivalent to $((g\ x)\ y)$.

Example 3.2.3:

Examples of η-conversion:

$\lambda y.((g\ a\ b)\ y) = (g\ a\ b)$

$\lambda x.\lambda y.(f\ x\ y) = f$

Note that types are preserved during λ-conversion.

Definition 3.12: A term is said to be in *normal form* iff it is a λ-term but not of the form $\mathcal{E}[(\lambda x.e_1\ e_2)]$.

Theorem 3.2: For every term e there exists a term e' in normal form derivable from e by λ-conversion. This term e' is unique modulo α-conversion, and is called *the normal form of* e.
Proof: see Fortune et al. (1983), page 158.

Definition 3.13: Let e be the term $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.(@\ e_1\ e_2\ \ldots\ e_p)$. Then the *head* of e is the atom $@$, and the *heading* of e is the term $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.@$. e is called *rigid* if $@ \in \mathcal{C} \cup \{x_1, \ldots, x_n\}$, and *flexible* otherwise.

Definition 3.14:

A *substitution pair* is a pair $\langle x, e\rangle$ where $x \in \mathcal{V}, e \neq x, \tau(x) = \tau(e)$ and e is reduced to normal form. We say this substitution pair *pertains to x*. A *substitution* is a finite set of substitution pairs pertaining to distinct variables:

$\sigma = \{\langle x_i, e_i\rangle | 1 \leq i \leq n\} \qquad \forall (1 \leq i, j \leq n)(x_i = x_j) \Rightarrow (i = j)$

We define $x\ \sigma$ as

$$\begin{cases} e & \text{if } \langle x, e\rangle \in \sigma, \\ x & \text{otherwise.} \end{cases}$$

If $\mathcal{T}$ denotes the set of terms in normal form, $x\sigma$ can be interpreted as a type-preserving mapping from $\mathcal{V}$ to $\mathcal{T}$. We extend this mapping in the following way: For all $E \in \mathcal{T}$, the application of σ to E, written as $E\ \sigma$, is defined as the normal form of the term $((\lambda x_1, \ldots, \lambda x_n.E)\ (e_1, e_2, \ldots, e_n))$.

Notational Conventions:

In computer implementations of higher-order unification, the infix symbol "\" is used instead of the combination of the Greek letter λ and the period. From now on, we also adopt the convention that symbols starting with capital letters are variables and symbols starting with lower case letters are constants, to be consistent with the Prolog-based

implementation of the system. For example, the term $\lambda X.\lambda Y.(loves\ X\ Y)$ is represented as `X\Y\(loves X Y)`. The standard way to express the relationship between operators or predicates (which are all considered constants in the λ-calculus) and their arguments is *prefix*-notation. For example, `(X + Y)` would be written as `(+ X Y)`. (Cambridge prefix notation is used for all terms; e.g., `(loves X Y)` is used instead of `loves(X,Y)`.) The above conventions will be used throughout the remainder of this dissertation.

3.3. Representation of Mathematical Objects

There are many ways to represent numbers, functions, truth values, data structures, and other objects in the typed λ-calculus. Numbers, for example, can be represented as "Church numerals" in the following way:

0 = `F\X\X`
1 = `F\X\(F X)`
2 = `F\X\(F (F X))`
3 = `F\X\(F (F (F X)))`
etc.

Assuming i is a primitive type, each such Church numeral can by assigned the simple type $(i \rightarrow i) \rightarrow i \rightarrow i$ (note that the $\rightarrow$ operator is right-associative).

Arithmetic functions like the successor function, the addition function, the multiplication function, or the conditional function can be implemented in the typed λ-calculus using the Church numerals and the function definitions given below. Such arithmetic operations can thus be performed using the conversion rules of the typed λ-calculus discussed above.

succ = `N\F\X\(N F (F X))`
add = `M\N\F\X\(M F (N F X))`
mult = `M\N\F\X\(N (M F) X)`
cond = `M\N\O\F\X\(M Y\(N F X) (O F X))`

Let $I = (i \rightarrow i) \rightarrow (i \rightarrow i)$. Then $\tau(succ) = I \rightarrow I$, $\tau(add) = I \rightarrow I \rightarrow I$, $\tau(mult) = I \rightarrow I \rightarrow I$, and $\tau(cond) = I \rightarrow I \rightarrow I \rightarrow I$. Application of the conditional function (**cond** $m\ n\ o$), where m, n, and o are Church numerals, reduces to n if m is greater than zero and to o if m is equal to zero.

Considering only natural number representations over a domain B, where B is a primitive type, the functions representable in the simply typed λ-calculus can be characterized in the following way:

Theorem 3.3: The functions representable by λ-expressions of the type $(I \rightarrow (I \rightarrow \ldots(I \rightarrow I)\ldots))$, where $I = (B \rightarrow B) \rightarrow (B \rightarrow B)$, where B is a primitive type, are exactly the functions generated by the constants **0** and **1** using the operations **add**, **mult**, and **cond**.

Proof: see Fortune et al. (1983), page 161.

Shown below are representations of the booleans along with typical operations. Suitable types can be given, but are omitted here for brevity.

true = `X\Y\X`
false = `X\Y\Y`
not = `T\(T false true )`
iszero = `N\(N (X\false) true)`
if-then-else = `(A B C)`
and = `A\B\(A B X\Y\Y)`

Even though any recursive function can be represented by a λ-expression (Gordon 1988), the *typed* λ-calculus is not powerful enough for representing all of them; instead, the *untyped* λ-calculus must be used in many cases. For example, the **or** function defined below cannot be assigned simple types since it contains a "self-application" term, i.e., application term of the form $(x\ x)$. Self-application terms are not legal in the typed λ-calculus since they cannot be assigned any type.

or = `A\B\((A A) B)`

3.4. Higher-Order Unification

3.4.1 Motivation

The unification of typed λ-terms is referred to as *higher-order unification*. It is the process of finding substitutions for variables (some of which may denote functions) such that the terms are equal by λ-conversion. As an example, consider the following two terms:

(1) `X\Y\(foo (a X Y) H)`

(2) `X\Y\(foo (G Y X) 2)`

By replacing `G` by `P\Q\(a Q P)` and replacing `H` by `2`, the two resulting terms are equal by the λ-conversion rules:

(1) `X\Y\(foo (a X Y) 2)`

(2) `X\Y\(foo (P\Q\(a Q P) Y X) 2)`
= `X\Y\(foo (Q\(a Q Y) X) 2)` (β-conversion)
= `X\Y\(foo (a X Y) 2)` (β-conversion)

However, the following two terms cannot be unified:

(3) `X\Y\(f X Y)`

(4) `X\Y\(g X Y)`

There are also no unifying substitutions for the two terms below,

(5) `X\Y\(a Y)`

(6) `X\Y\(F X)`

because `(a Y)` contains the prefix variable `Y`, but the substitution for the function variable `F` is independent of `Y` since `F` is applied only to `X`.

In general, a pair of typed λ-terms may not have a most general unifier. For example, `(F a)` and `a` have two unifiers (assuming, for example, that $\tau(a) = i$, and $\tau(F) = i \rightarrow i$):

`F` $\leftarrow$ `X\a`
`F` $\leftarrow$ `X\X`

neither one of which is more general than the other, i.e., one cannot be obtained from the other by a substitution to some of its free variables. Similarly, `(F a)` and `(g a a)` have four unifiers (assuming, $\tau(a) = i$, $\tau(F) = i \rightarrow i$, and $\tau(g) = i \rightarrow i \rightarrow i$):

`F` $\leftarrow$ `X\(g a X)`
`F` $\leftarrow$ `X\(g X a)`
`F` $\leftarrow$ `X\(g X X)`
`F` $\leftarrow$ `X\(g a a)`

It is even possible that two terms have countably infinite unifiers. The higher-order unification problem is only semi-decidable: While there is a procedure that always find a unifier if one exists, it is in general impossible to determine that there is no unifier. A systematic search procedure for unifiers was first given by Huet (1975). A simple example to motivate Huet's unification procedure is given below. The procedure is discussed in detail in the next section. Suppose that we want to unify `(X Y Z)` with `(foo a b)` (assuming suitable types). One may attempt to do that by having the head of the first term, `X`, "imitate" the head of the second term, `foo`. Thus the substitution for `X` is constructed as follows:

```
X ← V1\V2\(foo (H1 V1 V2) (H2 V1 V2))
```

Note that the arguments for `foo` are terms built up of function variables (`H1` and `H2`) applied to the sequence of binder variables. Such a substitution expresses a general form for the imitation. The introduction of new variables is a key reason for the potential nontermination of the unification process (contrast this situation with that of the first-order unification algorithm). Under this imitition substitution for `X`, the term `(X Y Z)` becomes

```
(V1\V2\(foo (H1 V1 V2) (H2 V1 V2)) Y Z)
```

which by β-reduction is equal to

```
(foo (H1 Y Z) (H2 Y Z)).
```

Now, in order to unify `(foo (H1 Y Z) (H2 Y Z))` with `(foo a b)` we have to unify each of their arguments: First we need to unify `(H1 Y Z)` with `a`. It is possible to attempt an imitation substitution, as before, for variable `H1`. However, it is also possible to construct a function that will return one of the arguments `Y` or `Z`, which in turn is to be unified with `a`. This is the idea behind the "projection" substitutions. Let us attempt the following projection:

```
H1 ← V1\V2\V1
```

Now `(H1 Y Z)` becomes `(V1\V2\V1 Y Z)`, which is equal to `Y`. Obviously the imitation substitution `Y ← a` takes care of the rest. Unifying the second arguments `(H2 V1 V2)` and `b` can be done similarly.

3.4.2. Higher-order unification procedure

Suppose that we are given a finite set of pairs of terms of the same type to be unified:

$$\{\langle u_1, v_1\rangle, \ldots, \langle u_n, v_n\rangle\}.$$

The higher-order unification problem involves finding a substitution σ such that $u_i\ \sigma$ is λ-convertible to $v_i\ \sigma$ by the rules of λ-conversion defined earlier. We assume each term is represented in head-normal form, as:

$$\lambda x_1 \ldots, \lambda x_n.(A\ t_1 \ldots t_m),$$

where A is a constant or variable of type $\alpha_1 \rightarrow \ldots \rightarrow \alpha_m \rightarrow \beta$. Given two rigid terms

$$\lambda x_1 \ldots . \lambda x_n.(F_1\ s_1 \ldots s_i) \text{ and } \lambda x_1 \ldots . \lambda x_n.(F_2\ r_1 \ldots r_i)$$

of the same type, they are unifiable only if F_1 and F_2 are identical, and they can be reduced to:

$$\{\langle \lambda x_1 \ldots . \lambda x_n.s_1, \lambda x_1 \ldots . \lambda x_n.r_1\rangle, \ldots, \langle \lambda x_1 \ldots . \lambda x_n.s_i, \lambda x_1 \ldots . \lambda x_n.r_i\rangle\}.$$

Huet observed that either such a set of pairs has no unifier or it can be reduced (by procedure SIMPL defined further below) to another set, having the same set of unifiers, in which each pair has at least one flexible term. For a set consisting only of flexible-flexible pairs, a unifier can be trivially constructed. For a flexible-rigid pair, Huet has shown that two kinds of substitutions are possible: imitation and projections.

Definition 3.15: Let $F = \lambda x_1 \ldots, \lambda x_n.(f\ t_1 \ldots t_k)$ and $R = \lambda x_1 \ldots . \lambda x_n, (c\ s_1 \ldots\ s_j)$ respectively be the flexible and rigid terms in head normal form; and let the type of f be $\alpha_1 \rightarrow \ldots \rightarrow \alpha_k \rightarrow \beta$. Then, if c is a constant, the *imitation substitution* is defined as:

$$f \leftarrow \lambda w_1 \ldots . \lambda w_k.(c\ (h_1\ w_1\ \ldots\ w_k)\ \ldots\ (h_j\ w_1\ \ldots\ w_k)),$$

where the h_i's are new variables of appropriate types.

Definition 3.16: If α_i is of the form $\beta_1 \rightarrow \ldots \rightarrow \beta_l \rightarrow \beta$, the i^{th} *projection substitution*, for $1 \leq i \leq k$, is defined as:

$$f \leftarrow \lambda w_1 \ldots . \lambda w_k.(w_i\ (h_1\ w_1\ \ldots\ w_k)\ \ldots\ (h_l\ w_1\ \ldots\ w_k)),$$

where the h_i's above are new variables of appropriate types. Note that these substitutions are determined entirely by the heads of the flexible and rigid terms.

Higher-order unification procedures can be conveniently described in terms of the function SIMPL (Huet 1975, Nadathur & Miller 1990).

Definition 3.17: The function SIMPL on sets of disagreement pairs $\mathcal{D}$ is defined as follows (a sequence of prefix variables x_i is denoted by $\vec{x}$).

(1) If $\mathcal{D} = \phi$ then SIMPL($\mathcal{D}$) = ϕ.

(2) If $\mathcal{D} = \{\langle F_1, F_2\rangle\}$, and

  (a) if F_1 is flexible then SIMPL($\mathcal{D}$) = $\mathcal{D}$; otherwise

  (b) if F_2 is flexible then SIMPL($\mathcal{D}$) = $\{\langle F_2, F_1\rangle\}$;

  (c) otherwise F_1 and F_2 are both rigid. Let $F_1 = \lambda\vec{x}.(C_1\ A_1\ \ldots\ A_r)$ and let $F_2 = \lambda\vec{x}.(C_2\ B_1\ \ldots\ B_s)$. If $C_1 \neq C_2$ then SIMPL($\mathcal{D}$) fails; otherwise SIMPL($\mathcal{D}$) = SIMPL($\{\langle\lambda\vec{x}.A_i, \lambda\vec{x}.B_i\rangle | 1 \leq i \leq r\}$).

(3) Otherwise $\mathcal{D}$ has more than one equations. Let $\mathcal{D} = \{\langle F_i, G_i\rangle | 1 \leq i \leq n\}$.

  (a) If SIMPL($\{\langle F_i, G_i\rangle\}$) fails for some i, then SIMPL($\mathcal{D}$) fails;

  (b) Otherwise SIMPL($\mathcal{D}$) = $\bigcup_{i=1}^{n}$ SIMPL($\{\langle F_i, G_i\rangle\}$).

A higher-order unification procedure would iteratively select an applicable substitution according to some scheme, reduce some or all terms to normal form, and simplify the result using SIMPL.

Example 3.4.1:

We illustrate how the imitation and projection substitutions and the SIMPL function can be used to resolve the set of equations (disagreement pairs) from the example in section 5.1. We have not yet specified how equations from this set are selected for substitutions, or in which order substitutions are enumerated. For the present we just assume that the set of equations is ordered and we always select the first one to be processed next. We also assume that substitutions are enumerated in some order of increasing complexity. A more precise higher-order unification procedure is given in section 5. Let $\mathcal{D}$ be the current set of disagreement pairs initialized to the following set:

```
{F1 = F\X\X,
 (F2 F1) = F\X\(F X),
 (F2 (F2 F1)) = F\X\(F (F X))}
```

We assume type $(i \to i) \to i \to i$ for each term on the right-hand side, and assume that all other terms have appropriate types. The type of `F1` is assumed to be $(i \to i) \to i \to i$,

and that of F2 assumed to be $((i \to i) \to i \to i) \to (i \to i) \to i \to i$. (For readability, types are omitted in the following discussion.) Selecting the first equation, we need to find a substitution for F1. Only projection substitutions are applicable since X, the head of the right hand side term, is not a constant. The two simplest applicable projection substitutions are K\L\L and K\L\K. The first term clearly is the right choice and converts the first equation to:

```
K\L\L = F\X\X.
```

After α-conversion both sides of this equation are identical. According to SIMPL this equation is now removed from the $\mathcal{D}$, and there are no equations added since the heads of these terms have no arguments. Therefore $\mathcal{D}$ becomes:

```
{(F2 F1) = F\X\(F X),
 (F2 (F2 F1)) = F\X\(F (F X))}
```

After substituting for all F1's in $\mathcal{D}$ and simplifying all terms we get:

```
{(F2 K\L\L) = F\X\(F X),
 (F2 (F2 K\L\L)) = F\X\(F (F X))}
```

Next, a substitution for F2 needs to be found. Applicable substitutions are:

```
(1)   F2 <- K\L\M\(M (H1 K L M) (H2 K L M))
(2)   F2 <- K\L\M\(L (H1 K L M) (H2 K L M))
(3)   F2 <- K\L\M\(K (H1 K L M) (H2 K L M))
```

Selecting the first substitution above, $\mathcal{D}$ is converted to:

```
{(K\L\M\(M (H1 K L M) (H2 K L M)) K\L\L) = F\X\(F X),
 (K\L\M\(M (H1 K L M) (H2 K L M))
   (K\L\M\(M (H1 K L M) (H2 K L M)) K\L\L)) = F\X\(F (F X))}
```

The first equation can be simplified through β-reduction to the following:

```
L\M\(M (H1 A\B\B L M) (H2 A\B\B L M)) = F\X\(F X)
```

Since the head of the left hand side term is a different prefix variable than the head of the right hand side term, SIMPL fails, and another substitution must be tried. The second choice will lead to failure as well. Using the third substitution $\mathcal{D}$ is converted to:

```
{(K\L\M\(K (H1 K L M) (H2 K L M)) K\L\L) = F\X\(F X),
 (K\L\M\(K (H1 K L M) (H2 K L M))
   (K\L\M\(K (H1 K L M) (H2 K L M)) K\L\L)) = F\X\(F (F X))}
```

which, after reducing all terms to normal form, becomes:

```
{A\B\(H1 K\L\L A B) = F\X\(F X),
 A\B\(H1 K\L\L (H2 K\L\(H1 M\N\N K L) A B)
               (H1 K\L\(H1 M\N\N K L) A B)) = F\X\(F (F X))}
```

The only viable substitution for H1 now is K\L\L, converting $\mathcal{D}$ to:

```
{A\B\(A B) = F\X\(F X),
 A\B\(H2 K\L\(K L) A B (A B)) = F\X\(F (F X))}
```

Applying α-conversion:

```
{F\X\(F X) = F\X\(F X),
 F\X\(H2 K\L\(K L) F X (F X)) = F\X\(F (F X))}
```

Since the headings of the first equation are identical, it is removed from $\mathcal{D}$ and another equation made from the argument of the heads is added as specified by SIMPL:

```
{F\X\X = F\X\X,
 F\X\(H2 K\L\(K L) F X (F X)) = F\X\(F (F X))}
```

The new equation is also removed, and the only equation remaining in $\mathcal{D}$ is:

```
{F\X\(H2 K\L\(K L) F X (F X)) = F\X\(F (F X))}
```

The next substitution would be:

```
 H2 <- K\L\M\N\(L (H3 K L M N))
```

converting $\mathcal{D}$ to:

```
{F\X\(F (H3 K\L\(K L) F X (F X))) = F\X\(F (F X))}
```

which according to SIMPL is simplied to:

```
{F\X\(H3 K\L\(K L) F X (F X)) = F\X\(F X)}
```

Finally, substituting K\L\M\N\N for H3 yields:

```
   {F\X\(F X) = F\X\(F X)}
 = {F\X\X = F\X\X}
 = {}
```

After substituting and reducing all terms, the final unifying substitutions are:

```
F1 <- K\L\L
F2 <- K\L\M\(K L (L M))
H1 <- K\L\L
H2 <- K\L\M\N\(L N)
H3 <- K\L\M\N\N
```

Example 3.4.2:

Next we give a somewhat larger example to illustrate the use of both imitation and projection substitutions. In this example we want to find substitutions for F1, F2, F3, F4, F5, and F7 so that (F1 (F2 F4) (F3 F7 (F2 F5))) = (saw mike mary). The sequence of steps needed to find a unifier is shown below. Note that a single equation does not provide enough constraints to restrict the number of possible unifiers to one. The derivation below represents only one solution path in the derivation tree. The derivation is presented in terms of triples of the form < T1 , T2 , Subst >, where T1 and T2 the two higher-order terms to be unified, and Subst is the set of substitutions applied so far.

```
           < (F1 (F2 F4) (F3 F7 (F2 F5))) , (saw mike mary) , {} >
Projection:    { F1 <- U1\U2\(U1 (H1 U1 U2)) }
---->      < (F2 F4 (H1 (F2 F4) (F3 F7 (F2 F5)))) , (saw mike mary) ,
                  { F1 <- U1\U2\(U1 (H1 U1 U2)) } >
Projection:    { F2 <- U1\U2\(U2 (H2 U1 U2)) }
---->      < (H1 ... (F3 F7 (F2 F5)) (H2 F4 (H1 ... (F3 F7 (F2 F5))))) ,
                  (saw mike mary) ,
                  { F1 <- U1\U2\(U1 (H1 U1 U2)),
                    F2 <- U1\U2\(U2 (H2 U1 U2)) } >
Projection:    { H1 <- U1\U2\U3\(U2 (H3 U1 U2 U3)) }
---->      < (F3 F7 (F2 F5) (H3 ... (F3 F7 (F2 F5)) (H2 F4 ...))) ,
                  (saw mike mary) ,
                  { F1 <- U1\U2\(U1 W3\(U2 (H3 U1 U2 W3))),
                    F2 <- U1\U2\(U2 (H2 U1 U2)) } >
Projection:    { F3 <- U1\U2\U3\(U2 (H4 U1 U2 U3)) }
```

```
---->      < (F2 F5 (H4 F7 (F2 F5) (H3 ... ... (H2 F4 ...))) ,
                  (saw mike mary) ,
                  { F1 <- U1\U2\(U1 W3\(U2 (H3 U1 U2 W3))),
                    F2 <- U1\U2\(U2 (H2 U1 U2)),
                    F3 <- U1\U2\U3\(U2 (H4 U1 U2 U3)) } >
Apply substitution for F2:
---->      < (H4 F7 ... (H3 ... ... (H2 F4 ...)) (H2 F5 ...)) ,
                  (saw mike mary) ,
                  { F1 <- U1\U2\(U1 W3\(U2 (H3 U1 U2 W3))),
                    F2 <- U1\U2\(U2 (H2 U1 U2)),
                    F3 <- U1\U2\U3\(U2 (H4 U1 U2 U3)) } >
Projection:     { H4 <- U1\U2\U3\U4\(U1 (H5 . . U3 .) (H6 . . . U4)) }
---->      < (F7 (H5 . . (H3 ... ... (H2 F4 ...)) .)
                 (H6 . . . (H2 F5 ...))) ,
                  (saw mike mary) ,
                  { F1 <- U1\U2\(U1 W3\(U2 (H3 U1 U2 W3))),
                    F2 <- U1\U2\(U2 (H2 U1 U2)),
                    F3 <- U1\U2\U3\(U2 W4\(U1 (H5 U1 U2 U3 W4)
                                              (H6 U1 U2 U3 W4))) } >
Imitation:      { F7 <- U1\U2\(saw (K1 U1 U2) (K2 U1 U2)) }
---->      < (saw (K1 (H5 . . (H3 . . (H2 F4 .)) .) (H6 . . . (H2 F5 .)))
                 (K2 (H5 . . (H3 . . (H2 F4 .)) .) (H6 . . . (H2 F5 .)))) ,
                  (saw mike mary) ,
                  { F1 <- U1\U2\(U1 W3\(U2 (H3 U1 U2 W3))),
                    F2 <- U1\U2\(U2 (H2 U1 U2)),
                    F3 <- U1\U2\U3\(U2 W4\(U1 (H5 U1 U2 U3 W4)
                                              (H6 U1 U2 U3 W4))),
                    F7 <- U1\U2\(saw (K1 U1 U2) (K2 U1 U2)) } >
Now unify first argument:
---->      < (K1 (H5 . . (H3 . . (H2 F4 .)) .) (H6 . . . (H2 F5 .))) ,
                  mike ,
                  { F1 <- U1\U2\(U1 W3\(U2 (H3 U1 U2 W3))),
                    F2 <- U1\U2\(U2 (H2 U1 U2)),
                    F3 <- U1\U2\U3\(U2 W4\(U1 (H5 U1 U2 U3 W4)
                                              (H6 U1 U2 U3 W4))),
                    F7 <- U1\U2\(saw (K1 U1 U2) (K2 U1 U2)) } >
Projection:     { K1 <- U1\U2\U1 }
Projection:     { H5 <- U1\U2\U3\U4\U3 }
Projection:     { H3 <- U1\U2\U3\U4\U3 }
Projection:     { H2 <- U1\U2\U1 }
---->      < F4 , mike ,
                  { F1 <- U1\U2\(U1 W3\(U2 W3)),
                    F2 <- U1\U2\(U2 U1),
                    F3 <- U1\U2\U3\(U2 W4\(U1 U3 (H6 U1 U2 U3 W4))),
```

```
                    F7 <- U1\U2\(saw U1 (K2 U1 U2)) } >
Imitation:      { F4 <- mike }
Unify second argument:
           < (K2 (H5 . . (H3 . . (H2 F4 .)) .) (H6 . . . (H2 F5 .))) ,
                   mary ,
                   { F1 <- U1\U2\(U1 W3\(U2 W3)),
                     F2 <- U1\U2\(U2 U1),
                     F3 <- U1\U2\U3\(U2 W4\(U1 U3 (H6 U1 U2 U3 W4))),
                     F7 <- U1\U2\(saw U1 (K2 U1 U2)),
                     F4 <- mike } >
Projection:     { K2 <- U1\U2\U2 }
Projection:     { H6 <- U1\U2\U3\U4\U4 }
Projection:     { H2 <- U1\U2\U1 }
---->       < F5 , mary ,
                   { F1 <- U1\U2\(U1 W3\(U2 W3)),
                     F2 <- U1\U2\(U2 U1),
                     F3 <- U1\U2\U3\(U2 W4\(U1 U3 W4)),
                     F7 <- U1\U2\(saw U1 U2),
                     F4 <- mike } >
Imitation:      { F5 <- mary }
```

Applying η-conversion to the substitutions for `F1` and `F3` we obtain the final set of substitutions:

```
                   { F1 <- U1\U2\(U1 U2),
                     F2 <- U1\U2\(U2 U1),
                     F3 <- U1\U2\U3\(U2 (U1 U3)),
                     F7 <- U1\U2\(saw U1 U2),
                     F4 <- mike,
                     F5 <- mary }
```

Example 3.4.3:

If there is no unifier the search may not terminate. For example, trying to unify the terms `(F a)` and `(b (F a))` would lead to an infinite computation:

(F a) = (b (F a))

Imitation: F → W \(b (H1 W))

⟶ (b (H1 a)) = (b (b (H1 a)))

⟶ (H1 a) = (b (H1 a))

Imitation: H1 → W \(b (H2 W))

⟶ (b (H2 a)) = (b (b (H2 a)))

⟶ (H2 a) = (b (H2 a))

etc.

4. Higher-Order Definite Clause Grammars

Higher-order definite clause grammars (DCGs) are a generalization of first-order DCGs, which are based on *definite (clause) programs*. DCGs are a special case of *metamorphosis grammars* (Colmerauer 1978). The rules of metamorphosis grammars are of the form:

$$S\alpha \rightarrow \beta$$

where S is a nonterminal grammar symbol, α is a string of terminals and nonterminals and β is a string of terminals, nonterminals and procedure calls. Metamorphosis grammars are a type of *logic grammar* (Abramson 1989). Logic grammars comprise generalized type-0 rewriting rules like ordinary grammars, but their grammar symbols may include arguments representing trees. They also differ from traditional grammars in the use of variables and unification, and the possibility of including tests in grammar rules. In addition, logic grammars can be endowed with procedural semantics through processors based on specialized theorem provers.

Other logic grammars, e.g. *extraposition grammars* (Pereira 1981), include special mechanisms to capture certain linguistic phenomena, but are equivalent in terms of computational power. Compared with other logic grammars, the main advantage of DCGs is that they can be easily and efficiently implemented. Since symbols can have arguments, DCGs can also describe type-0 languages, but in general not as straightforwardly as other grammar types due to the restrictions on the DCG rules discussed below.

4.1. Definite Clause Programs

In this section we define definite-clause programs and state the correctness theorems for their execution. Following the notation and terminology of Lloyd (1987), we start by defining first-order terms. The definitions below assume an alphabet consisting of *variables*, *constants*, *function symbols*, *predicate symbols*, *connectives* (including $\longleftarrow$, $\vee$, and $\sim$ (negation)), *quantifiers* (including $\exists$ and $\forall$), and usual the *punctuation symbols*.

Definition 4.1 A (first-order) *term* is defined inductively as follows:
(a) a variable is a term;

(b) a constant is a term;
(c) if f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

Definition 4.2 If p is an n-ary predicate symbol, and if $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atomic formula* or simply *atom*.[3]

Definition 4.3 A *literal* is an atom or the negation of an atom. A *positive literal* is an atom. A *negative literal* is the negation of an atom.

Definition 4.4 A *clause* is a formula of the form $\forall x_1 \ldots \forall x_s (L_1 \vee \ldots \vee L_m)$, where each L_i is a literal and $x_1, \ldots, x_s$ are all the variables occurring in $L_1 \vee \ldots \vee L_m$. The *empty clause* is denoted by $\Box$ and corresponds to the case where $m = 0$; it can be interpreted as contradiction.

Definition 4.5 A *definite program clause* is a clause of the form $A \vee \sim B_1 \vee \ldots \vee \sim B_n$, where A and each B_i is a positive literal (universal quantification is assumed). It is more usual to write such a clause as $A \longleftarrow B_1, \ldots, B_n$. A is therefore called the *head* of the program clause, and $B_1, \ldots, B_n$ is called the *body* of the program clause.

Definition 4.6 A *definite-clause program* (also called *definite program*) is a finite set of definite program clauses.

Definition 4.7 A *definite goal* is a clause of the form $\longleftarrow B_1, \ldots, B_n$ (which is equivalent to $\sim B_1 \vee \ldots \vee \sim B_n$ with all variables universally quantified).

Definition 4.8 A *substitution* θ is a finite set of the form $\{v_1/t_1, \ldots, v_n/t_n\}$, where each v_i is a variable, each t_i is a term distinct from v_i, and the variables $v_1, \ldots, v_n$ are distinct. Each element v_i/t_i is called a *binding* for v_i.

Definition 4.9 An *expression* is either a term, a literal, or a conjunction or disjunction of literals.

Definition 4.10 Two expressions E and F are *variants* if there exist substitutions θ and σ such that $E = F\theta$ and $F = E\sigma$.

Definition 4.11 Let P be a definite program and G a definite goal. An *answer* for $P \cup \{G\}$ is a substitution for variables of G.

[3]The term 'atom' here should not be confused with atom in Chapter 3.

Definition 4.12 Let P be a definite program, G a definite goal $\longleftarrow A_1, \ldots, A_k$, and θ an answer for $P \cup \{G\}$. We say that θ is a *correct answer* for $P \cup \{G\}$ if $\forall((A_1 \wedge \ldots \wedge A_k)\theta)$ is a logical consequence of P. ($\forall(T)$ means that all variables of T are universally quantified.)

Definition 4.13 Let G be $\longleftarrow A_1, \ldots, A_m, \ldots, A_k$ and C be $\longleftarrow B_1, \ldots, B_q$. Then G' is *derived* from G and C using the most general unifier (mgu) θ if the following conditions hold:
(a) A_m is an atom, called the *selected* atom, in G;
(b) θ is an mgu of A_m and A;
(c) G' is the goal $\longleftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$.
G' is called a *resolvent* of G and C.

Definition 4.14 Let P be a definite program and G a definite goal. An *SLD-derivation*[4] of $P \cup \{G\}$ consists of a sequence $G_0 = G, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of variants of program clauses of P, and a sequence $\theta_1, \theta_2, \ldots$ of mgu's such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}.

Definition 4.15 An *SLD-refutation* (also called *SLD-resolution*) of $P \cup \{G\}$ is a finite SLD-derivation of $P \cup \{G\}$ which has the empty clause $\Box$ as the last goal in the derivation. If $G_n = \Box$, we say the refutation has *length n*.

Definition 4.16 Let P be a definite program and G a definite goal. A *computed answer* θ for $P \cup \{G\}$ is the substitution obtained by restricting the composition of $\theta_1 \ldots \theta_n$ to the variables of G, where $\theta_1 \ldots \theta_n$ is the sequence of mgu's used in an SLD-refutation of $P \cup \{G\}$.

Theorem 4.17 *(Soundness of SLD-Resolution)*
Let P be a definite program and G a definite goal. Then every computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$.

Proof: see Lloyd (1987), page 43.

Theorem 4.18 *(Completeness of SLD-Resolution)*
Let P be a definite program and G a definite goal. For every correct anwer θ for $P \cup \{G\}$, there exists a computed answer σ for $P \cup \{G\}$ and a substitution γ such that $\theta = \sigma\gamma$.

Proof: see Lloyd (1987), page 49.

Below is a simple definite clause program for converting a list of English words into a list of French words, or vice versa. (Symbols starting with capital letters denote variables; "cons" and "nil" are arbitrary function symbols; clauses are delimited by periods.)

[4] *SLD* stands for **L**inear resolution for **D**efinite clauses with **S**election function.

Example 4.1 (Sterling 1986)

```
translate(cons(Word,Words),cons(Mot,Mots)) ←–
        dict(Word,Mot), translate(Words,Mots).
translate(nil,nil).

dict(the,le).
dict(dog,chien).
dict(chases,chasse).
dict(cat,chat).
```

A possible query (goal) for the program would be:

(1) translate(cons(the,cons(dog,cons(chases,cons(the,cons(cat,nil))))),Ms)

which would bind "Ms" to the corresponding list with French words:

(2) cons(le,cons(chien,cons(chasse,cons(le,cons(chat,nil)))))

The first step of this derivation involves unifying the query (1) with the head of a variant of the first program clause for "translate". Let the following clause be this variant:

```
translate(cons(Word1,Words1),cons(Mot1,Mots1)) ←–
        dict(Word1,Mot1), translate(Words1,Mots1).
```

Then "Word1" is unified with "the", and "Words1" is unified with

cons(dog,cons(chases,cons(the,cons(cat,nil)))).

With these bindings the subgoals "dict(Word1,Mot1)" and "translate(Words1,Mots1)" are executed next. Subgoal "dict(the,Mot1)" unifies with the first program clause for "dict" binding "Mot1" to "le". Subgoal "translate(cons(dog,...)),Mots1)" recursively invokes the first program clause for "translate" unifying with another variant of it. The process terminates when the last element "nil" of the input term is reached and the second program clause for "translate" is applied. At this point "Ms" of the original query has been bound to (2).

4.2. First-Order Definite Clause Grammars

Definite-clause grammars can be thought of as *context free grammars* (CFGs) augmented with parameters which specify semantic information. Below is a simple example illustrating our notation for a CFG. The start symbol of the grammar is the nonterminal on the left-hand side of the first production rule, and a terminal symbol is an identifier enclosed within [and].

Example 4.2

```
sentence → np, vp.
np → pn.
vp → tv, np.
pn → [julia].
pn → [tom].
tv → [hates].
```

A definite clause grammar (DCG) combines the concept of a CFG and a definite clause program. A DCG allows us to specify the syntax of a language using CFG-like rules and the semantics of the language using the terms of definite clause programs. A DCG enhances a CFG in three important ways: (1) grammar nonterminal symbols may include arguments; (2) rules may include predicates (for imposing conditions) in their bodies;[5] and (3) invocation of grammar rules requires unification of these arguments.

Syntax of (first-order) DCGs:

DCG ::= $clause$
DCG ::= $clause, DCG$

$clause$ ::= $rule$
$clause$ ::= $fact$

$rule$ ::= $head \rightarrow body$

$head$ ::= $posliteral$

$body$ ::= $posliteral$
$body$ ::= $posliteral, body$

[5] This feature is omitted from most of the discussion in this dissertation, since it does not affect the major techniques of my thesis.

$fact$ ::= $posliteral$

$posliteral$ ::= $functor(arg, \ldots, arg)$

$functor$ ::= ⟨ constant ⟩

arg ::= ⟨ first-order term ⟩

Arguments may be used to enforce context-sensitive features of the language or to build structures, such as parse trees, during parsing.

Example 4.3

```
sentence(sent(N,V)) → nounphrase(N, Num),
                      verbphrase(V, Num).

nounphrase(np(D,N), Num) → determiner(D, Num),
                           nounphrase2(N, Num).
nounphrase(np(N), Num) → nounphrase2(N, Num).

nounphrase2(np2(A,N), Num) → adjective(A),
                             nounphrase2(N, Num).
nounphrase2(np2(N), Num) → noun(N, Num).

verbphrase(vp(V), Num) → verb(V, Num).

determiner(det(the), Num) → [the].
determiner(det(a), singular) → [a].

noun(n(computer), singular) → [computer].
noun(n(computers), plural) → [computers].

adjective(adj(super)) → [super].
adjective(adj(mini)) → [mini].
adjective(adj(micro)) → [micro].

verb(v(runs), singular) → [runs].
verb(v(run), plural) → [run].
```

Example 4.3 shows a definite clause grammar for a trivial subset of English that illustrates parse tree construction and simple check for number agreement. (Square brackets `[...]` are a notational variant of the "dot" functor `.(...)`, used to contruct lists.) Each grammar rule can be read declaratively. For example, the first rule states that a sentence is composed of a noun phrase followed by a verb phrase, that the semantic representation of a sentence is composed from the semantic representations of the constituent noun phrase and verb phrase, and that the noun phrase and verb phrase have the same number (singular or plural); the second rule states that a noun phrase is composed of a determiner followed by another type of noun phrase (as defined by the rules for `nounphrase2`); etc. Disjunction is expressed by having more than one rule for a particular grammar constituent. Each nonterminal grammar symbol can have arguments which may contain variables (symbols that start with capital letters are variables). The scope of a variable is the rule in which it occurs. Note that, as with CFGs, terminal symbols are of the form `[Word]`, where `Word` is a word of the sentence being parsed.

Transformation from DCGs to definite clause programs:

To execute the above DCG as a relational program, two additional arguments are added to each grammar symbol in order to keep track of how much of the input sentence has been processed. In the following description, the notation $\bar{t_i}$ is used to denote a sequence of terms. Rules of the form

$$a(\bar{t}) \longrightarrow b_1(\bar{t_1}), b_2(\bar{t_2}), \ldots, b_k(\bar{t_k}).$$

are converted into

$$a(\bar{t}, P_0, P_k\ )\ \texttt{:-}\ \ b_1(\bar{t_1}, P_0, P_1), b_2(\bar{t_2}, P_1, P_2), \ldots, b_k(\bar{t_k}, P_{k-1}, P_k).$$

And terminals

$$a(\bar{t}) \longrightarrow [word].$$

are converted into

$$a(\bar{t}, [word|Rest], Rest).$$

We use the symbol `:-` as a variant of a left arrow to separate the head from the body in definite clauses.[6] The notation $[W|R]$ denotes a list whose first element is W and

[6]This is also the symbol used by Prolog interpreters.

the remainder R. Using a straightforward inductive argument, one can show that the resulting definite-clause program correctly parses the sentences of the language defined by the corresponding DCG. In order to do so, the semantics of DCGs is first defined as follows:

(1) For all ground instances $a(\bar{t}) \longrightarrow$ `[word]` of some rule, $\bar{t}$ is a semantic representation for the sentence `[word]`.

(2) For all ground instances $a(\bar{t}) \longrightarrow b_1(\bar{t_1}),\ldots,b_k(\bar{t_k})$, if $s_1,\ldots,s_k$ are sentences derivable from nonterminals $b_1,\ldots,b_k$ respectively, and $\bar{t_1},\ldots,\bar{t_k}$ are their respective semantic representations, then $\bar{t}$ is a semantic representation for $s_1\ s_2\ \ldots s_k$ (i.e. the concatenation of these sentences).

Theorem 4.1: Let D be a DCG and P the program obtained from D using the transformation described earlier. Then P correctly parses and computes the semantics of sentences of D.

Proof: Since there is a 1-1 correspondence between the DCG rules in D and the definite clauses in P, there is for any sentence s a 1-1 correspondence between a parse tree PT produced by D for s and a derivation tree DT produced by P for s. In order to show that the semantic representation computed by P for s is the same as that defined by D for s we need to show (1) that the sequence of terminals derived by DT is equal to that parsed in PT, and (2) that the semantic representation computed for the root of DT is correct with respect to the semantic representation at the root of PT.

Proof of (1): We need to prove that the sequence of terminals (leaf nodes) below each node of PT is exactly the difference between the input and output list of the corresponding literal in DT after it has been resolved. We prove this by induction on the height of the subtree corresponding to a node: It is obviously true for all nodes with subtree height 0 (leaf nodes), because leaf nodes are of the form:

$$a_0(\bar{t}, [\mathit{Word}|\mathit{Rest}\ ], \mathit{Rest}),$$

where *Word* is a word of category a_0. Induction step: Assume it is true for all nodes with subtree height n or less. Then it is also true for all nodes with subtree height $n+1$ because all clauses are of the form:

$$a_0(\bar{t}, P_0, P_k)\ \ \texttt{:-}\ \ a_1(\bar{t_1}, P_0, P_1), a_2(\bar{t_2}, P_1, P_2), \ldots, a_k(\bar{t_k}, P_{k-1}, P_k).$$

If in a particular derivation the subtree corresponding to a_0 has height n, then the subtrees corresponding to $a_1, a_2, \ldots$ will have at most height $n-1$. Therefore, by hypothesis, the difference between P_0 and P_1 is the list of words corresponding to a_1, the difference between P_1 and P_2 is the list of words corresponding to a_2, etc. Therefore, the difference between P_0 and P_k is the concatenation of the sublists corresponding to each constituent on the right-hand side.

Proof of (2): This can also be shown by induction on the length of a derivation, in a manner identical to the soundness of SLD-resolution (theorem 7.1, Lloyd 87). **Q.E.D.**

The DCG shown earlier is converted, by the transformation just described, into the following definite clause program:

```
sentence(sent(N,V), P0, P1) :- nounphrase(N, Num, P0, P2),
                               verbphrase(V, Num, P2, P1).

nounphrase(np(D,N), Num, P0, P1) :- determiner(D, Num, P0, P2),
                                    nounphrase2(N, Num, P2, P1).
nounphrase(np(N), Num, P0, P1) :- nounphrase2(N, Num, P0, P1).

nounphrase2(np2(A,N), Num, P0, P1) :- adjective(A, P0, P2),
                                      nounphrase2(N, Num, P2, P1).
nounphrase2(np2(N), Num, P0, P1) :- noun(N, Num, P0, P1).

verbphrase(vp(V), Num, P0, P1) :- verb(V, Num, P0, P1).

determiner(det(the), Num, [the|P1], P1).
determiner(det(a), singular, [a|P1], P1).

noun(n(computer), singular, [computer|P1], P1).
noun(n(computers), plural, [computers|P1], P1).

adjective(adj(super), [super|P1], P1).
adjective(adj(mini), [mini|P1], P1).
adjective(adj(micro), [micro|P1], P1).

verb(v(runs), singular, [runs|P1], P1).
verb(v(run), plural, [run|P1], P1).
```

To execute this program, one may enter a query such as:

```
(1) sentence( LF, [a,micro,computer,runs], [ ] ),
```

which matches the left-hand side (head) of the first rule, instantiating[7] `LF` to `sent(N,V)`, `P0` to `[a,micro,computer,runs]`, and `P1` to the empty list `[ ]`. Note that variables like `N` and `V` are renamed at each invocation of a rule. Next, the two subgoals on the right-hand side of that rule will be executed. When these two executions are completed, `N` will be bound to

[7] For logical variables the term "instantiating" is commonly used instead of "binding".

```
np(det(a),np2(adj(micro),np2(n(computer))))
```

and `V` will be bound to `vp(v(runs))`. So, the final binding of `LF` is the following parse tree:

```
sent(np(det(a),np2(adj(micro),np2(n(computer)))), vp(v(runs))).
```

In order to see how these variables were instantiated, we give below a trace for the above query (see explanation following this trace):

```
?- sentence(LF,[a,micro,computer,runs],[]).

   (1) 0 Call: sentence(L1,[a,micro,computer,runs],[]) ?
   (1) 1 Head [1]: sentence(L1,[a,micro,computer,runs],[]) ?
   (2) 1 Call: nounphrase(L2,N1,[a,micro,computer,runs],W1) ?
   (2) 2 Head [1->2]: nounphrase(L2,N1,[a,micro,computer,runs],W1) ?
   (3) 2 Call: determiner(L3,N1,[a,micro,computer,runs],W2) ?
   (3) 3 Head [1->2]: determiner(L3,N1,[a,micro,computer,runs],W2) ?
   (3) 3 Head [2]: determiner(L3,N1,[a,micro,computer,runs],W2) ?
   (3) 2 Done: determiner(det(a),singular,[a,micro,computer,runs],
               [micro,computer,runs]) ?
   (4) 2 Call: nounphrase2(L4,singular,[micro,computer,runs],W1) ?
   (4) 3 Head [1->2]: nounphrase2(L4,singular,[micro,computer,runs],W1) ?
   (5) 3 Call: adjective(L5,[micro,computer,runs],W3) ?
   (5) 4 Head [1->2]: adjective(L5,[micro,computer,runs],W3) ?
   (5) 4 Head [2->3]: adjective(L5,[micro,computer,runs],W3) ?
   (5) 4 Head [3]: adjective(L5,[micro,computer,runs],W3) ?
   (5) 3 Done: adjective(adj(micro),[micro,computer,runs],[computer,runs]) ?
   (6) 3 Call: nounphrase2(L6,singular,[computer,runs],W1) ?
   (6) 4 Head [1->2]: nounphrase2(L6,singular,[computer,runs],W1) ?
   (7) 4 Call: adjective(L7,[computer,runs],W4) ?
   (7) 5 Head [1->2]: adjective(L7,[computer,runs],W4) ?
   (7) 5 Head [2->3]: adjective(L7,[computer,runs],W4) ?
   (7) 5 Head [3]: adjective(L7,[computer,runs],W4) ?
   (7) 4 Fail: adjective(L7,[computer,runs],W4) ?
   (6) 4 Head [2]: nounphrase2(L6,singular,[computer,runs],W1) ?
   (8) 4 Call: noun(L7,singular,[computer,runs],W1) ?
   (8) 5 Head [1->2]: noun(L7,singular,[computer,runs],W1) ?
   (8) 4 Exit: noun(n(computer),singular,[computer,runs],[runs]) ?
   (6) 3 Exit: nounphrase2(np2(n(computer)),singular,
               [computer,runs],[runs]) ?
```

```
(4) 2 Exit: nounphrase2(np2(adj(micro),np2(n(computer))),singular,
            [micro,computer,runs],[runs]) ?
(2) 1 Exit: nounphrase(np(det(a),np2(adj(micro),np2(n(computer)))),
            singular,[a,micro,computer,runs],[runs]) ?
(9) 1 Call: verbphrase(L8,singular,[runs],[]) ?
(9) 2 Head [1]: verbphrase(L8,singular,[runs],[]) ?
(10) 2 Call: verb(L9,singular,[runs],[]) ?
(10) 3 Head [1->2]: verb(L9,singular,[runs],[]) ?
(10) 2 Exit: verb(v(runs),singular,[runs],[]) ?
(9) 1 Exit: verbphrase(vp(v(runs)),singular,[runs],[]) ?
(1) 0 Exit: sentence(sent(np(det(a),np2(adj(micro),np2(n(computer)))),
            vp(v(runs))),[a,micro,computer,runs],[]) ?

LF = sent(np(det(a),np2(adj(micro),np2(n(computer)))),vp(v(runs)))
```

The nodes of the derivation tree are numbered sequentially in the order in which they are called—we assume a Prolog-like, depth-first search strategy with backtracking. This number is indicated in this trace by the first number (given in parentheses). During backtracking, this number is decremented accordingly. The second number is the level (depth) of that node in the current derivation tree. Trace steps preceded by `Call` show the subgoal to be resolved. Steps preceded by `Head` indicate which clauses are being tried. The notation `[1->2]` means that the first clause for a particular goal is tried, and if this fails, the second clause is tried next. `Fail` means that this goal failed, and backtracking is initiated. `Done` means that this goal succeeded and there are no choice points left. `Exit` also indicates successful completion but other choice points remain in case of subsequent failure.

The reversibility property of a DCG is illustrated by the query

```
sentence(sent(np(det(a),np2(micro,np2(n(computer)))),
              vp(v(run))), P0, []),
```

which returns the sentence

```
[a,micro,computer,runs]
```

by binding this list to the variable `P0` of the query. The last argument in this query is again the empty list, since we require that the list returned by `P0` is a complete sentence; that is, there should be nothing left after it is parsed by the grammar.

4.3. Higher-Order Definite Clause Grammars

4.3.1. Higher-order definite clauses

The extension of first-order definite clauses to higher-order definite clauses is straightforward. The terms are now simply typed λ-terms, which can be considered a generalization of first-order terms. Essentially, first-order unification will be replaced by higher-order unification. The semantics of the resulting higher-order definite clause programs can be thought of as a special case of a theory of complete logic programs with equality as discussed by Jaffar, Lassez, and Maher (1984), who proved soundness and completeness for such systems. The conversion rules of the λ-calculus effectively serves as the equality theory that is of interest here. Below is a simple example of higher-order definite clause program:

Example 4.4

```
event((L2 L3 L1)) ⟵ subject(L1), predicate(L2), object(L3).
subject(julia).
object(tom).
predicate(X\Y\(hates Y X)).
predicate(X\Y\(loves Y X)).
```

Executing this program on the goal event(X) will result in one of the following two bindings for X (assuming all terms are reduced to their normal forms).

(hates julia tom)
(loves julia tom)

4.3.2. Higher-order DCGs

Higher-order DCGs are a generalization of first-order DCGs in the same way as higher-order definite clause programs are a generalization of first-order definite clause programs; that is, the terms that grammar symbols can have as arguments are now terms of the simply typed λ-calculus. We illustrate below the use of higher-order DCGs for translating sentences into their logical form (semantic representation). Consider the sentence:

Every man loves a woman.

In first-order logic, the semantic representation of this sentence might be:

$$\forall X(manp(X) \supset \exists Y(womanp(Y) \wedge lovesp(X,Y)))^{8}$$

The λ-calculus representation is very similar to that of first-order logic, except that quantifiers are treated as non-logical constants and are separated from their variables:

```
(all X\(implies (manp X)
                (exists Y\(and (womanp Y) (lovesp X Y)))))
```

The following is a higher-order DCG that synthesizes such logical forms for the sentences that it can parse.

```
sentence((P1 P2))  -->  np(P1),  vp(P2).

np((P1 P2))  -->  det(P1),  noun(P2).
np(P)  -->  prop_noun(P).

vp(X\(P2 (P1 X)))  -->  trans_verb(P1),  np(P2).

det(P1\P2\(all X\(implies (P1 X) (P2 X))))  -->  [every].
det(P1\P2\(exists X\(and (P1 X) (P2 X))))  -->  [a].

noun(manp)  -->  [man].
noun(womanp)  -->  [woman].

prop_noun(P\(P johnp))  -->  [john].
prop_noun(P\(P maryp))  -->  [mary].

trans_verb(lovesp)  -->  [loves].
```

As in the case of first-order DCGs, each grammar symbol must be extended with two more arguments for manipulating the sentence and its components, so that it can be executed as a definite clause program. Therefore, the above higher-order DCG would be converted into the set of clauses listed below.

```
sentence((P1 P2),In,Out)  :-  np(P1,In,J),  vp(P2,J,Out).
```

[8]The p-suffix is a standard way of naming predicate symbols in Lisp. In the above formula *manp*, *womanp*, and *lovesp* denote predications on individuals. It also emphasizes that these are arbitrary symbols that have nothing to do with the words 'man', 'woman', and 'loves'.

```
np((P1 P2),In,Out)  :-  det(P1,In,J),  noun(P2,J,Out).
np(P,In,Out)  :-  prop_noun(P,In,Out).

vp(X\(P2 (P1 X)),In,Out)  :-  trans_verb(P1,In,J),
                                np(P2,J,Out).

det(P1\P2\(all X\(implies (P1 X) (P2 X))), [every|Out], Out).
det(P1\P2\(exists X\(and (P1 X) (P2 X)))), [a|Out], Out).

noun(manp, [man|Out], Out).
noun(womanp, [woman|Out], Out).

prop_noun(P\(P johnp), [john|Out], Out).
prop_noun(P\(P maryp), [mary|Out], Out).

trans_verb(lovesp, [loves|Out], Out).
```

An example of a query for this program would be

```
sentence(L, [john,loves,a,woman], []).
```

The execution of the program using this query would instantiate `L` to

```
(exists X\(and (womanp X) (lovesp johnp X))).
```

We explain below how this answer is obtained. The query

```
sentence(L, [john,loves,a,woman], []).
```

matches the head (left-hand side) of the first rule:

```
(1) sentence((P1 P2), In, Out)  :-  np(P1, In, J), vp(P2, J, Out).
```

Therefore, this rule is activated and its body (right-hand side) executed (note that all variables of a rule are appropriately renamed for each activation). We therefore obtain the following instantiation of this rule:

```
(1) sentence((P1 P2), [john,loves,a,woman], [])  :-
       np(P1, [john,loves,a,woman], J),  vp(P2, J, []).
```

Next, the subgoal:

```
(2) np(P1, [john,loves,a,woman], J)
```

is executed. This matches both rules for `np`; the first rule is initially tried (shown below is its instantiation):

```
(3) np((P1 P2), [john,loves,a,woman], Out)  :-
       det(P1, [john,loves,a,woman], J),  noun(P2, J, Out).
```

The next subgoal to be executed therefore is:

```
(4) det(P1, [john,loves,a,woman], J).
```

There are two facts for `det`. The first one requires that the first word of the input list (second argument) is `every`; the second one requires that the first word is `a`. Neither of them matches the word `john`, the first word of the input list. Therefore, subgoal `(4)` fails. Since *all* subgoals of a rule must succeed in order for the head of the rule to succeed, `(3)` fails as well. Therefore, the second rule for `np` is instantiated next:

```
(5) np(P, [john,loves,a,woman], Out)  :-
       prop_noun(P, [john,loves,a,woman], Out).
```

The next subgoal would be:

```
(6) prop_noun(P, [john,loves,a,woman], Out)
```

which matches the first fact for `prop_noun`:

```
   prop_noun(P\(P johnp), [john|Out], Out)

==> (6) prop_noun(P\(P johnp), [john,loves,a,woman],
                                      [loves,a,woman])

==> (5) np(P\(P johnp), [john,loves,a,woman], [loves,a,woman]) :-
           prop_noun(P\(P johnp), [john,loves,a,woman],
                                        [loves,a,woman]).

==> (2) np(P\(P johnp), [john,loves,a,woman], [loves,a,woman])

==> (1) sentence((P\(P johnp) P2), [john,loves,a,woman], [])  :-
           np(P\(P johnp), [john,loves,a,woman], [loves,a,woman]),
           vp(P2, [loves,a,woman], []).
```

Now the second subgoal of (1) is activated:

```
 (7) vp(P2, [loves,a,woman], []).
```

The instantiated rule for vp is as follows:

```
 (8) vp(X\(P2 (P1 X)),[loves,a,woman],[])  :-
        trans_verb(P1,[loves,a,woman],J), np(P2,J,[]).
```

Therefore, the next subgoal is:

```
 (9) trans_verb(P1,[loves,a,woman],J).
```

This process continues by activating the fact for trans_verb:

```
(10) trans_verb(lovesp, [loves|Out], Out)

 ==> (9) trans_verb(lovesp, [loves,a,woman], [a,woman]).

 ==> (8) vp(X\(P2 (lovesp X)),[loves,a,woman],[])  :-
            trans_verb(lovesp,[loves,a,woman],[a,woman]),
            np(P2,[a,woman],[]).
```

The second subgoal of (8) now is:

```
(11) np(P2,[a,woman],[])
```

The first rule for np is now instantiated:

```
(12) np((P1 P2),[a,woman],[])  :-  det(P1,[a,woman],J), noun(P2,J,[]).
```

The next subgoal is:

```
(13) det(P1,[a,woman],J)
```

Only the second rule for det matches, because the first word of the input sentence is a:

```
(13) det(P1\P2\(exists X\(and (P1 X) (P2 X))), [a,woman], [woman]).

(12) np((P1\P2\(exists X\(and (P1 X) (P2 X))) P2),[a,woman],[])  :-
        det(P1\P2\(exists X\(and (P1 X) (P2 X))),[a,woman],[woman]),
        noun(P2,[woman],[]).
```

The last subgoal thus is:

```
(14) noun(P2,[woman],[]).
```

This goal instantiates the rule:

```
(15) noun(womanp, [woman|[]], []).

 ==> (14) noun(womanp,[woman],[]).

(12) np((P1\P2\(exists X\(and (P1 X) (P2 X))) womanp), [a,woman],[]) :-
        det(P1\P2\(exists X\(and (P1 X) (P2 X))),[a,woman],[woman]),
        noun(womanp,[woman],[]).
```

The first argument of `np` in `(12)` can be reduced as follows:

```
 ==> (12) np(P2\(exists X\(and (womanp X) (P2 X))),[a,woman],[])  :-
            det(P1\P2\(exists X\(and (P1 X) (P2 X))),[a,woman],[woman]),
            noun(womanp,[woman],[]).

 ==> (8) vp(X\(P2\(exists Y\(and (womanp Y) (P2 Y))) (lovesp X)),
                                          [loves,a,woman],[])  :-
            trans_verb(lovesp,[loves,a,woman],[a,woman]),
            np(P2\(exists X\(and (womanp X) (P2 X))),[a,woman],[]).
```

where variables have been renamed appropriately. The first argument of `vp` can likewise be reduced:

```
 ==> (8) vp(X\(exists Y\(and (womanp Y) (lovesp X Y))),
                                          [loves,a,woman],[])  :-
            trans_verb(lovesp,[loves,a,woman],[a,woman]),
            np(P2\(exists X\(and (womanp X) (P2 X))),[a,woman],[]).

 ==> (1) sentence((P\(P johnp) X\(exists Y\(and (womanp Y)
                                              (lovesp X Y)))),
                                    [john,loves,a,woman], [])  :-
            np(P\(P johnp), [john,loves,a,woman], [loves,a,woman]),
            vp(X\(exists X\(and (womanp X) (lovesp X))) (lovesp X)),
                                          [loves,a,woman], []).
```

Applying two β-reductions to the first argument yields the final semantic representation for the complete sentence ("john loves a woman"):

```
   (P\(P johnp) X\(exists Y\(and (womanp Y) (lovesp X Y))))
 = (X\(exists Y\(and (womanp Y) (lovesp X Y))) johnp)
 = (exists Y\(and (womanp Y) (lovesp johnp X)))
```

5. Synthesizing Higher-Order DCGs from Examples

This chapter forms the core of this dissertation: it presents the basic technique for generalizing semantic representations from examples. Section 5.1 describes the general technique; variations of this scheme are possible, and some are discussed later. I describe this technique in terms of four procedures, called SYNTH, SOLVE, SUBST, and CHECK. SYNTH is the main routine, which collects the examples and passes on to SOLVE a resulting set of higher-order equations. This set is solved by calling SUBST, which determines the next substitution to be performed. Procedure CHECK decomposes equations into simpler equations where possible, and determines whether the set of equations derived so far is consistent. Section 5.2 illustrates different features of the technique with the aid of two small examples. Section 5.3 discusses the notion of compositionality, which is crucial for the applicability of this technique; and Section 5.4 discusses the implications when multiple solutions exist for the set of higher-order equations.

5.1. Basic Technique

Below is the pseudo-code for procedure SYNTH, which takes as input a CFG along with a set of examples and returns a higher-order DCG. I assume, for simplicity of presentation, that a CFG rule has either a single terminal on its right-hand side or a sequence of one or more nonterminals (in practice, both terminals and nonterminals are permitted on the right-hand side).

5.1.1. Procedure SYNTH(G)

1. Let **G** be an unambiguous CFG having n rules, with start symbol s.

2. Construct the higher-order DCG as follows:
   If the i-th CFG rule is a_i --> $b_{i1} \ldots b_{ik_i}$, the i-th DCG rule will be

   $$a_i((F_i\ V_1 \ldots V_{k_i})) \texttt{ --> } b_{i1}(V_1),\ \ldots\ b_{ik_i}(V_{k_i}),$$

   where F_i is a function variable and each V_i is a variable.
   If the i-th CFG rule is a_i --> $[t]$, the i-th DCG rule will be

   $$a_i(F_i) \texttt{ --> } [t].$$

3. Determine the values for the function variables F_i in the above DCG as follows.

$E \leftarrow \phi$; *done* $\leftarrow$ *false*;

WHILE *not done* **DO**

a. Generate a set of new sentences se_i, $1 \leq i \leq k$, for some finite k (selection strategy for these sentences is discussed in Chapter 6). Query the user for the semantic representation of each se_i; let the user's input be n_i, assumed to be a typed λ-term.[9]

b. Execute the goal $s(M, se_i, [\,])$, $1 \leq i \leq k$, using the constructed DCG of step 2, but treating each F_i as a constant (the F_i are existential variables across the entire DCG). Let m_i, $1 \leq i \leq k$, be the term computed for variable M.

c. $E \leftarrow E \cup \{m_i = n_i | 1 \leq i \leq k\}$

d. Call SOLVE(E) to solve for the function variables F_i (see next section). In general, SOLVE may produce multiple maximally general solutions in case it succeeds. Assign *done* $\leftarrow$ *true* if *either* unification fails, *or* unification succeeds and all sentences of the CFG have been enumerated, *or* unification succeeds and the user accepts the resulting DCG after replacing all variables F_i in the DCG of step 2 according to one of the unifiers of E and reducing all λ-terms to their normal forms.

END WHILE

4. If unification failed in step **3d**, print "no solution", else print the DCG found.

5.1.2. Procedure SOLVE(E)

Procedure SOLVE tries to solve the set of higher-order equations $E \leftarrow \mathbf{E}$ by attempting to find substitutions for the free variables occurring in E.

1. Let F be the set of function variables (F_i) occurring in E.

2. Let $\sigma \leftarrow \emptyset$, the empty substitution.

3. WHILE $E \neq \emptyset$ **DO**

[9] Chapter 8 discusses a scheme for automatic type assignment that allows the user to enter untyped λ-terms as semantic representations

a. Select an equation e_1 from E, and call SUBST(e_1). If SUBST succeeds, it returns a substitution term t for the variable V at the head position of the left-hand side term of e_1.

b. $\sigma \leftarrow \sigma\{\langle V, t\rangle\}$ (composition of substitutions).

c. $E \leftarrow$ CHECK($E\sigma$) (see below).

END WHILE

4. Return $\sigma \uparrow F$ (the *restriction* of σ to F).

5.1.3. Procedure SUBST(e)

Procedure SUBST selects a substitution for the head of the left-hand side term of the input equation in the following way. Let $e \leftarrow \mathbf{e} = \langle e_1, e_2\rangle$, where

$$e_1 = \lambda u_1, \ldots, \lambda u_n.(f\ g_1\ g_2\ \ldots\ g_p),$$
$$e_2 = \lambda v_1, \ldots, \lambda v_m.(@\ h_1\ h_2\ \ldots\ h_q),$$

and $m \geq n$ (in order for a unifying substitution to exist). First, η-expand e_1 as follows:

$$e_1 \leftarrow \lambda u_1, \ldots, \lambda u_n.\lambda u_{n+1}, \ldots \lambda u_m.(f\ g_1\ g_2\ \ldots\ g_p\ u_{n+1}\ u_m).$$

1. Nondeterministically select a substitution σ according to the following options:

   a. *Projection* substitutions. Projection substitutions are of the form:
   $f \leftarrow \lambda w_1 \ldots . \lambda w_k.(w_i\ (h_1\ w_1\ \ldots\ w_k)\ \ldots\ (h_l\ w_1\ \ldots\ w_k))$, for each $1 \leq i \leq k$, where the type of f, $\tau(f) = \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_k \rightarrow \beta$, and $\tau(w_i) = \alpha_i$ (the α_i are type variables). The following parameters are chosen nondeterministically: (i) k, the number of prefix variables; (ii) w_i, the head of the substitution term; (iii) l, the number of arguments of w_i.

   b. *Imitation* substitution. The applicable imitation substitution would be:
   $f \leftarrow \lambda w_1 \ldots \lambda w_p.\lambda v_{n+1} \ldots \lambda v_m.(@\ E_1\ E_2\ \ldots\ E_q)$, where
   $E_i = (h_i\ w_1\ \ldots\ w_p\ v_{n+1}\ \ldots\ v_m)$, for $1 \leq i \leq q$, and $\tau(f) = \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_{p+m-n} \rightarrow \beta$. The type of the constant @ in the substitution term must be the same as the type of the atom @ in e_2.

2. Return the selected substitution σ.

5.1.4. Function CHECK(E)

The function CHECK takes as input a set of equations E, and is defined as follows. Types are assumed to be maintained in a global environment that is updated by procedure unify(x,y). A sequence of prefix variables x_i is denoted by $\vec{x}$.

(1) If $E = \emptyset$ then return $\emptyset$.

(2) If $E = \{\langle t_1, t_2 \rangle\}$, unify($\tau(t_1)$,$\tau(t_2)$), and

  (a) if t_1 is flexible then return E;

  (b) otherwise t_1 and t_2 are both rigid. Let $t_1 = \lambda\vec{x}.(@_1\ A_1\ \ldots\ A_r)$ and let $t_2 = \lambda\vec{x}.(@_2\ B_1\ \ldots\ B_s)$. If $@_1 \neq @_2$ then fail; otherwise unify($\tau(A_i)$,$\tau(B_i)$) for $1 \leq i \leq r$, and return CHECK($\{\langle \lambda\vec{x}.A_i, \lambda\vec{x}.B_i \rangle | 1 \leq i \leq r\}$).

(3) Otherwise E has more than one equation. Let $E = \{\langle F_i, G_i \rangle | 1 \leq i \leq n\}$.

  (a) if CHECK($\{\langle F_i, G_i \rangle\}$) fails for some i, then CHECK(E) fails;

  (b) otherwise return $\bigcup_{i=1}^{n}$ CHECK($\{\langle F_i, G_i \rangle\}$).

5.2. Two Examples of Synthesis

5.2.1. The Successor Function

Below is a very simple grammar which generates sentences of the form `[a]`, `[a,a]`, `[a,a,a]`, etc. The derivation of a higher-order DCG by procedure SYNTH is illustrated as follows.

(Step **1.**) The input CFG is:

```
s → [a].
s → [a], s.
```

(Step **2.**) The CFG augmented with function variables is:

```
s(F1) → [a].
s(F2 A) → [a], s(A).
```

(Step **3a.**) Suppose we wanted the following semantics: `[a]` means 0; `[a,a]` means 1; `[a,a,a]` means 2; and so on, the meaning of a sequence of length n is the number $n - 1$. Suppose further that we use *Church numerals* (see section 3.2) to encode these numbers: `0 = F\X\X`, `1 = F\X\(F X)`, `2 = F\X\(F (F X))`, etc. We will see that the desired DCG can be obtained with just three examples:

```
[a],
[a,a],
[a,a,a]
```

Next, the user is asked for the corresponding semantic representations, which would be respectively:

```
F\X\X,
F\X\(F X),
F\X\(F (F X))
```

(Step **3b.**) Executing the above DCG on the sentence `[a]`, the constructed semantic representation will be `F1`.

(Step **3c.**) Therefore, the equation

```
(e0)   F1 = F\X\X
```

is added to E. Similarly, the following equations are added to E:

```
(e1)   (F2 F1) = F\X\(F X)
(e2)   (F2 (F2 F1)) = F\X\(F (F X))
```

(Step **3d.**) The higher-order unification procedure SOLVE is called next. For readability I indicate types only for selected terms and variables in the derivation below (only one elementary type i is used). In this example types are provided by the user as follows: τ(`F\X\X`) = $(i \rightarrow i) \rightarrow i \rightarrow i$, τ(`F\X\(F X)`) = $(i \rightarrow i) \rightarrow i \rightarrow i$, and τ(`F\X\(F (F X))`) = $(i \rightarrow i) \rightarrow i \rightarrow i$.

Given the initial set of equations,

```
{F1 = F\X\X,
 (F2 F1) = F\X\(F X),
 (F2 (F2 F1)) = F\X\(F (F X))}
```

the types of the other terms are inferred in the following way: `F1`, `(F2 F1)`, and `(F2 (F2 F1))` must have type $(i \rightarrow i) \rightarrow i \rightarrow i$ as well. Since we now know the types of both `F1` and `(F2 F1)`, the type for `F2` is inferred as τ(`F2`) = $((i \rightarrow i) \rightarrow i \rightarrow i) \rightarrow ((i \rightarrow i) \rightarrow i \rightarrow i)$, which, due to the right-associativity of the $\rightarrow$ operator, is the same as $((i \rightarrow i) \rightarrow i \rightarrow i) \rightarrow (i \rightarrow i) \rightarrow i \rightarrow i$.

The following projection substitution immediately solves the first equation:

```
F1 <- K\L\L
```

The remaining equations to be solved are:

```
{(F2 K\L\L) = F\X\(F X),
 (F2 (F2 K\L\L)) = F\X\(F (F X))}
```

The following projection substitution is next attempted for `F2`:

```
F2 <- K\L\M\(K (H2 K L M) (H1 K L M))
```

This yields the following reduced equation set:

```
{A\B\(H1 K\L\L A B) = F\X\(F X),
 A\B\(H1 K\L\L (H2 K\L\(H1 M\N\N K L) A B)
               (H1 K\L\(H1 M\N\N K L) A B)) = F\X\(F (F X))}
```

Next, the following projection substitution is chosen for `H1`:

```
H1 <- K\L\L
```

which solves the first equation, so that the only remaining equation is:

```
{A\B\(H2 K\L\(K L) A B (A B)) = F\X\(F (F X))}
```

The correct projection substitution for `H2` now is:

```
H2 <- K\L\M\N\(L (H3 K L M N))
```

leading to:

```
{A\B\(H3 K\L\(K L) A B (A B)) = F\X\(F X)}
```

Finally, the substitution

```
H3 <- K\L\M\N\N
```

solves the remaining equation. After substituting and reducing all terms, the final substitutions are:

```
F1 <- K\L\L
F2 <- K\L\M\(K L (L M))
H1 <- K\L\L
H2 <- K\L\M\N\(L N)
H3 <- K\L\M\N\N
```

Procedure SOLVE returns the substitutions for `F1` and `F2` to procedure SYNTH:

(Step **4.**) The resulting (higher-order) DCG is:

```
s(A\B\B) --> [a].
s(A\B\C\(A B (B C)) D) --> [a], s(D).
```

where the term `A\B\C\(A B (B C))` in the second rule essentially performs the successor operation. For example, in order to parse the sentence `[a,a]`, the second rule is invoked first, which then calls the first rule instantiating `D` to `A\B\B`. Therefore, the argument of the head of the second rule becomes

```
  (A\B\C\(A B (B C))  A\B\B)
= B\C\(D\E\E B (B C))
= B\C\(B C)
```

which is the Church numeral for the number `1`.

Even though this is a very simple example, it shows how an infinite language can be inferred from a few examples. This small example makes use only of projection substitutions. The next example involves both projection and imitation substitutions.

5.2.2. Search Constraints through Multiple Examples

The example below illustrates how multiple examples constrain the search space. It also demonstrates the use of the imitation substitution rule and the type enumeration scheme.

(Step **1.**) The CFG is given by the following rules:

```
s → pn, iv.
pn → [shrdlu].
pn → [eliza].
iv → [runs].
iv → [halts].
```

(Step **2.**) The rules are augmented with function variables as follows:

```
s((F1 A B)) → pn(A),iv(B).
pn(F2) → [shrdlu].
```

```
pn(F3) → [eliza].
iv(F4) → [runs].
iv(F5) → [halts].
```

(Step **3a.**) Using the CFG from step 1, the system generates the following training sentences: `[shrdlu,runs]`, `[eliza,runs]`, and `[shrdlu,halts]`, for which the user provides the corresponding semantic representations: `(run shrdlu)`, `(run eliza)`, and `(halt shrdlu)`, where τ(`run`) = $i \rightarrow i$, τ(`halt`) = $i \rightarrow i$, τ(`shrdlu`) = i, and τ(`eliza`) = i.

(Step **3b.**) Executing each of these sentences on the skeletal DCG of step 2, the following terms are respectively obtained: `(F1 F2 F4)`, `(F1 F3 F4)`, and `(F1 F2 F5)`.

(Step **3c.**) The set of higher-order equations is as follows (for readability I indicate the types only selectively in this derivation):

```
{(F1 F2 F4) = (run shrdlu),
 (F1 F3 F4) = (run eliza),
 (F1 F2 F5) = (halt shrdlu)}.
```

(Step **3d.**) This set of equations is passed on to procedure SOLVE:

Since `F1` is at the head position of the first equation, a substitution is sought for `F1`. It can be easily seen that an imitation substitution won't work. There is only one applicable imitation substitution: `K\L\(run (H1 K L))`. However, applying this substitution in the third equation will put the constant `run` at the head position of the left-hand side term which is incompatible with the head of the right-hand side term. Thus, a projection substitution must be attempted. The simplest projection substitutions in this case would be `K\L\L` or `K\L\K`, both of which would lead to failure because they would require that either `F4` or `F2` will be bound to two different ground terms. The correct projection substitution is:

```
F1 <- K\L\ (L (H1 K L))
```

The type of `F1` would be $\alpha_1 \rightarrow \alpha_2 \rightarrow i$, where α_1 is the type of `F2`, and α_2 the type of `F4`. Both α_1 and α_2 are variables at this stage that will be instantiated later.

Replacing all occurrences of `F1` in the above equations with its substitution and simplifying leads to the following set of equations:

```
{(F4 (H1 F2 F4)) = (run shrdlu),
 (F4 (H1 F3 F4)) = (run eliza),
 (F5 (H1 F2 F5)) = (halt shrdlu)}
```

Now F4 is the head of the first equation and we need to find a substitution for it. The following imitation substitution is applicable:

```
F4 <- K\ (run (H2 K))
```

The type α_2 of F4 therefore is $\alpha_3 \rightarrow i$, which further instantiates the type of F1 to $\alpha_1 \rightarrow (\alpha_3 \rightarrow i) \rightarrow i$. The type of the argument (H2 K) of run is also inferred at this point: it must be of the same type as the corresponding argument of run in the right-hand side terms, namely i.

Replacing all occurrences of F4 by its substitution and simplifying results in the following set of equations:

```
{(run (H2 (H1 F2 K\(run (H2 K))))) = (run shrdlu),
 (run (H2 (H1 F3 K\(run (H2 K))))) = (run eliza),
 (F5 (H1 F2 F5)) = (halt shrdlu)}
```

Applying CHECK to the above equation set we get:

```
{(H2 (H1 F2 K\(run (H2 K)))) = shrdlu,
 (H2 (H1 F3 K\(run (H2 K)))) = eliza,
 (F5 (H1 F2 F5)) = (halt shrdlu)}
```

Choosing projection substitution K\K for H2 transforms the equations to:

```
{(H1 F2 run) = shrdlu,
 (H1 F3 run) = eliza,
 (F5 (H1 F2 F5)) = (halt shrdlu)}
```

Likewise, the projection substitution K\L\K for H1 yields:

```
{F2 = shrdlu,
 F3 = eliza,
 (F5 F2) = (halt shrdlu)}
```

This implies that both F2 and F3 are of type i, which implies H1 is of type $i \rightarrow i \rightarrow i$. This in turn instantiates the type of H2 to $i \rightarrow i$, and the type of F4 to $i \rightarrow i$. Therefore, F1 will have type $i \rightarrow (i \rightarrow i) \rightarrow i$. The obvious choices for F2 and F3 now are shrdlu and eliza, respectively, which leaves only one equation:

```
{(F5 shrdlu) = (halt shrdlu)}
```

The type of F5 is easily inferred to be $i \rightarrow i$. F5 will be replaced by K\(halt (H3 K)):

```
{(H3 shrdlu) = (shrdlu)}
```

The projection substitution K\K for H3 completes the derivation. The final substitutions with their types are:

F1:$(i \rightarrow (i \rightarrow i) \rightarrow i)$ = K\L\(L K)
F4:$(i \rightarrow i)$ = run
H2:$(i \rightarrow i)$ = K\K
H1:$(i \rightarrow i)$ = K\L\K
F2:i = shrdlu
F3:i = eliza
F5:$(i \rightarrow i)$ = halt
H3:$(i \rightarrow i)$ = K\K

(Step 4.) Substituting these in the grammar from step 2 yields the following higher-order DCG:

```
s((K\L\(L K) A B)) → pn(A), iv(B).
pn(shrdlu) → [shrdlu].
pn(eliza) → [eliza].
iv(run) → [runs].
iv(halt) → [halts].
```

In this example three training instances (higher-order equations) are sufficient to guarantee a unique solution, where unique means that the DCGs corresponding to each solution of the set of equations are equivalent in terms of input/output behavior.

5.3. Compositionality

5.3.1. Significance of Compositionality

It has been generally recognized that compositionality plays an important role in language semantics, and the technique discussed in this dissertation exploits compositionality to generalize semantic representations from examples. However, until recently, the

notion of compositionality was mostly intuitively defined as some functional dependence of the meaning of an expression on the meanings of its parts. But, as pointed out first by van Benthem (1986) and later by Zadrozny (1992), this definition is meaningless if there are no restrictions imposed on the types of functions being used for computing the meaning of an expression from the meanings of its parts. One can always find such functions, no matter what the meanings of the whole expression and its parts are.

Theorem (Zadrozny 1992) **:** Let L be a language and G_L a grammar describing how the sentences of L are composed from phrases and words. If m is a function that maps each sentence of L and its parts (phrases and words) on to some semantic representation (meaning), then there exists a function μ which computes the meaning of any sentence or phrase from the meanings of its parts.

Meaningful restrictions would be, for example, allowing only polynomial functions of a certain degree, or functions that can be expressed in the typed λ-calculus. Such restrictions are not only natural for certain domains, but they also allow a unique (presumably the correct) compositional semantics to be defined by specifying relatively few values (examples). Computability by itself would not be a meaningful restriction, because, if the meaning function is computable, then the corresponding composition function is computable as well (Zadrozny 1992).

The following example (due to Zadrozny) illustrates how appropriate restrictions can make the inference of a composition function tractable:

```
N --> N, D.
N --> D.
D --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.
```

For this grammar, the meaning of any numeral can be expressed as a polynomial in two variables with coefficients in natural numbers:

$\nu(\mathtt{N\ D}) = 10 * \nu(\mathtt{N}) + \nu(\mathtt{D})$

By restricting the meaning functions to polynomials of degree 1, a unique compositional semantics is defined by specifying the semantics for three sentences.

In this dissertation, the typed λ-calculus has been used for representing and composing meanings. That is, we have restricted the possible meaning functions to those that can be expressed by the typed λ-calculus. In general, any such restriction will work well only for limited domains. For example, polynomials may be suitable for certain aspects of mathematics, and the typed λ-calculus for certain aspects of natural language.

In addition, when we talk about compositional semantics, we expect the function that composes the meanings to be "simple" or easily definable.

The following example illustrates how our intuitive understanding of compositionality for certain aspects of natural language can be expressed in terms of boolean operations on sets. The grammar rule

```
NP --> Adj, N
```

can be interpreted compositionally by mapping adjectives and nouns into sets of objects with the corresponding properties, and by mapping concatenation into set intersection. This would provide the intuitively correct semantics for expressions such as "tall tree" or "blue bag". In the typed λ-calculus, sets can be expressed as terms of the form `X\(property X)`. For example, `X\(blue X)` would represent the set of all blue objects, and the set of all objects that are bags would be represented by `X\(bag X)`. These two sets can be intersected by combining the corresponding terms such that the scope of their variables includes both terms, e.g., `X\(and (bag X) (blue X))`.

However, in order to use such restrictions, it is sometimes necessary to enlarge the set of syntactic constructions (grammar rules) to distinguish those cases for which no single function (representable in the chosen formalism) exists which can handle all cases. For example, I have found that often certain grammar rules must be added in order to use the typed λ-calculus for semantic representations. Increasing the vocabulary poses a problem only if the semantic representations associated with the additional words are of different types than the words already in the grammar, in which case operations such as *type raising* are needed. These issues are discussed further in Chapter 8.

5.3.2. Can Compositionality be Expected?

As mentioned earlier, any grammar is compositional with respect to any semantics if arbitrary functions are allowed for computing the meaning of an expression from the meanings of its parts. Compositionality becomes meaningful only if the semantics is expressed through relatively simple functions such as those expressible by the typed λ-calculus. In order for a solution to exist for a particular synthesis problem, the grammar must be compositional with respect to the typed λ-calculus; that is, the composition functions must be expressible in the typed λ-calculus. Even though in many cases there is more than one CFG to express a particular language, in general the most natural grammar is one that possesses the kind of simple compositionality discussed above.

For example consider the language of arithmetic expressions containing expressions such as `12+3`, `78-413*9`, `67/5+851*10`, etc. This language could be generated by the grammar:

```
expr --> digits.
expr --> digits, a, digits.
a --> op.
a --> op, digits, op.
digits --> digit.
digits --> digits, digit.
digit --> [0] | [1] | [2] | [3] | ... | [9].
op --> [+] | [-] | [*] | [/].
```

Assuming the semantics of such an expression is its numeric value, a semantic augmentation for the above grammar would be hard to find. However, if the same language is defined by the following grammar, which decomposes expressions into meaningful subexpressions taking into account the precedence of operators, a semantic augmentation can be found quite easily.

```
expr --> term.
expr --> term, add_op, term.
term --> number.
term --> number, mult_op, term.
number --> digit.
number --> digit, number.
digit --> [0] | [1] | [2] | [3] | ... | [9].
add_op --> [+] | [-].
mult_op --> [*] | [/].
```

Since it can be assumed that grammar writers are familiar with the semantics of the corresponding languages, it would be natural for them to write grammars that are compositional in the sense discussed above.

5.4. Multiple Solutions

In general, the set of higher-order equations generated from a particular CFG and a set of training sentences has many solutions. However, some of these solutions may be

equivalent in the sense that the resulting DCGs have the same input/output behavior; that is, even if two DCGs are not identical, they may still produce the same semantic representations for all the sentences accepted by the grammar.

5.4.1. Equivalent solutions

Consider the following CFG and sentence-meaning pairs:

```
s --> pn, vp.
vp --> tv, pn.
pn --> [mike].
pn --> [mary].
pn --> [john].
tv --> [saw].
tv --> [visited].
```

Sentence	*Semantic representation*
[mike,saw,mary]	(saw mike mary)
[john,saw,mary]	(saw john mary)
[mike,visited,mary]	(visited mike mary)
[mike,saw,john]	(saw mike john)

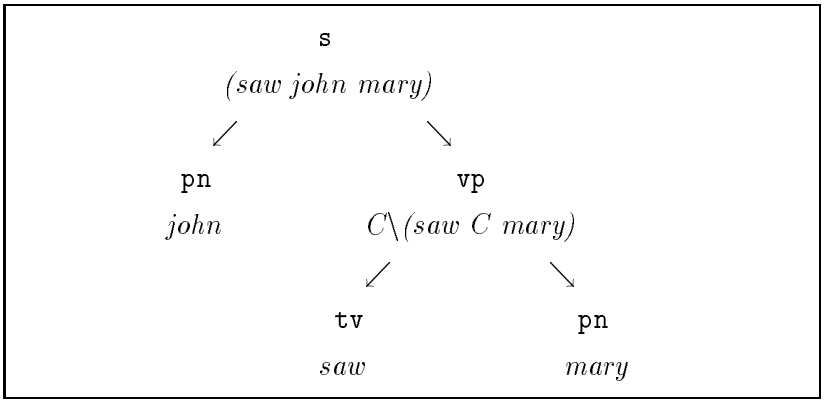
The following two DCGs can be derived from the above CFG and training instances:

(1)
```
s((D C)) --> pn(C), vp(D).
vp(C\(D C E)) --> tv(D), pn(E).
pn(mike) --> [mike].
pn(mary) --> [mary].
pn(john) --> [john].
tv(saw) --> [saw].
tv(visited) --> [visited].
```

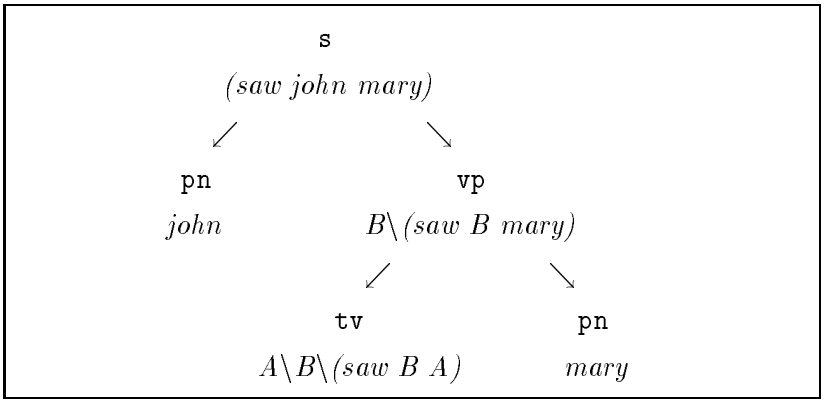
(2)
```
s((D C)) --> pn(C), vp(D).
vp((B C)) --> tv(B), pn(C).
pn(mike) --> [mike].
pn(mary) --> [mary].
pn(john) --> [john].
tv(A\B\(saw B A)) --> [saw].
tv(A\B\(visited B A)) --> [visited].
```

The difference between the two DCGs is that the arguments of the semantic representations of the verbs are in a different order (note that `saw` can be expanded to `A\B\(saw A B)` through η-conversion). This is compensated for by appropriately modifying the semantics of the verb phrase rule, as can be seen in the following derivations:

Using DCG `(1)`:



Using DCG `(2)`:



Even though the semantic representation of `tv` is *saw* in DCG `(1)` and *A\B\(saw B A)* in DCG `(2)`, the semantic representations of the complete sentence, `[john saw mary]`, are identical in both cases, namely *(saw john mary)*.

5.4.2. Solutions leading to distinct DCGs

If the problem is underconstrained, that is, if insufficient training instances are provided, there may be several solutions which lead to DCGs that do not compute the same semantic representations for all sentences of the language. For example, consider the DCG for the successor function mentioned in section 5.2.:

```
s --> [a].
```

```
s --> [a], s.
```

If only the following two training instances are used, both DCGs listed below can be derived.

Sentence	*Semantic representation*
[a]	F\X\X
[a,a]	F\X\(F X)

```
(1)    s(A\B\B) --> [a].
       s(A\B\C\(B C) D) --> [a],s(D).

(2)    s(A\B\B) --> [a].
       s(A\B\C\(A B (B C)) D) --> [a],s(D).
```

Even though both DCGs compute the same semantic representations for the two training sentences given above, they compute different semantic representations for the sentences `[a,a,a]`, `[a,a,a,a]`, etc.: DCG `(1)` will compute `B\C\(B C)` for all such sentences, whereas DCG `(2)` computes `F\X\(F (F X))`, `F\X\(F (F (F X)))`, etc. That is, DCG `(1)` effectively computes a constant function, whereas DCG `(2)` computes a linear function.

Section 6.1. discusses criteria that can be used to determine whether a set of training sentences is complete, that is, whether all DCGs that can be derived from them by SYNTH are equivalent.

5.5. Summary

This chapter has described the basic procedure for generating a higher-order DCG from a CFG and sample sentence-meaning pairs. The main steps of this synthesis procedure involve generating a set of representative sentences from the input CFG, asking the user for their semantic representations, forming a set of higher-order equations over function variables, and using the solution for these variables in order to construct a higher-order DCG. This chapter has also discussed the significance and plausibility of compositionality. When applying this technique to grammars more complex than those mentioned in this chapter, various optimizations are required to contain the nondeterminism inherent in the higher-order unification procedure. Such optimizations are discussed in Chapter 6. Chapter 7 is concerned with efficient execution of the synthesized DCGs.

6. Efficient Synthesis

The previous chapter introduced the basic technique for synthesizing definite clause grammars. This chapter investigates various techniques for increasing the efficiency of the synthesis process. Not surprisingly, choosing the right training sentences in the right order can be crucial for achieving acceptable performance. I will show that significant improvements in performance can also be achieved by adjusting the basic higher-order unification procedure to our problem domain, and by using a suitable search strategy.

6.1. Generation of Training Instances

The objective of this section is to develop criteria that can be used to determine whether a given set of training instances is "complete" for a given CFG, that is, whether all higher-order DCGs that can be inferred from these training instances and the given CFG implement the same sentence-meaning function. Even though training instances can be provided incrementally, it is in general more efficient to process all training instances simultaneously. However, using more training instances than necessary is counterproductive, since additional computations would have to be performed without further reducing the search space. Therefore, it is desirable to have criteria for determining a minimal set of sentences such that their semantic representations induce a unique solution.

The underlying assumption in this section is that for any given CFG and sentence-meaning function, there exists a DCG that computes the correct semantic representations for all sentences of the language. Thus, the question is how many and which training sentences are required to uniquely determine such an augmentation. First I show that, even for CFGs generating infinite languages, there is a finite set of training instances guaranteeing a unique solution. Then I discuss how many training instances one can expect to be sufficient for a unique solution in relation to the size of the grammar. Finally I will discuss criteria that I believe are sufficient to characterize a complete set of training instances for realistic applications.

Theorem 6.1. Let G be an unambiguous CFG and m a meaning function[10] that maps sentences of G to their semantic representations. Let D be the smallest[11] higher-order DCG that computes m. There exists an integer k such that D can be identified by

[10] By "meaning functions" I mean functions that map sentences to semantic representations.

[11] A suitable measure for size would be the number of symbols in a DCG; other measures are possible as well.

comparing its output with the value defined by m on at most k sentences of the language $L(G)$ defined by G.

Proof: Let the enumeration of DCGs in order of increasing size be $D_0,\ D_1,\ \ldots$. Therefore D is D_i for some i. Since it is the *smallest* DCG computing the meaning function m, its meaning function differs from that of each DCG in $D_0,\ D_1,\ \ldots,\ D_{i-1}$. Let s_0 be the smallest sentence in $L(G)$—using for example the lexicographic ordering of words—whose meaning as given by D_0 differs from that as given by D_i. Similarly, let $s_1,\ s_2, \ldots,\ s_{i-1}$ be the corresponding (smallest) sentences that distinguish respectively $D_1,\ D_2,\ \ldots,\ D_{i-1}$ from D_i. Then the integer k satisfying the theorem is the index of the largest sentence in the set $\{s_0,\ s_1, \ldots,\ s_{i-1}\}$. **Q.E.D.**

While Theorem 6.1 states that there is a finite minimal set of training instances, one can show along the lines of Budd and Angluin (1982) that this k is not computable. The reason is that the computability of k is equivalent to the problem of deciding the equivalence of two higher-order DCGs. Since even first-order DCGs have the power of Turing machines (Pereira and Warren 1980), clearly there is no algorithm to decide the equivalence of two higher-order DCGs. Hence k is not computable. This result in turn means that one cannot preprocess a grammar in order to determine k. Instead one should be content with an interactive process of generating examples until the user is satisfied with the generated DCG after some point. Notwithstanding this result, for practical applications, it may be that such sets are reasonably small. Intuitively it is clear that the larger the size k of the set of training instances is, compared with the size/footnoteA suitable measure for the size of terms would again be the number of symbols. of the terms representing the meaning function, the higher the probability that the (unique) correct meaning function has been found. For example, it is unlikely that a meaning function m_1 of size 10 (i.e., the smallest representation of the meaning function has size[12] 10) computes the correct meanings for all sentences of size[13] < 1000000, but computes an incorrect meaning for some sentence of size > 1000000. Even though in principle it is possible that m_1 turns out to be incorrect after testing it on some sentence of size > 1000000, it is highly unlikely one can obtain such a function as a higher-order DCG.

For each DCG rule, only a small number of decisions regarding the processing of the semantic terms returned by each nonterminal needs to be made, since these terms are built up from λ-abstraction and application. Since each training instance in general contributes several constraints, one can expect the size of a minimal set of training

[12] Again measured as the number of symbols.

[13] Measured as the number of words in a sentence.

instances to be of the same order as the number of grammar rules. This observation has been confirmed by the numerous grammars on which the system has been tested. The number of training instances can be reduced by using longer sentences, so that each training instance covers more rules and contributes more constraints. However, using longer sentences has negative implications for the complexity of the search, since inconsistent substitutions cannot be detected as early.

6.1.1. Criteria for selecting training instances

Under additional assumptions, it seems possible to determine a complete set of training sentences. In order to do so, I assume that the grammar has no chain rules[14] (and all unreachable[15] rules are deleted). Furthermore, I assume the following:

1. The semantic terms of each grammar rule with nonterminals on the right-hand side do not discard any of the semantic terms returned by those non-terminals (that is, each prefix variable must occur at least once in the body of the λ-term).

2. The semantic terms corresponding to all uses of a particular grammar symbol (terminal or nonterminal) have the same type.

What follows is a set of criteria that can be used to identify a complete set of training sentences.

Definition 6.1. A *semantic rule* corresponding to a syntactic (CFG) rule is a substitution found by SYNTH for the function variable corresponding to that rule.

Conjecture: Let L be the language generated by the given grammar G, and let $TS = \{s_1, s_2, \ldots\}$ be a set of training sentences. Then TS guarantees a unique solution if it satisfies the following criteria:

(1) Each grammar rule must be used by at least one training instance.

(2) For each use of a grammar rule `a --> b1, ..., bn` by some training instance, there is for each nonterminal `bi` another use of that grammar rule (by the same or different training instance) such that the terminals (phrase) being parsed by each

[14] A chain rule is a rule whose left-hand side is a nonterminal that does not form the left-hand side of any other rule, and whose right-hand side consists of a single nonterminal.

[15] A rule r is said to be unreachable if there exists no parse tree (with the start symbol as root) involving r.

`bj` $\neq$ `bi` are identical in the two uses, and those being parsed by `bi` are different in such a way that the corresponding semantic term is also different from that of `bi` (if such a use is possible).

(3) For each nonterminal `a` occurring on the left-hand side of $k > 1$ grammar rules, there must be k training instances involving `a` whose parse trees are identical except for the subtree corresponding to `a`. This subtree is varied according to the k different rules for `a`. If these training instances involve more than one application of a rule with left-hand side `a` only one of these rule applications should be varied.

(4) Training sentences should not be derived by applying recursive rules unless necessary to satisfy condition **(2)**.

(5) A sentence with repeated occurrences of terminals should be omitted if possible.

Informal Justification: Let l_0 be the semantic term corresponding to some use u_0 of grammar rule g_0. Then, varying a string of terminals W being parsed by g_0 and the corresponding semantic terms as described in the conjecture defines a semantic rule m_0 for g_0 that is unique, except possibly for variable ordering (changing the order of the prefix variables without changing the body of a term) and constant terms that are independent of W for the following reasons:

Criterion **(2)** ensures that the semantic representation returned by each nonterminal `bi` is correctly incorporated into the semantic representation of the head `a` of each rule. In the case that the semantic representation of `bi` is part of the semantic representation of the complete sentence, this criterion also ensures that the term returned by `bi` is not discarded and incorrectly reconstructed for example by "hard coding" it into the rule. Criterion **(3)** ensures that the semantic terms of the heads of the k rules headed by `a` are of the same type, use the same variable orderings, and handle constants in the same way. At the same time it forces the semantic rule of the head of the grammar rule that contains the nonterminal `a` in its body to be general enough to correctly handle all possible semantic terms that can be returned by `a`. Criterion **(4)** ensures that the semantic term of the head of a recursive rule is not specific to certain kinds of semantic terms returned by nonterminals in the body of the rule only periodically. For example, a particular nonterminal may return a certain kind of semantic term only for an even number of recursive calls, in which case semantic rules may be inferred that are too specific if the training instances don't include cases involving both, odd and even number of recursive calls. Given criterion **(2)**, criterion **(5)** doesn't seem to be required for correctness, but it

avoids unnecessary training instances and unnecessary computations during higher-order unification.

The reason why variable ordering or constant terms may not be uniquely determined is that fixed combinations of grammar rules may compensate these factors in such a way that their combined behavior is identical even though the semantic rules of the individual grammar rules may vary (see section 5.4). That is, there may be some flexibility for the semantic rules of individual grammar rules of a fixed combination, but the overall behavior of such a combination is uniquely determined. Since the training instances define a unique semantic function for all such fixed combinations, any two grammars augmented using TS have the same input/output behavior.

6.1.2. Efficiency issues

The examples below illustrate important factors in choosing training instances (sentence-meaning pairs).

- To achieve optimal performance, training instances must be as small as possible.
- The set of training instances should be considered an *ordered* set, since the performance of the unification procedure crucially depends on the order in which substitutions are assigned to the variables occurring in the training instances. In particular, small training instances should come before larger ones, since inconsistent substitutions can be ruled out sooner in small equations.
- If larger training instances are needed to satisfy the criteria given in the previous section, it is beneficial to put smaller training instances that exercise the rules also used by the larger ones first.
- Even though redundant small training instances may reduce the search space considerably if processed before larger training instances that use the same rules, using unnecessary training instances at the end can significantly increase time and space complexity, especially if recursive rule applications are involved.

Example 6.1.1:

In the example below the first training instance isn't really needed to ensure a correct augmentation. However, the search space expands considerably if the first training instance is left out, because, using the first training instance, the semantic augmentation for the first rule can be determined independently of the second rule.

```
s --> [0].                  train(1,[0],F\X\X).
s --> [succ], s.            train(2,[succ,0],F\X\(F X)).
                            train(3,[succ,succ,0],F\X\(F (F X))).
```

Example 6.1.2:

Consider the grammar in example 6.1. augmented with function variables:

```
s(F1) --> [0].
s(F2 A) --> [succ], s(A).
```

Assume this DCG is executed on the additional training sentence `[succ, succ, succ, 0]`. The DCG would return the term `(F2 (F2 (F2 F1)))`; that is, the function variable `F2` is repeated for each recursive call. This means, each time a substitution is tried for `F2` all occurrences of `F2` must be replaced by that substitution and the corresponding β-reductions must be performed. This can be disastrous if substitutions like Huet's projection substitutions are allowed. For example, the substitution

```
F2 <- X\Y\Z\(X (H1 X Y Z) (H2 X Y Z))
```

would convert the term `(F2 (F2 (F2 F1)))` to

```
(X1\Y1\Z1\(X1 (H1 X1 Y1 Z1) (H2 X1 Y1 Z1))
  (X2\Y2\Z2\(X2 (H1 X2 Y2 Z2) (H2 X2 Y2 Z2))
     Y3\Z3\(F1 (H1 F1 Y3 Z3) (H2 F1 Y3 Z3))))
=
(X1\Y1\Z1\(X1 (H1 X1 Y1 Z1) (H2 X1 Y1 Z1))
  (Y2\Z2\(Y3\Z3\(F1 (H1 F1 Y3 Z3) (H2 F1 Y3 Z3))
              (H1 Y4\Z4\(F1 (H1 F1 Y4 Z4) (H2 F1 Y4 Z4)) Y2 Z2)
                 (H2 Y5\Z5\(F1 (H1 F1 Y5 Z5) (H2 F1 Y5 Z5)) Y2 Z2))))
=
(X1\Y1\Z1\(X1 (H1 X1 Y1 Z1) (H2 X1 Y1 Z1))
 (Y2\Z2\(F1 (H1 F1 (H1 Y4\Z4\(F1 (H1 F1 Y4 Z4) (H2 F1 Y4 Z4)) Y2 Z2)
     (H2 Y5\Z5\(F1 (H1 F1 Y5 Z5) (H2 F1 Y5 Z5)) Y2 Z2))
         (H2 F1 (H1 Y6\Z6\(F1 (H1 F1 Y6 Z6) (H2 F1 Y6 Z6)) Y2 Z2)
     (H2 Y5\Z5\(F1 (H1 F1 Y5 Z5) (H2 F1 Y5 Z5)) Y2 Z2)))))
=
(X1\Y1\Z1\(Y2\Z2\(F1 (H1 F1 (H1 Y4\Z4\(F1 (H1 F1 Y4 Z4) (H2 F1 Y4 Z4)) Y2 Z2)
     (H2 Y5\Z5\(F1 (H1 F1 Y5 Z5) (H2 F1 Y5 Z5)) Y2 Z2))
         (H2 F1 (H1 Y6\Z6\(F1 (H1 F1 Y6 Z6) (H2 F1 Y6 Z6)) Y2 Z2)
     (H2 Y5\Z5\(F1 (H1 F1 Y5 Z5) (H2 F1 Y5 Z5)) Y2 Z2)))
 (H1 Y2\Z2\(F1 (H1 F1 (H1 Y4\Z4\(F1 (H1 F1 Y4 Z4) (H2 F1 Y4 Z4)) Y2 Z2)
```

```
    (H2 Y5\Z5\(F1 (H1 F1 Y5 Z5) (H2 F1 Y5 Z5)) Y2 Z2))
        (H2 F1 (H1 Y6\Z6\(F1 (H1 F1 Y6 Z6) (H2 F1 Y6 Z6)) Y2 Z2)
    (H2 Y5\Z5\(F1 (H1 F1 Y5 Z5) (H2 F1 Y5 Z5)) Y2 Z2))) Y1 Z1)
(H2 Y2\Z2\(F1 (H1 F1 (H1 Y4\Z4\(F1 (H1 F1 Y4 Z4) (H2 F1 Y4 Z4)) Y2 Z2)
    (H2 Y5\Z5\(F1 (H1 F1 Y5 Z5) (H2 F1 Y5 Z5)) Y2 Z2))
        (H2 F1 (H1 Y6\Z6\(F1 (H1 F1 Y6 Z6) (H2 F1 Y6 Z6)) Y2 Z2)
    (H2 Y5\Z5\(F1 (H1 F1 Y5 Z5) (H2 F1 Y5 Z5)) Y2 Z2))) Y1 Z1)))
```

That is, the size of the resulting term increases exponentially with the number of occurrences of the variable being replaced and the number of arguments of the head of the substitution term.

Example 6.1.3:

Consider the following grammar augmentation:

CFG	*Training instances*	
`s --> [0].`	`[0]`	`F\X\X`
`s --> [s,0].`	`[s,0]`	`F\X\(F X)`
`s --> [s,s,0].`	`[s,s,0]`	`F\X\(F (F X))`
`s --> [s,s,s], s.`	`[s,s,s,0]`	`F\X\X`
	`[s,s,s,s,0]`	`F\X\(F X)`

The correct higher-order DCG can be found with the above training instances:

```
s(A\B\B) --> [0].
s(A\B\(A B)) --> [s],[0].
s(A\B\(A (A B))) --> [s],[s],[0].
s(A\B\C\(A B C) D) --> [s],[s],[s],s(D).
```

However, if `[s,s,s,s,s,0], F\X\(F (F X))` is used instead of the first training instance, execution time will be much longer since the intermediate terms produced by that training instance would be much larger than that of the first training instance.

Providing semantic rules with the syntactic rules:

Complexity can also be reduced by specifying the semantic rules directly with the corresponding syntactic rules if they are known to the user. Also, incremental synthesis of semantics by breaking down a large grammar into independent smaller ones can be used to control time and space complexity (see discussion in chapter 8).

6.2. Search Control and Combination Rules

The nondeterministic selection of substitution terms is implemented by backtracking. The fact that all right-hand side terms are known to be ground can be exploited to improve the search procedure in various ways (see below). Without these optimizations a simple depth-first search procedure is incomplete, since there could be, in general, infinite branches. This is because the projection and imitation substitutions in Huet's procedure can introduce new variables that in general have no restrictions. Therefore, in order to ensure exhaustive traversal of the search space, an upper limit is placed on the depth of the nesting of terms and on the number of arguments a function can have, and this limit is successively increased until a solution is found (a term has "nesting of depth n" if it has n nested parentheses). If there is no solution, the search may not terminate.

Therefore the function SUBST must be augmented in such a way that the number of arguments and the depth of nesting of terms doesn't exceed the limit value lim: Let $N(V)$ denote the depth of nesting of variable V in the original equation.

1. Input a higher-order equation e. If $N(V) > lim$ then fail.

2. Nondeterministically select a substitution S according to the following options:

a. *Projection* substitutions. Projection substitutions are of the form: $f \leftarrow \lambda w_1 \ldots . \lambda w_k . (w_i\ (h_1\ w_1\ \ldots\ w_k)\ \ldots\ (h_l\ w_1\ \ldots\ w_k))$, for each $1 \leq i \leq k$, where $\tau(f) = \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_k \rightarrow \beta$, and $\tau(w_i) = \alpha_i$ (the α_i are type variables). The following parameters are chosen nondeterministically: (i) $k \leq lim$, the number of prefix variables (see discussion on type inference below); (ii) w_i, the head of the substitution term; (iii) $l \leq lim$, the number of arguments of w_i, which is often determined through unification of w_i with the corresponding argument to which the substitution term is being applied.

b. *Imitation* substitution. Imitation substitution are of the form of the form: $f \leftarrow \lambda w_1 \ldots . \lambda w_k . (c\ (h_1\ w_1\ \ldots\ w_k)\ \ldots\ (h_j\ w_1\ \ldots\ w_k))$, where $\tau(f) = \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_k \rightarrow \beta$. The number k of prefix variables is equal to the number of arguments of f, except if the right-hand side term is a λ-abstraction, i.e., has additional prefix variables. In that case the number of prefix variables of the substitution term must be increased accordingly. The number of arguments j and the type of the constant c obviously must be the same as that of the constant being imitated.

3. Return the selected substitution S.

If only Huet's imitation and projection substitutions are allowed, the enumeration procedures are comparatively simple (as described above). However, the price one pays for simplicity of enumeration procedures is increased complexity in search. By using simpler substitution rules (which can be thought of as precompiled combinations of Huet's imitation and projection substitutions), and by using more sophisticated enumeration procedures, time and space complexity can be reduced dramatically, especially if combined with other optimization techniques that take into account the fact that one side is ground, because the semantic representations provided by the user are usually ground.

Huet's imitation and projection substitutions require that if the head of the substitution term has one or more arguments, then a new function variable must be introduced which is applied to all prefix variables. This implies that long "detours" must be taken in order to synthesize even simple terms (see for example the case discussed in Appendix D). A more direct way to resolve higher-order equations is possible if the kinds of substitutions discussed below are used instead of the imitation and projection substitutions of Huet's procedure.

6.2.1. More Efficient Substitutions

Specifying abstractions of substitutions

For debugging purposes or for assisting the system in resolving a set of higher-order equations, it is useful to be able to specify abstractions of substitutions at various levels. For example, sometimes one would like to specify that a projection substitution has three prefix variables, without knowing anything else about this substitution. Or sometimes one would like to specify that a projection substitution with three prefix variables takes the second prefix variable as its head without knowing the type of that variable.

The following notation is used to specify such abstractions. The term used by the system to characterize the *projection substitutions* of Huet's procedure is `proj(L,M,N)`, where `L` is the number of prefix variables of the substitution term, `M` specifies which prefix variable is chosen as head of the substitution term, and `N` is the number of arguments of that head. For example, the substitution term

```
A\B\C\(C (H1 A B C) (H2 A B C))
```

would be characterized as `proj(3,1,2)`, meaning that there are *three* prefix variables `A\B\C\`, the *first* prefix variable `C` (counting from right to left) is chosen as the head

which has *two* arguments. By replacing one or more arguments with variables, one obtains partial specifications of substitutions. E.g., `proj(3,X,Y)` only specifies that it is a projection substitution with three prefix variables. Other uses of this notation are discussed in the section on debugging and diagnostics in the appendix. Apart from debugging and analysis purposes, this compact notation is also convenient for discussing classes of substitutions.

The term used by the system to characterize the imitation substitutions of Huet's procedure is `imitation(L)`, where `L` is the number of prefix variables of the substitution term. For example, the substitution term

```
A\B\C\(and (H1 A B C) (H2 A B C))
```

would be characterized as `imitation(3)`. The number of arguments of the head '`and`' does not need to be specified since it is determined by the term on the right-hand side of the higher-order equation (disagreement pair), which is always ground.

Variable-free substitution rules

If the following three classes of substitution rules are used instead of Huet's imitation and projection rules, the type of higher-order equations arising in our synthesis are resolved much more effectively, since these substitutions do not introduce any new free variables.

(1) *Closed projection substitutions:*

Linear substitutions without vacuous abstractions (i.e., each prefix variable occurs exactly once) are common, e.g., `A\B\(A B)`. If one is restricted to the substitutions used by Huet's procedure, the following sequence of substitutions is needed to obtain the above term:

```
(1)   V <- A\B\(A (H1 A B))
(2)   H1 <- A\B\B
```

Substituting for `H1` in `(1)` and simplifying the result `A\B\(A (X\Y\Y A B))`, the desired substitution for `V` is obtained. This type of combination of substitution rules is necessary for constructing almost any substitution term: first new variables that are dependent on all prefix variables are introduced, and then suitable projection substitutions are used to discard "unwanted" terms. However, these combinations involve a lot of overhead because, for nontrivial substitutions, many

copies of large terms have to be temporarily stored and manipulated until they are finally discarded.

Therefore it would clearly be desirable to have the search procedure directly generate substitutions like `A\B\(A B)`. A procedure for generating such substitutions has been implemented and is termed `cproj(L,M,N)` (the arguments `L`, `M`, and `N` have the same meaning as in the `proj(L,M,N)`-case discussed above). Instead of introducing new function variables with unnecessary arguments, it directly generates arguments that are made up of only the prefix variables. An additional restriction is that each prefix variable occurs exactly once in the body of the substitution term (linearity). It is straightforward to enumerate such substitutions as there are no infinite branches; that is, for any given values of `L`, `M`, and `N`, that are only a small, finite number of substitutions.

Clearly, these substitution rules are not as general as Huet's projection substitution rules; however, in conjunction with the other two classes of substitution rules discussed below, they are sufficient for the kinds of unification problems that arise from non-pathological DCG synthesis tasks.

Example 6.2.1:

Here are some examples (in no particular order) of substitutions the closed projection rules would generate.

```
cproj(1,1,0):  V <- A\A
cproj(2,2,1):  V <- A\B\(A B)
cproj(3,3,2):  V <- A\B\C\(A C B)
cproj(2,1,1):  V <- A\B\(B A)
cproj(3,2,2):  V <- A\B\C\(B A C)
cproj(3,2,1):  V <- A\B\C\(B (A C))
cproj(4,4,3):  V <- A\B\C\D\(A C D B)
cproj(4,4,2):  V <- A\B\C\D\(A (C D) B)
```

Note that, when the types are known, most substitutions are immediately eliminated, since the types restrict the number of prefix variables as well as the number of arguments a function can have. For example, assume a substitution is to be found for `V` in the term `(V X Y)`, and the types for `V`, `X`, and `Y` are known to be

$(i \to o) \to i \to o$, $i \to o$, and i, respectively. Then the only possible closed projection substitution is `A\B\(A B)`, since the type of `V` restricts the number of prefix variables to two, and the head of the substitution term must be a function type.

(2) *Imitation-projection substitutions:*

In order to synthesize a term such as `A\B\C\(and (A C) (B C))` using Huet's imitation and projection substitutions, the following sequence of substitutions is needed:

```
(1) V <- A\B\C\(and (H1 A B C) (H2 A B C))
(2) H1 <- A\B\C\(A (K1 A B C))
(3) K1 <- A\B\C\C
(4) H2 <- A\B\C\(B (K2 A B C))
(5) K2 <- A\B\C\C
```

However, performing these substitutions and simplifying involves a complex derivation with a lot of overhead. Hence, as in the previous case, it would be desirable to have the search procedure generate those terms directly. A procedure for generating such substitutions has been implemented and is termed `imit_proj(N)`, where `N` is the number of prefix variables. This procedure doesn't introduce new variables, but selects prefix variables nondeterministically to serve as functions at the argument positions. The remaining prefix variables are used as arguments to those functions. A prefix variable can occur at most once in a function position, but any number of times as an argument. (I refer to this as the almost-linear restriction.) It is also possible to introduce free variables as arguments.

Example 6.2.2:

Examples of substitutions generated by this procedure are (the constants appearing in these substitutions would in general be determined from the matching ground term):

```
imit_proj(3):  V <- A\B\C\(and (A C) (B C))
imit_proj(3):  V <- A\B\C\(not (A C B))
imit_proj(3):  V <- A\B\C\(and (B D) (A D C))
imit_proj(2):  V <- A\B\(latitude A B)
imit_proj(4):  V <- A\B\C\D\(and (B C) (A C D))
```

```
imit_proj(0):  V <- 2
imit_proj(5):  V <- A\B\C\D\E\(and (B E C) (A D E))
imit_proj(3):  V <- A\B\C\(setof (A C) B)
imit_proj(4):  V <- A\B\C\D\(and (B E C) (A E D))
imit_proj(3):  V <- A\B\C\(numberof (A C) B)
imit_proj(2):  V <- A\B\(not (A B))
imit_proj(4):  V <- A\B\C\D\(and (A D E) (B E C))
```

Again, the number of possible substitution terms for any fixed number of prefix variables is relatively small. There are no new variables introduced as in the case of Huet's imitation and projection substitutions, and the duplication of variables is also very limited. Therefore, even if there appear to be a larger choice of substitutions the overall complexity is significantly reduced as no new variables are being introduced for which viable substitutions have to be found.

(3) *Generalized imitation-projection substitutions:*

In order to synthesize substitution terms of arbitrary nesting, e.g.,
`A\B\(exists X\(and (A X) (B X)))`,
a generalized version of the imitation-projection rule is needed. These rules are similar to Huet's imitation substitution rules except that the new function variables being introduced may not have all of the prefix variables as arguments. This enhancement makes it possible to altogether disallow substitutions that discard arguments (like `A\B\C\C`, or `A\B\C\(A C)`). Substitutions that discard arguments are usually only needed to remove extraneous subterms in a derivation. However, the use of generalized imitation-projection substitutions makes it possible to avoid the generation of extraneous subterms in the first place. This property in turn implies a certain *monotonicity* during the resolution process, which allows further optimizations as discussed below.

The term used by the system for this class of substitutions is `imitation(N)`, where `N` again is the number of prefix variables.

Example 6.2.3:

Typical substitutions of this type are:

```
imitation(2):  V <- A\B\(setof (H1 A) (H2 B))
```

```
imitation(3):  V <- A\B\C\(numberof (H1 A B C) H2)
imitation(3):  V <- A\B\C\(setof (H1 A B) (H2 C))
imitation(3):  V <- A\B\C\(exists (H1 A B C))
```

Since the generalized imitation-projection substitutions introduce new variables and therefore have a potentially high branching factor, the closed projection substitutions and the imitation-projection substitutions should be tried first. Many equations occurring during the resolution of a set of higher-order equations can be solved without the generalized imitation-projection substitutions.

6.2.2. Effect of Combination Rules on Search Complexity

As noted earlier, if only the basic projection and imitation substitutions are used, the size of the terms increases exponentially with the number of prefix variables and the number of occurrences of a variable in the term. Even for simple grammars terms can grow very large. As a result the execution time increases exponentially as well.

Example 6.2.4:

CFG:

```
s --> [0].
s --> [s], s.
```

Training instances:

```
train(1,[0],X\Y\X).
train(2,[s,0],X\Y\Y).
train(3,[s,s,0],X\Y\X).
train(4,[s,s,s,0],X\Y\Y).
```

Sequence of substitutions:

```
s(A) --> [0].
s(D E) --> [s],s(E).

[[A,I\J\I],
 [D A,I\J\J],
 [D (D A),I\J\I],
 [D (D (D A)),I\J\J]]

proj(2,2,0):  A <- B4\C4\B4
```

```
[[D K\L\K,A\B\B],
 [D (D M\N\M),A\B\A],
 [D (D (D O\P\O)),A\B\B]]

   proj(3,3,2):  D <- Y3\Z3\A4\ (Y3 (Q Y3 Z3 A4) (Z Y3 Z3 A4))

[[Q R\S\R A B,B],
 [Q T\U\T (Q V\W\ (Q X\Y\X V W) A B) (Z A1\B1\ (Q C1\D1\C1 A1 B1) A B),A],
 [Q E1\F1\E1 (Q G1\H1\ (Q I1\J1\I1 G1 H1) (Q K1\L1\ (Q M1\N1\M1 (Q O1\P1\
(Q Q1\R1\Q1 O1 P1) K1 L1)
  (Z S1\T1\ (Q U1\V1\U1 S1 T1) K1 L1)) A B) (Z W1\X1\ (Q Y1\Z1\Y1 (Q A2\B2\
(Q C2\D2\C2 A2 B2) W1 X1)
  (Z E2\F2\ (Q G2\H2\G2 E2 F2) W1 X1)) A B)) (Z I2\J2\ (Q K2\L2\K2 I2 J2)
  (Q M2\N2\ (Q O2\P2\O2 (Q Q2\R2\ (Q S2\T2\S2 Q2 R2) M2 N2) (Z U2\V2\ (Q
W2\X2\W2 U2 V2) M2 N2)) A B)
  (Z Y2\Z2\ (Q A3\B3\A3 (Q C3\D3\ (Q E3\F3\E3 C3 D3) Y2 Z2) (Z G3\H3\ (Q
I3\J3\I3 G3 H3) Y2 Z2)) A B)),B]]

   proj(3,1,0):  Q <- V3\W3\X3\X3

[[Z K3\L3\L3 A B,A],
 [Z M3\N3\N3 B (Z O3\P3\ (Z Q3\R3\R3 O3 P3) A B),B]]

   proj(3,2,0):  Z <- S3\T3\U3\T3

[]
      A = B4\C4\B4
      D = Y3\Z3\A4\ (Y3 A4 Z3)
      Q = V3\W3\X3\X3
      Z = S3\T3\U3\T3
```

As can be seen at the substitution `proj(3,3,2)`, because of the multiple occurrences of the prefix variables in the projection substitution, a term can be duplicated $m * n$ times, where m is the number of occurrences of the variable being substituted for, and the n is the number of times a prefix variable occurs in the substitution term.

Example 6.2.5:

The substitution rules `itpr` below are combinations of projection rules such that no new variables are introduced and no variables are duplicated or omitted.

```
s(A B C) --> pn(B),vp(C).
vp(G H I) --> tv(H),pn(I).
```

```
pn(M) --> [mike].
pn(P) --> [mary].
pn(S) --> [john].
tv(V) --> [saw].
tv(Y) --> [visited].

[[A M (G V P),saw mike mary],
 [A S (G V P),saw john mary],
 [A M (G Y P),visited mike mary],
 [A M (G V S),saw mike john]]

   cproj(2,1,0):  A <- U1\V1\(V1 U1)

[[G V P M,saw mike mary],
 [G V P S,saw john mary],
 [G Y P M,visited mike mary],
 [G V S M,saw mike john]]

   cproj(3,3,0):  G <- R1\S1\T1\(R1 T1 S1)

[[V M P,saw mike mary],
 [V S P,saw john mary],
 [Y M P,visited mike mary],
 [V M S,saw mike john]]

   imitation(2):  V <- P1\Q1\(saw (C1 P1 Q1) (B1 P1 Q1))

[[B1 M P,mary],
 [C1 M P,mike],
 [B1 S P,mary],
 [C1 S P,john],
 [Y M P,visited mike mary],
 [B1 M S,john],
 [C1 M S,mike]]

   proj(2,1,0):  B1 <- N1\O1\O1

[[P,mary],
 [C1 M P,mike],
 [P,mary],
 [C1 S P,john],
 [Y M P,visited mike mary],
 [S,john],
 [C1 M S,mike]]
```

```
   imitation(0):  P <- mary

[[C1 M mary,mike],
 [C1 S mary,john],
 [Y M mary,visited mike mary],
 [S,john],
 [C1 M S,mike]]

   proj(2,2,0):  C1 <- L1\M1\L1

[[M,mike],
 [S,john],
 [Y M mary,visited mike mary],
 [S,john],
 [M,mike]]

   imitation(0):  M <- mike

[[S,john],
 [Y mike mary,visited mike mary],
 [S,john]]

   imitation(0):  S <- john

[[Y mike mary,visited mike mary]]

   imitation(2):  Y <- J1\K1\(visited (E1 J1 K1) (D1 J1 K1))

[[D1 mike mary,mary],
 [E1 mike mary,mike]]

   proj(2,1,0):  D1 <- H1\I1\I1

[[E1 mike mary,mike]]

   proj(2,2,0):  E1 <- F1\G1\F1

[]
      A = U1\V1\(V1 U1)
      G = R1\S1\T1\(R1 T1 S1)
      V = P1\Q1\(saw P1 Q1)
      B1 = N1\O1\O1
      P = mary
```

```
C1 = L1\M1\L1
M = mike
S = john
Y = J1\K1\(visited J1 K1)
D1 = H1\I1\I1
E1 = F1\G1\F1
```

6.2.3. Monotonicity and Linearity

Since the higher-order equations obtained by equating the terms that result from executing sentences on an augmented DCG with the terms (semantic representations) that the user provides always have one side ground, the higher-order unification problem is actually a higher-order *matching* problem. Even though decidability of higher-order matching is still an open problem, in enumerating matching substitutions various optimizations of the unification search procedure are possible for this subclass of unification problems. Such optimizations can best be achieved by using a set of substitution rules that observe the following condition.

Definition 6.3.1: A set of substitutions satisfies the *monotonicity condition* if none of them contains vacuous abstractions; i.e., all prefix variables of an abstraction must occur at least once in the body of the term.

This condition implies that terms introduced by previous substitutions cannot be discarded later on.

Obviously, completeness is lost under this condition if one is limited to Huet's *imitation* and *projection* substitutions. Therefore new substitution rules must be introduced which satisfy the monotonicity condition without giving up completeness. These alternative substitutions can still be classified as imitation and projection substitutions, as they are essentially precompiled combinations of Huet's imitation and projection substitutions, but using at most one imitation substitution. (Section 6.2.1 discusses these alternative substitution rules in more detail.)

Definition 6.3.2: A λ-abstraction is said to be *linear* if each of its prefix-variables occurs at most once in its body.

Definition 6.3.3: A λ-abstraction is said to be *almost-linear* if all prefix-variables that are used at function positions in the body of the λ-abstraction occur at most once.

Given monotonicity and linearity, the fact that the right-hand side of each equation is

ground allows to predict the number of imitation substitutions required to make the left-hand side term identical to the right-hand side term:

Proposition 6.3.4: If monotonicity and linearity hold, the number of imitation substitutions needed to convert the left-hand side term of an equation into the right-hand side term is computed by subtracting the number of constants occurring in the right-hand side term from the number of constants occurring in the left-hand side term.

Proof: Due to linearity, each imitation substitution introduces exactly one constant, and due to monotonicity, none of these constants will be discarded later. Since the left-hand side term may already contain constants, the number of imitation substitutions is determined as specified in the proposition. **Q.E.D**.

This is an important constraint which helps to reduce the search space. However, in many cases the linearity constraint has to be relaxed, since the duplication of individuals in natural language is fairly common, for example, when using reflexives. Therefore, we have to allow almost-linear substitutions, in which case the number of imitation substitutions must be reduced accordingly.

The number of projection substitutions cannot be predicted as accurately, but it is constrained in the following way. The number of projection substitutions is equal to the total number of free variables in the left-hand side term plus the number of free variables being introduced by substitution rules minus the number of imitation substitutions (as discussed above).

6.3. Dependency Directed Backtracking

During the process of solving a set of higher-order equations many substitutions are tried for each variable until a consistent set of substitutions is found. If failure occurs for a particular set of substitutions, the equation at which the failure occurred can be easily located. For example, after a particular substitution has been made in some equation e_1, the heads of the left-hand-side term and right-hand-side term of another equation e_2 may be different constants; that is, e_2 cannot be resolved with the current set of substitutions. Being able to localize failure in this way allows the system to analyze the cause of that failure and continue the search such that only those substitutions which have contributed to the failure are changed.

The procedure described below is a refinement of depth-first search, and is therefore applicable only for (higher-order) unification problems for which depth-first search is a

complete search strategy. Such unification problems arise for example if at least one side of each equation is ground, and if the substitution rules satisfy the monotonicity condition, i.e., none of the substitution rules discards any of its arguments.

Basic Idea:

For example, consider the set of equations below. (`e(V11,V12,...)` represents an equation containing variables `V11,V12,...`):

```
e(V11,V12),
e(V21,V22),
e(V11,V31,V32),
e(V21,V41,V42),
```

The equations are organized on a stack. If the head of the top-most equation is a variable, it will get a substitution by one the substitution rules. Note that in this example, variables `V12` and `V22` do not occur in the third equation. Now assume all variables until `V32` have obtained a substitution; also assume that after `V32` is assigned, failure occurs in the third equation. It would not make sense to backtrack for example to `V12` or `V22` as they have no influence on the equation in which the failure occurred; instead the system should backtrack to the assignment of `V11`.

Method:

The scheme for dependency-directed backtracking described here is based on the concept of an *influence set*. There is an influence set for each equation, which is initially defined as the variables occurring in the equation. If an equation is decomposed, each of the generated equations inherits the influence set of the original equation. If a substitution introduces new variables, these new variables are added to the influence sets of the equations in which the substitutions took place (note that all occurrences of a particular variable in the set of equations are replaced simultaneously). As soon as failure is detected in an equation E, the set of variables of the influence set of E that have already been bound is asserted as "backtrack set". The backtrack set indicates to which variable assignments one should backtrack to try to overcome the failure. The backtrack set is removed as soon as one of its variables obtained a new assignment.

Example 6.4.1:

Below is a set of equations that arises while processing a small grammar.

```
 1 [[A (B C) (D E),sings mary],
 2  [A (B F) (D E),sings john],
 3  [A (B C) (D G),sleeps mary],
 4  [A (B C) (H (I (J K))),girl mary],
 5  [A (B C) (H (I (J L))),student mary],
 6  [A (B C) (H (I (M (N O) K))),and (girl mary) (good mary)],
 7  [A (B C) (H (I (M (N P) K))),and (girl mary) (smart mary)],
 8  [A (B C) (H (I (M (Q O (N P)) K))),and (girl mary) (and (good mary) (smart mary))],
 9  [A (B F) (R S (B C)),saw john mary],
10  [A (B F) (R T (B C)),visited john mary],
11  [A (U V (I (J K))) (D E),exists W\(and (girl W) (sings W))],
12  [A (U X (I (J K))) (D E),all W\(implies (girl W) (sings W))],
13  [A (B F) (R S (U V (Y (J K) (Z A1 (D E))))),exists W\(and (and (girl W) (sings W))
                                                          (saw john W))],
14  [B1 (C1 (B C) (D1 E1)) (F1 (B F)),saw john mary],
15  [B1 (C1 (B C) (D1 G1)) (F1 (B F)),visited john mary]]
```

If equation 11 fails due to a substitution for example to `K` in equation 4, a backtrack set is defined as the set of variables occurring in equation 11 that have been assigned so far, which would be {`A`,`D`,`E`,`I`,`J`}. This means it only makes sense to backtrack to one of these variable assignments, and not for example to `B`, `C`, `F` or `G`, since these variables have no influence on equation 11 where the failure occurred.

6.4. Summary

This chapter discussed optimizations of the synthesis procedure given in Chapter 5. The efficiency of DCG synthesis can be increased by:

- using training instances that contribute a sufficient number of constraints; the training sentences should be as short as possible, and no unnecessary training instances should be included;
- presenting shorter training instances before longer ones such that the constraints introduced by the short training instances constrain the search for substitutions in the longer training instances;
- using suitable combinations of substitution rules during higher-order unification;
- using the constraints provided by the fact that the right-hand sides of the higher-order equations are always ground (higher-order matching);

- incorporating a scheme for dependency directed backtracking in the higher-order unification procedure.

7. Efficient Execution

The higher-order DCGs of Chapter 5 are not as efficient as equivalent first-order DCGs, since the complexity of applying λ-conversion rules is greater than the complexity of first-order unification. However, it turns out that for most practical cases higher-order DCGs can be converted into first-order DCGs by precompiling all β-reductions involved in the execution of the DCGs. This conversion can be considered a form of *partial execution*. Partial execution means that a program is partly executed at compile time, which requires that the operations performed during partial execution must be independent of any input data. A simple example of partial execution is "constant folding" or "strength reduction" in traditional compilers, e.g., the replacement of the term `x+0` by the term `x`. In general, declarative programming languages offer more opportunities for partial execution than procedural languages, since the former are free from the complexities of side-effects and control. This topic has been explored to some extent in functional programs (Burstall 1977, Kahn 1982) and logic programs (Clark 1977, Tamaki 1984, Takeuchi 1985).

Below I describe a novel technique for partially executing a higher-order DCG and show that the resulting first-order DCG correctly computes the semantic representations for all sentences. The basic idea is to replace β-reduction by first-order unification for "forward execution," i.e., computing the semantic representation of a given sentence. For "reverse execution" of the DCG, additional checks are imposed to ensure correctness. Nevertheless, the efficiency of both forward and reverse execution of the partially executed DCG is better than those of the corresponding higher-order DCG. In fact the efficiency improvement for reverse execution is more dramatic since we are effectively replacing higher-order unification by first-order unification.

Section 7.1 gives the basic procedure for partial execution and discusses its correctness and limitations. Section 7.2 introduces various enhancements of the basic partial execution procedure, and Section 7.3 analyzes the implications of partial execution on reversibility.

7.1. Basic Procedure for Partial Execution

The input to the partial execution procedure is a higher-order DCG, i.e., the output of procedure SYNTH of Chapter 5. The terms to be considered for partial execution are the application terms occurring on the left-hand sides of DCG rules. To simplify the

initial presentation, the following two assumptions will be made, which will be relaxed later on:

> *Assumption 7.1.* All application terms or subterms are reduced in computing the semantic representation of the sentence being parsed, unless the term at the function position (of the application term) is a constant.
>
> *Assumption 7.2.* A prefix variable x must occur at most once in the body t of a term $x\backslash t$ unless that prefix variable remains a variable during grammar execution.

Procedure for basic partial execution

Under the above two assumptions, a higher-order DCG is partially executed as follows:

(1) Rename variables such that all prefix variables of each rule are distinct;

(2) FOR EACH rule r DO
FOR EACH application term $(t_1\ t_2)$ in r DO
IF t_1 is a variable THEN

(a) replace all occurrences of t_1 in r by an abstraction `X\Y`, where `X` and `Y` are new variables;

(b) replace `(X\Y` t_2`)` by `Y` and all occurrences of `X` in r by t_2.

End of Procedure.

In this partially executed form, the symbol `\` is simply an infix binary constructor, and therefore can now take structured terms in both of its argument positions. It is possible to express steps (a) and (b) above in a more compact manner, but the above form makes it easier to demonstrate correctness in the next section.

Example 7.1.1:

Partial execution is illustrated for the following DCG:

```
(r1) s((A B)) --> np(A), vp(B).
(r2) np(Y\(Y A)) --> pn(A).
(r3) vp(Z\(B (A Z))) --> tv(A), np(B).
(r4) pn(mike) --> [mike].
(r5) pn(mary) --> [mary].
(r6) pn(john) --> [john].
```

```
(r7) tv(A\B\(saw A B)) --> [saw].
(r8) tv(A\B\(visited A B)) --> [visited].
```

The semantic terms of all rules are in normal form. Rule (r1) is partially executed in the following way:

```
     s((A B)) --> np(A), vp(B).
=    s((C\D B)) --> np(C\D), vp(B).
=    s(D) --> np(B\D), vp(B).
```

Rule (r2) can be similarly converted:

```
     np(Y\(Y A)) --> pn(A).
=    np((B\C)\(B\C A)) --> pn(A).
=    np((A\C)\C) --> pn(A).
```

Likewise rule (r3):

```
     vp(Z\(B (A Z))) --> tv(A), np(B).
=    vp(Z\(B (C\D Z))) --> tv(C\D), np(B).
=    vp(Z\(B D)) --> tv(Z\D), np(B).
=    vp(Z\(E\F D)) --> tv(Z\D), np(E\F).
=    vp(Z\F) --> tv(Z\D), np(D\F).
```

As there are no applications satisfying assumption 7.1 in the semantic representations of the terminal symbols here, the following first-order DCG is obtained:

```
(r1)  s(A) --> np(B\A), vp(B).
(r2)  np((A\B)\B) --> pn(A).
(r3)  vp(A\B) --> tv(A\C), np(C\B).
(r4)  pn(mike) --> [mike].
(r5)  pn(mary) --> [mary].
(r6)  pn(john) --> [john].
(r7)  tv(A\B\(saw A B) --> [saw].
(r8)  tv(A\B\(visited A B) --> [visited].
```

The following diagram shows the expanded parse tree for the sentence "`mike saw mary`", indicating how its semantic representation is computed through first-order unification.

```
s(A0) -->
      np(   B0    \A0)
      np((mike\B1)\B1) -->
                       pn(mike)
                       pn(mike) --> [mike]
      vp( B0  )
      vp(A2\B2) -->    tv(A2\    C2     ),
                       tv(A \B\(saw A B)) --> [saw]
                       np(    C2   \B2)
                       np((mary\B3)\B3)   --> pn(mary)
                                              pn(mary) --> [mary]
```

C2 is first unified with B\(saw A B) and then with mary\B3, which assigns mary to B and (saw A mary) to B3. The term A2\B2, which now denotes A\mary\(saw A mary) is then unified with B0 which has been bound to (mike\B1). Therefore A is bound to mike. Finally, when B0\A0 is unified with mike\mary\(saw mike mary), we obtain (saw mike mary) as the semantic representation A0 of the input sentence. Note that, since \ is an infix binary constructor, its use in terms such as A\mary\(saw A mary) is legal.

7.1.1. Correctness of Partial Execution

Theorem 7.1.1: Under the two assumptions 7.1 and 7.2 given earlier, the partially executed DCG obtained using the procedure given above computes the same semantic representations for all sentences as the corresponding higher-order DCG.

Proof: We need to show (1) that all terms produced by a partially executed DCG are correct in the sense that they can be converted (using the λ-conversion rules) into the semantic representation that would be returned by the corresponding higher-order DCG; (2) that the semantic representations produced by the partially executed DCG are in normal form, that is, all β-reductions have been performed.

In order to show these two points we need to show (a) how λ-calculus terms and λ-calculus substitution can be correctly simulated by first-order terms and first-order substitution; (b) how β-reduction can be correctly simulated in FOL; and (c) how normal forms can be correctly simulated in FOL.

(a) *Simulating λ-calculus terms and substitution correctly:*
Let sub(t,X,u) abbreviate the operation of substitution. In a first-order language, it refers to the result of textually replacing all occurrences of variable X in t by u. In

λ-calculus, it refers to the result of a similar replacement except that variables in `t` may have to be renamed to avoid variable capture. There are two restrictions under which λ-terms may be simulated by first-order terms (in which all prefix variables are treated as logical variables) and λ-calculus substitution simulated by first-order substitution (where renaming is absent): (i) all prefix variables must have distinct names, and (ii) each prefix variable may occur at most once in the body of the λ-term (linearity). (To see what goes wrong without distinct prefix variables, consider the result of `sub(X\(foo Y), Y, X\X)`. The result `X\(foo X\X)` is incorrect as a first-order term, since all occurrences of `X` would refer to the same object. The result of `sub(Z\(foo Z Y Y), Y, X\X)` illustrates what can go wrong without linearity. The result `Z\(foo Z X\X X\X)` again is incorrect as a first order term.) Restrictions (i) and (ii) are sufficient for correct simulation of λ-calculus terms and substitution since they ensure that each prefix variable occurs at most once in a term, thus avoiding the problem that in a first-order term all prefix variables with the same name are bound to each other.

(b) *Simulating β-reduction correctly:*
β-reduction makes use of substitution to reduce an application term of the form `(X\T B)` to `sub(T,X,B)`. Simulating β-reduction in a first-order setting reduces to correct simulation of substitution. Since these λ-terms are used in the context of a DCG, we must ensure that the two requirements for correct simulation of substitution are met in the partially executed DCG. We guarantee distinct prefix variables through a combination of compile-time variable renaming and the use of SLD-resolution which makes distinct variants of clauses at each step. Semantic terms returned by different nonterminals on the right-hand side of a rule cannot have any variables in common since distinct variants of clauses are used for each call. Linearity ensures that distinct prefix variables occur within each such term. We guarantee linearity by explicitly requiring it or by using an expensive "copying" mechanism (see section 7.2.2).

(c) *Correctness of Normal Forms:*
If each β-reduction step is correctly simulated, all we need to ensure now is that all β-reduction steps and only the needed β-reduction steps are applied; but this is assumption 7.1. The operations necessary for correct simulation of β-reduction are performed by a DCG if the arguments for the semantic representations are structured appropriately:

Let t be the semantic term of the head of a DCG rule. Assume `(A B)` is an application occurring in t, and assume that `A` is a function variable. We consider the two cases, where `A` is a prefix variable of t, `A` is a free variable in t but occurring in one of the semantic terms on the right-hand side of the rule. In case 1, t is of the form `...\A\...(...(A B)...)`.

The operations for β-reduction can now be performed by binding the prefix variable `A` to a term `X\Y`, where `X` and `Y` are new variables, unifying `X` with `B`, and replacing `(A B)` by `Y`, so that the reduced version becomes `...\(B\Y)\...(...Y...)`. Therefore, no matter what the prefix variable `A` is bound to when t is applied to some argument at execution time, the term `Y` will be the reduced term, and the prefix variable will be removed. The reduced term is then returned in the head of the rule that removes this prefix variable.

In case 2, t contains a term `(A B)`, where `A` is a function variable occurring on the right-hand side of the rule, i.e., the rule is of the form

```
a(...(A B)...) --> ..., b(A), ...
```

The operations of β-reduction are now implemented by replacing this rule with a rule that is identical except for the terms indicated:

```
a(...Y...) --> ..., b(B\Y), ...
```

As can be easily verified, at execution time `Y` will become the reduced version of `(A B)`. Note that this operation effectively removes the prefix variable of `A` after unifying it with `B` so that the same term `Y` is returned on the left-hand side as the term that would be produced by β-reduction of `(A B)`.

Therefore, under assumptions 7.1 and 7.2, the scheme for partial execution described above ensures that all β-reductions are correctly simulated when the DCG is executed. **Q.E.D.**

Note that the procedure for basic partial execution need not distinguish between the two cases considered in case (c)—the proof of construction of the normal forms—since the unification and replacement operations specified in the procedure have the intented effect in both situations.

Example 7.1.2:

In order to illustrate case 1 and case 2 of the proof, we partially execute the following higher-order DCG, which extends the grammar of the previous example with determiners and nouns:

```
(r1)  s((A B)) --> np(A), vp(B).
(r2)  np(Y\(Y A)) --> pn(A).
(r3)  np(Z\(A Z B)) --> det(A),n(B).
(r4)  det(A\B\(exists C\(and (B C) (A C)))) --> [a].
```

```
(r5)  det(A\B\(all C\(implies (B C) (A C)))) --> [every].
(r6)  vp(Z\(B (A Z))) --> tv(A), np(B).
(r7)  pn(mike) --> [mike].
(r8)  pn(mary) --> [mary].
(r9)  pn(john) --> [john].
(r10) n(A\(student A)) --> [student].
(r11) n(A\(professor A)) --> [professor].
(r12) tv(A\B\(saw A B)) --> [saw].
(r13) tv(A\B\(visited A B)) --> [visited].
```

Rules (r1), (r2) and (r6) are partially executed as shown in Example 7.1.1. Rule (r3) is executed as follows:

```
     np(Z\(A Z B)) --> det(A),n(B)
  =  np(Z\(U\V Z B)) --> det(U\V),n(B)
  =  np(Z\(V B)) --> det(Z\V),n(B)
  =  np(Z\(K\L B)) --> det(Z\K\L),n(B)
  =  np(Z\L) --> det(Z\B\L),n(B)
```

Case 1 is illustrated by the terminal rules for the determiners a and every, in which the function variables of the applications being reduced are prefix variables. For example, the rule for determiner a is partially executed in the following way:

```
     det(A\B\(exists C\(and (B C) (A C)))) --> [a].
  = det((K\L)\B\(exists C\(and (B C) (K\L C)))) --> [a].
  = det((C\L)\B\(exists C\(and (B C) L))) --> [a].
  = det((C\L)\(M\N)\(exists C\(and (M\N C) L))) --> [a].
  = det((C\L)\(C\N)\(exists C\(and N L))) --> [a].
```

The rule for determiner every is converted similarly:

```
     det(A\B\(all C\(implies (B C) (A C)))) --> [every].
  = det((C\L)\(C\N)\(all C\(implies N L))) --> [every].
```

Therefore, the complete partially executed DCG is:

```
(r1)  s(A) --> np(B\A), vp(B).
(r2)  np((A\B)\B) --> pn(A).
(r3)  np(A\B) --> det(A\C\B),n(C)
```

```
(r4)  det((C\A)\(C\B)\(exists C\(and B A))) --> [a].
(r5)  det((C\A)\(C\B)\(all C\(implies B A))) --> [every].
(r6)  vp(A\B) --> tv(A\C), np(C\B).
(r7)  pn(mike) --> [mike].
(r8)  pn(mary) --> [mary].
(r9)  pn(john) --> [john].
(r10) n(A\(student A)) --> [student].
(r11) n(A\(professor A)) --> [professor].
(r12) tv(A\B\(saw A B)) --> [saw].
(r13) tv(A\B\(visited A B)) --> [visited].
```

This scheme for partial execution effectively shifts the β-reductions into the argument positions of the grammar constituents. This is crucial for efficient reversibility because it enforces the constraints provided by the semantic representation as soon as each constituent is invoked, rather than to wait until all constituents have been executed nondeterministically.

7.2. Enhanced Partial Execution Procedure

7.2.1. Improved Treatment of Application Terms

The basic procedure of section 7.1 needs to be enhanced to take into consideration that not all applications with function variables should be reduced. For example, in representations such as Church numerals, the applications occurring in the numerals should not be partially executed, e.g., `F\X\(F X)` to `(X\B)\X\B`. The solution to this problem is to trace the β-reductions performed in the higher-order DCG (for the training sentences) to see which applications actually need to be reduced.

For each application `(A B)` occurring in the semantic terms of the grammar rules, we have to consider the following cases. Since all such terms are assumed to be reduced to normal form, `A` is either a constant or a variable, but not an abstraction. During execution of the grammar we can therefore distinguish the following two cases: (1) `A` remains a variable in the final semantic representation; (2) `A` will be bound to an abstraction so that the application `(A B)` will be reduced eventually. Assuming that such an application `(A B)` is either reduced in all training instances or is never reduced, one can distinguish accordingly which applications can be partially executed and which cannot.

Example 7.1.3:

Consider the following CFG:

```
s --> [0].
s --> [s,0].
s --> [s,s], s.
```

Using the training instances

Sentence	*Semantic representation*
[0]	F\X\X
[s,0]	F\X\(F X)
[s,s,0]	F\X\X
[s,s,s,0]	F\X\(F X)

the following higher-order DCG is obtained:

```
s(A\B\B) --> [0].
s(A\B\(A B)) --> [s],[0].
s(A\B\C\(A B C) D) --> [s],[s],s(D).
```

There is one application in the second rule, and three applications in the third rule. Only the applications in the third rule are actually β-reduced, as can be seen by executing the DCG on the training sentences. The first training sentence, `[0]`, uses only the first rule which has no applications. The second training sentence, `[s 0]`, uses only the second rule, which provides the correct semantic representation, `F\X\(F X)`, without reducing the application occurring in it. The third training sentence uses the third rule and the first rule. In order to obtain its semantic representation in reduced form, `F\X\X`, all applications in the third rule have to be reduced:

```
  (A\B\C\(A B C) D)
= (A\B\C\(A B C) F\X\X)
= B\C\(F\X\X B C)
= B\C\(X\X C)
= B\C\C
```

Therefore, the third rule can be partially executed accordingly:

```
s(A\B\C\(A B C) D) --> [s],[s],s(D).
s(B\C\(D B C)) --> [s],[s],s(D).
s(B\C\(K\L B C)) --> [s],[s],s(K\L).
s(B\C\(L C)) --> [s],[s],s(B\L).
s(B\C\F) --> [s],[s],s(B\C\F).
```

In certain cases a particular application is reduced in parsing certain sentences but not others.

Example 7.1.4:

Consider the following DCG:

```
s(A\B\B) --> [0].
s(A\B\(A B)) --> [s],[0].
s(A\B\B) --> [s],[s],[0].
s(A\B\C\(A D\C (B C)) E) --> [s],[s],[s],s(E).
```

This higher-order DCG cannot be partially executed with any of the schemes discussed so far. This is because the application in `A\B\(A B)` is reduced in computing the semantic representation for certain sentences but not others. One way to solve this problem would be to partially execute all applications, including those occurring in the final representations. For example, the term `A\B\(A B)` would reduced to `(B\D)\B\D`, and the rules of the above grammar would have to be changed accordingly:

```
s(A\B\B) --> [0].
s((B\D)\B\D) --> [s],[0].
s(A\B\B) --> [s],[s],[0].
s((C\H)\C\J) --> [s],[s],[s],s(((D\C)\H\J)).
```

If such a completely reduced DCG is used, not all generated semantic terms would be legal terms of the λ-calculus. Therefore, partial execution has to be reversed for some terms, e.g., `(B\D)\B\D` has to be converted back to `A\B\(A B)`, and `(D\B)\D\D`, which is generated for the sentence `[s,s,s,s,0]`, has to be converted to `E\D\D`.

A partially executed term is converted back to a legal λ-term by replacing any prefix term of the form `(A\B)` by a new variable `C`, and then replacing all occurrences of `B` by `(C A)`.

Example 7.1.5:

The conversion of `(C\B)\C\B` to `A\B\(A B)` is done by replacing `(C\B)` with the new variable `A`, and then replacing `B` with `(A B)`.

7.2.2. The Need for Copying

Another potential problem arises when a function returned by one of the constituents of a grammar rule is applied more than once. This can happen if not all substitution terms satisfy the linearity assumption. For example, consider the rule

```
a((B (B C))) --> b(B), c(C).
```

Now, if `B` is an abstraction then there are two applications to be performed in the term `(B (B C))`. However, the simple scheme for partial execution cannot be used here since the two applications have different arguments. Replacing `B` with `X\Y` would result in:

```
a(((X\Y) ((X\Y) C))) --> b(X\Y), c(C).
```

Now `X` would have to be bound to `((X\Y C))` *and* to `C`, which is impossible.

A solution to this problem would be to make a "copy" of the term returned by `b` for each occurrence of `B`.[16]. In the example above, we would have:

```
a((B1 (B2 C))) --> b(B1), copy(B1,B2), c(C).
```

The predicate `copy` produces a copy of `B1` so that the function represented by `B1` can be applied twice (to different arguments). Now `B1` can be replaced by `X1\Y1`:

```
a(((X1\Y1) (B2 C))) --> b((X1\Y1)), copy((X1\Y1),B2), c(C).
```

Reducing the applications gives:

```
a(Y1) --> b((B2 C)\Y1), copy(((B2 C)\Y1),B2), c(C).
```

Next, `B2` is replaced by `X2\Y2`:

```
a(Y1) --> b((X2\Y2 C)\Y1), copy((X2\Y2 C)\Y1,X2\Y2), c(C).
```

The final partially executed rule is obtained by reducing the remaining applications:

```
a(Y1) --> b(Y2\Y1), copy(Y2\Y1,C\Y2), c(C).
```

[16] A copy of term `B` leaves constants unchanged, but variables are consistently renamed; e.g., `foo(C,D,D)` would be a copy of `foo(A,B,B)`

Example 7.1.6:

A more systematic way to handle multiple applications of a function is illustrated by the following example. The term `F\X\(F (F X))` can be converted into reduced form in the following way:

```
    F\X\(F (F X))
=   F\X\(F1 (F2 X)),  copy(F, F1),  copy(F, F2)
=   F\X\(A1\B1 (A2\B2 X)),  copy(F, A1\B1),  copy(F, A2\B2)
=   F\X\(A1\B1 B2),  copy(F, A1\B1),  copy(F, X\B2)
=   F\X\B1,  copy(F, B2\B1),  copy(F, X\B2)
```

The reduced form is converted back into a regular λ-term as follows:

```
=   F\X\B1,  copy(F, B2\B1),  copy(F, X\B2)
=   F\X\(F1 B2),  copy(F, F1),  copy(F, X\B2)
=   F\X\(F1 (F2 X)),  copy(F, F1),  copy(F, F2)
=   F\X\(F (F X))
```

7.2.3. General Procedure for Partial Execution

The following procedure incorporates the enhancements of the previous subsection. Assume $HG0$ is the higher-order DCG constructed as described in Chapter 5 (none of the applications have been reduced yet). Assume $s_1 \ldots s_k$ are the training sentences used for the construction of $HG0$.

(1) Make a copy $HG1$ of $HG0$. ($HG1$ will be transformed into the partially executed DCG.)

(2) FOR $i = 1$ to k **DO**

(2a) Make a copy HG of $HG0$.

(2b) Execute HG on sentence s_i to instantiate all variables in the semantic representations of those rules of HG used for parsing s_i, without reducing any applications. (If a rule is used more than once by a sentence, it is sufficient to keep track of the instantiations resulting from the first use; i.e., instantiations of subsequent uses are ignored as they may not be unifiable with the instantiations resulting from the first use.)

(2c) Execute HG and $HG1$ simultaneously on sentence s_i in the following way: for each pair of rules $r \in HG$ and $r1 \in HG1$ being called during this execution, exhaustively reduce all applications of the semantic representations m and $m1$ associated with rules r and $r1$. Since m is already completely instantiated, it is used as a guide to decide which applications of $m1$ are supposed to be partially executed. If a particular abstraction is applied more than once in a term, a copy operation must be inserted in the body of $r1$, so that a separate copy of this abstraction is available for each β-reduction.

END FOR

(3) Asserting the elements of the final list $HG1$ yields the partially executed DCG, which can be directly executed by a Prolog interpreter.

By iterating through all training sentences all grammar rules are partially executed. There may be redundant partial executions of a rule if that rule is used by more than one training sentence. Unless there are application terms that should be reduced for some but not all training instances (as discussed above), there should be no conflicting partial executions. This procedure also ensures that an application is reduced only if it is supposed to be reduced according to the higher-order DCG.

To illustrate this procedure for partial execution consider the following CFG and training instances:

```
s --> [0].
s --> [sc,0].
s --> [sc,sc], s.
```

Sentence	*Semantic representation*
[0]	F\X\X
[sc,0]	F\X\(F X)
[sc,sc,0]	F\X\X
[sc,sc,sc,0]	F\X\(F X)

The higher-order DCG $HG0$ is as follows:

```
s(A\B\B) --> [0].
s(A\B\(A B)) --> [sc],[0].
s(A\B\C\(A B C) D) --> [sc],[sc],s(D).
```

In order to obtain a partially executed version we follow the procedure given above:

(1) First a copy $HG1$ of $HG0$ is made.

(2) First iteration ($i = 1$):

(2a) Another copy HG of $HG0$ is made.

(2b) HG is executed on the first training sentence `[0]` returning its semantic representation `A\B\B`.

(2c) Since there are no applications in the semantic representation returned by this sentence, there is no change in HG and $HG1$.

(2) Second iteration ($i = 2$):

(2a) A fresh copy HG of $HG0$ is made.

(2b) HG is executed on the second training sentence `[sc,0]` returning the semantic representation `A\B\(A B)`.

(2c) Since `A` in this term is not an abstraction there are again no β-reductions performed; that is, $HG1$ remains unchanged.

(2) Third iteration ($i = 3$):

(2a) A fresh copy HG of $HG0$ is made.

(2b) HG is executed on the third training sentence `[sc,sc,0]` returning as semantic representation `(A\B\C\(A B C) F\X\X)`.

(2c) HG and $HG1$ now look as follows:

HG :

```
s(A\B\B) --> [0].
s(A\B\(A B)) --> [sc],[0].
s(A\B\C\(A B C) A\B\B) --> [sc],[sc],s(A\B\B).
```

$HG1$:

```
s(A\B\B) --> [0].
s(A\B\(A B)) --> [sc],[0].
s(A\B\C\(A B C) D) --> [sc],[sc],s(D).
```

The term representing the semantics of the third rule of HG can be reduced by performing three β-reductions:

```
  (A\B\C\(A B C)  F\X\X)
= B\C\(F\X\X B C)
= B\C\(X\X C)
= B\C\C
```

which transforms the third rule of $HG1$ in the following way:

```
  s(A\B\C\(A B C) D) --> [sc],[sc],s(D).
= s(B\C\(D B C)) --> [sc],[sc],s(D).
= s(B\C\(K\L B C)) --> [sc],[sc],s(K\L).
= s(B\C\(L C)) --> [sc],[sc],s(B\L).
= s(B\C\(E\F C)) --> [sc],[sc],s(B\E\F).
= s(B\C\F) --> [sc],[sc],s(B\C\F).
```

There are no other applications to be performed in HG, therefore the process continues with the next training sentence.

(2) Fourth iteration ($i = 4$):

The iteration for the training sentence `[sc,sc,sc,0]` is basically the same as the third iteration and doesn't further change $HG1$.

(3) Therefore the final version of $HG1$ is:

```
s(A\B\B) --> [0].
s(A\B\(A B)) --> [sc],[0].
s(B\C\F) --> [sc],[sc],s(B\C\F).
```

which is the partially executed version of the original higher-order DCG.

7.3. Reversibility

Dymetman and Isabelle (1990) point out important theoretical and practical benefits of DCG *reversibility*. A DCG is reversible if it is possible to use it not only for computing

the semantic representation of each sentence of the language, but also for generating the set of sentences corresponding to a particular semantic representation. Reversibility makes it easier to construct DCGs which neither *overgenerate* nor *undergenerate*, i.e., they generate (or accept) all correct sentences for a particular semantic representation no more and no less. One benefit, of course, is that one doesn't need to write a separate DCG for generation. However, even if one is only interested in translating sentences into their semantic representations, it is in general useful to have the DCG detect ungrammatical sentences, rather than generating a "wrong" semantic representation without giving any indication that the parsed sentence is not part of the intended language. Without reversibility, it is hard to determine whether the DCG only accepts "correct" sentences, whereas a reversible DCG would generate all sentences for a particular semantic representation, and the grammar designer can easily check whether all of them should have this semantic representation.

The higher-order DCGs of Chapter 5 can be used for computing the semantics of a sentence quite efficiently, but not so efficiently for generating a sentence given its semantic representation. For example, if the higher-order rule $s((F\ A\ B)) \rightarrow np(A),\ vp(B)$ in the above grammar is used for parsing, the semantics A for np and B for vp are computed first, and then F is applied to A and B to obtain the semantics for s. If, however, the rule is used for generation, A and B would have to be assigned nondeterministically using higher-order unification.

7.3.1. Correctness for Reverse Execution

One of the motivations for developing a procedure for partial execution is its use for efficient reverse execution of DCGs. However, in certain cases correctness problems of reverse execution arise, as illustrated by the following example. The grammar

```
s(X\Y\Y) --> [0].
s(X\Y\X) --> [1].
```

will be expanded to the following clauses:

```
s(X\Y\Y, [0|T], T).
s(X\Y\X, [1|T], T).
```

When used in parsing mode, this grammar correctly computes `X\Y\Y` for `[0]`, and `X\Y\X` for `[1]` using the following queries:

```
?- s(Sem, [0], []).          --->   Sem = X\Y\Y
```

and

```
?- s(Sem, [1], []).          --->   Sem = X\Y\X
```

However, consider its use in generation mode, for example on the query:

```
?- s(A\B\B, Sent, []).
```

Now, both clauses match: the first clause matches by binding `A` to `X` and `B` to `Y`, and the second clause matches by binding all four variables to each other. Therefore, the incorrect sentence `[1]` would also be generated for the above query.

Next follows an analysis of the conditions under which reverse execution of partially executed DCGs is correct and the corresponding correctness proof. It has been shown in the previous section that a partially executed DCG G_p is correct in the forward direction; i.e., given a sentence s, it generates the correct semantic representation m_s, as defined by the original, higher-order DCG. We assume that the grammar is unambiguous, i.e., for each sentence there is only one semantic representation. However, there can be more than one sentence (paraphrases) for a particular semantic representation.

In order to ensure correctness when using the partially executed DCG in the reverse direction "freezing" is applied to the input semantic representation; that is, all of its variables are turned into constants (different constants are used for different variables). This operation prevents possible bindings of different variables to the same term or to each other.

Theorem 7.3.3: Assuming that an input semantic representation m has no free variables, reverse execution of a partially executed DCG D from the frozen form n of m computes exactly the set of sentences whose semantics is m.

Proof: Let s_1, ..., s_k be all the sentences with semantics m. By the correctness of forward execution (from s_1, ..., s_k) and the completeness of SLD-resolution, there exists a reverse execution from m for computing each of these sentences. Such a derivation would not instantiate any variable in m (since these represent binder variables of lambda-terms). Hence there is also a successful reverse execution from the frozen form n of m.

The remaining issue is that of soundness of reverse execution: to show that reverse execution does not compute any incorrect sentence from n. We should prove that the only way that reverse execution can compute an incorrect sentence is by instantiating some variables of m. Suppose otherwise, i.e., suppose that reverse execution of m computes an

incorrect sentence s without instantiating any variables of m. By completeness of SLD-resolution, there is a forward execution from s that computes m. But this contradicts the assumption that s was an incorrect sentence. Now the proof is easy to complete: freezing variables in m prevents any possible instantiation, and therefore reverse execution from n is correct. **Q.E.D.**

7.3.2. Efficient Control for Reverse Execution

One of the problems of bidirectional grammars is the inefficiency (due to nondeterminism) in the generation mode.

Example 7.3.1:

If the grammar below is invoked with `s((saw john mary))`, the subgoal `np(D)` will be executed completely uninstantiated; i.e., all possibilities for this subgoal have to be explored until one is found that is compatible with the subgoal `vp(D\(saw john mary))`.

```
s(A) --> np(D),vp(D\A).
np(A) --> pn(A).
vp(A\B) --> tv(A\E\B),np(E).
pn(john) --> [john].
pn(mary) --> [mary].
tv(A\B\ (saw A B)) --> [saw].
```

At first it may seem that problem can be solved by changing the order in which these subgoals are executed, but in certain cases there does not exist an ordering that avoids all such unconstrained executions.

Example 7.3.2:

Assume the following grammar is invoked by the query `s((saw john mary),S,[])`. (The DCG shown below is in definite-clause format.)

```
s(A,W,V) :- np(D\A,W,U),vp(D,U,V).
np((A\B)\B,W,V) :- pn(A,W,V).
vp(A\B,W,V) :- tv(A\E,W,U),np(E\B,U,V).
pn(john,[john|V],V).
pn(mary,[mary|V],V).
tv(A\B\(saw A B),[saw|V],V).
```

The subgoal `np(D\(saw john mary),S,U)` should be executed first since it is constrained by the given semantic representation. However, the rule for `np` will then call the goal `pn(A,S,U)` which again is completely uninstantiated. The solution to this problem is to *suspend* the goal `pn(A,S,U)` and select the goal `vp(D,U,[])` for execution at this point, because `D` has been instantiated to `A\(saw john mary)` when the rule for `np` was invoked.

This technique can be implemented by maintaining a list of goals (initially containing the goal entered at the top level). On each iteration, one goal from this list whose semantic representation contains constraints from the original goal is selected, and it is replaced by the subgoals on the right-hand side of the grammar rule whose head unifies with it. This process continues until the list is empty.

Example 7.3.3:

In the example above, the list would be initially `[s((saw john mary),S,[])]`. Next, the only element of that list is selected and replaced by the right-hand side of the rule for `s`, so that the list becomes `[np(D\(saw john mary),S,U),vp(D,U,[])]`. Next the goal `np(D\(saw john mary),S,U)` is selected and replaced by `pn(D,S,U)`. The list now is `[pn(A,S,U), vp(A\(saw john mary),U,[])]`. The goal `vp(A\(saw john mary),U,[])` is selected next, etc. Below is a complete trace.

List	*Subst. for* `S`	*Select*
`[s((saw john mary),S,[])]`	`S`	`s(...)`
`[np(D\(saw john mary),S,U),vp(D,U,[])]`	`S`	`np(...)`
`[pn(A,S,U), vp(A\(saw john mary),U,[])]`	`S`	`vp(...)`
`[pn(A,S,U), tv(A\E,U,V), np(E\(saw john mary),V,[])]`	`S`	`np(...)`
`[pn(A,S,U), tv(A\(A1\(saw john mary)),U,V), pn(A1,V,[])]`	`S`	`tv(...)`
`[pn(john,S,[saw|V]), pn(mary,V,[])]`	`S`	`pn(john)`
`[pn(mary,V,[])]`	`[john,saw|V]`	`pn(mary)`
`[]`	`[john,saw,mary]`	

As can be seen, no search (nondeterminism) is involved.

7.3.3. Application of Reversibility

The benefits of bidirectionality become apparent in the construction of natural language interfaces to data bases or knowledge bases. For demonstration purposes a simple meta-interpreter of DCG-rules has been implemented. It interprets and answers questions that can be parsed by the DCG. In order for the grammar to parse questions, the following kinds of rules have been added to the DCG:

```
np --> wh([who]).
np --> wh([what]).
vp --> wh([did,what]).
vp --> wh([does,what]).
etc.
```

meaning that a noun phrase can be replaced by the words `[who]` or `[what]`, and a verb phrase can be replaced by `[did,what]` or `[does,what]`. Any nonterminal of the grammar can obtain such rules.

When a statement is parsed, its semantic representation is simply added to the data base (knowledge base). However, if a question is parsed the interpreter discards the interrogative words (who, what, etc.), and tries to generate a sequence of words for the corresponding grammar constituent such that there exists a fact in the data base (or one can be inferred) whose corresponding natural language sentence is identical to the sentence completed by the interpreter. The sequence of words generated by the interpreter for the missing grammar constituent thus is the answer to the question. If there are multiple answers, all of them are generated. A question that doesn't contain any interrogative words is a yes/no question, and the answer returned by the interpreter is "yes" if the corresponding fact can be inferred from the data base, and "I don't know" otherwise.

Below is a demonstration of how such a DCG (see Appendix C) can be used to answer questions.

```
|: john saw mary.                              % Statement entered by the user.
                     (saw john mary).          % Semantic representation
                                               % generated by the grammar.
|: mary was seen by every smart person.

                     (all A\ (implies  (and  (person A)  (smart A))
                                       (saw A mary))).
```

```
|: every student is a smart person.

                    (all A\ (implies  (student A)
                                      (and  (person A) (smart A)))).

|: nancy is a student.
                    (student nancy).

|: who saw mary?                              % Question entered by the user.
  - john.                                     % The interpreter returns three
  - every smart person.                       % answers.
  - nancy.

|: every person that read a book is a student.

                    (all A\ (implies  (and  (person A)
                                            (exists B\ (and  (book B)
                                                             (read A B))))
                                      (student A))).

|: mike saw a professor that read every book.

                    (exists A\ (and  (and  (professor A)
                                           (all B\ (implies  (book B)
                                                             (read A B))))
                                     (saw mike A))).

|: mike saw who?
  - a professor that read every book.

|: mike saw a professor that read what?
  - every book.

|: mike saw a professor that did what?
  - read every book.

|: mike did what?
  - saw a professor that read every book.

|: who saw who?
  - john  - mary.
  - every smart person  - mary.
  - mike  - a professor that read every book.
```

```
  - nancy  - mary.

|: who did what?
  - john  - saw mary.
  - every smart person  - saw mary.
  - mike  - saw a professor that read every book.
  - nancy  - saw mary.

|: who does what?
  - every student  - is a smart person.
  - nancy  - is a student.
  - every person that read a book  - is a student.
  - nancy  - is a smart person.

|: john saw mary?                                % Yes/no questions.
  - yes.

|: bertrand saw mary?
  - I don't know.
```

Not all logically entailed answers are generated in this demonstration since only a very simple, incomplete inference engine was used.

7.4. Summary

In this chapter I have described a general procedure for converting higher-order DCGs into first-order DCGs. This conversion can be considered a form of partial execution since all β-reductions that would be reduced at execution time in the higher-order DCG are effectively reduced at "compile time", that is, during the conversion. Execution is more efficient, especially in the reverse direction, since first-order unification is more efficient than higher-order unification.

This type of partial execution is correct in the sense that the partially executed DCGs compute the same semantic representations as the corresponding higher-order DCGs. Also, partially executed DCGs can be used to compute the correct sentences for a given semantic representation if they are used in conjunction with a simple meta-interpreter that eliminates incorrect results.

8. Application to Natural Query Languages

The development of natural query languages, that is, subsets of natural languages suitable for querying databases or knowledge bases, is becoming increasingly important as the number and sizes of databases increases. Therefore, a system that can automatically synthesize or modify natural query language interfaces would be desirable. Such query languages typically satisfy the requirements of the system discussed in this dissertation, such as non-ambiguity, and the semantic representations can often be easily adapted to the λ-calculus. This chapter discusses issues related to the application of my system to realistically sized query languages.

In order to synthesize larger DCGs from examples, the syntactic rules and the training instances must be specified in such a way that the resulting higher-order equations can be solved reasonably fast. In the following sections I describe a methodology for developing the syntactic rules in an incremental, modular fashion to facilitate efficient synthesis of large grammars. The DCGs generated by the system can be used in basically two ways. Either a grammar is used to directly convert sentences into structures that form the input to some application, or two grammars are used, one in parsing mode, the other in generation mode, to translate from one language to another, with appropriate semantic representations as an interlingua. Section 8.3 discusses one application of each type.

8.1. Pragmatic Enhancements to SYNTH

8.1.1. Type Assignment and Type Inference

The higher-order unification procedure used by SOLVE requires knowledge of the types for variables. However, specifying types for the training instances is optional, since the system uses the conventions given below to assign primitive and function types. Together with the type inference mechanism implemented by procedure CHECK, these initial type assignments allow the types of all terms to be completely determined during the unification process. In the current implementation, only one primitive type i is used, so that for example "individuals" and "booleans" are not differentiated.

The semantic representations given by the user are assigned types in the following way:

(1) Since the semantic representations given by the user are assumed to be ground, each function application (together with *all* of its arguments) is assigned primitive type i. E.g., `(saw john mary)` has type i. Constants `john` and `mary` have type i as well. Therefore, `(saw john mary)` would be augmented in the following way: `(saw john:`i `mary:`i `):` i.

(2) Function types are inferred from the types of the arguments in the obvious way: the type of the term T at the function position of a term is a function type of the form $\alpha_1 \rightarrow (\alpha_2 \rightarrow (\alpha_3 \rightarrow \ldots))$, where the first argument of T is of type α_1, the second argument of type α_2, etc. (α_1, α_2, etc. are type variables). For example, if `mary` is of type i and `(saw john mary)` is of type i then `(saw john)` must be of type $i \rightarrow i$, and `saw` must be of type $i \rightarrow (i \rightarrow i)$. Therefore, the whole term would be further augmented as follows:

`((saw:`$(i \rightarrow (i \rightarrow i))$ `john:`i`):`$(i \rightarrow i)$ `mary:`i`):` i.

(3) If a term has prefix variables it must be a function that takes as arguments terms of the same types as the corresponding prefix variables. For example, the term `X\(saw X mary)` would be a function of type $i \rightarrow i$, and `X` would be of type i.

This scheme for type assignment has proved sufficient for all the test cases I have investigated, and I believe it will work for most practical applications. However, the user is free to override this scheme by providing types along with the semantic representations. In natural language semantics separate primitive types for individuals and booleans are typically used. However, this distinction is not necessary for SYNTH to correctly infer semantic representations (types are not part of the final DCGs).

Example 8.1.1:

In example 5.2.1 the CFG

```
s --> [a].
s --> [a], s.
```

was converted into a higher-order DCG using the following training instances:

Sentence	*Semantic representation*
`[a]`	`F\X\X`
`[a,a]`	`F\X\(F X)`
`[a,a,a]`	`F\X\(F (F X))`

which gave rise to these higher-order equations:

```
F1 = F\X\X
(F2 F1) = F\X\(F X)
(F2 (F2 F1)) = F\X\(F (F X)).
```

If no types are given for the semantic representation given by the user, the system would assign type i for `X`, `(F X)`, and `(F (F X))` according to the rules described above. Since both `X` and `(F X)` have type i, the type of `F` is inferred to be $i \rightarrow i$. Therefore, using these conventions for type assignment and type inference, the same types are obtained as those given in Chapter 5.

Example 8.1.2:

Consider the following set of higher-order equations from example 5.2.2:

```
{(F1 F2 F4) = (run shrdlu),
 (F1 F3 F4) = (run eliza),
 (F1 F2 F5) = (halt shrdlu)}.
```

Following the above conventions for type assignment, the system assigns the elementary type i to the semantic representation given by the user and the non-function constants in those terms; that is, $\tau($`(run shrdlu)`$) = i$, $\tau($`(run eliza)`$) = i$, $\tau($`(halt shrdlu)`$) = i$, $\tau($`shrdlu`$) = i$, and, $\tau($`eliza`$) = i$. Therefore, $\tau($`run`$) = i \rightarrow i$ and $\tau($`halt`$) = i \rightarrow i$. Recall that the system uses i for both individual and boolean types.

8.1.2. Semantic Rules Provided by the User

Users can optionally provide the semantic rules for any grammar rules for which they happen to know the semantic rules. Providing this information for terminals is often easy. For example, most users who can provide the semantic representations for the sentences generated by the following CFG, could probably also specify the semantic representations for the terminals `program`, `computer`, `runs`, and `halts`.

```
s --> np, iv.
np --> det, n.
det --> [a].
det --> [every].
n --> [program].
n --> [computer].
iv --> [runs].
iv --> [halts].
```

A natural semantics for these terminals would be respectively `A\(prog A)`, `A\(comp A)`, `A\(run A)`, and `A\(halt A)`, and this information can be provided as follows:

```
n(A\(prog A)) --> [program].
n(A\(comp A)) --> [computer].
iv(A\(run A)) --> [runs].
iv(A\(halt A)) --> [halts].
```

Users may also specify semantic rules for individual grammar rules. For example, for the rule `s --> np, iv`, the semantics for `s` is computed by applying the semantics of `np` to the semantics of `iv`. This information could be communicated to the system simply by augmenting the original CFG rule as follows:

```
s((A B))--> np(A), iv(B).
```

Providing semantics along with grammar rules is an effective way reduce the search space since the number of free variables for which substitutions need to found is reduced. It gives rise to a different set of equations than described in Chapter 5, because the augmented grammar constructed at step 2 of procedure SYNTH can now contain ground terms in addition to function variables. For example, including the semantics for the terminals and the first grammar rule, the above grammar will be converted into the following DCG:

```
s((A B))--> np(A), iv(B).
np((F1 A B) --> det(A), n(B).
det(F2) --> [a].
det(F3) --> [every].
n(A\(prog A)) --> [program].
n(A\(comp A)) --> [computer].
iv(A\(run A)) --> [runs].
iv(A\(halt A)) --> [halts].
```

Now, executing this DCG on the training sentence `[a,program,runs]`, will produce the semantic representation

```
((F1 F2 A\(prog A)) A\(run A)).
```

If the user specifies `(run prog)` as the semantic representation for `[a,program,runs]`, the following equation is obtained:

```
((F1 F2 A\(prog A)) A\(run A)) = (run prog).
```

Similar equations would be set up for the other training sentences. However, elementary types can now also be assigned to subterms of the left-hand side terms using the following rule: If a term is not completely ground but contains a subterm that has a constant at its function position, that function application together with all of its arguments is assigned type i. For example, in the above equation we can assign type i to the subterms `(prog A)` and `(run A)`.

This feature can also be used to incrementally synthesize the semantics for larger grammars. Large grammars often can be broken down into independent smaller grammars (modules) that still parse complete sentences but are less complex. These modules can then be trained separately. The semantic rules found for a set of grammar rules this way can then be provided directly with those rules when the modules are merged, so that most of the semantics is already fixed when the system searches for consistent augmentations of the whole grammar, which can cut down time complexity considerably.

8.1.3. Generalization of Lexical Rules

The scope of an augmented DCG can be enlarged by exploiting the similarity of the representations of words of the same syntactic category. In most cases it is possible to generalize the specific associations between words and their λ-term meanings to the categories of those words and their corresponding meanings. For example, in a machine translation application, assuming the representation for any transitive verb TV is of the form $\lambda x.\lambda y.TV(y,x)$, then the representation of a transitive verb of another language could sometimes be obtained by first consulting a dictionary to obtain the root of the corresponding English word, and then extending the root into the λ-term. It should be noted, however, that my system does *not* rely on the fact that a word (or its root) occurs in its semantic representation. In order for the system to infer those semantic representations from the examples, a word does not need to bear any resemblance to its semantic representation.

8.2. Methodology for Larger Applications

8.2.1. Compositionality and Grammatical Structure

In section 5.3 it was noted that the grammatical rules must be specified in such a way that they reflect the compositionality of the language with respect to the λ-calculus. In this

section I discuss a methodology for developing the syntactic rules for large applications in such a way that compositionality is maintained.

Basically, the syntax must be defined such that the corresponding semantics is compositional with respect to the typed λ-calculus; i.e., for any grammar rule there must exist a function in the typed λ-calculus that computes the semantic representation for the phrase corresponding to that rule from the semantic representations of the constituents. For example, if the semantic representation of a phrase has a right-recursive structure the phrase must be generated by right-recursive grammar rules rather than left-recursive rules (see example given in Appendix E.2).

As another example of how semantics imposes restrictions on the syntactic rules, consider the following four pairs of sentences and representations from the CHAT-80 language:

Example 8.2.1:

```
(1)   what is bordering italy?
      X\(borders X italy)

(2)   what is not bordering italy?
      X\(not (borders X italy))

(3)   what is bordering any country?
      X\(exists C\(and (country C) (borders X C)))).

(4)   what is not bordering any country?
      X\(not (exists C\(and (country C) (borders X C))))).
```

The syntax for cases (1) and (2) can be naturally expressed by the following rules:

```
s --> [what,is], vppr, [?].
vppr --> tv_pr_not, pn.
tv_pr_not --> tv_pr.
tv_pr_not --> [not], tv_pr.
tv_pr --> [bordering].
```

The corresponding augmentations would be:

```
s(B) --> [what,is],vppr(B),[?].
vppr((C D)) --> tv_pr_not(C),pn(D).
```

```
tv_pr_not(B\C\(D C B)) --> tv_pr(D).
tv_pr_not(B\C\(not (D C B))) --> [not],tv_pr(D).
tv_pr(A\B\(borders A B)) --> [bordering].
```

Note that the scope of `not` is just the verb (`tv_pr`). However, in case (4) the scope of the negation should be the whole expression headed by `exists`. That is, the simple approach of generalizing the rule

```
vppr --> tv_pr_not, pn.
```

to

```
vppr --> tv_pr_not, np.
```

where

```
np --> pn.
np --> [any], noun.
```

would not allow compositional semantics. Instead new rules for `vppr` need to be introduced:

```
vppr --> [not], vppr_any.
vppr --> vppr_any.
vppr_any --> tv_pr, [any], np0.
```

The corresponding augmentations are:

```
vppr(B\(not (C B))) --> [not],vppr_any(C).
vppr(B) --> vppr_any(B).
vppr_any(C\(exists D\(and (F D) (E C D)))) -->
   tv_pr(E),[any],np0(F).
```

8.2.2. Reversible Grammars

The syntactic rules should be written such that the corresponding DCG does not overgenerate. That is, if the DCG is used in the reverse mode, only grammatically and semantically correct sentences should be generated for a particular semantic representation. The following example illustrates grammatical overgeneration, that is, generation of sentences from a particular semantic representation that are semantically correct but grammatically incorrect:

```
s((B A)) --> np(A), vp(B).
np(john) --> [john].
np(people) --> [people].
vp(X\(run X)) --> [runs].
vp(X\(run X)) --> [run].
```

This DCG computes the semantic representation `(run john)` for the sentence `[john runs]`, and `(run people)` for `[people run]`. However, when used in the reverse direction, this DCG computes both `[john runs]` *and* `[john run]` for the semantic representation `(run john)`. Similarly, two sentences, one grammatically correct the other one incorrect, are generated for the semantic representation `(run people)`.

Grammatical overgeneration is best eliminated by using attributes that are added as arguments to the grammar symbols. For example, number and gender agreement can be achieved by such attributes. For the above DCG this could be achieved, for example, by adding arguments that prevent incorrect rule combinations in the following way:

Example 8.2.2:

```
s((B A),N) --> np(A,N), vp(B,N).
np(john,singular) --> [john].
np(people,plural) --> [people].
vp(X\(run X),singular) --> [runs].
vp(X\(run X),plural) --> [run].
```

See Appendix C for a more elaborate example of this technique.

Semantic overgeneration is exemplified by the following DCG:

Example 8.2.3:

```
s((A B)) --> np(A), vp(B).
vp((B A)) --> ntv(A), np(B).
vp(C\(not (A B C))) --> [didn't], ntv1(A), np(B).
ntv(B\C\(not (D C B))) --> [didn't], tv(D).
ntv(B\C\(D C B)) --> [did], tv(D).
ntv1(B\C\(exists D\(and (B D) (E C D)))) --> tv(E).
np(B\(B A)) --> pn(A).
np(A) --> [any], noun(A).
```

```
np(B\C(exists E\(and (A E) (B E C)))) --> [a], noun(A).
tv(A\B\(read A B)) --> [read].
tv(A\B\(know A B)) --> [know].
pn(john) --> [john].
pn(principia) --> [principia].
noun(X\(logician X)) --> [logician].
```

This DCG correctly parses the following types of sentences and generates their semantic representations as indicated.

```
(1)    john didn't read principia
       (not (read john principia))

(2)    john didn't know any logician
       (not (exists X\(and (logician X) (know john X))))

(3)    mary did know a logician
       (exists X\(and (logician X) (know mary X)))
```

However, for the semantic representation

```
       (exists X\(and (logician X) (not (know john X))))
```

whose meaning should be "there exists a logician that john didn't know", the above DCG would generate the somewhat ungrammatical sentence

```
       john didn't know a logician
```

which is likely to be understood as "john didn't know any logician."

The compositionality constraint takes care of a large part of the overgeneration problem since it requires that separate rules be given for semantically different phrases. Overgeneration in example 8.2.3 can be eliminated by separating the rules for certain nonterminals just as the rules for `vppr` were separated in example 8.2.1 to achieve compositionality.

If the training instances are used as a basis for specifying the syntax of the language, one can consider the process of constructing syntactic rules in the following way: Initially there is only one grammar rule for each training sentence. Each of them has the starting symbol on the left-hand side and a sequence of words (terminals) making up the training

sentence on the right-hand side. Generalization takes place by replacing certain sequences of terminals and nonterminals with a nonterminal that may be used repeatedly in the same or other grammar rules. A new rule is added which has that nonterminal on the left-hand side and the sequence of terminals and nonterminals that it replaced on the right-hand side.

As a simple example consider the following sentences (1)-(3).

```
(1)   s --> [john], [met], [mary].
(2)   s --> [mike], [met], [mary].
(3)   s --> [john], [met], [mike].
```

This grammar can be generalized by introducing a new nonterminal in the following way:

```
(4)   s --> pn1, [met], [mary].
(5)   s --> pn1, [met], [mike].
(6)   pn1 --> [john].
(7)   pn1 --> [mike].
```

This can be further generalized as follows:

```
(8)   s --> pn1, [met], pn2.
(9)   pn2 --> [mary].
(10)  pn2 --> [mike].
```

Further generalization is obtained now by merging pn1 and pn2:

```
(11)  s --> pn, [met], pn.
(12)  pn --> [john].
(13)  pn --> [mike].
(14)  pn --> [mary].
```

This process can be continued for example by changing (11) to:

```
(15)  s --> pn, vp.
(16)  vp --> [met], pn.
```

etc.

As demonstrated by examples 8.2.1 and 8.2.3, both the compositionality and reversibility constraint work to inhibit this generalization process in certain situations. This means, in general achieving compositionality increases reversibility and vice versa.

8.2.3. Efficiency

In general the syntactic rules can be written in many different ways without affecting the language defined by those rules. Similarly, there are many "minimal" sets of training instances that define the same unique augmentation. Even though the final DCGs are equivalent, the choice of syntactic rules and training instances can significantly affect the time and space requirements for computing the correct augmentation.

For example, one can always add an arbitrary number of "chain rules" to the set of rules specifying the syntax without changing the language defined by that grammar. However, each unnecessary chain rule will introduce an additional function variable for which a substitution needs to be found. Having more function variables than necessary implies that there are many solutions to the higher-order equations. Even though all such solutions lead to DCGs that are equivalent in terms of input/output behavior, the time complexity to find just one solution can increase dramatically since all branches have to be explored even if they are not the source of the failure. This problem is partly alleviated by the dependency directed backtracking scheme discussed earlier.

It is also important for efficiency that training instances are provided in order of increasing size. Each training instance should involve as few rules/function variables as possible, so that the constraints given by the higher-order matching problem take effect as early as possible and the search space is kept small. Training instances involving a large number of rules/function variables should be provided only after as many of those rules as possible have be trained individually, so that some of the function variables are already instantiated.

8.2.4. Type Raising

Type raising (Andrews 1986) is the conversion of a semantic representation to a term that has a higher type. A term t_1 has a higher type than a term t_2 if the type of one of the arguments of t_1 is higher than all the types of the arguments of t_2. The type of a function that only takes terms of primitive types as arguments is higher than any of these primitive types. A term t can be type raised by applying `A\B\(B A)` to it. The type of the resulting term, `B\(B t)`, is of the form $\beta \rightarrow \gamma$, and the type of `B` is of the form $\alpha \rightarrow \gamma$, where α is the type of t. This means $\beta = (\alpha \rightarrow \gamma)$, so that the type of `B\(B t)` is actually of the form $(\alpha \rightarrow \gamma) \rightarrow \gamma$. Therefore, whatever the type α of t, the type of `B\(B t)` will be one level higher.

Type raising is necessary in certain cases so that semantic representations from different parts of the grammar have the same type if they are used by the same grammar

symbol. This is illustrated in the grammar below where a noun phrase can be either a proper noun or a quantified noun phrase. This example also shows how type raising is automatically performed by the higher-order unification procedure.

```
s --> np, iv.
np --> pn.
np --> det, n.
det --> [a].
det --> [every].
n --> [program].
n --> [computer].
iv --> [runs].
iv --> [halts].
pn --> [shrdlu].
pn --> [eliza].
```

Sentence	*Semantic representation*
[shrdlu,runs]	(run shrdlu)
[eliza,runs]	(run eliza)
[shrdlu,halts]	(halt shrdlu)
[a,program,runs]	(exists X\(and (program X) (run X)))
[every,program,runs]	(all X\(implies (program X) (run X)))
[a,computer,runs]	(exists X\(and (computer X) (run X)))
[a,program,halts]	(exists X\(and (program X) (halt X)))

The following higher-order DCG is obtained:

```
s((C D)) --> np(C),iv(D).
np(B\(B C)) --> pn(C).
np(C\(D C E)) --> det(D),n(E).
det(A\B\(exists C\(and (B C) (A C)))) --> [a].
det(A\B\(all C\(implies (B C) (A C)))) --> [every].
n(A\(program A)) --> [program].
n(A\(computer A)) --> [computer].
iv(A\(run A)) --> [runs].
iv(A\(halt A)) --> [halts].
pn(shrdlu) --> [shrdlu].
pn(eliza) --> [eliza].
```

where the first three rules can be reduced to:

```
s((C D)) --> np(C),iv(D).
np(B\(B C)) --> pn(C).
np(C\(D C E)) --> det(D),n(E).
```

As can be seen, a proper noun is "type raised", and thus being made type-compatible with quantified noun phrases.

8.3. Case Studies

8.3.1. CHAT-80′

This section discusses the application of my DCG-synthesis system to a variation of the natural query language of the CHAT-80 system (Warren & Pereira 1982). I call this variation CHAT-80′. CHAT-80 is a system, implemented in Prolog, that stores geographic information and accepts queries about this domain in English. It uses extraposition grammars, a logic-based grammar formalism, to translate English questions into the Prolog subset of first-order logic. The resulting logical expression (semantic representation) is then transformed by a planning algorithm into an optimized Prolog query, which is executed to yield the answer. Below are a few typical queries together with their semantic representations and the responses of the CHAT-80 system.[17]

```
Question:  Does Afghanistan border China?
Semantics: (borders afghanistan china)
Answer:    yes.

Question:  Which country's capital is Ouagadougou?
Semantics: C\(and (country C) (capital C ouagadougou))
Answer:    C = upper_volta.

Question:  Which is the ocean that borders African countries and
           that borders Asian countries?
Semantics: X\(and (ocean X) (and (country C) (and (african C)
           (and (borders X C) (and (country C1) (and (asian C1)
           (borders X C1)))))))
```

[17] I have changed the syntax for the semantic representations slightly so that it matches the λ-calculus syntax notation used in this dissertation.

```
Answer:    X = indian_ocean.

Question:  What is the capital of each country bordering the Baltic?
Semantics: C\X\(and (country C) (and (borders C baltic)
                                    (capital C X)))
Answers:   C-X = denmark-copenhagen,
           C-X = east_germany-east_berlin,
           C-X = finland-helsinki,
           C-X = poland-warsaw,
           C-X = soviet_union-moscow,
           C-X = sweden-stockholm,
           C-X = west_germany-bonn.

Question:  What are the latitudes of the countries north of the
           United Kingdom?
Semantics: C\Ls\(and (country C)
                     (and (northof C united_kingdom)
                          (setof L\(latitude C L) Ls)))
Answers:   C-Ls = canada-60 degrees,
           C-Ls = denmark-55 degrees,
           C-Ls = finland-65 degrees,
           C-Ls = iceland-65 degrees,
           C-Ls = norway-64 degrees,
           C-Ls = soviet-union-57 degrees,
           C-Ls = sweden-63 degrees.

Question:  Which country is bordered by two seas?
Semantics: C\(and (country C)
                  (number_of X\(and (sea X) (borders C X)) 2))
Answers:   C = egypt, C = iran, C = israel, C = saudi_arabia,
           C = turkey.

Question:  How many countries does the Danube flow through?
Semantics: N\(number_of C\(and (country C) (flows danube C)) N)
Answer:    N = 6.
```

```
Question:  From what country does a river flow into the Persian
           Gulf?
Semantics: C\(and (river R)
                  (and (country C) (flows R C person_gulf)))
Answer:    C = iraq.

Question:  What is the total area of countries south of the
           Equator not in Australasia?
Semantics:
 T\(and (setof A\C\(and (area C A)
                        (and (country C)
                             (and (southof C equator)
                                  (not (in C australasia))))) S)
        (aggregate total S T))
Answer:    T = 10228 ksqmiles.

Question:  Which country bordering the Mediterranean borders a
           country that is bordered by a country whose
           population exceeds the population of India?
Semantics:
 C\(and (country C)
        (and (borders C mediterranean)
             (and (country C1)
                  (and (country C2)
                       (and (population C2 X)
                            (and (population indea Y)
                                 (and (exceeds X Y)
                                      (and (borders C2 C1)
                                           (borders C C1)))))))))
Answer:    C = turkey.
```

As exemplified by the above queries, CHAT-80 allows yes/no questions, single answer questions, and multiple answer questions. The semantic representations of queries may involve abstractions such as `(setof L\(latitude C L) Ls)`, or `(number_of X\(and (sea X) (borders C X)) 2)`. The semantics of quantifications and aggregations such

as "how many" and "total area" provide particular challenges regarding compositionality as can be seen below. The last query above illustrates how essentially unlimited qualifications can be used in a question.

The aspect of CHAT-80 that is of interest in of this dissertation is the translation of the natural language queries into their semantic representations. I will show that the DCG synthesis system can generate such a translater by specifying the grammar along with typical input/output examples. For larger grammars it is recommended that the translater (the DCG) be synthesized in a modular fashion; that is, the DCGs for small independent subgammars should first be synthesized separately, if possible. If independent subgrammars cannot be isolated, a grammar should be developed incrementally, starting with short sentences and simple semantics. This allows efficient computation of the semantics of many grammar rules and terminals so that the search space is reduced when the semantics for more complex sentences needs to be determined. To demonstrate this point, consider the fraction of the syntax of the CHAT-80′ language exemplified by the following training instances (training instances are specified by a ternary predicate `train(Number,Sentence,Semantics)`, where `Number` would be a parameter that can be used for indexing, but is not used by the current system).

```
train(_,[does,italy,border,france,?],
        (borders italy france)).
train(_,[does,italy,border,a,sea,?],
        (and (sea X) (borders italy X))).
```

A possible grammar for such sentences is:

```
s --> [does], pn, vp0, [?].
vp0 --> tv, pn.
vp0 --> tv, [a], np0.
pn(france) --> [france].
pn(italy) --> [italy].
np0 --> n1.
n1(X\(sea X)) --> [sea].
tv(X\Y\(borders X Y)) --> [border].
```

If the semantic representations for some of the rules are already known, the rules can be augmented as shown for the terminals `france`, `italy`, `sea`, and `border` in this case. The semantics will be inferred for any rules or terminals for which it is not given.

The above training instances and grammar rules will lead to the following initial set of higher-order equations each of which is given as a list of two elements, `[D1,D2]`, where `D1` is the left-hand side and `D2` the right-hand side of the equation. (Note that the outermost parentheses of a λ-term may be omitted.)

```
[F1 italy (F2 K\L\(borders K L) france),
    borders italy france],
[F1 italy (F3 K\L\(borders K L) (F58 K\(sea K))),
    and (sea H2) (borders italy H2)],
```

Solving these equations for the function variables, and substituting in the DCG in the usual way leads to the following higher-order DCG:

```
s((B A)) --> [does],pn(A),vp0(B),[?].
vp0((A M B)) --> tv(A),pn(B).
vp0(M\(and (B N) (A M N))) --> tv(A),[a],np0(B).
pn(france) --> [france].
pn(italy) --> [italy].
np0(K\K A) --> n1(A).
n1(X\(sea X)) --> [sea].
tv(X\Y\(borders X Y)) --> [border].
```

Let us next consider a grammar for the following two examples:

```
train(_,[what,is,the,capital,of,italy,?],
   Y\(capital italy Y)).
train(_,[what,is,the,capital,of,each,country,?],
   X\Y\(and (country X) (capital X Y))).
```

Such a grammar would be:

```
s --> [what,is,the], n2, [of], pn, [?].
s --> [what,is,the], n2, [of,each], np0, [?].
pn(italy) --> [italy].
np0(K\K A) --> n1(A).
n1(X\(country X)) --> [country].
n2(X\Y\(capital Y X)) --> [capital].
```

Again, we assume that the semantic representations for the terminals are known (the order of the arguments `X` and `Y` of `capital` is not critical; both choices lead to equivalent solutions). The semantics for the rule `np0 --> n1` has been determined by the other grammar fraction above and can therefore also be provided at this point. Hence only the semantics for the first two rules remain to be determined. The following equations are used to infer the semantics for those two rules (represented by `F19` and `F18`):

```
[F19 K\L\(capital L K) italy,
    K\(capital italy K)],
[F18 K\L\(capital L K) (K\K K\(country K)),
    K\L\(and (country K) (capital K L))],
```

The resulting DCG rules are:

```
s(K\L\M\(K M L) A B) -->
    [what], [is], [the], n2(A), [of], pn(B), [?].
s(K\L\M\N\(and (L M) (K N M)) A B) -->
    [what], [is], [the], n2(A), [of], [each], np0(B), [?].
```

Next, we want to incorporate grammar rules for the following three examples which have a somewhat more complex semantics:

```
train(_,[what,are,the,latitudes,of,italy,?],
   Y\(setof Z\(latitude italy Z) Y)).
train(_,[what,are,the,longitudes,of,italy,?],
   Y\(setof Z\(longitude italy Z) Y)).
train(_,[what,are,the,latitudes,of,each,country,?],
   X\Y\(and (country X) (setof Z\(latitude X Z) Y))).
```

The syntax for these sentences could be described by the following rules (the semantics for some of these rules and terminals is also given):

```
s --> [what,are], np_set, [of], pn, [?].
s --> [what,are], np_set, [of,each], np0, [?].
np0(K\K A) --> n1(A).
np_set --> [the,latitudes].
np_set --> [the,longitudes].
n1(X\(country X)) --> [country].
pn(italy) --> [italy].
```

This will lead to the following equations:

```
[F20 F79 italy,
    K\(setof L\(latitude italy L) K)],
[F20 F80 italy,
    K\(setof L\(longitude italy L) K)],
[F21 F79 (F58 K\(country K)),
    K\L\(and (country K) (setof M\(latitude K M) L))],
```

After solving for the function variables in these equations, the following augmented rules are obtained:

```
s(K\L\(K L) A B) --> [what],[are],np_set(A),[of],pn(B),[?].
s(K\L\M\N\(and (L M) (K M N)) A B) --> [what],[are],np_set(A),
                                       [of],[each],np0(B),[?].
np0(K\K A) --> n1(A).
np_set(K\L\(setof M\(latitude K M) L)) --> [the],[latitudes].
np_set(K\L\(setof M\(longitude K M) L)) --> [the],[longitudes].
n1(X\(country X)) --> [country].
pn(italy) --> [italy].
```

As demonstrated by the above example, a large grammar can be synthesized in an incremental fashion by considering more or less independent subsets of the whole language. The semantics for each rule needs to be inferred only once. That is, once the semantics for a rule has been determined, it can be used in augmentation of other rules (see for example rule `np0 --> n1` above). This helps to reduce the search space. However, in some cases backtracking may be necessary if the semantics for a rule inferred earlier is not general enough to correctly handle the semantics of later rules.

The syntactic rules are different from those in the original CHAT-80 grammar for the following reasons: (1) Certain syntactic constructs that can be expressed very concisely in an extraposition grammar require more rules when one is restricted to definite clauses. (2) No attributes for features such as number or gender agreement are used in this grammar; adding such attributes would allow to merge some of the rules (see for example the grammar given in appendix C). (3) In order to avoid introduction of unnecessary function variables, which would degrade the performance of the synthesis process, rules with the same left-hand side and similar right-hand sides have not been combined. For example, the two rules

```
s --> [which,are,the], np0_pl, [?].
s --> [which,are], np0_pl, [?].
```

could be combined to:

```
s --> [which,are], s1.
s1 --> [the], np0_pl, [?].
s1 --> np0_pl, [?].
```

However, one more rule means one more function variable:

```
s((F1 A)) --> [which,are], s1(A).
s1((F2 A)) --> [the], np0_pl(A), [?].
s1((F3 A)) --> np0_pl(A), [?].
```

which increases the complexity of the search for a solution.

The complete set of grammar rules specifying the syntax of the query language is given in appendix B. Appendix B also lists the training instances used for this application as well as the corresponding higher-order and partially executed DCGs. It also gives the complete sequence of substitutions found by the system to resolve the set of higher-order equations. Sample executions of the partially executed DCG are provided at the end of Appendix B.

8.3.2. SEQUEL′

To demonstrate how the system can be used to generate natural language interfaces to databases with standard query languages, consider the task of converting simple instructions expressed in English into the corresponding instructions in the database query language SEQUEL (Date 1989)./footnoteSEQUEL′ is the subset of SEQUEL I am considering in this section. SEQUEL is very similar to SQL. Below is a CFG defining a set of English sentences.

```
s --> s1, [.].
s1 --> [print], np1.
s1 --> [print], rel0.
s1 --> [bind], np3.
np1 --> np2, pp0.
np2 --> [the], attr.
np3 --> rel0, pp3.
attr --> [names].
attr --> [addresses].
rel0 --> [the,join,of], j1.
rel0 --> rel.
rel --> [members].
rel --> [customers].
rel --> [r1].
```

```
pp0 --> pp1, pp2.
pp1 --> [of], rel0.
pp2 --> [with], cond.
pp3 --> [to], rel0.
j1 --> rel0, rel1.
rel1 --> [and], rel0.
cond --> [negative,balance].
cond --> [positive,balance].
```

As an interlingua we choose λ-terms which closely resemble the corresponding expressions of the query language. The training instances to convert English sentences into the interlingual representation would be as follows:

```
train(1,[print,the,names,of,members,with,negative,balance,.],
          (select name (from (rel members) neg_balance))).
train(2,[print,the,addresses,of,members,with,negative,balance,.],
          (select address (from (rel members) neg_balance))).
train(3,[print,the,names,of,customers,with,negative,balance,.],
          (select name (from (rel customers) neg_balance))).
train(4,[print,the,names,of,members,with,positive,balance,.],
          (select name (from (rel members) pos_balance))).
train(5,[print,members,.],(rel members)).
train(6,[print,the,join,of,members,and,customers,.],
          (join (rel members) (rel customers))).
train(7,[bind,members,to,r1,.],(assign (rel r1) (rel members))).
```

The resulting augmented partially executed DCG is:

```
s(A) --> s1(A),['.'].

s1(A) --> [print],np1(A).

s1(A) --> [print],rel0(A).
s1(A) --> [bind],np3(A).

np1(A) --> np2(D),pp0(D\A).

np2(A\(select B A)) --> [the],attr(B).

np3(assign A B) --> rel0(B),pp3(A).

pp0((A\B)\B) --> pp1(E),pp2(E\A).

pp1(A\(from B A)) --> [of],rel0(B).

pp2((A\B)\B) --> [with],cond(A).

attr(name) --> [names].

attr(address) --> [addresses].

rel0(A) --> [the],[join],[of],j1(A).
rel0(rel A) --> rel(A).

rel(members) --> [members].

rel(customers) --> [customers].

rel(r1) --> [r1].

cond(neg_balance) -->
                [negative],[balance].
cond(pos_balance) -->
                [positive],[balance].
```

```
pp3(A) --> [to],rel0(A).
j1(A) --> rel0(D),rel1(D\A).
rel1(A\(join A B)) --> [and],rel0(B).
```

The CFG which defines the corresponding phrases in SEQUEL would be:

```
s --> s1, [.].
s --> s2, [.].
s1 --> [select], np1.
s2 --> [assign,to], np2.
np1 --> n, pp1.
np1 --> [*,from], rel0.
np2 --> rel0, np3.
np3 --> [:], s1.
pp1 --> [from], tup.
tup --> rel0, c.
c --> [where], cond.
j1 --> [','], rel0.
```

```
rel0 --> rel.
rel0 --> rel0, j1.
rel --> [members].
rel --> [customers].
rel --> [r1].
cond --> [balance,<,0].
cond --> [balance,>,0].
n --> [name].
n --> [address].
```

And the corresponding training instances would be:

```
train(1,[select,name,from,members,where,balance,<,0,.],
          (select name (from (rel members) neg_balance))).
train(2,[select,address,from,members,where,balance,<,0,.],
          (select address (from (rel members) neg_balance))).
train(3,[select,name,from,customers,where,balance,<,0,.],
          (select name (from (rel customers) neg_balance))).
train(4,[select,name,from,members,where,balance,>,0,.],
          (select name (from (rel members) pos_balance))).
train(5,[select,*,from,members,.],(rel members)).
train(6,[select,*,from,members,',',customers,.],
          (join (rel members) (rel customers))).
train(7,[assign,to,r1,:,select,*,from,members,.],
          (assign (rel r1) (rel members))).
```

The resulting augmented partially executed DCG is:

```
s(A) --> s1(A),['.'].
s(A) --> s2(A),['.'].
s1(A) --> [select],np1(A).
s2(A) --> [assign],[to],np2(A).
np1(A) --> n(D),pp1(D\A).
np1(A) --> [*],[from],rel0(A).

np2(assign A B) --> rel0(A),np3(B).
```

```
rel0(rel A) --> rel(A).
rel0(A) --> rel0(D),j1(D\A).
rel(members) --> [members].
rel(customers) --> [customers].
rel(r1) --> [r1].
cond(neg_balance) -->
                [balance],[<],[0].
cond(pos_balance) -->
```

```
                                                    [balance],[>],[0].
np3(A) --> [:],s1(A).                       n(name) --> [name].
pp1(A\(select A B)) --> [from],tup(B).      n(address) --> [address].
tup(A) --> rel0(D),c(D\A).
c(A\(from A B)) --> [where],cond(B).
j1(A\(join A B)) --> [','],rel0(B).
```

In order to translate from English to SEQUEL the first DCG is used in parsing mode to obtain the interlingual representation, then the result is given as input to the second DCG operating in generation mode to synthesize the corresponding SEQUEL query. Here are some sample runs:

```
|: print the names of members with negative balance.
   select name from members where balance < 0.

|: print the names of the join of customers and members with positive balance.
   select name from customers , members where balance > 0.

|: print the join of members and customers.
   select * from members , customers.

|: print customers.
   select * from customers.

|: bind the join of customers and members to r1.
   assign to r1 : select * from customers , members.

|: bind customers to r1.
   assign to r1 : select * from customers.
```

8.4. Summary

In this chapter I have discussed various extensions of the system that help to synthesize larger DCGs more efficiently:

- For most practical applications it is possible to automatically assign and infer simple types for all terms, so that the user can omit all or some of the type specification when providing semantic representations.
- The user can specify semantic information along with the syntactic rules. This allows to cut down the search space if semantic information is already known. This also facilitates efficient synthesis of large DCGs since a large grammar can

be broken down into smaller pieces with reasonable-size search space, and the semantics for rules and terminals found for subgrammars can then be incorporated into larger grammars. Without this facility the synthesis of larger DCGs would be impractical.

- The system can use the fact that terminals (words) of the same syntactic category usually have a similar structure, so that the semantics of terminals often can be inferred by generalizing this similarity.

I have also discussed methodologies for developing grammars and semantic representations so that large DCGs can be synthesized efficiently, and I have shown how type raising, which is sometimes necessary to "align" semantic representations returned by different parts of the grammar, is automatically taken care of by the technique proposed in this dissertation. The system has been applied to a variation of the CHAT-80 database query language to demonstrate these techniques and methodologies. The SEQUEL query language was chosen to illustrate how the system can be used to translate between languages using an interlingua.

9. Conclusions and Further Work

9.1. Summary and Contributions

This dissertation is related to the following four general research areas: (1) relationship between syntax and formal semantics, (2) generalization from examples, (3) higher-order logic programming, and (4) machine learning. The contributions of my research are primarily to the first three areas.

The principal contribution of my research has been to show that it is possible, under reasonable assumptions, to mechanically transform a context-free grammar into a definite-clause grammar using a finite set of examples. This problem is not only of technical interest but also has potential applications, and, to the best of my knowledge, the problem has been not been addressed in the literature. The key idea needed to solve this problem was to adopt the simply-typed λ-calculus as the semantic representation language and to assume the principle of compositionality, which requires that the syntactic rules partition a sentence into meaningful phrases, such that the meaning of the sentence can be computed from the meanings of its parts. Writing grammar rules such that they satisfy this requirement is usually the most natural way for humans to define a language. These restrictions are generally satisfied by the class of natural query languages. With these assumptions, I showed that one can cast the problem of generalization from examples as a unification problem over simply-typed λ-terms. However, Huet's procedure (Huet 1975) could not be directly used for this purpose due to the lack of complete knowledge of types. Therefore, I developed a combination of type enumeration and type inference in order to solve this problem.

Higher-order logics (or typed λ-calculi) are useful for synthesizing and manipulating programs, since programs can be viewed as functions and therefore can be represented by variables in the logic. However, inference in higher-order logic is more complex than in first-order logic. To obtain an efficient search for solutions, I showed that it is necessary to implement the unification procedure so that the constraints from several examples are enforced simultaneously. From the perspectives of both search and inference, my system marks an interesting point of departure from the higher-order logic programming language λProlog (Nadathur and Miller 1988). To further improve performance, I showed how to make the execution of the resulting higher-order DCG more efficient by the

technique of partial execution, which effectively turns higher-order rules into first-order rules where possible. With the aid of a simple checking routine, the resulting first-order DCG is capable of efficient forward as well as reverse execution.

I have implemented a system that incorporates the above ideas, and have tested it on a substantial application—synthesis of a variant of the CHAT 80 natural query language—in order to demonstrate their practicality. When inferring the semantic rules for a large grammar, it is in general beneficial to isolate small independent "subgrammars" for which the semantic rules can be found relatively easily. The semantic rules for the complete grammar can then be found by incrementally combining these subgrammars and their semantic rules. The techniques discussed in this dissertation allow one to associate semantics with grammar rules or terminals (words) if their semantics is known or has already been inferred, so that the semantic rules for the remaining grammar rules and terminals can be found faster. This approach allows the user to supervise and direct the generalization process, so that large and complex grammars can be processed quickly. It can also help in identifying syntactic rules that should be rewritten to facilitate the inference process.

9.2. Generalizing the Paradigm to Other Applications

The techniques developed in this dissertation could be used to obtain insights in other areas:

1. The techniques may be used as a basis for the automatic generation of machine translation systems. In principle, both the *transfer approach* and the *interlingua approach* (Hutchins 1986, Nirenburg 1987) could be realized by such a system. The transfer approach translates the natural language input directly into the natural language output. In the interlingua approach, the input is first converted into an intermediate representation, which is then converted to the output language; that is, two grammars would have to be generated in this case. The interlingua approach was illustrated in chapter 8 by a simple natural query language that interfaces with the database query language SEQUEL.

2. In combination with a system that learns the syntax of languages from examples, the results of this research may provide insights into how intelligent agents may efficiently acquire language from examples; that is, the results may contribute to the development of comprehensive grammar induction systems that include both syntactic and semantic generalization.

3. These techniques may provide a systematic way to generate efficient reversible grammars. Grammar bidirectionality provides important theoretical benefits because it insures that a grammar generates the "right" set of sentences (paraphrases) for a particular semantic representation—no more and no less (Dymetman and Isabelle 1990).

As noted earlier, there is a strong connection between DCG synthesis from grammars and examples, on the one hand, and program synthesis from schemas and examples, on the other. Next, I will discuss a variation of this problem. A DCG that computes semantic representations from sentences can be considered as a program whose input is a sentence and whose output is the corresponding semantic representation. In fact, any program can be implemented by a definite-clause logic program, where each predicate has two arguments: the first argument would be the input, and the second argument the output. In the top-level query, the second argument is typically a variable to be instantiated during the execution.

In order to solve a particular problem, a program has to recursively reduce a complex problem to simpler problems. The solutions of the simpler problems are then composed in some way to form the solution of the complex problem. The problem of inferring a definite-clause logic program from examples, therefore, can be divided into two parts: first, one has to decide how the problem can be reduced to simpler problems; second, one has to decide how the solutions of the simpler problems are composed to form the solutions of the complex problems. My proposed scheme assumes that the user provides a "skeleton" of the logic program, specifying the reduction of complex problems to simpler problems. The system then uses this skeleton and a set of sample input/output pairs to infer the rules that combine the solutions of the subproblems—in the case of computing semantic representations from sentences, the CFG would be such a skeleton.

This approach allows us to efficiently infer programs that are much more complex than those which can be efficiently inferred with other approaches to programming by examples (Summers 1977, Bauer 1979, Kodratoff 1979, Biermann et al. 1984). The use of program skeletons allows us to use the the typed λ-calculus and higher-order unification to infer the input/output behaviour of programs from examples. This approach effectively reduces (first-order) inductive inference to (second-order) deductive inference, thus suggesting a general formal framework for integrating inductive and deductive learning processes.

Example 9.1

The following problem is taken from Dietzen and Pfenning (1989). Consider the following set of skeletal clauses for symbolic integration (F1-F4 are variables to be determined):

```
intgr( X\(expn X C), (F1 A) )  :-  const( C, A ).
intgr( X\(C * (F X)), (F2 A B) )  :-  const( C, A ),  intgr( F, B ).
intgr( X\((F X) + (G X)), (F3 A B) )  :-  intgr( F, A ),  intgr( G, B ).
intgr( cos, F4 ).
const( 2, F5 ).
const( 3, F6 ).
```

Consider the following training instances:

```
(1) intgr( X\(expn X 2), X\(expn X (2+1) div (2+1)) ).
(2) intgr( X\(expn X 3), X\(expn X (3+1) div (3+1)) ).
(3) intgr( X\(2 * (cos X)), X\(2 * (sin X)) ).
(4) intgr( X\((expn X 2) + (cos X)), X\((expn X (2+1) div (2+1)) + (sin X)) ).
(5) intgr( cos, sin ).
```

Higher-order equations obtained by applying the training instances to the program clauses are as follows:

```
(1) (F1 F5) = X\(expn X (2+1) div (2+1))
(2) (F1 F6) = X\(expn X (3+1) div (3+1))
(3) (F2 F5 F4) = X\(2 * (sin X))
(4) (F3 (F1 F5) F4) = X\((expn X (2+1) div (2+1)) + (sin X))
(5) F4 = sin
```

The solutions to the above equations are:

```
F1 = V\X\(expn X (V+1) div (V+1))
F2 = V\W\X\(V * (W X))
F3 = V\W\X\((V X) + (W X))
F4 = sin
F5 = 2
F6 = 3
```

Thus the resulting program clauses are:

```
intgr( X\(expn X C), (V\X\(expn X (V+1) div (V+1)) A) )  :-  const( C, A ).
intgr( X\(C * (F X)), (V\W\X\(V * (W X)) A B) )  :-  const( C, A ),
                                                     intgr( F, B ).
intgr( X\((F X) + (G X)), (V\W\X\((V X) + (W X)) A B) )  :-  intgr( F, A ),
                                                             intgr( G, B ).
intgr( cos, sin ).
const( 2, 2 ).
const( 3, 3 ).
```

After evaluation and partial execution:

```
intgr( X\(expn X C), X\(expn X (A+1) div (A+1)) )  :-  const( C, A ).
intgr( X\(C * (F X)), X\(A * D) )  :-  const( C, A ),  intgr( F, X\D ).
intgr( X\((F X) + (G X)), X\(D + E) )  :-  intgr( F, X\D ),
                                           intgr( G, X\E ).
intgr( cos, sin ).
const( 2, 2 ).
const( 3, 3 ).
```

For more complex applications it may not always be possible to determine a complete "correct" decomposition (program skeleton) right away. As in the case of grammar induction, it will probably be necessary to develop and infer the program skeleton and the combination rules in an incremental and interactive fashion, so as to complete the induction process reasonably fast. This is a topic for further investigation.

9.3. Syntactic and Semantic Ambiguities

Syntactic ambiguities (i.e., a sentence with more than one parse) are common in grammars for formal languages, in particular programming languages, as well as natural languages, whereas semantic ambiguities (i.e., a sentence with more than one semantic representation) occur mainly in natural languages.[18] Therefore, if the system is to be used for a larger class of languages than restricted natural query languages, it is necessary to have a scheme for dealing with ambiguous grammars and languages.

Certain types of syntactic and semantic ambiguities could actually be handled by the current system with only minor extensions. In the case where there are several parse trees for a sentence but only one semantic representation, the task is basically to select a parse tree which expresses the correct compositionality with respect to the given semantic representation. Therefore, the system would simply backtrack and select another parse, until the resulting set of higher-order equations can be solved.

Semantic ambiguities can be resolved by this system only if there are different parse trees for each interpretation of an ambiguous sentence. In order to preserve compositionality it may be necessary to have separate sets of grammar rules for each interpretation; that is, grammar rules may have to be duplicated to accommodate the different semantic rules corresponding to each interpretation. If compositionality holds, the system can

[18] Semantic ambiguity implies syntactic ambiguity under the assumption of compositionality.

resolve semantic ambiguities in the following way. Let $\{s_1, \ldots, s_k\}$ be the set of training sentences, where each s_i has $n_i \geq 1$ parse trees and interpretations. For each sentence s_i, the system would then ask the user for n_i semantic representations. Each semantic representation has to be paired up with a parse tree to set up higher-order equations as described in Section 5.1. Since the correct pairing is unknown, the system has to try all combinations until the resulting set of higher-order equations can be solved in the usual way.

Example 9.2:

The following grammar can parse the sentence `[john saw the girl in the park]` in two ways:

```
s --> pn, vp, pp.          s((F1 A B C)) --> pn(A), vp(B), pp(C).
vp --> tv, np.             vp((F2 A B)) --> tv(A), np(B).
np --> det, n, pp.         np((F3 A B C)) --> det(A), n(B), pp(C).
pn --> [john].             pn(F4) --> [john].
tv --> [saw].              tv(F5) --> [saw].
det --> [the].             det(F6) --> [the].
n --> [girl].              n(F7) --> [girl].
pp --> [in,the,park].      pp(F8) --> [in,the,park].
pp --> [].                 pp(F9) --> [].
```

The first parse (`(F1 F4 (F2 F5 (F3 F6 F7 F8)) F9)`) corresponds to the structure:

```
[john [saw [the girl in the park]]]
```

The second parse (`(F1 F4 (F2 F5 (F3 F6 F7 F9)) F8)`) corresponds to:

```
[john [saw [the girl]] in the park]
```

Each parse corresponds to a different semantic interpretation, which could be expressed for example as:

```
(exists X\(and (girl X) (saw john (in X park))))
```

and

```
(exists X\(and (girl X) (in (saw john X) park)))
```

respectively. Therefore, the system would ask the user for two semantic representations for the above sentence. These two semantic representations are then paired up nondeterministically with the two ways of parsing the sentence so that two higher-order equations can be set up (and solved in the usual way):

```
{(F1 F4 (F2 F5 (F3 F6 F7 F8)) F9)
            = (exists X\(and (girl X) (saw john (in X park)))),
 (F1 F4 (F2 F5 (F3 F6 F7 F9)) F8)
            = (exists X\(and (girl X) (in (saw john X) park)))}
```

9.4. Limitations and Extensions

Grammatical agreement of gender, number, and tense:

Grammatical constraints such as gender, number, and tense agreement can be enforced in DCGs using additional arguments. For example, two constituents on the right-hand side of a rule can be forced to have the same number by making the variables representing number in each case be identical. For example, in the rule

```
s --> np(N), vp(N).
```

the number of `np` and `vp` would be required to be the same. The example in Appendix C uses this feature. However, this approach has the disadvantage that disjunctions often can only be handled by duplicating grammar rules. While it is possible to leave a variable completely unconstrained to permit any value, one cannot easily say for example that the gender of a particular constituent can be male *or* neuter but *not* female without using a separate grammar rule for each case.

Representation of time:

It appears representation of temporal information cannot be easily handled in a compositional way. Clifford (1990) gives the example "John worked yesterday", where the "ed" in "worked" is redundant in this case; so it shouldn't have a semantic representation, whereas in the case of "John worked", it should have one. The reason is that, if the meaning of "ed" is represented in the typed λ-calculus by a term m, then "yesterday" would have to produce m as well. But in the final representation of "John worked yesterday" m must appear only once (there should only be one representation for a particular meaning). Since in the case of "John worked" the meaning m of "ed" cannot

be discarded, a conditional would be required that discards the semantic representation of "ed" in one case but not in the other, which is impossible in the typed λ-calculus. Therefore, representations of temporal relations should be processed separately, e.g., by using additional arguments as discussed below.

Pronoun resolution:

For natural language applications such as machine translation, the compositionality assumption causes problems, since the resolution of pronouns depends on the context. For example:

```
That is a house.  It is big.  This is a car.  It is big.
```

The pronoun "It" in "It is big." refers to two different things. Unless the sentence "It is big." is interpreted in a context, one cannot get the right referent. But requiring a context goes against the notion of compositionality, since compositionality implies that the semantics of a sentence is computed from the semantics of its parts rather than from the semantics of its context. However, it can be argued that the meaning of "It" is the same in both cases, namely a variable. A similar problem arises in relative clauses: the reference of the (omitted) subject of the relative clause depends on the main clause, but, as a variable, the compositionality principle still applies. E.g.:

```
That is a house that is big. --->  exists(X, house(X) & big(X))
That is a car that is big.   --->  exists(X, car(X) & big(X))
```

Initially, the semantics of the omitted subject ("the house/the car") would just be a free variable which is bound by the grammar rules to the quantified variable. Similarly:

```
John saw Mary.  She is big.  --->  saw(john,mary) & big(mary)
John saw Nancy.  She is big. --->  saw(john,nancy) & big(nancy)
```

The type of DCGs discussed so far cannot accommodate general pronoun resolution since there is no notion of context. For example, a typical augmented grammar rule would be:

```
a((F X Y)) --> b(X), c(Y).
```

If `b` or `c` had a pronoun, its meaning could not be resolved due to lack of context. One way to restore compositionality is to assume that every phrase of the language is interpreted in a context, and returns a new context. So, the augmentation might be:

```
a((F X Y), Cin, Cout) --> b(X, Cin, C), c(Y, C, Cout).
```

With this scheme the pronoun resolution problem can be separated from what can be done in a strictly compositional manner. Basically, two additional arguments can be used for holding the attributes of the terminals encountered during the parse (the attributes can simply be provided in the lexicon). These attribute lists (given in the context) can then be used to assign variables corresponding to pronouns. For example, if the pronoun is "she" with attributes `[female, singular]` and the context contains

```
[(john,[male,singular]),(mary,[female,singular])],
```

the variable corresponding to "she" would be bound to `mary`.

Equivalence of semantic formulas:

Semantic formulas often satisfy various kinds of equivalence laws, such as the commutativity of disjunction and conjunction. The system described in this dissertation may fail if the user does not provide the semantic representations such that arguments of disjunction, conjunction, or similar operators are in consistent order. This problem could be corrected by building such equivalences into the higher-order unification procedure.

Bibliography

Abramson, H. & Dahl, V. (1989) *Logic Grammars*, Springer-Verlag, New York.

Anderson, J. R. (1977) "Induction of Augmented Transition Networks," *Cognitive Science*, 1, p. 125-157.

Anderson, J. R. (1981) "A Theory of Language Acquisition Based on General Learning Principles," *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, p. 97-103, Vancouver, B. C., Canada, Morgan-Kaufmann Publishers.

Andrews, P.B. (1986) *An Introduction to Mathematical Logic and Type Theory: to Truth Through Proof.* Orlando, Florida: Academic Press.

Barzdin, J.M. (1977) "Inductive Inference of Automata, Functions and Programs," *Amer. Math. Soc. Transactions*, Vol. 109(2), p. 107-112.

Bauer, M. (1979) "Programming by Examples," *Artificial Intelligence*, 12, p. 1-21.

Berwick, R.C. (1985) *The Acquisition of Syntactic Knowledge*, MIT Press, Cambridge, Massachusetts.

Biermann, A.W., Guiho, G, and Kodratoff, Y. (eds.) (1984) *Automatic Program Construction Techniques*, Macmillan Publishing Company, London.

Blum, L., and Blum, M. (1975) "Toward a Mathematical Theory of Inductive Inference," *Inform. Control*, 28, p. 125-155.

Brachman, R.J. (1979) "On the Epistemological Status of Semantic Networks," in N. V. Findler (Ed.), *Associative Networks: Representation and use of knowledge by computers*, p. 3-50, New York: Academic Press.

Budd, T.A. and D. Angluin (1982) " Two Notions of Correctness and their Relation to Testing", *Acta Informatica*, 18, p. 31-45.

Burstall, R.M. and J. Darlington (1977) "A Transformation System for Developing Recursive Programs," *Journal of the ACM*, 24(1), p. 44-67.

Castaing, J., and Kodratoff, Y. (1984) "Theorem Proving By The Study Of Example Proof Traces or Theorem Proving By A Correct Theorem Statement," In: *Automatic Program Construction Techniques*, A. W. Biermann, G. Guiho, and Y. Kodratoff (eds.), p. 421-431, Macmillan Publishing Company, London.

Charniak, E., and McDermott, D. V. (1985) *Introduction to Artificial Intelligence*, Addison-Wesley: Reading, MA.

Church, A. (1940) "A Formulation of the Simple Theory of Types," *J. Symb. Logic*, 5 (1), p. 56-68.

Clark, K. L. and S. Sickel (1977) "Predicate Logic: A Calculus for Deriving Programs," in *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, 419-420, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Clifford, F. (1990) *Formal Semantics and Pragmatics for Natural Language Querying*, Cambridge University Press.

Colmerauer, A. (1978) "Metamorphosis Grammars," In: L. Bolc (ed), Lecture Notes in Computer Science, Springer-Verlag, vol. 63, p. 133-189.

Dahl, V., and Saint-Dizier, P. (eds) (1988) *Natural Language Understanding and Logic Programming II*, North-Holland.

Date, C. J. (1989) *A Guide to the SQL Standard: a User's Guide to the Standard Relational Language SQL*, 2nd ed., Reading, Mass.: Addison-Wesley Pub. Co.

Dietzen, S., and Pfenning, F. (1989) "Higher-Order and Modal Logic as a Framework for Explanation-Based Generalization," (Extended abstract), in Alberto Maria Segre (ed), *Sixth International Workshop on Machine Learning*, pp. 447-449, San Mateo, California, June 1989, Morgan-Kaufmann Publishers.

Dowek, G. (1992) "Third Order Matching is Decidable," in *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, June 22-25, 1992, Santa Cruz, CA.

Dowty, D.R., Wall, R.E. and Peters, S. (1981) *Introduction to Montague Semantics*, Dordrecht, Holland; Boston: D. Reidel Pub. Co.; Hingham, MA.

Dymetman, M., and Isabelle, P. (1990) "Grammar Bidirectionality through Controlled Backward Deduction", in *Logic and Logic Grammars for Language Processing*, Patrick Saint-Dizier and Stan Szpakowicz (eds.), Ellis Horwood, p. 275-293.

Fortune, S., Leivant, D., L., and O'Donnell, M. (1983) "The Expressiveness of Simple and Second-Order Type Structure," *Journal of the Association for Computing Machinery*, Vol. 30, No. 1, January 1983, p. 151-185.

Gordon, M.J.C. (1988) *Programming Language Theory and its Implementation*, Prentice Hall, New York.

Grandy, R.E. (1990) "Understanding and the Principle of Compositionality," in James E. Tomberlin (ed.), *Philosophical Perspectives*, Vol. 4: Action Theory and Philosophy of Mind, p. 557-572, Atascadero, CA: Ridgeview.

Guard, J.R. (1964) "Automated Logic for Semi-Automated Mathematics," Scientific Report No. 1., A F C R L, p. 64-411, AD 602 710.

Hauptmann, A. G. (1991) "Meaning from Structure in Natural Language Processing," Technical Report CMU-CS-91-158, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

Hagiya, M. (1990) "Programming by Example and Proving by Example Using Higher-Order Unification," In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, p. 588–602, Kaiserslautern, Germany, July 1990. Springer-Verlag LNAI 449.

Hagiya, M. (1991) "From Programming-by-Example to Proving-by-Example," In T. Ito and A. R. Meyer, editors, *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, p. 387–419, Sendai, Japan, September 1991. Springer-Verlag LNCS 526.

Huet, G. P. (1975) "A Unification Algorithm for Typed λ-Calculus," *Theoretical Computer Science*, 1, p. 27-57.

Hutchins, W. J. (1986) *Machine Translation: Past, Present, Future*, Ellis Horwood Limited.

Isabelle, P., Dymetman, M. and Macklovitch, E. (1988) "CRITTER: A Translation System for Agricultural Market Reports," *Proc. 12th International Conference on Computational Linguistics*, p. 261-266. Budapest, August.

Ishizaka, H. (1990) "Polynomial Time Learnability of Simple Deterministic Languages," *Machine Learning*, 5, p. 151-164.

Jaffar, J., Lassez J.-L., Maher M. J. (1984) "A Theory of Complete Logic Programs with Equality," In *Journal of Logic Programming*, p. 211-223.

Jensen, D. C., and Pietrzykowski, T. (1976) "Mechanizing ω-Order Type Theory through Unification," *Theoretical Computer Science* 3, p. 123-171.

Kahn, K. M. (1982) "A Partial Evaluator of Lisp Programs Written in Prolog", in Michel van Caneghem, ed., *First International Logic Programming Conference*, p. 19-25, ADDP-GIA, Faculte' des Sciences de Luminy, Marseille, France.

Kodratoff, Y. (1979) "A Class of Functions Synthesized from a Finite Number of Examples and a LISP Program Scheme," *International Journal of Computer and Information Sciences*, 8, p. 489-521.

Landsbergen, J. (1987) *Montague Grammar and Machine Translation*, Eindhoven, Holland, Philips Research M.S. 14.026.

Lassez, J. L., Maher, M. J. and Marriott K. (1988) "Unification Revisited," in *Foundations of Deductive Databases and Logic Programming*, Jack Minker (ed.), Morgan Kaufmann, Los Altos, CA, p. 587-625.

Lebowitz, M. (1990) "The Utility of Similarity-Based Learning in a World Needing Explanantion," in *Machine Learning, An Artificial Intelligence Approach*, Vol. III, Yves Kodratoff and Ryszard Michalski (eds.), Morgan Kaufmann, Palo Alto, CA, p. 399-422.

Lehnert, W. G. (1987) "Learning to Integrate Syntax and Semantics," Proceedings of the Fourth International Workshop on Machine Learning, Irvine, 1987, Morgan Kaufmann, Los Altos, CA, p. 179-190.

Lloyd, J. W. (1987) *Foundations of Logic Programming*, Springer-Verlag, New York.

Matsumoto, Y., Tanaka, H., Hirakawa, H., Miyoshi, H. and Yasukawa, H. (1983) (1983) "BUP: A Bottom-Up Parser Embedded in PROLOG," *New Generation Computing*, 1, p. 145-58.

Michalski, R. S., Carbonell, J. G., and Mitchell, T. M. (1983), *Machine Learning: An Artificial Intelligence Approach*, vol. 1-3, Los Altos, CA: Morgan Kaufmann.

Miller, D. A. and Nadathur G. (1986a) "Higher Order Logic Programming," *Proceedings of the third international Conference on Logic Programming*, London UK, July 1986, p. 448-462.

Miller, D. A. and Nadathur G. (1986b) "Some Uses of Higher-Order Logic in Computational Linguistics," (technical report?) Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104-3897.

Mitchell, T.M. (1978) "Version Spaces: An Approach to Concept Learning," Stanford, CA: Stanford University, (Ph.D. dissertation).

Montague, R. (1974) *Formal Philosophy*, Yale University Press: New Haven.

Nadathur, G. and Miller, D. (1988) "An Overview of λProlog," *Proc. 5th ICLP*, p. 810-827.

Nadathur, G. and Miller, D. (1990) "Higher-Order Horn Clauses," *Journal of the ACM*, p. 777-814.

Nadathur, G. and Jayaraman, B. (1989) "Towards a WAM Model for λProlog," *Proc. North American Conference on Logic Programming*, Cleveland, p. 1180-1198.

Nirenburg, S. (1987) *Machine translation: Theoretical and Methodological Issues*, Cambridge University Press.

Pereira, F.C.N. (1981) "Extraposition Grammars," *American Journal of Computational Linguistics*, vol. 9, no. 4, p. 243-255.

Pereira, F.C.N. (1990) "Semantic Interpretation as Higher-Order Deduction," *Proc. Second European Workshop on Logics and AI*, Amsterdam.

Pereira, F. C. N., Shieber, S. M. (1987) *Prolog and Natural-Language Analysis*, CSLI Lecture Notes, 10, Chicago University Press, Stanford, 1987.

Pereira, F.C.N., and Warren, D.H.D. (1980) "Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Transition Networks," *Artificial Intelligence*, vol. 13, p. 231-278.

Plotkin, G.D. (1970) "A Note on Inductive Generalization," *Machine Intelligence 5*, B. Meltzer and D. Michie (eds.), p. 153-163, American Elsevier, New York.

Plotkin, G.D. (1971) "A Further Note on Inductive Generalization," *Machine Intelligence 6*, B. Meltzer and D. Michie (eds.), p. 101-124, American Elsevier, New York.

Robinson, J.A. (1965) "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the Association for Computing Machinery*, vol. 12, no. 1.

Sandewall, E. (1970) "Representing Natural Language Information in Predicate Calculus," In *Machine Intelligence 5*, B. Meltzer and D. Michie (Eds.), Edinburgh University Press, Edinburgh.

Schank, R. C. (1973) "Identification of Conceptualizations Underlying Natural Language," in R. C. Schank & and K. M. Colby (Eds.), *Computer models of thought and language*, San Francisco: W. H. Freeman.

Schank, R. (1975) *Conceptual Information Processing*, New York, NY: American Elsevier.

Sedlock, D. (1988) "Natural Language Generation in a Large Expert System," 8th International Workshop. Expert Systems and their Applications, Avignon, France, 1988. (Nanterre, France: EC 2), p. 401-12, vol. 1.

Selfridge, M. (1981) "A Computer Model of Child Language Acquisition," In: *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, p. 92-96, Vancouver, B. C., Canada.

Selfridge, M. (1986) "A Computer Model of Child Language Learning," *Artificial Intelligence*, 29, p. 171-216.

Shavlik, J.W. (1990) "Acquiring Recursive and Iterative Concepts with Explanation-Based Learning," *Machine Learning*, 5, p. 39-70.

Shieber, S.M. (1986) "A Uniform Architecture for Parsing and Generation," *Proc. 12th International Conference on Computational Linguistics*, p. 614-619, Budapest, August.

Siskind, J. M. (1990) "Acquiring Core Meanings of Words, Represented as Jackendoff-Style Conceptual Structures, from Correlated Streams of Linguistic and non-Linguistic Input," in *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics*, p. 143-164, University of Pittsburgh, Pittsburgh, PA, June 1990.

Siskind, J. M. (1993) "Lexical Acquisition as Constraint Satisfaction," Colloquium at the University of Pennsylvania, Philadelphia, PA, February 1993.

Smith, D. R. (1984) "The Synthesis of LISP Programs from Examples: A Survey," In: *Automatic Program Construction Techniques*, A. W. Biermann, G. Guiho, and Y. Kodratoff (eds.), p. 307-324, Macmillan Publishing Company, London.

Sterling, L. and Shapiro, E. (1986) *The Art of Prolog*, MIT Press, Cambridge, MA.

Summers, P.D. (1977) "A Methodology for LISP Program Construction from Examples," *Journal of the Association of Computing Machinery*, 24, p. 161-175.

Takeuchi, A. and K. Furukawa (1985) *Partial Evaluation of Prolog Programs and its Application to Meta Programming.* Technical Report TR-126, ICOT, Tokyo, Japan.

Tamaki, H. and Taisuke S. (1984) "Unfold/Fold Transformation of Logic Programs," in *Proceedings of the Second International Logic Programming Conference*, p. 127-138, Uppsala University, Uppsala, Sweden.

van Benthem, J.F.A.K. (1986) *Essays in Logical Semantics*, chapter 10, Reidel Pub. Co., Hingham, MA, USA, distributed by Kluwer Academic Pub.

Velardi, P. (1989) "Natural Language Interfaces to Databases: Features and Limitations," Entity-Relationship Approach: A Bridge to the User. Proceedings of the Seventh International Conference, Rome, Italy, 1988. (Amsterdam, Netherlands: North-Holland), p. 35-40.

Vere, S.A. (1975) "Induction of Concepts in the Predicate Calculus," In: *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, USSR, p. 281-287.

Wallace, M. (1984) "Communicating with Databases in Natural Language," Ellis Horwood Services (A-1), European Computer-Industry Research Center GMBH.

Warren, D.H.D., and Pereira, F.C.N. (1982) "An Efficient Easily Adaptable System for Interpreting Natural Language Queries," *American Journal of Computational Linguistics*, 8 (3-4), p. 110-22.

Warren, D.S. (1983) "Using λ-Calculus to Represent Meanings in Logic Grammars," *ACL Proceedings, 21th Annual Meeting*, p. 51-6.

Warren, D.S. (1985) "Using Montague Semantics in Natural Language Understanding," in *Theoretical Approaches to Natural Language Understanding, a Workshop at Halifax, Nova Scotia*, p. 61-8.

Winston, P. H. (1975) "Learning Structural Descriptions from Examples," In: *The Psychology of Computer Vision*, P. H. Winston (ed.), New York: McGraw-Hill Book Company, p. 157-209.

Winston, P. H. (1984) *Artificial Intelligence*, 2nd edn. Addison-Wesley: Reading, MA.

Woods, W. A. (1970) "Transition Network Grammars for Natural Language Analysis," *Communications of the ACM*, 13: p. 591-606.

Zadrozny, W. (1992) "On Compositional Semantics," *Proc. Coling 1992*, p. 260-266, Nantes, France.

Appendix

A: How to Use the System

A.1. The user interface

The system discussed in this dissertation is available from the author as a Quintus Prolog program, and as executable for UNIX systems. After starting it up it displays the following instructions:

```
Escape to Prolog by typing "prolog".
Terminate Prolog session by typing "halt".


Please enter name of file containing the grammar: test
```

If the system is at the Prolog level and shows the prompt `| ?-` it can be started (or restarted) by entering `start` followed by a period.

The system prompts the user for the name of the file containing the input grammar. The rules of this grammar must be specified using the `-->` operator. The constituents on the right-hand side of the rules are separated by commas, and terminated with a period. Terminals are inclosed in `[ ]`'s. This format is exemplified by the following grammar:

```
s --> [which], n1, pvp, [?].
pvp --> [is], tv_pp, [by], np.
np --> number, n1_pl.
number --> [two].
number --> [three].
n1_pl --> [seas].
n1_pl --> [oceans].
n1 --> [country].
n1 --> [nation].
tv_pp --> [bordered].
tv_pp --> [contained].
```

If the semantics for some of the rules or terminals of a grammar is already known, it can specified by simply adding arguments specifying the semantics. For example, the user may already know the semantics for all terminals and for the third rule of the

above grammar. This information can be specified along with the input grammar in the following way:

```
s --> [which], n1, pvp, [?].
pvp --> [is], tv_pp, [by], np.
np(A\B\C\(A‘C‘B)‘D‘E) --> number(D),n1_pl(E).
number --> [two].
number --> [three].
n1_pl(A\(sea‘A)) --> [seas].
n1_pl(A\(ocean‘A)) --> [oceans].
n1(A\(country‘A)) --> [country].
n1(A\(nation‘A)) --> [nation].
tv_pp(A\B\(borders‘A‘B)) --> [bordered].
tv_pp(A\B\(contains‘A‘B)) --> [contained].
```

Note that the backquote (‘) must be used instead of space to indicate function application in the λ-calculus. E.g., the term `(saw john mary)` should be encoded as `(saw‘john‘mary)`.

After entering the name of the file containing the input grammar, the system will try to read the training instances from a file with the same name but extended with `_t`:

```
Read training instances from file test_t? (y/n) y
```

If the user answers `no`, it will generated training sentences and ask the user for their semantic representations interactively.
The file with the training instances contains clauses of the form

```
train(_, <sentence>, <semantic representation> ).
```

The file with training instances for the above grammar might look like this:

```
train(_,[which,continent,is,bordered,by,two,seas,?],
 X\(and‘(continent‘X)‘(numberof‘Y\(and‘(sea‘Y)‘(borders‘X‘Y))‘2))).
train(_,[which,country,is,bordered,by,two,seas,?],
 X\(and‘(country‘X)‘(numberof‘Y\(and‘(sea‘Y)‘(borders‘X‘Y))‘2))).
train(_,[which,country,is,bordered,by,three,seas,?],
 X\(and‘(country‘X)‘(numberof‘Y\(and‘(sea‘Y)‘(borders‘X‘Y))‘3))).
```

Next, the system will contruct the set of higher-order equations corresponding to these training instances and start the unification procedure (SOLVE). Depending to what degree the user wants to control the search for a consistent set of substitutions, he or she can choose among the following options:

```
Starting unification procedure ...
     Mode 0: General projection rule only.
     Mode 1: General projection and imitation rules only.
     Mode 2: Use additional rule for rule variables.
     Mode i: Interactive mode.
     Mode s: Specify a sequence of substitutions.
     Mode x: No types used.

     Enter mode: 0
```

Mode `0` can be chosen for those cases where the semantic representations given by the user contain no constants. The imitation substitution rule is not needed in those cases, because each imitation substitution introduces a constant. Mode `1` is the general case. Mode `2` is an optimization that uses an additional rule for function variables that corresponds to grammar rules that have no terminals, since the substitution terms for these variables in general are of a restricted form. Mode `i` allows the user to specify substitutions interactively. The options in this mode are discussed below. Mode `s` allows to specify paritially or completely the sequence of substitutions to be selected for the current set of equations. It is also discussed in more detail below. Finally, mode `x` invokes a unification procedure that doesn't use types.

If the uses choses mode `0`, the system will ask whether it should print all substitutions being tried (verbose). This option is useful for example for the analysis of the search procedure and substitution rules being used.

```
     Verbose? (y/n): n
```

Assuming the user replys with no, the system will proceed and try to solve the set of equations. The parameter `Lim` is indicates the maximum order of types, and the maximum depth of nesting allowed for terms. This parameter is incremented successively to ensure exhaustive traversal of the search space.

```
Lim = 0      cpu: 0
```

```
Lim = 1      cpu: 50
Lim = 2      cpu: 650
Lim = 3      cpu: 2000
Writing to file "result"...
(Case memory disabled.)
     Solution found.
     See file "test_h" for the higher order DCG.
     See file "test_p" for the partially executed version.
```

After a solution is found, the system writes the complete derivation (the augmented grammar, all substitutions and the set of equations after each substitution) to the file `result`. The synthesized higher-order DCG will be written to the file `test_h`, if `test` is the file name of the original grammar, and the partially executed version would be written to `test_p`.

At this point the user can force the system to backtrack and find another solution:

```
     Get another solution? (y/n) n
```

If the answer is no, the system is automatically restarted:

```
     Escape to Prolog by typing "prolog".
     Terminate Prolog session by typing "halt".

     Please enter name of file containing the grammar:
```

In case mode `i` is chosen above, the augmented initial grammar and the current set of higher-order equations is displayed. In addition the user has a choice of actions that control which substitutions are selected:

```
     Enter mode: i
Lim = 0      cpu: 0

s(F1) --> [0].
s(F2‘A) --> [s],s(A).

[[F1,K\L\L],
 [F2‘F1,K\L\(K‘L)],
```

```
[F2‘(F2‘F1),K\L\(K‘(K‘L))]]

    Please specify next substitution (e.g., proj(2,2,1)),
    or type "c" to change interactive mode,
    or type "a" to let the system finish,
    or type "quit" to quit session,
    or type "abort" to abort,
    or type "b" to backtrack,
    or type "l" to place restrictions in projections: c
```

The user may either specify the next substitution by entering an instruction of the form `proj(K,L,M)` or `imit(K)`, or change the interactive mode as described below. A projection substitution is specified for example by `proj(3,1,2)`, which would be a projection with three prefix variables, the first of which (counting from right to left) is chosen as head, and the head has two arguments; so the substitution would be

```
A\B\C\(C (H1 A B C) (H2 A B C)).
```

Imitation substitutions are specify by terms like `imit(2)`, which would be an imitation with two prefix variables. A substitution can be partially specified through the use of the "don't care" symbol '`_`'. For example, `proj(3,_,_)` requires that the next substitution is a projection with three prefix variables, but leaves the remaining parameters for the system to decide. It is also possible to leave the substitution completely unspecified by simply entering `_`.

Selecting option `c` above will bring up the following menu for selecting the amount of information to be displayed after each substitution:

```
    Type "1" to show everything,
    type "2" to show substitutions, first and last set of equations,
    type "3" to show first and last set of equations,
    type "4" to show last set of equations only:
```

Choice `1` of this menu means that the initial augmented grammar, all substitutions, and the set of equations after each substitution will be displayed. The other choices would display the derivation less detailed as indicated.

Option `a` of the previous menu effectively causes the system to leave the interactive mode and finish search, if possible, without further guidance by the user. As in the

non-interactive mode, the user has the option to see all attempted substitutions. Option `quit` will terminate the search and write the current status to the file `result`. The whole Prolog process can be interrupted with option `abort`. The user can try different substitutions by forcing the system to backtrack using option `b`. As indicated, option `l` is used to limit to choice of projection substitutions to speed up the search; option `l` will display the following message:

```
Enter proj(lim1,_,lim2) for projection substitutions,
where lim1 and lim2 are two numbers; e.g., proj(5,_,4) :
```

The maximum number of prefix variables is determined by `lim1`, and the maximum number of arguments of the head of the substitution term is determined by `lim2`.

Choice `s` of the main menu prompts the user for a list of substitution specifications that determine the initial or all substitutions chosen by the system:

```
Enter list of specifications.
E.g., [proj(2,1,0),imit(1),_,stop].
   or [proj(2,1,0),imit(1)|_].
(Check file "result" after each run.)
```

If the last element of the list is `stop`, the derivation will terminate at that point and write the current state to file `result`. If the rest of the list is left unspecified (`[...|_]`), the system will try to find consistent substitutions for the remaining equations. Again, substitutions can be partially specified using the "don't care" symbol '`_`'.

A.2. Debugging and diagnostic facilities

If the system fails to find a solution for a particular application it may to due to a variety of reasons, such as typos, or the grammar may not be compositional with respect to the given semantics, or the corresponding higher-order unification problem may be too complex for the system to find a solution in a reasonable time. Most of these problems can be eliminated by restructuring the grammar or reorganizing the set of training instances. In the previous section we mentioned various diagnostic facilities, such as the interactive mode, that can be used to determine the problem with a grammar or its training instances.

One approach to locating a problem would be reducing the number of training instances until the system is able to find a solution, and then adding the other training instances one by one until an incompatible training instance is located. In general, it is advisable to break down a large grammar into small modules to derive the semantics for individual rules. As described in the previous section, the semantics can then be specified along with the input grammar when the complete grammar is being synthesized.

In the current implementation the system records the sequence of substitutions and the set of disagreement pairs after each substitution a file called `result`. This information can be used in subsequent runs to specify the initial sequence of substitutions to avoid repeated search for the same substitutions, by using the `s`-mode. The `s`-mode allows to specify completely or partially the sequence of substitutions the system will choose. It also allows to specify just the number of substitutions so that the unification procedure is terminated early and the sequence of substitutions found by the system can be analyzed.

Additional debugging or diagnostic facilities apart from those discussed above and those of the underlying implementation language (Quintus Prolog) would be desirable. In general, a grammar will exhibit a certain modularity, and it should be possible for the system to isolate a small number of rules and training instances that prevent the system from finding a solution. Such a strategy is closely related to the dependency directed backtracking scheme discussed earlier. Even if there is no clear modularity it would be helpful if the system indicated until which training instance or disagreement pair it was able to find consistent substitutions, as this is usually an indication that there is a problem with the subsequent disagreement pair and the corresponding training instance.

Various kinds of typos or "syntax errors" can be detected even before the higher-order unification procedure is envoked. For example, if the semantic representation of

a training instance contains the term `saw(john,mary)` instead of `(saw john mary)`, a simple routine would detect the error and indicate to the user in which training instance it occurred.

B: Details of the CHAT-80′ Application

Syntax:

Below I list the set of rules specifying the syntax of the query language.

Syntax for the CHAT-80′ language:

```
s --> [does], pn, vp0, [?].
   vp0 --> tv, pn.
   vp0 --> tv, [a], np0.
s --> [is,there,a], np0, vppr, [?].
s --> qnp, [is], pn, [?].
   qnp --> [which], n1, [s], n2.
s --> [which,is,a], np0, [?].
s --> [which,is,the], np0, [?].
s --> [which,are,the], np0_pl, [?].
s --> [which,are], np0_pl, [?].
s --> [which], np0, vp, [?].
s --> [which], np0, pvp, [?].
s --> [what], pvp, [?].
   pvp --> [is], tv_pp, [by], np2n.
s --> [what], vp, [?].
s --> [what,is], vppr, [?].
s --> [what,is,there], vppr, [?].
s --> [what,is,the], n2, [of,each], np0, [?].
s --> [what,is,the], n2, [of], pn, [?].
s --> [what,are], np_set, [of], pn, [?].
s --> [what,are], np_set, [of,each], np0, [?].
s --> [how,many], s1, [?].
s --> s1.
   s1 --> n1_pl, [are,there].
   s1 --> n1_pl, vp1.
      vp1 --> [does], pn, tv.
      vp1 --> tv, pn.
s --> [from,what], n1, s2, [?].
s --> [from,where], s2, [?].
   s2 --> [does], pn, vp_prep.
   s2 --> [does,a], np0, vp_prep.
      vp_prep --> tv3, pn.
s --> [what,is,the], quant, s3, [?].
s --> [what,is,the], s3, [?].
s --> [what,are], np_pl, [?].
   s3 --> n2.
```

```
   s3 --> n2, [of], np_pl.
      np_pl --> n1_pl.
      np_pl --> n1_pl, vppr.
s --> [what,is,the], quant, s4, [?].
s --> [what,is,the], s4, [?].
s --> [what,is,the], set2, [?].
s --> [what,is,the], np3, [what,?].
s --> [what,is,the], np3, [something,?].
s --> [what,are], np_pl2, [?].
   s4 --> set2, [each], np0.
   s4 --> set2.
      set2 --> np3.
         np3 --> n3.
         np3 --> n2, [of], np_pl2.
            np_pl2 --> [the], n1_pl, tv_pr_not.
s --> s5, [each], np0, [?].
s --> s5, [what,?].
s --> [what], np2, [what,?].
s5 --> [how,many], np2.
np2 --> n1_pl, [are,there], tv_pr.

s --> vppr, [?].
   np0 --> n1.
   np0 --> n1, vppr.
      vppr --> vppr1, [and], vppr.
      vppr --> vppr1.
      vppr --> [not], vppr_any.
      vppr --> vppr_any.
         vppr1 --> [that], tv_not, pn.
         vppr1 --> [that], tv_not, [a], np0.
         vppr1 --> [that], tv_not, np0_pl.
         vppr1 --> [whose], n2, vp.
         vppr1 --> tv_pr_not, pn.
         vppr1 --> tv_pr_not, [a], np0.
         vppr1 --> tv_pr_not, [the], np0.
         vppr1 --> tv_pr_not, np0_pl.
         vppr_any --> tv_pr, [any], np0.
            tv_not --> tv_s.
            tv_not --> [does,not], tv.
            tv_pr_not --> tv_pr.
            tv_pr_not --> [not], tv_pr.
   np0_pl --> adj, n1_pl.
   np2n --> number, n1_pl.
   np_set --> [the,latitudes].
```

```
   np_set --> [the,longitudes].

   vp --> tv_s, pn.
   vp --> tv_s, [the], n2, [of], pn.
   vp --> tv_s, [the], np0.
   vp --> tv_s, [a], np0.

tv(X\Y\(borders X Y)) --> [border].
tv(X\Y\(contains X Y)) --> [contain].
tv(X\Y\(flows X Y)) --> [flow,through].
tv_s(X\Y\(borders X Y)) --> [borders].
tv_s(X\Y\(contains X Y)) --> [contains].
tv_s(X\Y\(flows X Y)) --> [flows,through].
tv_s(X\Y\(exceeds X Y)) --> [exceeds].
tv_pr(X\Y\(borders X Y)) --> [bordering].
tv_pr(X\Y\(contains X Y)) --> [containing].
tv_pr(X\Y\(northof X Y)) --> [north,of].
tv_pr(X\Y\(southof X Y)) --> [south,of].
tv_pr(X\Y\(in X Y)) --> [in].
tv_pp(X\Y\(borders X Y)) --> [bordered].
tv_pp(X\Y\(contains X Y)) --> [contained].
tv3(X\Y\Z\(flows X Y Z)) --> [flow,into].
pn(africa) --> [africa].
pn(france) --> [france].
pn(india) --> [india].
pn(italy) --> [italy].
pn(paris) --> [paris].
pn(spain) --> [spain].
pn(rome) --> [rome].
pn(baltic) --> [the,baltic].
pn(danube) --> [the,danube].
pn(equator) --> [the,equator].
pn(persian_gulf) --> [the,persian,gulf].
pn(united_kingdom) --> [the,united,kingdom].
pn(1000000) --> [1,million].
n1(X\(country X)) --> [country].
n1_pl(X\(country X)) --> [countries].
n1(X\(nation X)) --> [nation].
n1_pl(X\(nation X)) --> [nations].
n1(X\(ocean X)) --> [ocean].
n1_pl(X\(ocean X)) --> [oceans].
n1(X\(sea X)) --> [sea].
n1_pl(X\(sea X)) --> [seas].
n1(X\(continent X)) --> [continent].
```

```
n1_pl(X\(continent X)) --> [continents].
n1(X\(river X)) --> [river].
n1_pl(X\(river X)) --> [rivers].
n2(X\Y\(capital Y X)) --> [capital].
n2(X\Y\(president Y X)) --> [president].
n2(X\Y\(area Y X)) --> [area].
n2(X\Y\(population Y X)) --> [population].
n3(Z\X\Y\(area Y X Z)) --> [area,adjacent,to].
adj(X\(african X)) --> [african].
adj(X\(asian X)) --> [asian].
number --> [two].
number --> [three].
quant(X\Y\(aggregate total X Y)) --> [total].
quant(X\Y\(aggregate average X Y)) --> [average].
```

Training instances:

```
train(_,[which,is,a,country,?], X\(country X)).
train(_,[which,are,the,african,countries,?],
   X\(and (african X) (country X))).
train(_,[which,are,african,countries,?],
   X\(and (african X) (country X))).

train(_,[what,is,there,that,borders,italy,?],
   X\(borders X italy)).
train(_,[what,is,there,that,does,not,border,italy,?],
   X\(not (borders X italy))).
train(_,[what,is,there,that,borders,a,sea,?],
   X\(and (sea Y) (borders X Y))).
train(_,[what,is,there,that,borders,african,countries,?],
   X\(and (and (african Y) (country Y)) (borders X Y))).
train(_,[what,is,bordering,italy,?],
   X\(borders X italy)).
train(_,[what,is,not,bordering,italy,?],
   X\(not (borders X italy))).
train(_,[what,is,bordering,a,sea,?],
   X\(and (sea Y) (borders X Y))).
train(_,[what,is,bordering,the,sea,?],
   X\(and (sea Y) (borders X Y))).
train(_,[what,is,bordering,african,countries,?],
   X\(and (and (african Y) (country Y)) (borders X Y))).
train(_,[what,is,containing,italy,and,bordering,spain,?],
   X\(and (contains X italy) (borders X spain))).
train(_,[which,is,the,country,bordering,the,baltic,?],
```

```
   X\(and (country X) (borders X baltic))).

train(_,[does,italy,border,france,?],(borders italy france)).
train(_,[does,italy,border,a,sea,?],(and (sea X) (borders italy X))).

train(_,[what,is,the,capital,of,italy,?],
   Y\(capital italy Y)).
train(_,[what,is,the,capital,of,each,country,?],
   X\Y\(and (country X) (capital X Y))).

train(_,[which,country,s,capital,is,rome,?],
   X\(and (country X) (capital X rome))).

train(_,[what,are,the,latitudes,of,italy,?],
   Y\(setof Z\(latitude italy Z) Y)).
train(_,[what,are,the,longitudes,of,italy,?],
   Y\(setof Z\(longitude italy Z) Y)).
train(_,[what,are,the,latitudes,of,each,country,?],
   X\Y\(and (country X) (setof Z\(latitude X Z) Y))).

train(_,[what,is,bordered,by,two,seas,?],
   X\(numberof Y\(and (sea Y) (borders X Y)) 2)).
train(_,[what,is,bordered,by,three,seas,?],
   X\(numberof Y\(and (sea Y) (borders X Y)) 3)).
train(_,[which,country,is,bordered,by,two,seas,?],
   X\(and (country X) (numberof Y\(and (sea Y) (borders X Y)) 2))).

train(_,[countries,are,there], X\(country X)).
train(_,[nations,are,there], X\(nation X)).
train(_,[how,many,countries,are,there,?],
   X\(numberof Y\(country Y) X)).
train(_,[how,many,countries,does,the,baltic,border,?],
   X\(numberof Y\(and (country Y) (borders baltic Y)) X)).

train(_,[how,many,oceans,border,spain,?],
   X\(numberof Y\(and (ocean Y) (borders Y spain)) X)).

train(_,[from,where,does,the,danube,flow,into,the,persian,gulf,?],
   X\(flows danube X persian_gulf)).
train(_,[from,what,country,does,the,danube,flow,into,the,persian,gulf,?],
   X\(and (country X) (flows danube X persian_gulf))).
train(_,[from,where,does,a,river,flow,into,the,persian,gulf,?],
   X\(and (river Y) (flows Y X persian_gulf))).
```

```
train(_,[what,are,countries,?],
   X\(country X)).
train(_,[what,are,countries,bordering,the,baltic,?],
   X\(and (country X) (borders X baltic))).
train(_,[what,is,the,area,?],
   Z\(setof X\Y\(area Y X) Z)).
train(_,[what,is,the,area,of,countries,?],
   Z\(setof X\Y\(and (country Y) (area Y X)) Z)).
train(_,[what,is,the,total,area,?],
   Z\(and (setof X\Y\(area Y X) S) (aggregate total S Z))).

train(_,[what,are,the,countries,bordering,?],
   Y\X\(and (country Y) (borders Y X))).
train(_,[what,is,the,area,adjacent,to,what,?],
   X\B\Y\(area Y B X)).
train(_,[what,is,the,area,of,the,countries,bordering,something,?],
   X\B\Y\(and (and (country Y) (borders Y X)) (area Y B))).
train(_,[what,is,the,area,adjacent,to,?],
   S\X\(setof B\Y\(area Y B X) S)).
train(_,[what,is,the,area,adjacent,to,each,ocean,?],
   S\X\(and (ocean X) (setof B\Y\(area Y B X) S))).
train(_,[what,is,the,average,area,adjacent,to,?],
   X\A\(and (setof B\Y\(area Y B X) S) (aggregate average S A))).

train(_,[what,countries,are,there,in,what,?],
   X\C\(and (country C) (in C X))).
train(_,[how,many,countries,are,there,in,what,?],
   X\N\(numberof C\(and (country C) (in C X)) N)).
train(_,[how,many,countries,are,there,in,each,continent,?],
   X\N\(and (continent X) (numberof C\(and (country C) (in C X)) N))).

train(_,[what,is,bordering,any,country,?],
   X\(exists C\(and (country C) (borders X C)))).
train(_,[what,is,not,bordering,any,country,?],
   X\(not (exists C\(and (country C) (borders X C))))).
train(_,[is,there,a,sea,bordering,any,country,?],
   (and (sea X) (exists C\(and (country C) (borders X C))))).

train(_,[what,exceeds,1,million,?],
   P\(exceeds P 1000000)).
train(_,[what,exceeds,the,population,of,india,?],
   X\(and (exceeds X P) (population india P))).
train(_,[whose,population,exceeds,1,million,?],
   X\(and (population X P) (exceeds P 1000000))).
```

```
train(_,[what,borders,the,ocean,?],
   X\(and (ocean Y) (borders X Y))).
train(_,[what,borders,a,sea,?],
   X\(and (sea Y) (borders X Y))).
train(_,[which,country,borders,the,ocean,?],
   X\(and (country X) (and (ocean Y) (borders X Y)))).
```

Higher-order DCG:

Below is the higher-order DCG derived from the original grammar and the training instances.

```
s(K\L\(L K) A B) --> [does],pn(A),vp0(B),[?].
vp0(K\L\M\(K M L) A B) --> tv(A),pn(B).
vp0(K\L\M\(and (L N) (K M N)) A B) --> tv(A),[a],np0(B).
s(K\L\(and (K M) (L M)) A B) --> [is],[there],[a],np0(A),vppr(B),[?].
s(K\L\(K L) A B) --> qnp(A),[is],pn(B),[?].
qnp(K\L\M\N\(and (K N) (L M N)) A B) --> [which],n1(A),[s],n2(B).
s(K\K A) --> [which],[is],[a],np0(A),[?].
s(K\K A) --> [which],[is],[the],np0(A),[?].
s(K\K A) --> [which],[are],[the],np0_pl(A),[?].
s(K\K A) --> [which],[are],np0_pl(A),[?].
s(K\L\M\(and (K M) (L M)) A B) --> [which],np0(A),vp(B),[?].
s(K\L\M\(and (K M) (L M)) A B) --> [which],np0(A),pvp(B),[?].
s(K\K A) --> [what],pvp(A),[?].
pvp(K\L\(L K) A B) --> [is],tv_pp(A),[by],np2n(B).
s(K\K A) --> [what],vp(A),[?].
s(K\K A) --> [what],[is],vppr(A),[?].
s(K\K A) --> [what],[is],[there],vppr(A),[?].
s(K\L\M\N\(and (L M) (K N M)) A B) --> [what],[is],[the],n2(A),
                                       [of],[each],np0(B),[?].
s(K\L\M\(K M L) A B) --> [what],[is],[the],n2(A),[of],pn(B),[?].
s(K\L\(K L) A B) --> [what],[are],np_set(A),[of],pn(B),[?].
s(K\L\M\N\(and (L M) (K M N)) A B) --> [what],[are],np_set(A),
                                       [of],[each],np0(B),[?].
s(K\L\(numberof K L) A) --> [how],[many],s1(A),[?].
s(K\K A) --> s1(A).
s1(K\K A) --> n1_pl(A),[are],[there].
s1(K\L\M\(and (K M) (L M)) A B) --> n1_pl(A),vp1(B).
vp1(K\L\M\(L K M) A B) --> [does],pn(A),tv(B).
vp1(K\L\M\(K M L) A B) --> tv(A),pn(B).
s(K\L\M\(and (K M) (L M)) A B) --> [from],[what],n1(A),s2(B),[?].
s(K\K A) --> [from],[where],s2(A),[?].
s2(K\L\(L K) A B) --> [does],pn(A),vp_prep(B).
```

```
s2(K\L\M\(and (K N) (L N M)) A B) --> [does],[a],np0(A),vp_prep(B).
vp_prep(K\L\M\N\(K M N L) A B) --> tv3(A),pn(B).
s(K\L\M\(and (L N) (K N M)) A B) --> [what],[is],[the],quant(A),s3(B),[?].
s(K\K A) --> [what],[is],[the],s3(A),[?].
s(K\K A) --> [what],[are],np_pl(A),[?].
s3(K\L\(setof K L) A) --> n2(A).
s3(K\L\M\(setof N\O\(and (L O) (K N O)) M) A B) --> n2(A),[of],np_pl(B).
np_pl(K\K A) --> n1_pl(A).
np_pl(K\L\M\(and (K M) (L M)) A B) --> n1_pl(A),vppr(B).
s(K\L\M\N\(and (L O M) (K O N)) A B) --> [what],[is],[the],
                                          quant(A),s4(B),[?].
s(K\K A) --> [what],[is],[the],s4(A),[?].
s(K A) --> [what],[is],[the],set2(A),[?].
s(K\K A) --> [what],[is],[the],np3(A),[what],[?].
s(K\K A) --> [what],[is],[the],np3(A),[something],[?].
s(K\K A) --> [what],[are],np_pl2(A),[?].
s4(K\L\M\N\(and (L N) (K M N)) A B) --> set2(A),[each],np0(B).
s4(K\K A) --> set2(A).
set2(K\L\M\(setof (K M) L) A) --> np3(A).
np3(K\K A) --> n3(A).
np3(K\L\M\N\O\(and (L O M) (K N O)) A B) --> n2(A),[of],np_pl2(B).
np_pl2(K\L\M\N\(and (K M) (L N M)) A B) --> [the],n1_pl(A),tv_pr_not(B).
s(K\L\M\N\(and (L M) (K N M)) A B) --> s5(A),[each],np0(B),[?].
s(K\L\M\(K M L) A) --> s5(A),[what],[?].
s(K\K A) --> [what],np2(A),[what],[?].
s5(K\L\M\(numberof (K M) L) A) --> [how],[many],np2(A).
np2(K\L\M\N\(and (K N) (L N M)) A B) --> n1_pl(A),[are],[there],tv_pr(B).
s(K\K A) --> vppr(A),[?].
np0(K\K A) --> n1(A).
np0(K\L\M\(and (K M) (L M)) A B) --> n1(A),vppr(B).
vppr(K\L\M\(and (K M) (L M)) A B) --> vppr1(A),[and],vppr(B).
vppr(K\K A) --> vppr1(A).
vppr(K\L\(not (K L)) A) --> [not],vppr_any(A).
vppr(K\K A) --> vppr_any(A).
vppr1(K\L\(K L) A B) --> [that],tv_not(A),pn(B).
vppr1(K\L\M\(and (L N) (K N M)) A B) --> [that],tv_not(A),[a],np0(B).
vppr1(K\L\M\(and (L N) (K N M)) A B) --> [that],tv_not(A),np0_pl(B).
vppr1(K\L\M\(and (K N M) (L N)) A B) --> [whose],n2(A),vp(B).
vppr1(K\L\(K L) A B) --> tv_pr_not(A),pn(B).
vppr1(K\L\M\(and (L N) (K N M)) A B) --> tv_pr_not(A),[a],np0(B).
vppr1(K\L\M\(and (L N) (K N M)) A B) --> tv_pr_not(A),[the],np0(B).
vppr1(K\L\M\(and (L N) (K N M)) A B) --> tv_pr_not(A),np0_pl(B).
vppr_any(K\L\M\(exists N\(and (L N) (K M N))) A B) --> tv_pr(A),[any],
                                                       np0(B).
```

```
tv_not(K\L\M\(K M L) A) --> tv_s(A).
tv_not(K\L\M\(not (K M L)) A) --> [does],[not],tv(A).
tv_pr_not(K\L\M\(K M L) A) --> tv_pr(A).
tv_pr_not(K\L\M\(not (K M L)) A) --> [not],tv_pr(A).
np0_pl(K\L\M\(and (K M) (L M)) A B) --> adj(A),n1_pl(B).
np2n(K\L\M\(K M L) A B) --> number(A),n1_pl(B).
np_set(K\L\(setof M\(latitude K M) L)) --> [the],[latitudes].
np_set(K\L\(setof M\(longitude K M) L)) --> [the],[longitudes].
vp(K\L\M\(K M L) A B) --> tv_s(A),pn(B).
vp(K\L\M\N\(and (K N O) (L O M)) A B C) --> tv_s(A),[the],n2(B),[of],pn(C).
vp(K\L\M\(and (L N) (K M N)) A B) --> tv_s(A),[the],np0(B).
vp(K\L\M\(and (L N) (K M N)) A B) --> tv_s(A),[a],np0(B).
tv(K\L\(borders K L)) --> [border].
tv(K\L\(contains K L)) --> [contain].
tv(K\L\(flows K L)) --> [flow],[through].
tv_s(K\L\(borders K L)) --> [borders].
tv_s(K\L\(contains K L)) --> [contains].
tv_s(K\L\(flows K L)) --> [flows],[through].
tv_s(K\L\(exceeds K L)) --> [exceeds].
tv_pr(K\L\(borders K L)) --> [bordering].
tv_pr(K\L\(contains K L)) --> [containing].
tv_pr(K\L\(northof K L)) --> [north],[of].
tv_pr(K\L\(southof K L)) --> [south],[of].
tv_pr(K\L\(in K L)) --> [in].
tv_pp(K\L\(borders K L)) --> [bordered].
tv_pp(K\L\(contains K L)) --> [contained].
tv3(K\L\M\(flows K L M)) --> [flow],[into].
pn(africa) --> [africa].
pn(france) --> [france].
pn(india) --> [india].
pn(italy) --> [italy].
pn(paris) --> [paris].
pn(spain) --> [spain].
pn(rome) --> [rome].
pn(baltic) --> [the],[baltic].
pn(danube) --> [the],[danube].
pn(equator) --> [the],[equator].
pn(persian_gulf) --> [the],[persian],[gulf].
pn(united_kingdom) --> [the],[united],[kingdom].
pn(1000000) --> [1],[million].
n1(K\(country K)) --> [country].
n1_pl(K\(country K)) --> [countries].
n1(K\(nation K)) --> [nation].
n1_pl(K\(nation K)) --> [nations].
```

```
n1(K\(ocean K)) --> [ocean].
n1_pl(K\(ocean K)) --> [oceans].
n1(K\(sea K)) --> [sea].
n1_pl(K\(sea K)) --> [seas].
n1(K\(continent K)) --> [continent].
n1_pl(K\(continent K)) --> [continents].
n1(K\(river K)) --> [river].
n1_pl(K\(river K)) --> [rivers].
n2(K\L\(capital L K)) --> [capital].
n2(K\L\(president L K)) --> [president].
n2(K\L\(area L K)) --> [area].
n2(K\L\(population L K)) --> [population].
n3(K\L\M\(area M L K)) --> [area],[adjacent],[to].
adj(K\(african K)) --> [african].
adj(K\(asian K)) --> [asian].
number(K\L\M\(numberof N\(and (L N) (K M N)) 2)) --> [two].
number(K\L\M\(numberof N\(and (L N) (K M N)) 3)) --> [three].
quant(K\L\(aggregate total K L)) --> [total].
quant(K\L\(aggregate average K L)) --> [average].
```

Derivation of the higher-order order DCG for CHAT-80′

Below is the grammar for the CHAT-80′ language augmented with function variables expressing compositionality. `F1`, `F2`, `F3`, etc. will be instantiated by the higher-order unification procedure. Most of the terminal symbols are given their semantic representations already to reduce to amount of search necessary to find a solution.

```
s(F1 A B) --> [does],pn(A),vp0(B),[?].
vp0(F2 A B) --> tv(A),pn(B).
vp0(F3 A B) --> tv(A),[a],np0(B).
s(F4 A B) --> [is],[there],[a],np0(A),vppr(B),[?].
s(F5 A B) --> qnp(A),[is],pn(B),[?].
qnp(F6 A B) --> [which],n1(A),[s],n2(B).
s(F7 A) --> [which],[is],[a],np0(A),[?].
s(F8 A) --> [which],[is],[the],np0(A),[?].
s(F9 A) --> [which],[are],[the],np0_pl(A),[?].
s(F10 A) --> [which],[are],np0_pl(A),[?].
s(F11 A B) --> [which],np0(A),vp(B),[?].
s(F12 A B) --> [which],np0(A),pvp(B),[?].
s(F13 A) --> [what],pvp(A),[?].
pvp(F14 A B) --> [is],tv_pp(A),[by],np2n(B).
s(F15 A) --> [what],vp(A),[?].
s(F16 A) --> [what],[is],vppr(A),[?].
s(F17 A) --> [what],[is],[there],vppr(A),[?].
```

```
s(F18 A B) --> [what],[is],[the],n2(A),[of],[each],np0(B),[?].
s(F19 A B) --> [what],[is],[the],n2(A),[of],pn(B),[?].
s(F20 A B) --> [what],[are],np_set(A),[of],pn(B),[?].
s(F21 A B) --> [what],[are],np_set(A),[of],[each],np0(B),[?].
s(F22 A) --> [how],[many],s1(A),[?].
s(F23 A) --> s1(A).
s1(F24 A) --> n1_pl(A),[are],[there].
s1(F25 A B) --> n1_pl(A),vp1(B).
vp1(F26 A B) --> [does],pn(A),tv(B).
vp1(F27 A B) --> tv(A),pn(B).
s(F28 A B) --> [from],[what],n1(A),s2(B),[?].
s(F29 A) --> [from],[where],s2(A),[?].
s2(F30 A B) --> [does],pn(A),vp_prep(B).
s2(F31 A B) --> [does],[a],np0(A),vp_prep(B).
vp_prep(F32 A B) --> tv3(A),pn(B).
s(F33 A B) --> [what],[is],[the],quant(A),s3(B),[?].
s(F34 A) --> [what],[is],[the],s3(A),[?].
s(F35 A) --> [what],[are],np_pl(A),[?].
s3(F36 A) --> n2(A).
s3(F37 A B) --> n2(A),[of],np_pl(B).
np_pl(F38 A) --> n1_pl(A).
np_pl(F39 A B) --> n1_pl(A),vppr(B).
s(F40 A B) --> [what],[is],[the],quant(A),s4(B),[?].
s(F41 A) --> [what],[is],[the],s4(A),[?].
s(F42 A) --> [what],[is],[the],set2(A),[?].
s(F43 A) --> [what],[is],[the],np3(A),[what],[?].
s(F44 A) --> [what],[is],[the],np3(A),[something],[?].
s(F45 A) --> [what],[are],np_pl2(A),[?].
s4(F46 A B) --> set2(A),[each],np0(B).
s4(F47 A) --> set2(A).
set2(F48 A) --> np3(A).
np3(F49 A) --> n3(A).
np3(F50 A B) --> n2(A),[of],np_pl2(B).
np_pl2(F51 A B) --> [the],n1_pl(A),tv_pr_not(B).
s(F52 A B) --> s5(A),[each],np0(B),[?].
s(F53 A) --> s5(A),[what],[?].
s(F54 A) --> [what],np2(A),[what],[?].
s5(F55 A) --> [how],[many],np2(A).
np2(F56 A B) --> n1_pl(A),[are],[there],tv_pr(B).
s(F57 A) --> vppr(A),[?].
np0(F58 A) --> n1(A).
np0(F59 A B) --> n1(A),vppr(B).
vppr(F60 A B) --> vppr1(A),[and],vppr(B).
vppr(F61 A) --> vppr1(A).
```

```
vppr(F62 A) --> [not],vppr_any(A).
vppr(F63 A) --> vppr_any(A).
vppr1(F64 A B) --> [that],tv_not(A),pn(B).
vppr1(F65 A B) --> [that],tv_not(A),[a],np0(B).
vppr1(F66 A B) --> [that],tv_not(A),np0_pl(B).
vppr1(F67 A B) --> [whose],n2(A),vp(B).
vppr1(F68 A B) --> tv_pr_not(A),pn(B).
vppr1(F69 A B) --> tv_pr_not(A),[a],np0(B).
vppr1(F70 A B) --> tv_pr_not(A),[the],np0(B).
vppr1(F71 A B) --> tv_pr_not(A),np0_pl(B).
vppr_any(F72 A B) --> tv_pr(A),[any],np0(B).
tv_not(F73 A) --> tv_s(A).
tv_not(F74 A) --> [does],[not],tv(A).
tv_pr_not(F75 A) --> tv_pr(A).
tv_pr_not(F76 A) --> [not],tv_pr(A).
np0_pl(F77 A B) --> adj(A),n1_pl(B).
np2n(F78 A B) --> number(A),n1_pl(B).
np_set(F79) --> [the],[latitudes].
np_set(F80) --> [the],[longitudes].
vp(F81 A B) --> tv_s(A),pn(B).
vp(F82 A B C) --> tv_s(A),[the],n2(B),[of],pn(C).
vp(F83 A B) --> tv_s(A),[the],np0(B).
vp(F84 A B) --> tv_s(A),[a],np0(B).
tv(K\L\(borders K L)) --> [border].
tv(K\L\(contains K L)) --> [contain].
tv(K\L\(flows K L)) --> [flow],[through].
tv_s(K\L\(borders K L)) --> [borders].
tv_s(K\L\(contains K L)) --> [contains].
tv_s(K\L\(flows K L)) --> [flows],[through].
tv_s(K\L\(exceeds K L)) --> [exceeds].
tv_pr(K\L\(borders K L)) --> [bordering].
tv_pr(K\L\(contains K L)) --> [containing].
tv_pr(K\L\(northof K L)) --> [north],[of].
tv_pr(K\L\(southof K L)) --> [south],[of].
tv_pr(K\L\(in K L)) --> [in].
tv_pp(K\L\(borders K L)) --> [bordered].
tv_pp(K\L\(contains K L)) --> [contained].
tv3(K\L\M\(flows K L M)) --> [flow],[into].
pn(africa) --> [africa].
pn(france) --> [france].
pn(india) --> [india].
pn(italy) --> [italy].
pn(paris) --> [paris].
pn(spain) --> [spain].
```

```
pn(rome) --> [rome].
pn(baltic) --> [the],[baltic].
pn(danube) --> [the],[danube].
pn(equator) --> [the],[equator].
pn(persian_gulf) --> [the],[persian],[gulf].
pn(united_kingdom) --> [the],[united],[kingdom].
pn(1000000) --> [1],[million].
n1(K\(country K)) --> [country].
n1_pl(K\(country K)) --> [countries].
n1(K\(nation K)) --> [nation].
n1_pl(K\(nation K)) --> [nations].
n1(K\(ocean K)) --> [ocean].
n1_pl(K\(ocean K)) --> [oceans].
n1(K\(sea K)) --> [sea].
n1_pl(K\(sea K)) --> [seas].
n1(K\(continent K)) --> [continent].
n1_pl(K\(continent K)) --> [continents].
n1(K\(river K)) --> [river].
n1_pl(K\(river K)) --> [rivers].
n2(K\L\(capital L K)) --> [capital].
n2(K\L\(president L K)) --> [president].
n2(K\L\(area L K)) --> [area].
n2(K\L\(population L K)) --> [population].
n3(K\L\M\(area M L K)) --> [area],[adjacent],[to].
adj(K\(african K)) --> [african].
adj(K\(asian K)) --> [asian].
number(F132) --> [two].
number(F133) --> [three].
quant(K\L\(aggregate total K L)) --> [total].
quant(K\L\(aggregate average K L)) --> [average].
```

By executing the augmented DCG on the training instances list above, the following higher-order equations (disagreement-pairs) are generated.

```
[[F7 (F58 K\(country K)),
     K\(country K)],
 [F9 (F77 K\(african K) K\(country K)),
     K\(and (african K) (country K))],
 [F10 (F77 K\(african K) K\(country K)),
     K\(and (african K) (country K))],
 [F17 (F61 (F64 (F73 K\L\(borders K L)) italy)),
     K\(borders K italy)],
 [F17 (F61 (F64 (F74 K\L\(borders K L)) italy)),
     K\(not (borders K italy))],
```

```
[F17 (F61 (F65 (F73 K\L\(borders K L)) (F58 K\(sea K)))),
    K\(and (sea H1) (borders K H1))],
[F17 (F61 (F66 (F73 K\L\(borders K L)) (F77 K\(african K)
                                              K\(country K)))),
    K\(and (and (african H1) (country H1)) (borders K H1))],
[F16 (F61 (F68 (F75 K\L\(borders K L)) italy)),
    K\(borders K italy)],
[F16 (F61 (F68 (F76 K\L\(borders K L)) italy)),
    K\(not (borders K italy))],
[F16 (F61 (F69 (F75 K\L\(borders K L)) (F58 K\(sea K)))),
    K\(and (sea H1) (borders K H1))],
[F16 (F61 (F70 (F75 K\L\(borders K L)) (F58 K\(sea K)))),
    K\(and (sea H1) (borders K H1))],
[F16 (F61 (F71 (F75 K\L\(borders K L)) (F77 K\(african K)
                                              K\(country K)))),
    K\(and (and (african H1) (country H1)) (borders K H1))],
[F16 (F60 (F68 (F75 K\L\(contains K L)) italy)
      (F61 (F68 (F75 K\L\(borders K L)) spain))),
    K\(and (contains K italy) (borders K spain))],
[F8 (F59 K\(country K) (F61 (F68 (F75 K\L\(borders K L)) baltic))),
    K\(and (country K) (borders K baltic))],
[F1 italy (F2 K\L\(borders K L) france),
    borders italy france],
[F1 italy (F3 K\L\(borders K L) (F58 K\(sea K))),
    and (sea H2) (borders italy H2)],
[F19 K\L\(capital L K) italy,
    K\(capital italy K)],
[F18 K\L\(capital L K) (F58 K\(country K)),
    K\L\(and (country K) (capital K L))],
[F5 (F6 K\(country K) K\L\(capital L K)) rome,
    K\(and (country K) (capital K rome))],
[F20 F79 italy,
    K\(setof L\(latitude italy L) K)],
[F20 F80 italy,
    K\(setof L\(longitude italy L) K)],
[F21 F79 (F58 K\(country K)),
    K\L\(and (country K) (setof M\(latitude K M) L))],
[F13 (F14 K\L\(borders K L) (F78 F132 K\(sea K))),
    K\(numberof L\(and (sea L) (borders K L)) 2)],
[F13 (F14 K\L\(borders K L) (F78 F133 K\(sea K))),
    K\(numberof L\(and (sea L) (borders K L)) 3)],
[F12 (F58 K\(country K)) (F14 K\L\(borders K L) (F78 F132 K\(sea K))),
    K\(and (country K) (numberof L\(and (sea L) (borders K L)) 2))],
[F23 (F24 K\(country K)),
```

```
    K\(country K)],
[F23 (F24 K\(nation K)),
    K\(nation K)],
[F22 (F24 K\(country K)),
    K\(numberof L\(country L) K)],
[F22 (F25 K\(country K) (F26 baltic K\L\(borders K L))),
    K\(numberof L\(and (country L) (borders baltic L)) K)],
[F22 (F25 K\(ocean K) (F27 K\L\(borders K L) spain)),
    K\(numberof L\(and (ocean L) (borders L spain)) K)],
[F29 (F30 danube (F32 K\L\M\(flows K L M) persian_gulf)),
    K\(flows danube K persian_gulf)],
[F28 K\(country K) (F30 danube (F32 K\L\M\(flows K L M) persian_gulf)),
    K\(and (country K) (flows danube K persian_gulf))],
[F29 (F31 (F58 K\(river K)) (F32 K\L\M\(flows K L M) persian_gulf)),
    K\(and (river H1) (flows H1 K persian_gulf))],
[F35 (F38 K\(country K)),
    K\(country K)],
[F35 (F39 K\(country K) (F61 (F68 (F75 K\L\(borders K L)) baltic))),
    K\(and (country K) (borders K baltic))],
[F34 (F36 K\L\(area L K)),
    K\(setof L\M\(area M L) K)],
[F34 (F37 K\L\(area L K) (F38 K\(country K))),
    K\(setof L\M\(and (country M) (area M L)) K)],
[F33 K\L\(aggregate total K L) (F36 K\L\(area L K)),
    K\(and (setof L\M\(area M L) H3) (aggregate total H3 K))],
[F45 (F51 K\(country K) (F75 K\L\(borders K L))),
    K\L\(and (country K) (borders K L))],
[F43 (F49 K\L\M\(area M L K)),
    K\L\M\(area M L K)],
[F44 (F50 K\L\(area L K) (F51 K\(country K) (F75 K\L\(borders K L)))),
    K\L\M\(and (and (country M) (borders M K)) (area M L))],
[F41 (F47 (F48 (F49 K\L\M\(area M L K)))),
    K\L\(setof M\N\(area N M L) K)],
[F41 (F46 (F48 (F49 K\L\M\(area M L K))) (F58 K\(ocean K))),
    K\L\(and (ocean L) (setof M\N\(area N M L) K))],
[F40 K\L\(aggregate average K L) (F47 (F48 (F49 K\L\M\(area M L K)))),
    K\L\(and (setof M\N\(area N M K) H4) (aggregate average H4 L))],
[F54 (F56 K\(country K) K\L\(in K L)),
    K\L\(and (country L) (in L K))],
[F53 (F55 (F56 K\(country K) K\L\(in K L))),
    K\L\(numberof M\(and (country M) (in M K)) L)],
[F52 (F55 (F56 K\(country K) K\L\(in K L))) (F58 K\(continent K)),
    K\L\(and (continent K) (numberof M\(and (country M) (in M K)) L))],
[F16 (F63 (F72 K\L\(borders K L) (F58 K\(country K)))),
```

```
    K\(exists L\(and (country L) (borders K L)))],
[F16 (F62 (F72 K\L\(borders K L) (F58 K\(country K)))),
    K\(not (exists L\(and (country L) (borders K L))))],
[F4 (F58 K\(sea K)) (F63 (F72 K\L\(borders K L) (F58 K\(country K)))),
    and (sea H2) (exists K\(and (country K) (borders H2 K)))],
[F15 (F81 K\L\(exceeds K L) 1000000),
    K\(exceeds K 1000000)],
[F15 (F82 K\L\(exceeds K L) K\L\(population L K) india),
    K\(and (exceeds K H1) (population india H1))],
[F57 (F61 (F67 K\L\(population L K) (F81 K\L\(exceeds K L) 1000000))),
    K\(and (population K H1) (exceeds H1 1000000))],
[F15 (F83 K\L\(borders K L) (F58 K\(ocean K))),
    K\(and (ocean H1) (borders K H1))],
[F15 (F84 K\L\(borders K L) (F58 K\(sea K))),
    K\(and (sea H1) (borders K H1))],
[F11 (F58 K\(country K)) (F83 K\L\(borders K L) (F58 K\(ocean K))),
    K\(and (country K) (and (ocean H1) (borders K H1)))]]
```

The following sequence of substitutions resolves these higher-order equations.

```
 cproj(1,1,0):  F7 <- K\K
 cproj(1,1,0):  F58 <- K\K
 cproj(1,1,0):  F9 <- K\K
 imit_proj(3):  F77 <- K\L\M\(and (K M) (L M))
 cproj(1,1,0):  F10 <- K\K
 cproj(1,1,0):  F17 <- K\K
 cproj(1,1,0):  F61 <- K\K
 cproj(2,2,1):  F64 <- K\L\(K L)
 cproj(3,3,2):  F73 <- K\L\M\(K M L)
 imit_proj(3):  F74 <- K\L\M\(not (K M L))
 imit_proj(3):  F65 <- K\L\M\(and (L B) (K B M))
 imit_proj(3):  F66 <- K\L\M\(and (L B) (K B M))
 cproj(1,1,0):  F16 <- K\K
 cproj(2,2,1):  F68 <- K\L\(K L)
 cproj(3,3,2):  F75 <- K\L\M\(K M L)
 imit_proj(3):  F76 <- K\L\M\(not (K M L))
 imit_proj(3):  F69 <- K\L\M\(and (L B) (K B M))
 imit_proj(3):  F70 <- K\L\M\(and (L B) (K B M))
 imit_proj(3):  F71 <- K\L\M\(and (L B) (K B M))
 imit_proj(3):  F60 <- K\L\M\(and (K M) (L M))
 cproj(1,1,0):  F8 <- K\K
 imit_proj(3):  F59 <- K\L\M\(and (K M) (L M))
 cproj(2,1,1):  F1 <- K\L\(L K)
 cproj(3,3,2):  F2 <- K\L\M\(K M L)
```

```
imit_proj(3):  F3 <- K\L\M\(and (L A) (K M A))
cproj(3,3,2):  F19 <- K\L\M\(K M L)
imit_proj(4):  F18 <- K\L\M\N\(and (L M) (K N M))
cproj(2,2,1):  F5 <- K\L\(K L)
imit_proj(4):  F6 <- K\L\M\N\(and (K N) (L M N))
cproj(2,2,1):  F20 <- K\L\(K L)
imitation(2):  F79 <- K\L\(setof (H6 K) (H5 L))
cproj(1,1,0):  H5 <- K\K
imit_proj(2):  H6 <- K\L\(latitude K L)
imitation(2):  F80 <- K\L\(setof (H8 K) (H7 L))
cproj(1,1,0):  H7 <- K\K
imit_proj(2):  H8 <- K\L\(longitude K L)
imit_proj(4):  F21 <- K\L\M\N\(and (L M) (K M N))
cproj(1,1,0):  F13 <- K\K
cproj(2,1,1):  F14 <- K\L\(L K)
cproj(3,3,2):  F78 <- K\L\M\(K M L)
imitation(3):  F132 <- K\L\M\(numberof (H9 K L M) H10)
imit_proj(0):  H10 <- 2
imit_proj(4):  H9 <- K\L\M\N\(and (L N) (K M N))
imitation(3):  F133 <- K\L\M\(numberof (H11 K L M) H15)
imit_proj(0):  H15 <- 3
imit_proj(4):  H11 <- K\L\M\N\(and (L N) (K M N))
imit_proj(3):  F12 <- K\L\M\(and (K M) (L M))
cproj(1,1,0):  F23 <- K\K
cproj(1,1,0):  F24 <- K\K
imit_proj(2):  F22 <- K\L\(numberof K L)
imit_proj(3):  F25 <- K\L\M\(and (K M) (L M))
cproj(3,2,2):  F26 <- K\L\M\(L K M)
cproj(3,3,2):  F27 <- K\L\M\(K M L)
cproj(1,1,0):  F29 <- K\K
cproj(2,1,1):  F30 <- K\L\(L K)
cproj(4,4,3):  F32 <- K\L\M\N\(K M N L)
imit_proj(3):  F28 <- K\L\M\(and (K M) (L M))
imit_proj(3):  F31 <- K\L\M\(and (K B) (L B M))
cproj(1,1,0):  F35 <- K\K
cproj(1,1,0):  F38 <- K\K
imit_proj(3):  F39 <- K\L\M\(and (K M) (L M))
cproj(1,1,0):  F34 <- K\K
imit_proj(2):  F36 <- K\L\(setof K L)
imitation(3):  F37 <- K\L\M\(setof (H13 K L) (H12 M))
cproj(1,1,0):  H12 <- K\K
imit_proj(4):  H13 <- K\L\M\N\(and (L N) (K M N))
imit_proj(3):  F33 <- K\L\M\(and (L D) (K D M))
cproj(1,1,0):  F45 <- K\K
```

```
imit_proj(4):  F51 <- K\L\M\N\(and (K M) (L N M))
cproj(1,1,0):  F43 <- K\K
cproj(1,1,0):  F49 <- K\K
cproj(1,1,0):  F44 <- K\K
imit_proj(5):  F50 <- K\L\M\N\O\(and (L O M) (K N O))
cproj(1,1,0):  F41 <- K\K
cproj(1,1,0):  F47 <- K\K
imit_proj(3):  F48 <- K\L\M\(setof (K M) L)
imit_proj(4):  F46 <- K\L\M\N\(and (L N) (K M N))
imit_proj(4):  F40 <- K\L\M\N\(and (L E M) (K E N))
cproj(1,1,0):  F54 <- K\K
imit_proj(4):  F56 <- K\L\M\N\(and (K N) (L N M))
cproj(3,3,2):  F53 <- K\L\M\(K M L)
imit_proj(3):  F55 <- K\L\M\(numberof (K M) L)
imit_proj(4):  F52 <- K\L\M\N\(and (L M) (K N M))
cproj(1,1,0):  F63 <- K\K
imitation(3):  F72 <- K\L\M\(exists (H14 K L M))
imit_proj(4):  H14 <- K\L\M\N\(and (L N) (K M N))
imit_proj(2):  F62 <- K\L\(not (K L))
imit_proj(2):  F4 <- K\L\(and (K A) (L A))
cproj(1,1,0):  F15 <- K\K
cproj(3,3,2):  F81 <- K\L\M\(K M L)
imit_proj(4):  F82 <- K\L\M\N\(and (K N B) (L B M))
cproj(1,1,0):  F57 <- K\K
imit_proj(3):  F67 <- K\L\M\(and (K B M) (L B))
imit_proj(3):  F83 <- K\L\M\(and (L B) (K M B))
imit_proj(3):  F84 <- K\L\M\(and (L B) (K M B))
imit_proj(3):  F11 <- K\L\M\(and (K M) (L M))
```

Some of these substitution terms are too deeply nested to be generated by just one substitution. In those cases free variables are introduced which are instantiated by subsequent substitutions. (In this example variables whose name start with `H` are such free variables.) These free variables then have to be replaced by their substitution terms, and the terms in which they occur are simplified according to the λ-conversion rules. Therefore, the final substitutions of those variables with deeply nested substitution terms are:

```
F79 <- K\L\(setof M\(latitude K M) L)
F80 <- K\L\(setof M\(longitude K M) L)
F132 <- K\L\M\(numberof N\(and (L N) (K M N)) 2)
F133 <- K\L\M\(numberof N\(and (L N) (K M N)) 3)
F37 <- K\L\M\(setof N\O\(and (L O) (K N O)) M)
F72 <- K\L\M\(exists N\(and (L N) (K M N)))
```

Partially executed (first-order) DCG:

```
s(A) --> [does],pn(E),vp0(E\A),[?].
vp0(A\B) --> tv(A\E\B),pn(E).
vp0(A\(and B C)) --> tv(A\F\C),[a],np0(F\B).
s(and A B) --> [is],[there],[a],np0(H\A),vppr(H\B),[?].
s(A) --> qnp(D\A),[is],pn(D),[?].
qnp(A\B\(and C D)) --> [which],n1(B\C),[s],n2(A\B\D).
s(A) --> [which],[is],[a],np0(A),[?].
s(A) --> [which],[is],[the],np0(A),[?].
s(A) --> [which],[are],[the],np0_pl(A),[?].
s(A) --> [which],[are],np0_pl(A),[?].
s(A\(and B C)) --> [which],np0(A\B),vp(A\C),[?].
s(A\(and B C)) --> [which],np0(A\B),pvp(A\C),[?].
s(A) --> [what],pvp(A),[?].
pvp(A) --> [is],tv_pp(E),[by],np2n(E\A).
s(A) --> [what],vp(A),[?].
s(A) --> [what],[is],vppr(A),[?].
s(A) --> [what],[is],[there],vppr(A),[?].
s(A\B\(and C D)) --> [what],[is],[the],n2(B\A\D),[of],[each],np0(A\C),[?].
s(A\B) --> [what],[is],[the],n2(A\H\B),[of],pn(H),[?].
s(A) --> [what],[are],np_set(F\A),[of],pn(F),[?].
s(A\B\(and C D)) --> [what],[are],np_set(A\B\D),[of],[each],np0(A\C),[?].
s(A\(numberof B A)) --> [how],[many],s1(B),[?].
s(A) --> s1(A).
s1(A) --> n1_pl(A),[are],[there].
s1(A\(and B C)) --> n1_pl(A\B),vp1(A\C).
vp1(A\B) --> [does],pn(F),tv(F\A\B).
vp1(A\B) --> tv(A\E\B),pn(E).
s(A\(and B C)) --> [from],[what],n1(A\B),s2(A\C),[?].
s(A) --> [from],[where],s2(A),[?].
s2(A) --> [does],pn(E),vp_prep(E\A).
s2(A\(and B C)) --> [does],[a],np0(H\B),vp_prep(H\A\C).
vp_prep(A\B\C) --> tv3(A\B\F\C),pn(F).
s(A\(and B C)) --> [what],[is],[the],quant(I\A\C),s3(I\B),[?].
s(A) --> [what],[is],[the],s3(A),[?].
s(A) --> [what],[are],np_pl(A),[?].
s3(A\(setof B A)) --> n2(B).
s3(A\(setof B\C\(and D E) A)) --> n2(B\C\E),[of],np_pl(C\D).
np_pl(A) --> n1_pl(A).
np_pl(A\(and B C)) --> n1_pl(A\B),vppr(A\C).
s(A\B\(and C D)) --> [what],[is],[the],quant(J\B\D),s4(J\A\C),[?].
s(A) --> [what],[is],[the],s4(A),[?].
s(A) --> [what],[is],[the],set2(G),[?].
```

```
s(A) --> [what],[is],[the],np3(A),[what],[?].
s(A) --> [what],[is],[the],np3(A),[something],[?].
s(A) --> [what],[are],np_pl2(A),[?].
s4(A\B\(and C D)) --> set2(A\B\D),[each],np0(B\C).
s4(A) --> set2(A).
set2(A\B\(setof C A)) --> np3(B\C).
np3(A) --> n3(A).
np3(A\B\C\(and D E)) --> n2(B\C\E),[of],np_pl2(C\A\D).
np_pl2(A\B\(and C D)) --> [the],n1_pl(A\C),tv_pr_not(B\A\D).
s(A\B\(and C D)) --> s5(B\A\D),[each],np0(A\C),[?].
s(A\B\C) --> s5(B\A\C),[what],[?].
s(A) --> [what],np2(A),[what],[?].
s5(A\B\(numberof C A)) --> [how],[many],np2(B\C).
np2(A\B\(and C D)) --> n1_pl(B\C),[are],[there],tv_pr(B\A\D).
s(A) --> vppr(A),[?].
np0(A) --> n1(A).
np0(A\(and B C)) --> n1(A\B),vppr(A\C).
vppr(A\(and B C)) --> vppr1(A\B),[and],vppr(A\C).
vppr(A) --> vppr1(A).
vppr(A\(not B)) --> [not],vppr_any(A\B).
vppr(A) --> vppr_any(A).
vppr1(A) --> [that],tv_not(E\A),pn(E).
vppr1(A\(and B C)) --> [that],tv_not(G\A\C),[a],np0(G\B).
vppr1(A\(and B C)) --> [that],tv_not(G\A\C),np0_pl(G\B).
vppr1(A\(and B C)) --> [whose],n2(G\A\B),vp(G\C).
vppr1(A) --> tv_pr_not(D\A),pn(D).
vppr1(A\(and B C)) --> tv_pr_not(F\A\C),[a],np0(F\B).
vppr1(A\(and B C)) --> tv_pr_not(F\A\C),[the],np0(F\B).
vppr1(A\(and B C)) --> tv_pr_not(F\A\C),np0_pl(F\B).
vppr_any(A\(exists B\(and C D))) --> tv_pr(A\B\D),[any],np0(B\C).
tv_not(A\B\C) --> tv_s(B\A\C).
tv_not(A\B\(not C)) --> [does],[not],tv(B\A\C).
tv_pr_not(A\B\C) --> tv_pr(B\A\C).
tv_pr_not(A\B\(not C)) --> [not],tv_pr(B\A\C).
np0_pl(A\(and B C)) --> adj(A\B),n1_pl(A\C).
np2n(A\B) --> number(A\E\B),n1_pl(E).
np_set(A\B\(setof C\(latitude A C) B)) --> [the],[latitudes].
np_set(A\B\(setof C\(longitude A C) B)) --> [the],[longitudes].
vp(A\B) --> tv_s(A\E\B),pn(E).
vp(A\(and B C)) --> tv_s(A\F\B),[the],n2(F\I\C),[of],pn(I).
vp(A\(and B C)) --> tv_s(A\F\C),[the],np0(F\B).
vp(A\(and B C)) --> tv_s(A\F\C),[a],np0(F\B).
tv(A\B\(borders A B)) --> [border].
tv(A\B\(contains A B)) --> [contain].
```

```
tv(A\B\(flows A B)) --> [flow],[through].
tv_s(A\B\(borders A B)) --> [borders].
tv_s(A\B\(contains A B)) --> [contains].
tv_s(A\B\(flows A B)) --> [flows],[through].
tv_s(A\B\(exceeds A B)) --> [exceeds].
tv_pr(A\B\(borders A B)) --> [bordering].
tv_pr(A\B\(contains A B)) --> [containing].
tv_pr(A\B\(northof A B)) --> [north],[of].
tv_pr(A\B\(southof A B)) --> [south],[of].
tv_pr(A\B\(in A B)) --> [in].
tv_pp(A\B\(borders A B)) --> [bordered].
tv_pp(A\B\(contains A B)) --> [contained].
tv3(A\B\C\(flows A B C)) --> [flow],[into].
pn(africa) --> [africa].
pn(france) --> [france].
pn(india) --> [india].
pn(italy) --> [italy].
pn(paris) --> [paris].
pn(spain) --> [spain].
pn(rome) --> [rome].
pn(baltic) --> [the],[baltic].
pn(danube) --> [the],[danube].
pn(equator) --> [the],[equator].
pn(persian_gulf) --> [the],[persian],[gulf].
pn(united_kingdom) --> [the],[united],[kingdom].
pn(1000000) --> [1],[million].
n1(A\(country A)) --> [country].
n1_pl(A\(country A)) --> [countries].
n1(A\(nation A)) --> [nation].
n1_pl(A\(nation A)) --> [nations].
n1(A\(ocean A)) --> [ocean].
n1_pl(A\(ocean A)) --> [oceans].
n1(A\(sea A)) --> [sea].
n1_pl(A\(sea A)) --> [seas].
n1(A\(continent A)) --> [continent].
n1_pl(A\(continent A)) --> [continents].
n1(A\(river A)) --> [river].
n1_pl(A\(river A)) --> [rivers].
n2(A\B\(capital B A)) --> [capital].
n2(A\B\(president B A)) --> [president].
n2(A\B\(area B A)) --> [area].
n2(A\B\(population B A)) --> [population].
n3(A\B\C\(area C B A)) --> [area],[adjacent],[to].
adj(A\(african A)) --> [african].
```

```
adj(A\(asian A)) --> [asian].
number((A\B\C)\(B\D)\A\(numberof B\(and D C) 2)) --> [two].
number((A\B\C)\(B\D)\A\(numberof B\(and D C) 3)) --> [three].
quant(A\B\(aggregate total A B)) --> [total].
quant(A\B\(aggregate average A B)) --> [average].
```

Sample executions of the partially executed DCG:

```
|: does italy border spain?

(borders italy spain)


|: which country s capital is paris?

A\(and (country A) (capital A paris))


|: which is the ocean bordering france?

A\(and (ocean A) (borders A france))


|: which is the nation bordering the ocean containing italy?

A\(and (nation A) (and (and (ocean B) (contains B italy)) (borders A B)))


|: which is the ocean bordering italy and bordering spain?

A\(and (ocean A) (and (borders A italy) (borders A spain)))


|: which is the ocean bordering african countries and bordering the ocean
   containing italy?

A\(and (ocean A)
       (and (and (and (african B) (country B))
                 (borders A B))
            (and (and (ocean C) (contains C italy))
                 (borders A C))))


|: what is the capital of each country bordering the baltic?
```

```
A\B\(and (and (country A) (borders A baltic)) (capital A B))


|: what are the latitudes of france?

A\(setof B\(latitude france B) A)


|: what are the longitudes of each ocean bordering asian countries?

A\B\(and (and (ocean A) (and (and (asian C) (country C))
                              (borders A C)))
         (setof D\(longitude A D) B))


|: which country is bordered by two seas?

A\(and (country A) (numberof B\(and (sea B) (borders A B)) 2))


|: how many countries does france border?

A\(numberof B\(and (country B) (borders france B)) A)


|: how many oceans border spain?

A\(numberof B\(and (ocean B) (borders B spain)) A)


|: from what country does the danube flow into the persian gulf?

A\(and (country A) (flows danube A persian_gulf))


|: from what country does a river flow into the persian gulf?

A\(and (country A) (and (river B) (flows B A persian_gulf)))


|: from where does a river flow into the persian gulf?

A\(and (river B) (flows B A persian_gulf))
```

```
|: what is the area of each country bordering spain?

A\B\(and (and (country A) (borders A spain)) (area A B))


|: what is the total area of countries bordering the persian gulf?

A\(and (setof B\C\(and (and (country C)
                           (borders C persian_gulf))
                       (area C B)) D)
       (aggregate total D A))


|: what is the average area of countries bordering the persian gulf and
   bordering african countries?

A\(and (setof B\C\(and (and (country C)
                           (and (borders C persian_gulf)
                                (and (and (african D) (country D))
                                     (borders C D))))
                       (area C B)) E)
       (aggregate average E A))


|: what is the total area of countries not bordering the baltic?

A\(and (setof B\C\(and (and (country C)
                           (not (borders C baltic)))
                       (area C B)) D)
       (aggregate total D A))


|: what is the total area of countries not bordering the ocean containing
   italy?

A\(and (setof B\C\(and (and (country C)
                           (and (and (ocean D)
                                     (contains D italy))
                                (not (borders C D))))
                       (area C B)) E)
       (aggregate total E A))
```

```
|: what is the average area of the countries bordering each ocean?

A\B\(and (and (ocean A)
              (setof C\D\(and (and (country D)
                                   (borders D A))
                              (area D C)) E))
         (aggregate average E B))


|: what is the average area of the countries in each continent south of the
   united kingdom and north of the continent bordering the ocean containing
   india?

A\B\(and (and (and (continent A)
                   (and (southof A united_kingdom)
                        (and (and (continent C)
                                  (and (and (ocean D)
                                            (contains D india))
                                       (borders C D)))
                             (northof A C))))
              (setof E\F\(and (and (country F) (in F A)) (area F E)) G))
         (aggregate average G B))


|: how many countries are there in each continent?

A\B\(and (continent A) (numberof C\(and (country C) (in C A)) B))


|: how many countries are there in each continent south of the equator and not
   bordering the persian gulf?

A\B\(and (and (continent A)
              (and (southof A equator)
                   (not (borders A persian_gulf))))
         (numberof C\(and (country C) (in C A)) B))


|: is there a sea not bordering any country?

(and (sea A) (not (exists B\(and (country B) (borders A B)))))
```

```
|: is there a sea south of the united kingdom not bordering any country
   north of the equator?

(and (and (sea A) (southof A united_kingdom))
     (not (exists B\(and (and (country B) (northof B equator))
                         (borders A B)))))


|: what is bordering a sea and not bordering any river?

A\(and (and (sea B) (borders A B))
       (not (exists C\(and (river C) (borders A C)))))


|: what is bordering the ocean that does not border africa?

A\(and (and (ocean B) (not (borders B africa))) (borders A B))


|: which is the ocean that borders african countries and that borders asian
   countries?

A\(and (ocean A) (and (and (and (african B) (country B))
                                (borders A B))
                      (and (and (asian C) (country C))
                           (borders A C))))


|: which country borders the country whose population exceeds 1 million?

A\(and (country A)
       (and (and (country B)
                 (and (population B C) (exceeds C 1000000)))
            (borders A B)))


|: which country bordering the persian gulf borders a country that borders
   a country whose population exceeds the population of india?

A\(and (and (country A) (borders A persian_gulf))
       (and (and (country B)
                 (and (and (country C)
                           (and (population C D)
                                (and (exceeds D E) (population india E))))
```

```
          (borders B C)))
(borders A B)))
```

C: Application to a Larger Subset of English

Below is a CFG to be augmented with semantic interpretation rules. Note that the CFG has already been augmented with an extra argument to enforce context-sensitive features (e.g., tense, number, gender). However, this extra argument has no influence on the semantic representations. A terminal of the form `wh(List)` indicates that `List` is an interrogative word compound, and a terminal of the form `pron(W)` indicates that `W` is a pronoun. Other terminals are of the form `[...]`.

```
s(_) --> s1_p(_).
s(_) --> s1_p(_), s(_).
s1_p(_) --> s1(_), ['.'].
s1(_) --> np([_,N,G,nom|_]),vp([_,N,G|_]).
s1(A) --> pvp(A), pnp(A).
np(A) --> det(A), adjs_n_rel(A).
np(A) --> pn(A).
np([_,_,neut|_]) --> wh([what]).
np(_) --> wh([who]).
np([_,sing,masc,nom|_]) --> pron(he).
np([_,sing,masc,acc|_]) --> pron(him).
np([_,sing,masc,dat|_]) --> pron(him).
np([_,sing,fem,nom|_]) --> pron(she).
np([_,sing,fem,acc|_]) --> pron(her).
np([_,sing,fem,dat|_]) --> pron(her).
np([_,sing,neut|_]) --> pron(it).
vp([past|A]) --> tv([past|A]), np([_,_,_,acc|_]).
vp(A) --> iv(A).
vp([pres|_]) --> [is,a], adjs_n_rel(_).
vp([past|_]) --> wh([did,what]).
vp([pres,sing|_]) --> wh([does,what]).
adjs_n_rel(A) --> adjs_n(A).
adjs_n_rel(A) --> adjs_n(A), rel(_).
adjs_n(A) --> adjs(A), n(A).
adjs_n(A) --> n(A).
adjs(A) --> adj(A).
adjs(A) --> adj(A), adjs(A).
rel(A) --> rel_pro(A), vp(A).
pnp(_) --> [by], np([_,_,_,dat|_]).
pvp(_) --> np([_,N,_,nom|_]), ptv([_,N|_]).
ptv(_) --> [was], tv([pp|_]).
det([_,sing|_]) --> [a].
det([_,sing|_]) --> [every].
n([_,sing,fem|_]) --> [girl].
```

```
n([_,sing|_]) --> [student].
pn([_,sing,fem|_]) --> [mary].
pn([_,sing,masc|_]) --> [john].
iv([pres,sing|_]) --> [sings].
iv([pres,sing|_]) --> [sleeps].
tv([past|_]) --> [saw].
tv([past|_]) --> [visited].
tv([pp|_]) --> [seen].
tv([pp|_]) --> [visited].
adj(_) --> [good].
adj(_) --> [smart].
rel_pro(_) --> [that].
```

The training instances for this example are:

```
train(1,[mary,sings,'.'],(sings mary)).
train(2,[john,sings,'.'],(sings john)).
train(3,[mary,sleeps,'.'],(sleeps mary)).
train(4,[mary,is,a,girl,'.'],(girl mary)).
train(5,[mary,is,a,student,'.'],(student mary)).
train(6,[mary,is,a,good,girl,'.'],(and (girl mary) (good mary))).
train(7,[mary,is,a,smart,girl,'.'],(and (girl mary) (smart mary))).
train(8,[mary,is,a,good,smart,girl,'.'],(and (girl mary)
                                        (and (good mary) (smart mary)))).
train(9,[john,saw,mary,'.'],(saw john mary)).
train(10,[john,visited,mary,'.'],(visited john mary)).
train(11,[a,girl,sings,'.'],(exists X\(and (girl X) (sings X)))).
train(12,[every,girl,sings,'.'],(all X\(implies (girl X) (sings X)))).
train(13,[john,saw,a,girl,that,sings,'.'],
         (exists X\(and (and (girl X) (sings X)) (saw john X)))).
train(14,[mary,was,seen,by,john,'.'],(saw john mary)).
train(15,[mary,was,visited,by,john,'.'],(visited john mary)).
train(16,[mary,saw,john,'.',john,saw,mary,'.'],
         (and (saw mary john) (saw john mary))).
```

Here is the augmented higher-order DCG in which each grammar symbol obtained an additional argument representing the semantics of the corresponding grammar constituent:

```
s(A,B\B C) --> s1_p(F,C).
s(A,B\C\(and B C) D E) --> s1_p(H,D),s(J,E).
s1_p(A,B\B C) --> s1(F,C),[.].
s1(A,B\C\(B C) D E) --> np([H,I,J,nom|K],D),vp([M,I,J|N],E).
s1(A,B\C\(C B) D E) --> pvp(A,D),pnp(A,E).
np(A,B\C\D\(B D C) E F) --> det(A,E),adjs_n_rel(A,F).
```

```
np(A,B\C\(C B) D) --> pn(A,D).
np([A,B,neut|C],D) --> wh([what]).
np(A,B) --> wh([who]).
np([A,sing,masc,nom|B],C) --> pron(he).
np([A,sing,masc,acc|B],C) --> pron(him).
np([A,sing,masc,dat|B],C) --> pron(him).
np([A,sing,fem,nom|B],C) --> pron(she).
np([A,sing,fem,acc|B],C) --> pron(her).
np([A,sing,fem,dat|B],C) --> pron(her).
np([A,sing,neut|B],C) --> pron(it).
vp([past|A],B\C\D\(C (B D)) E F) --> tv([past|A],E),np([J,K,L,acc|M],F).
vp(A,B\C\(B C) D) --> iv(A,D).
vp([pres|A],B\C\(B C) D) --> [is],[a],adjs_n_rel(I,D).
vp([past|A],B) --> wh([did,what]).
vp([pres,sing|A],B) --> wh([does,what]).
adjs_n_rel(A,B\C\(B C) D) --> adjs_n(A,D).
adjs_n_rel(A,B\C\D\(C D B) E F) --> adjs_n(A,E),rel(J,F).
adjs_n(A,B\C\D\(and (C D) (B D)) E F) --> adjs(A,E),n(A,F).
adjs_n(A,B\C\(B C) D) --> n(A,D).
adjs(A,B\C\(B C) D) --> adj(A,D).
adjs(A,B\C\D\(and (B D) (C D)) E F) --> adj(A,E),adjs(A,F).
rel(A,B\C\D\E\(B E C D) F G) --> rel_pro(A,F),vp(A,G).
pnp(A,B\C\(C B) D) --> [by],np([H,I,J,dat|K],D).
pvp(A,B\C\D\(D (C B)) E F) --> np([I,J,K,nom|L],E),ptv([N,J|O],F).
ptv(A,B\C\D\(C (B D)) E) --> [was],tv([pp|I],E).
det([A,sing|B],C\D\(exists E\(and (D E) (C E)))) --> [a].
det([A,sing|B],C\D\(all E\(implies (D E) (C E)))) --> [every].
n([A,sing,fem|B],C\(girl C)) --> [girl].
n([A,sing|B],C\(student C)) --> [student].
pn([A,sing,fem|B],mary) --> [mary].
pn([A,sing,masc|B],john) --> [john].
iv([pres,sing|A],B\(sings B)) --> [sings].
iv([pres,sing|A],B\(sleeps B)) --> [sleeps].
tv([past|A],B\C\(saw B C)) --> [saw].
tv([past|A],B\C\(visited B C)) --> [visited].
tv([pp|A],B\C\(saw B C)) --> [seen].
tv([pp|A],B\C\(visited B C)) --> [visited].
adj(A,B\(good B)) --> [good].
adj(A,B\(smart B)) --> [smart].
rel_pro(A,B\C\D\(and (B D) (C D))) --> [that].
```

After partial execution and generalization of the representations of terminals, where possible, the following first-order DCG is obtained.

```
s(A,B) --> s1_p(E,B).
```

```
s(A,and B C) --> s1_p(F,B),s(H,C).
s1_p(A,B) --> s1(E,B),[.].
s1(A,B) --> np([E,F,G,nom|H],I\B),vp([K,F,G|L],I).
s1(A,B) --> pvp(A,E),pnp(A,E\B).
np(A,B\C) --> det(A,B\F\C),adjs_n_rel(A,F).
np(A,(B\C)\C) --> pn(A,B).
np([A,B,neut|C],D) --> wh([what]).
np(A,B) --> wh([who]).
np([A,sing,masc,nom|B],C) --> pron(he).
np([A,sing,masc,acc|B],C) --> pron(him).
np([A,sing,masc,dat|B],C) --> pron(him).
np([A,sing,fem,nom|B],C) --> pron(she).
np([A,sing,fem,acc|B],C) --> pron(her).
np([A,sing,fem,dat|B],C) --> pron(her).
np([A,sing,neut|B],C) --> pron(it).
vp([past|A],B\C) --> tv([past|A],B\F),np([H,I,J,acc|K],F\C).
vp(A,B\C) --> iv(A,B\C).
vp([pres|A],B\C) --> [is],[a],adjs_n_rel(H,B\C).
vp([past|A],B) --> wh([did,what]).
vp([pres,sing|A],B) --> wh([does,what]).
adjs_n_rel(A,B\C) --> adjs_n(A,B\C).
adjs_n_rel(A,B\C) --> adjs_n(A,F),rel(H,B\F\C).
adjs_n(A,B\(and C D)) --> adjs(A,B\D),n(A,B\C).
adjs_n(A,B\C) --> n(A,B\C).
adjs(A,B\C) --> adj(A,B\C).
adjs(A,B\(and C D)) --> adj(A,B\C),adjs(A,B\D).
rel(A,B\C\D) --> rel_pro(A,C\G\B\D),vp(A,G).
pnp(A,(B\C)\C) --> [by],np([G,H,I,dat|J],B).
pvp(A,(B\C)\C) --> np([F,G,H,nom|I],J),ptv([L,G|M],J\B).
ptv(A,(B\C)\D\C) --> [was],tv([pp|H],D\B).
det([A,sing|B],(C\D)\(C\E)\(exists C\(and E D))) --> [a].
det([A,sing|B],(C\D)\(C\E)\(all C\(implies E D))) --> [every].
n(A,B\(C B)) --> [C],{lex(C,n,A)}.
pn(A,B) --> [B],{lex(B,pn,A)}.
iv(A,B\(C B)) --> [C],{lex(C,iv,A)}.
tv(A,B\C\(D B C)) --> [D],{lex(D,tv,A)}.
tv([pp|A],B\C\(saw B C)) --> [seen].
tv([pp|A],B\C\(visited B C)) --> [visited].
adj(A,B\(C B)) --> [C],{lex(C,adj,A)}.
rel_pro(A,(B\C)\(B\D)\B\(and C D)) --> [that].
```

For some of the lexical categories in this example it is possible to systematically construct the semantic representations from the words themselves. E.g., if `TV` is a transitive verb in past tense, its representation is always of the form `A\B\(TV A B)`. Therefore this

construction can be generalized to all transitive verbs in past tense. The lexicon used for this example is listed below. Each entry is of the form `lex( Word, Category, Attributes )`. Note, however, that in general there doesn't need to be any resemblance between a word and its semantic representation in order for our system to infer the representations of those words from the examples. The additional generalizations are performed only if there exist such regularities.

```
lex( a, det, [_,sing|_] ).
lex( every, det, [_,sing|_] ).

lex( that, rel_pro, _ ).
lex( which, rel_pro, [_,_,neut|_] ).

lex( being, n, [_,sing,neut|_] ).
lex( book, n, [_,sing,neut|_] ).
lex( boy, n, [_,sing,masc|_] ).
lex( child, n, [_,sing,neut|_] ).
lex( girl, n, [_,sing,fem|_] ).
lex( person, n, [_,sing|_] ).
lex( picture, n, [_,sing,neut|_] ).
lex( professor, n, [_,sing|_] ).
lex( program, n, [_,sing,neut|_] ).
lex( pupil, n, [_,sing|_] ).
lex( student, n, [_,sing|_] ).

lex( bertrand, pn, [_,sing,masc|_] ).
lex( eliza, pn, [_,sing,fem|_] ).
lex( john, pn, [_,sing,masc|_] ).
lex( mary, pn, [_,sing,fem|_] ).
lex( mike, pn, [_,sing,masc|_] ).
lex( nancy, pn, [_,sing,fem|_] ).
lex( principia, pn, [_,sing|_] ).
lex( shrdlu, pn, [_,sing,neut|_] ).
lex( terry, pn, [_,sing|_] ).

lex( drew, tv, [past|_] ).
lex( ran, tv, [past|_] ).
lex( read, tv, _ ).
lex( saw, tv, [past|_] ).
lex( studied, tv, [past|_] ).
lex( taught, tv, [past|_] ).
lex( visited, tv, [past|_] ).
lex( wrote, tv, [past|_] ).
```

```
lex( halts, iv, [pres,sing|_] ).
lex( laughs, iv, [pres,sing|_] ).
lex( learns, iv, [pres,sing|_] ).
lex( plays, iv, [pres,sing|_] ).
lex( runs, iv, [pres,sing|_] ).
lex( sings, iv, [pres,sing|_] ).
lex( talks, iv, [pres,sing|_] ).
lex( works, iv, [pres,sing|_] ).

lex( red, adj, _ ).
lex( blue, adj, _ ).
lex( good, adj, _ ).
lex( bad, adj, _ ).
lex( smart, adj, _ ).
lex( little, adj, _ ).
lex( big, adj, _ ).
```

Below are some sample executions of the grammar constructed above, demonstrating recursive rule applications for relative clauses, adjectives, and multiple sentences, passive voice, reversibility, paraphrasing, and quantified noun phrases, as well as pronoun resolution and pronoun generation.

```
|: john saw mary.                                    % Natural language input.

   (saw john mary).                                  % Semantic representation
                                                     % computed by the grammar.
|: john was visited by a student that read principia.

   (exists A\(and  (and  (student A)  (read A principia))  (visited A john))).

|: terry wrote a big program.

   (exists A\(and  (and  (program A)  (big A))  (wrote terry A))).

|: every smart student wrote a big program.

   (all A\(implies  (and  (student A)  (smart A))
                    (exists B\(and  (and  (program B)  (big B))
                                    (wrote A B))))).

|: every smart little girl read a book.

   (all A\(implies  (and  (girl A)  (and  (smart A)  (little A)))
```

```
                    (exists B\(and  (book B)  (read A B))))).

|: mike is a child that read a book.

   (and  (child mike)  (exists A\(and  (book A)  (read mike A)))).

|: john taught every student that wrote a program that drew a picture.

   (all A\(implies (and (student A)
                        (exists B\(and (and (program B)
                                            (exists C\(and (picture C)
                                                           (drew B C))))
                                       (wrote A B))))
                   (taught john A))).

|: john saw mary. she visited him. he wrote a big program.

and (saw john mary)
    (and (visited mary john)
         (exists A\(and (and (program A) (big A)) (wrote john A)))).

|: a student saw a girl that visited him.

exists A\(and (student A)
              (exists B\(and (and (girl B) (visited B A))
                             (saw A B)))).


|: (saw john mary).                              % For generation, the semantic
                                                 % representation is the input.
   john saw mary.                                % Two sentences are generated
   mary was seen by john.                        % for this semantic
                                                 % representation.

|: (exists A\(and  (and  (student A)  (read A principia))  (visited A john))).

   a student that read principia visited john.
   john was visited by a student that read principia.

|: (exists A\(and  (and  (program A)  (big A))  (wrote terry A))).
                                                 % No passive version is con-
   terry wrote a big program.                    % structed in this case, since
                                                 % "written" was not covered
                                                 % by the training instances.
```

```
|: (all A\(implies  (and  (student A)  (smart A))
                    (exists B\(and  (and  (program B)  (big B))
                                    (wrote A B))))).
```

```
   every smart student wrote a big program.
```

```
|: (all A\(implies  (and  (girl A)  (and  (smart A)  (little A)))
                    (exists B\(and  (book B)  (read A B))))).
```

```
   every smart little girl read a book.
   a book was read by every smart little girl.
```

```
|: (and  (child mike)  (exists A\(and  (book A)  (read mike A)))).
```

```
   mike is a child that read a book.
```

```
|: (all A\(implies (and (student A)
                        (exists B\(and (and (program B)
                                            (exists C\(and (picture C)
                                                           (drew B C))))
                                       (wrote A B))))
                   (taught john A))).
```

```
   john taught every student that wrote a program that drew a picture.
```

```
|: and (saw john mary)
       (and (visited mary john)
            (exists A\(and (and (program A) (big A)) (wrote john A)))).
```

```
john saw mary. she visited him. he wrote a big program.
john saw mary. he was visited by her. he wrote a big program.
mary was seen by john. she visited him. he wrote a big program.
mary was seen by john. he was visited by her. he wrote a big program.
```

```
|: exists A\(and (student A)
                 (exists B\(and (and (girl B) (visited B A))
                                (saw A B)))).
```

```
a student saw a girl that visited him.
a student saw a girl that visited her.
a girl that visited a student was seen by him.
a girl that visited a student was seen by her.
```

D: Illustration of Combination Rules

Without combination rules:

```
The CFG is:

   s(A B C) --> np(B),iv(C).
   np(G H I) --> det(H),n(I).
   det(M) --> [a].
   det(P) --> [every].
   n(S) --> [program].
   n(V) --> [computer].
   iv(Y) --> [runs].
   iv(B1) --> [halts].

Training instances:
   train(1,[a,program,runs],(exists X\(and (program X) (run X)))).
   train(2,[every,program,runs],(all X\(implies (program X) (run X)))).
   train(3,[a,computer,runs],(exists X\(and (computer X) (run X)))).
   train(4,[a,program,halts],(exists X\(and (program X) (halt X)))).

Higher-order DCG:
   s(A\B\(A B) C D) --> np(C),iv(D).
   np(A\B\C\(A C B) D E) --> det(D),n(E).
   det(A\B\(exists C\(and (B C) (A C)))) --> [a].
   det(A\B\(all C\(implies (B C) (A C)))) --> [every].
   n(program) --> [program].
   n(computer) --> [computer].
   iv(run) --> [runs].
   iv(halt) --> [halts].

Partially executed DCG:
   s(A) --> np(D\A),iv(D).
   np(A\B) --> det(A\E\B),n(E).
   det(A\B\(exists C\(and (B C) (A C)))) --> [a].
   det(A\B\(all C\(implies (B C) (A C)))) --> [every].
   n(program) --> [program].
   n(computer) --> [computer].
   iv(run) --> [runs].
   iv(halt) --> [halts].


Here is the sequence of substitutions.  "proj(...)" are Huet's projection
substitutions, "imitation(.)" are Huet's imitation substitutions.
```

```
Further below are the disagreement pair sets after each substitution listed.
Note, that some of the intermediate terms are rather large.

   proj(2,2,1):  A <- T27\U27\(T27 (F1 T27 U27))
   proj(3,3,2):  G <- Q27\R27\S27\(Q27 (G1 Q27 R27 S27) (I1 Q27 R27 S27))
   imitation(2):  M <- O27\P27\(exists (Q1 O27 P27))
   imitation(3):  Q1 <- M27\N27\A\(and (L3 M27 N27 A) (H3 M27 N27 A))
   proj(3,3,1):  H3 <- J27\K27\L27\(J27 (B10 J27 K27 L27))
   proj(3,1,0):  G1 <- G27\H27\I27\I27
   proj(2,1,0):  F1 <- E27\F27\F27
   imitation(1):  Y <- D27\(run (T23 D27))
   proj(1,1,0):  T23 <- C27\C27
   proj(3,1,0):  B10 <- Z26\A27\B27\B27
   proj(3,2,1):  L3 <- W26\X26\Y26\(X26 (T24 W26 X26 Y26))
   proj(3,2,0):  I1 <- T26\U26\V26\U26
   imitation(1):  S <- S26\(program (J25 S26))
   proj(1,1,0):  J25 <- R26\R26
   proj(3,1,0):  T24 <- O26\P26\Q26\Q26
   imitation(2):  P <- M26\N26\(all (N25 M26 N26))
   imitation(3):  N25 <- K26\L26\A\(implies (P25 K26 L26 A) (O25 K26 L26 A))
   proj(3,3,1):  O25 <- H26\I26\J26\(H26 (Q25 H26 I26 J26))
   proj(3,1,0):  Q25 <- E26\F26\G26\G26
   proj(3,2,1):  P25 <- B26\C26\D26\(C26 (R25 B26 C26 D26))
   proj(3,1,0):  R25 <- Y25\Z25\A26\A26
   imitation(1):  V <- X25\(computer (S25 X25))
   proj(1,1,0):  S25 <- W25\W25
   imitation(1):  B1 <- V25\(halt (T25 V25))
   proj(1,1,0):  T25 <- U25\U25


[[A (G M S) Y,exists E1\(and (program E1) (run E1))],
 [A (G P S) Y,all E1\(implies (program E1) (run E1))],
 [A (G M V) Y,exists E1\(and (computer E1) (run E1))],
 [A (G M S) B1,exists E1\(and (program E1) (halt E1))]]

   proj(2,2,1):  A <- T27\U27\(T27 (F1 T27 U27))

[[G M S (F1 (G M S) Y),exists A\(and (program A) (run A))],
 [G P S (F1 (G P S) Y),all A\(implies (program A) (run A))],
 [G M V (F1 (G M V) Y),exists A\(and (computer A) (run A))],
 [G M S (F1 (G M S) B1),exists A\(and (program A) (halt A))]]

   proj(3,3,2):  G <- Q27\R27\S27\(Q27 (G1 Q27 R27 S27) (I1 Q27 R27 S27))
```

```
[[M (G1 M S (F1 H1\(M (G1 M S H1) (I1 M S H1)) Y)) (I1 M S (F1 J1\(M (G1 M
S J1) (I1 M S J1)) Y)),exists A\(and (program A) (run A))],
 [P (G1 P S (F1 K1\(P (G1 P S K1) (I1 P S K1)) Y)) (I1 P S (F1 L1\(P (G1 P
S L1) (I1 P S L1)) Y)),all A\(implies (program A) (run A))],
 [M (G1 M V (F1 M1\(M (G1 M V M1) (I1 M V M1)) Y)) (I1 M V (F1 N1\(M (G1 M
V N1) (I1 M V N1)) Y)),exists A\(and (computer A) (run A))],
 [M (G1 M S (F1 O1\(M (G1 M S O1) (I1 M S O1)) B1)) (I1 M S (F1 P1\(M (G1 M
S P1) (I1 M S P1)) B1)),exists A\(and (program A) (halt A))]]
```

```
   imitation(2):  M <- O27\P27\(exists (Q1 O27 P27))
```

```
[[Q1 (G1 R1\S1\(exists (Q1 R1 S1)) S (F1 T1\(exists (Q1 (G1 U1\V1\(exists
(Q1 U1 V1)) S T1) (I1 W1\X1\(exists (Q1 W1 X1)) S T1))) Y)) (I1
Y1\Z1\(exists (Q1 Y1 Z1)) S (F1 A2\(exists (Q1 (G1 B2\C2\(exists (Q1 B2
C2)) S A2) (I1 D2\E2\(exists (Q1 D2 E2)) S A2))) Y)),A\(and (program A)
(run A))],
 [P (G1 P S (F1 K1\(P (G1 P S K1) (I1 P S K1)) Y)) (I1 P S (F1 L1\(P (G1 P
S L1) (I1 P S L1)) Y)),all A\(implies (program A) (run A))],
 [Q1 (G1 F2\G2\(exists (Q1 F2 G2)) V (F1 H2\(exists (Q1 (G1 I2\J2\(exists
(Q1 I2 J2)) V H2) (I1 K2\L2\(exists (Q1 K2 L2)) V H2))) Y)) (I1
M2\N2\(exists (Q1 M2 N2)) V (F1 O2\(exists (Q1 (G1 P2\Q2\(exists (Q1 P2
Q2)) V O2) (I1 R2\S2\(exists (Q1 R2 S2)) V O2))) Y)),A\(and (computer A)
(run A))],
 [Q1 (G1 T2\U2\(exists (Q1 T2 U2)) S (F1 V2\(exists (Q1 (G1 W2\X2\(exists
(Q1 W2 X2)) S V2) (I1 Y2\Z2\(exists (Q1 Y2 Z2)) S V2))) B1)) (I1
A3\B3\(exists (Q1 A3 B3)) S (F1 C3\(exists (Q1 (G1 D3\E3\(exists (Q1 D3
E3)) S C3) (I1 F3\G3\(exists (Q1 F3 G3)) S C3))) B1)),A\(and (program A)
(halt A))]]
```

```
   imitation(3):  Q1 <- M27\N27\A\(and (L3 M27 N27 A) (H3 M27 N27 A))
```

```
[[H3 (G1 I3\J3\(exists K3\(and (L3 I3 J3 K3) (H3 I3 J3 K3))) S (F1
M3\(exists N3\(and (L3 (G1 O3\P3\(exists Q3\(and (L3 O3 P3 Q3) (H3 O3 P3
Q3))) S M3) (I1 R3\S3\(exists T3\(and (L3 R3 S3 T3) (H3 R3 S3 T3))) S M3)
N3) (H3 (G1 U3\V3\(exists W3\(and (L3 U3 V3 W3) (H3 U3 V3 W3))) S M3) (I1
R3\S3\(exists X3\(and (L3 R3 S3 X3) (H3 R3 S3 X3))) S M3) N3))) Y)) (I1
Y3\Z3\(exists A4\(and (L3 Y3 Z3 A4) (H3 Y3 Z3 A4))) S (F1 B4\(exists
C4\(and (L3 (G1 D4\E4\(exists F4\(and (L3 D4 E4 F4) (H3 D4 E4 F4))) S B4)
(I1 G4\H4\(exists I4\(and (L3 G4 H4 I4) (H3 G4 H4 I4))) S B4) C4) (H3 (G1
J4\K4\(exists L4\(and (L3 J4 K4 L4) (H3 J4 K4 L4))) S B4) (I1 G4\H4\(exists
M4\(and (L3 G4 H4 M4) (H3 G4 H4 M4))) S B4) C4))) Y)) A,run A],
 [L3 (G1 N4\O4\(exists P4\(and (L3 N4 O4 P4) (H3 N4 O4 P4))) S (F1
Q4\(exists R4\(and (L3 (G1 S4\T4\(exists U4\(and (L3 S4 T4 U4) (H3 S4 T4
U4))) S Q4) (I1 V4\W4\(exists X4\(and (L3 V4 W4 X4) (H3 V4 W4 X4))) S Q4)
```

```
R4) (H3 (G1 Y4\Z4\(exists A5\(and (L3 Y4 Z4 A5) (H3 Y4 Z4 A5))) S Q4) (I1
V4\W4\(exists B5\(and (L3 V4 W4 B5) (H3 V4 W4 B5))) S Q4) R4))) Y)) (I1
Y3\Z3\(exists C5\(and (L3 Y3 Z3 C5) (H3 Y3 Z3 C5))) S (F1 B4\(exists
D5\(and (L3 (G1 E5\F5\(exists G5\(and (L3 E5 F5 G5) (H3 E5 F5 G5))) S B4)
(I1 H5\I5\(exists J5\(and (L3 H5 I5 J5) (H3 H5 I5 J5))) S B4) D5) (H3 (G1
K5\L5\(exists M5\(and (L3 K5 L5 M5) (H3 K5 L5 M5))) S B4) (I1 H5\I5\(exists
N5\(and (L3 H5 I5 N5) (H3 H5 I5 N5))) S B4) D5))) Y)) A,program A],
 [P (G1 P S (F1 K1\(P (G1 P S K1) (I1 P S K1)) Y)) (I1 P S (F1 L1\(P (G1 P
S L1) (I1 P S L1)) Y)),all A\(implies (program A) (run A))],
 [H3 (G1 O5\P5\(exists Q5\(and (L3 O5 P5 Q5) (H3 O5 P5 Q5))) V (F1
R5\(exists S5\(and (L3 (G1 T5\U5\(exists V5\(and (L3 T5 U5 V5) (H3 T5 U5
V5))) V R5) (I1 W5\X5\(exists Y5\(and (L3 W5 X5 Y5) (H3 W5 X5 Y5))) V R5)
S5) (H3 (G1 Z5\A6\(exists B6\(and (L3 Z5 A6 B6) (H3 Z5 A6 B6))) V R5) (I1
W5\X5\(exists C6\(and (L3 W5 X5 C6) (H3 W5 X5 C6))) V R5) S5))) Y)) (I1
D6\E6\(exists F6\(and (L3 D6 E6 F6) (H3 D6 E6 F6))) V (F1 G6\(exists
H6\(and (L3 (G1 I6\J6\(exists K6\(and (L3 I6 J6 K6) (H3 I6 J6 K6))) V G6)
(I1 L6\M6\(exists N6\(and (L3 L6 M6 N6) (H3 L6 M6 N6))) V G6) H6) (H3 (G1
O6\P6\(exists Q6\(and (L3 O6 P6 Q6) (H3 O6 P6 Q6))) V G6) (I1 L6\M6\(exists
R6\(and (L3 L6 M6 R6) (H3 L6 M6 R6))) V G6) H6))) Y)) A,run A],
 [L3 (G1 S6\T6\(exists U6\(and (L3 S6 T6 U6) (H3 S6 T6 U6))) V (F1
V6\(exists W6\(and (L3 (G1 X6\Y6\(exists Z6\(and (L3 X6 Y6 Z6) (H3 X6 Y6
Z6))) V V6) (I1 A7\B7\(exists C7\(and (L3 A7 B7 C7) (H3 A7 B7 C7))) V V6)
W6) (H3 (G1 D7\E7\(exists F7\(and (L3 D7 E7 F7) (H3 D7 E7 F7))) V V6) (I1
A7\B7\(exists G7\(and (L3 A7 B7 G7) (H3 A7 B7 G7))) V V6) W6))) Y)) (I1
D6\E6\(exists H7\(and (L3 D6 E6 H7) (H3 D6 E6 H7))) V (F1 G6\(exists
I7\(and (L3 (G1 J7\K7\(exists L7\(and (L3 J7 K7 L7) (H3 J7 K7 L7))) V G6)
(I1 M7\N7\(exists O7\(and (L3 M7 N7 O7) (H3 M7 N7 O7))) V G6) I7) (H3 (G1
P7\Q7\(exists R7\(and (L3 P7 Q7 R7) (H3 P7 Q7 R7))) V G6) (I1 M7\N7\(exists
S7\(and (L3 M7 N7 S7) (H3 M7 N7 S7))) V G6) I7))) Y)) A,computer A],
 [H3 (G1 T7\U7\(exists V7\(and (L3 T7 U7 V7) (H3 T7 U7 V7))) S (F1
W7\(exists X7\(and (L3 (G1 Y7\Z7\(exists A8\(and (L3 Y7 Z7 A8) (H3 Y7 Z7
A8))) S W7) (I1 B8\C8\(exists D8\(and (L3 B8 C8 D8) (H3 B8 C8 D8))) S W7)
X7) (H3 (G1 E8\F8\(exists G8\(and (L3 E8 F8 G8) (H3 E8 F8 G8))) S W7) (I1
B8\C8\(exists H8\(and (L3 B8 C8 H8) (H3 B8 C8 H8))) S W7) X7))) B1)) (I1
I8\J8\(exists K8\(and (L3 I8 J8 K8) (H3 I8 J8 K8))) S (F1 L8\(exists
M8\(and (L3 (G1 N8\O8\(exists P8\(and (L3 N8 O8 P8) (H3 N8 O8 P8))) S L8)
(I1 Q8\R8\(exists S8\(and (L3 Q8 R8 S8) (H3 Q8 R8 S8))) S L8) M8) (H3 (G1
T8\U8\(exists V8\(and (L3 T8 U8 V8) (H3 T8 U8 V8))) S L8) (I1 Q8\R8\(exists
W8\(and (L3 Q8 R8 W8) (H3 Q8 R8 W8))) S L8) M8))) B1)) A,halt A],
 [L3 (G1 X8\Y8\(exists Z8\(and (L3 X8 Y8 Z8) (H3 X8 Y8 Z8))) S (F1
A9\(exists B9\(and (L3 (G1 C9\D9\(exists E9\(and (L3 C9 D9 E9) (H3 C9 D9
E9))) S A9) (I1 F9\G9\(exists H9\(and (L3 F9 G9 H9) (H3 F9 G9 H9))) S A9)
B9) (H3 (G1 I9\J9\(exists K9\(and (L3 I9 J9 K9) (H3 I9 J9 K9))) S A9) (I1
F9\G9\(exists L9\(and (L3 F9 G9 L9) (H3 F9 G9 L9))) S A9) B9))) B1)) (I1
```

```
I8\J8\(exists M9\(and (L3 I8 J8 M9) (H3 I8 J8 M9))) S (F1 L8\(exists
N9\(and (L3 (G1 O9\P9\(exists Q9\(and (L3 O9 P9 Q9) (H3 O9 P9 Q9))) S L8)
(I1 R9\S9\(exists T9\(and (L3 R9 S9 T9) (H3 R9 S9 T9))) S L8) N9) (H3 (G1
U9\V9\(exists W9\(and (L3 U9 V9 W9) (H3 U9 V9 W9))) S L8) (I1 R9\S9\(exists
X9\(and (L3 R9 S9 X9) (H3 R9 S9 X9))) S L8) N9))) B1)) A,program A]]

   proj(3,3,1):  H3 <- J27\K27\L27\(J27 (B10 J27 K27 L27))

[[G1 Y9\Z9\(exists A10\(and (L3 Y9 Z9 A10) (Y9 (B10 Y9 Z9 A10)))) S (F1
C10\(exists D10\(and (L3 (G1 E10\F10\(exists G10\(and (L3 E10 F10 G10) (E10
(B10 E10 F10 G10)))) S C10) (I1 H10\I10\(exists J10\(and (L3 H10 I10 J10)
(H10 (B10 H10 I10 J10)))) S C10) D10) (G1 K10\L10\(exists M10\(and (L3 K10
L10 M10) (K10 (B10 K10 L10 M10)))) S C10 (B10 (G1 N10\O10\(exists P10\(and
(L3 N10 O10 P10) (N10 (B10 N10 O10 P10)))) S C10) (I1 Q10\R10\(exists
S10\(and (L3 Q10 R10 S10) (Q10 (B10 Q10 R10 S10)))) S C10) D10)))) Y) (B10
(G1 T10\U10\(exists V10\(and (L3 T10 U10 V10) (T10 (B10 T10 U10 V10)))) S
(F1 W10\(exists X10\(and (L3 (G1 Y10\Z10\(exists A11\(and (L3 Y10 Z10 A11)
(Y10 (B10 Y10 Z10 A11)))) S W10) (I1 B11\C11\(exists D11\(and (L3 B11 C11
D11) (B11 (B10 B11 C11 D11)))) S W10) X10) (G1 E11\F11\(exists G11\(and (L3
E11 F11 G11) (E11 (B10 E11 F11 G11)))) S W10 (B10 (G1 H11\I11\(exists
J11\(and (L3 H11 I11 J11) (H11 (B10 H11 I11 J11)))) S W10) (I1
K11\L11\(exists M11\(and (L3 K11 L11 M11!
) (K11 (B10 K11 L11 M11)))) S W10)
 X10)))) Y)) (I1 N11\O11\(exists P11\(and (L3 N11 O11 P11) (N11 (B10 N11
O11 P11)))) S (F1 Q11\(exists R11\(and (L3 (G1 S11\T11\(exists U11\(and (L3
S11 T11 U11) (S11 (B10 S11 T11 U11)))) S Q11) (I1 V11\W11\(exists X11\(and
(L3 V11 W11 X11) (V11 (B10 V11 W11 X11)))) S Q11) R11) (G1 Y11\Z11\(exists
A12\(and (L3 Y11 Z11 A12) (Y11 (B10 Y11 Z11 A12)))) S Q11 (B10 (G1
B12\C12\(exists D12\(and (L3 B12 C12 D12) (B12 (B10 B12 C12 D12)))) S Q11)
(I1 E12\F12\(exists G12\(and (L3 E12 F12 G12) (E12 (B10 E12 F12 G12)))) S
Q11) R11)))) Y)) A),run A],
 [L3 (G1 N4\O4\(exists P4\(and (L3 N4 O4 P4) (N4 (B10 N4 O4 P4)))) S (F1
Q4\(exists R4\(and (L3 (G1 S4\T4\(exists U4\(and (L3 S4 T4 U4) (S4 (B10 S4
T4 U4)))) S Q4) (I1 V4\W4\(exists X4\(and (L3 V4 W4 X4) (V4 (B10 V4 W4
X4)))) S Q4) R4) (G1 H12\I12\(exists J12\(and (L3 H12 I12 J12) (H12 (B10
H12 I12 J12)))) S Q4 (B10 (G1 K12\L12\(exists M12\(and (L3 K12 L12 M12)
(K12 (B10 K12 L12 M12)))) S Q4) (I1 N12\O12\(exists P12\(and (L3 N12 O12
P12) (N12 (B10 N12 O12 P12)))) S Q4) R4)))) Y)) (I1 Y3\Z3\(exists C5\(and
(L3 Y3 Z3 C5) (Y3 (B10 Y3 Z3 C5)))) S (F1 B4\(exists D5\(and (L3 (G1
E5\F5\(exists G5\(and (L3 E5 F5 G5) (E5 (B10 E5 F5 G5)))) S B4) (I1
H5\I5\(exists J5\(and (L3 H5 I5 J5) (H5 (B10 H5 I5 J5)))) S B4) D5) (G1
Q12\R12\(exists S12\(and (L3 Q12 R12 S12) (Q12 (B10 Q12 R12 S12)))) S B4
(B10 (G1 T12\U12\(exists V12\(and (L3 T12 U12 V12) (T12 (B10 T12 U12
V12)))) S B4) (I1 W12\X12\(exists Y12\(and (L3 W12 X12 Y12) (W12 (B10 W12
```

```
X12 Y12)))) S B4) D5)))) Y)) A,program A],
 [P (G1 P S (F1 K1\(P (G1 P S K1) (I1 P S K1)) Y)) (I1 P S (F1 L1\(P (G1 P
S L1) (I1 P S L1)) Y)),all A\(implies (program A) (run A))],
 [G1 Z12\A13\(exists B13\(and (L3 Z12 A13 B13) (Z12 (B10 Z12 A13 B13)))) V
(F1 C13\(exists D13\(and (L3 (G1 E13\F13\(exists G13\(and (L3 E13 F13 G13)
(E13 (B10 E13 F13 G13)))) V C13) (I1 H13\I13\(exists J13\(and (L3 H13 I13
J13) (H13 (B10 H13 I13 J13)))) V C13) D13) (G1 K13\L13\(exists M13\(and (L3
K13 L13 M13) (K13 (B10 K13 L13 M13)))) V C13 (B10 (G1 N13\O13\(exists
P13\(and (L3 N13 O13 P13) (N13 (B10 N13 O13 P13)))) V C13) (I1
Q13\R13\(exists S13\(and (L3 Q13 R13 S13) (Q13 (B10 Q13 R13 S13)))) V C13)
D13)))) Y) (B10 (G1 T13\U13\(exists V13\(and (L3 T13 U13 V13) (T13 (B10 T13
U13 V13)))) V (F1 W13\(exists X13\(and (L3 (G1 Y13\Z13\(exists A14\(and (L3
Y13 Z13 A14) (Y13 (B10 Y13 Z13 A14)))) V W13) (I1 B14\C14\(exists D14\(and
(L3 B14 C14 D14) (B14 (B10 B14 C14 D14)))) V W13) X13) (G1 E14\F14\(exists
G14\(and (L3 E14 F14 G14) (E14 (B10 E14 F14 G14)))) V W13 (B10 (G1
H14\I14\(exists J14\(and (L3 H14 I14 J14) (H14 (B10 H14 I14 J14)))) V W13)
(I1 K14\L14\(exists M14\(and (L3 K14 L14 M14) (K14 (B10 K14 L14 M14))))
 V W13) X13)))) Y)) (I1 N14\O14\(exists P14\(and (L3 N14 O14 P14) (N14 (B10
N14 O14 P14)))) V (F1 Q14\(exists R14\(and (L3 (G1 S14\T14\(exists U14\(and
(L3 S14 T14 U14) (S14 (B10 S14 T14 U14)))) V Q14) (I1 V14\W14\(exists
X14\(and (L3 V14 W14 X14) (V14 (B10 V14 W14 X14)))) V Q14) R14) (G1
Y14\Z14\(exists A15\(and (L3 Y14 Z14 A15) (Y14 (B10 Y14 Z14 A15)))) V Q14
(B10 (G1 B15\C15\(exists D15\(and (L3 B15 C15 D15) (B15 (B10 B15 C15
D15)))) V Q14) (I1 E15\F15\(exists G15\(and (L3 E15 F15 G15) (E15 (B10 E15
F15 G15)))) V Q14) R14)))) Y)) A),run A],
 [L3 (G1 S6\T6\(exists U6\(and (L3 S6 T6 U6) (S6 (B10 S6 T6 U6)))) V (F1
V6\(exists W6\(and (L3 (G1 X6\Y6\(exists Z6\(and (L3 X6 Y6 Z6) (X6 (B10 X6
Y6 Z6)))) V V6) (I1 A7\B7\(exists C7\(and (L3 A7 B7 C7) (A7 (B10 A7 B7
C7)))) V V6) W6) (G1 H15\I15\(exists J15\(and (L3 H15 I15 J15) (H15 (B10
H15 I15 J15)))) V V6 (B10 (G1 K15\L15\(exists M15\(and (L3 K15 L15 M15)
(K15 (B10 K15 L15 M15)))) V V6) (I1 N15\O15\(exists P15\(and (L3 N15 O15
P15) (N15 (B10 N15 O15 P15)))) V V6) W6)))) Y)) (I1 D6\E6\(exists H7\(and
(L3 D6 E6 H7) (D6 (B10 D6 E6 H7)))) V (F1 G6\(exists I7\(and (L3 (G1
J7\K7\(exists L7\(and (L3 J7 K7 L7) (J7 (B10 J7 K7 L7)))) V G6) (I1
M7\N7\(exists O7\(and (L3 M7 N7 O7) (M7 (B10 M7 N7 O7)))) V G6) I7) (G1
Q15\R15\(exists S15\(and (L3 Q15 R15 S15) (Q15 (B10 Q15 R15 S15)))) V G6
(B10 (G1 T15\U15\(exists V15\(and (L3 T15 U15 V15) (T15 (B10 T15 U15
V15)))) V G6) (I1 W15\X15\(exists Y15\(and (L3 W15 X15 Y15) (W15 (B10 W15
X15 Y15)))) V G6) I7)))) Y)) A,computer A],
 [G1 Z15\A16\(exists B16\(and (L3 Z15 A16 B16) (Z15 (B10 Z15 A16 B16)))) S
(F1 C16\(exists D16\(and (L3 (G1 E16\F16\(exists G16\(and (L3 E16 F16 G16)
(E16 (B10 E16 F16 G16)))) S C16) (I1 H16\I16\(exists J16\(and (L3 H16 I16
J16) (H16 (B10 H16 I16 J16)))) S C16) D16) (G1 K16\L16\(exists M16\(and (L3
K16 L16 M16) (K16 (B10 K16 L16 M16)))) S C16 (B10 (G1 N16\O16\(exists
```

```
P16\(and (L3 N16 O16 P16) (N16 (B10 N16 O16 P16)))) S C16) (I1
Q16\R16\(exists S16\(and (L3 Q16 R16 S16) (Q16 (B10 Q16 R16 S16)))) S C16)
D16)))) B1) (B10 (G1 T16\U16\(exists V16\(and (L3 T16 U16 V16) (T16 (B10
T16 U16 V16)))) S (F1 W16\(exists X16\(and (L3 (G1 Y16\Z16\(exists A17\(and
(L3 Y16 Z16 A17) (Y16 (B10 Y16 Z16 A17)))) S W16) (I1 B17\C17\(exists
D17\(and (L3 B17 C17 D17) (B17 (B10 B17 C17 D17)))) S W16) X16) (G1
E17\F17\(exists G17\(and (L3 E17 F17 G17) (E17 (B10 E17 F17 G17)))) S W16
(B10 (G1 H17\I17\(exists J17\(and (L3 H17 I17 J17) (H17 (B10 H17 I17
J17)))) S W16) (I1 K17\L17\(exists M17\(and (L3 K17 L17 M17) (K17 (B10 K17
L17 M17)))) S W16) X16)))) B1)) (I1 N17\O17\(exists P17\(and (L3 N17 O17
P17) (N17 (B10 N17 O17 P17)))) S (F1 Q17\(exists R17\(and (L3 (G1
S17\T17\(exists U17\(and (L3 S17 T17 U17) (S17 (B10 S17 T17 U17)))) S Q17)
(I1 V17\W17\(exists X17\(and (L3 V17 W17 X17) (V17 (B10 V17 W17 X17)))) S
Q17) R17) (G1 Y17\Z17\(exists A18\(and (L3 Y17 Z17 A18) (Y17 (B10 Y17 Z17
A18)))) S Q17 (B10 (G1 B18\C18\(exists D18\(and (L3 B18 C18 D18) (B18 (B10
B18 C18 D18)))) S Q17) (I1 E18\F18\(exists G18\(and (L3 E18 F18 G18) (E18
(B10 E18 F18 G18)))) S Q17) R17)))) B1)) A),halt A],
 [L3 (G1 X8\Y8\(exists Z8\(and (L3 X8 Y8 Z8) (X8 (B10 X8 Y8 Z8)))) S (F1
A9\(exists B9\(and (L3 (G1 C9\D9\(exists E9\(and (L3 C9 D9 E9) (C9 (B10 C9
D9 E9)))) S A9) (I1 F9\G9\(exists H9\(and (L3 F9 G9 H9) (F9 (B10 F9 G9
H9)))) S A9) B9) (G1 H18\I18\(exists J18\(and (L3 H18 I18 J18) (H18 (B10
H18 I18 J18)))) S A9 (B10 (G1 K18\L18\(exists M18\(and (L3 K18 L18 M18)
(K18 (B10 K18 L18 M18)))) S A9) (I1 N18\O18\(exists P18\(and (L3 N18 O18
P18) (N18 (B10 N18 O18 P18)))) S A9) B9)))) B1)) (I1 I8\J8\(exists M9\(and
(L3 I8 J8 M9) (I8 (B10 I8 J8 M9)))) S (F1 L8\(exists N9\(and (L3 (G1
O9\P9\(exists Q9\(and (L3 O9 P9 Q9) (O9 (B10 O9 P9 Q9)))) S L8) (I1
R9\S9\(exists T9\(and (L3 R9 S9 T9) (R9 (B10 R9 S9 T9)))) S L8) N9) (G1
Q18\R18\(exists S18\(and (L3 Q18 R18 S18) (Q18 (B10 Q18 R18 S18)))) S L8
(B10 (G1 T18\U18\(exists V18\(and (L3 T18 U18 V18) (T18 (B10 T18 U18
V18)))) S L8) (I1 W18\X18\(exists Y18\(and (L3 W18 X18 Y18) (W18 (B10 W18
X18 Y18)))) S L8) N9)))) B1)) A,program A]]
```

```
   proj(3,1,0):  G1 <- G27\H27\I27\I27
```

```
[[F1 Z18\(exists A19\(and (L3 Z18 (I1 B19\C19\(exists D19\(and (L3 B19 C19
D19) (B19 (B10 B19 C19 D19)))) S Z18) A19) (Z18 (B10 Z18 (I1
E19\F19\(exists G19\(and (L3 E19 F19 G19) (E19 (B10 E19 F19 G19)))) S Z18)
A19)))) Y (B10 (F1 H19\(exists I19\(and (L3 H19 (I1 J19\K19\(exists
L19\(and (L3 J19 K19 L19) (J19 (B10 J19 K19 L19)))) S H19) I19) (H19 (B10
H19 (I1 M19\N19\(exists O19\(and (L3 M19 N19 O19) (M19 (B10 M19 N19 O19))))
S H19) I19)))) Y) (I1 P19\Q19\(exists R19\(and (L3 P19 Q19 R19) (P19 (B10
P19 Q19 R19)))) S (F1 S19\(exists T19\(and (L3 S19 (I1 U19\V19\(exists
W19\(and (L3 U19 V19 W19) (U19 (B10 U19 V19 W19)))) S S19) T19) (S19 (B10
S19 (I1 X19\Y19\(exists Z19\(and (L3 X19 Y19 Z19) (X19 (B10 X19 Y19 Z19))))
```

```
S S19) T19)))) Y)) A),run A],
 [L3 (F1 A20\(exists B20\(and (L3 A20 (I1 C20\D20\(exists E20\(and (L3 C20
D20 E20) (C20 (B10 C20 D20 E20)))) S A20) B20) (A20 (B10 A20 (I1
F20\G20\(exists H20\(and (L3 F20 G20 H20) (F20 (B10 F20 G20 H20)))) S A20)
B20)))) Y) (I1 Y3\Z3\(exists C5\(and (L3 Y3 Z3 C5) (Y3 (B10 Y3 Z3 C5)))) S
(F1 B4\(exists D5\(and (L3 B4 (I1 H5\I5\(exists J5\(and (L3 H5 I5 J5) (H5
(B10 H5 I5 J5)))) S B4) D5) (B4 (B10 B4 (I1 I20\J20\(exists K20\(and (L3
I20 J20 K20) (I20 (B10 I20 J20 K20)))) S B4) D5)))) Y)) A,program A],
 [P (F1 L20\(P L20 (I1 P S L20)) Y) (I1 P S (F1 L1\(P L1 (I1 P S L1))
Y)),all A\(implies (program A) (run A))],
 [F1 M20\(exists N20\(and (L3 M20 (I1 O20\P20\(exists Q20\(and (L3 O20 P20
Q20) (O20 (B10 O20 P20 Q20)))) V M20) N20) (M20 (B10 M20 (I1
R20\S20\(exists T20\(and (L3 R20 S20 T20) (R20 (B10 R20 S20 T20)))) V M20)
N20)))) Y (B10 (F1 U20\(exists V20\(and (L3 U20 (I1 W20\X20\(exists
Y20\(and (L3 W20 X20 Y20) (W20 (B10 W20 X20 Y20)))) V U20) V20) (U20 (B10
U20 (I1 Z20\A21\(exists B21\(and (L3 Z20 A21 B21) (Z20 (B10 Z20 A21 B21))))
V U20) V20)))) Y) (I1 C21\D21\(exists E21\(and (L3 C21 D21 E21) (C21 (B10
C21 D21 E21)))) V (F1 F21\(exists G21\(and (L3 F21 (I1 H21\I21\(exists
J21\(and (L3 H21 I21 J21) (H21 (B10 H21 I21 J21)))) V F21) G21) (F21 (B10
F21 (I1 K21\L21\(exists M21\(and (L3 K21 L21 M21) (K21 (B10 K21 L21 M21))))
V F21) G21)))) Y)) A),run A],
 [L3 (F1 N21\(exists O21\(and (L3 N21 (I1 P21\Q21\(exists R21\(and (L3 P21
Q21 R21) (P21 (B10 P21 Q21 R21)))) V N21) O21) (N21 (B10 N21 (I1
S21\T21\(exists U21\(and (L3 S21 T21 U21) (S21 (B10 S21 T21 U21)))) V N21)
O21)))) Y) (I1 D6\E6\(exists H7\(and (L3 D6 E6 H7) (D6 (B10 D6 E6 H7)))) V
(F1 G6\(exists I7\(and (L3 G6 (I1 M7\N7\(exists O7\(and (L3 M7 N7 O7) (M7
(B10 M7 N7 O7)))) V G6) I7) (G6 (B10 G6 (I1 V21\W21\(exists X21\(and (L3
V21 W21 X21) (V21 (B10 V21 W21 X21)))) V G6) I7)))) Y)) A,computer A],
 [F1 Y21\(exists Z21\(and (L3 Y21 (I1 A22\B22\(exists C22\(and (L3 A22 B22
C22) (A22 (B10 A22 B22 C22)))) S Y21) Z21) (Y21 (B10 Y21 (I1
D22\E22\(exists F22\(and (L3 D22 E22 F22) (D22 (B10 D22 E22 F22)))) S Y21)
Z21)))) B1) (B10 (F1 G22\(exists H22\(and (L3 G22 (I1 I22\J22\(exists
K22\(and (L3 I22 J22 K22) (I22 (B10 I22 J22 K22)))) S G22) H22) (G22 (B10
G22 (I1 L22\M22\(exists N22\(and (L3 L22 M22 N22) (L22 (B10 L22 M22 N22))))
S G22) H22)))) B1) (I1 O22\P22\(exists Q22\(and (L3 O22 P22 Q22) (O22 (B10
O22 P22 Q22)))) S (F1 R22\(exists S22\(and (L3 R22 (I1 T22\U22\(exists
V22\(and (L3 T22 U22 V22) (T22 (B10 T22 U22 V22)))) S R22) S22) (R22 (B10
R22 (I1 W22\X22\(exists Y22\(and (L3 W22 X22 Y22) (W22 (B10 W22 X22 Y22))))
S R22) S22)))) B1)) A),halt A],
 [L3 (F1 Z22\(exists A23\(and (L3 Z22 (I1 B23\C23\(exists D23\(and (L3 B23
C23 D23) (B23 (B10 B23 C23 D23)))) S Z22) A23) (Z22 (B10 Z22 (I1
E23\F23\(exists G23\(and (L3 E23 F23 G23) (E23 (B10 E23 F23 G23)))) S Z22)
A23)))) B1) (I1 I8\J8\(exists M9\(and (L3 I8 J8 M9) (I8 (B10 I8 J8 M9)))) S
(F1 L8\(exists N9\(and (L3 L8 (I1 R9\S9\(exists T9\(and (L3 R9 S9 T9) (R9
```

```
(B10 R9 S9 T9)))) S L8) N9) (L8 (B10 L8 (I1 H23\I23\(exists J23\(and (L3
H23 I23 J23) (H23 (B10 H23 I23 J23)))) S L8) N9)))) B1)) A,program A]]

   proj(2,1,0):  F1 <- E27\F27\F27

[[Y (B10 Y (I1 K23\L23\(exists M23\(and (L3 K23 L23 M23) (K23 (B10 K23 L23
M23)))) S Y) A),run A],
 [L3 Y (I1 Y3\Z3\(exists C5\(and (L3 Y3 Z3 C5) (Y3 (B10 Y3 Z3 C5)))) S Y)
A,program A],
 [P Y (I1 P S Y),all A\(implies (program A) (run A))],
 [Y (B10 Y (I1 N23\O23\(exists P23\(and (L3 N23 O23 P23) (N23 (B10 N23 O23
P23)))) V Y) A),run A],
 [L3 Y (I1 D6\E6\(exists H7\(and (L3 D6 E6 H7) (D6 (B10 D6 E6 H7)))) V Y)
A,computer A],
 [B1 (B10 B1 (I1 Q23\R23\(exists S23\(and (L3 Q23 R23 S23) (Q23 (B10 Q23
R23 S23)))) S B1) A),halt A],
 [L3 B1 (I1 I8\J8\(exists M9\(and (L3 I8 J8 M9) (I8 (B10 I8 J8 M9)))) S B1)
A,program A]]

   imitation(1):  Y <- D27\(run (T23 D27))

[[T23 (B10 U23\(run (T23 U23)) (I1 V23\W23\(exists X23\(and (L3 V23 W23
X23) (V23 (B10 V23 W23 X23)))) S Y23\(run (T23 Y23))) A),A],
 [L3 Z23\(run (T23 Z23)) (I1 Y3\Z3\(exists C5\(and (L3 Y3 Z3 C5) (Y3 (B10
Y3 Z3 C5)))) S A24\(run (T23 A24))) A,program A],
 [P B24\(run (T23 B24)) (I1 P S C24\(run (T23 C24))),all A\(implies
(program A) (run A))],
 [T23 (B10 D24\(run (T23 D24)) (I1 E24\F24\(exists G24\(and (L3 E24 F24
G24) (E24 (B10 E24 F24 G24)))) V H24\(run (T23 H24))) A),A],
 [L3 I24\(run (T23 I24)) (I1 D6\E6\(exists H7\(and (L3 D6 E6 H7) (D6 (B10
D6 E6 H7)))) V J24\(run (T23 J24))) A,computer A],
 [B1 (B10 B1 (I1 Q23\R23\(exists S23\(and (L3 Q23 R23 S23) (Q23 (B10 Q23
R23 S23)))) S B1) A),halt A],
 [L3 B1 (I1 I8\J8\(exists M9\(and (L3 I8 J8 M9) (I8 (B10 I8 J8 M9)))) S B1)
A,program A]]

   proj(1,1,0):  T23 <- C27\C27

[[B10 run (I1 K24\L24\(exists M24\(and (L3 K24 L24 M24) (K24 (B10 K24 L24
M24)))) S run) A,A],
 [L3 run (I1 Y3\Z3\(exists C5\(and (L3 Y3 Z3 C5) (Y3 (B10 Y3 Z3 C5)))) S
run) A,program A],
 [P run (I1 P S run),all A\(implies (program A) (run A))],
 [B10 run (I1 N24\O24\(exists P24\(and (L3 N24 O24 P24) (N24 (B10 N24 O24
```

```
P24)))) V run) A,A],
 [L3 run (I1 D6\E6\(exists H7\(and (L3 D6 E6 H7) (D6 (B10 D6 E6 H7)))) V
run) A,computer A],
 [B1 (B10 B1 (I1 Q23\R23\(exists S23\(and (L3 Q23 R23 S23) (Q23 (B10 Q23
R23 S23)))) S B1) A),halt A],
 [L3 B1 (I1 I8\J8\(exists M9\(and (L3 I8 J8 M9) (I8 (B10 I8 J8 M9)))) S B1)
A,program A]]

   proj(3,1,0):  B10 <- Z26\A27\B27\B27

[[L3 run (I1 Y3\Z3\(exists C5\(and (L3 Y3 Z3 C5) (Y3 C5))) S run) A,program A],
 [P run (I1 P S run),all A\(implies (program A) (run A))],
 [L3 run (I1 D6\E6\(exists H7\(and (L3 D6 E6 H7) (D6 H7))) V run) A,
computer A],
 [B1 A,halt A],
 [L3 B1 (I1 I8\J8\(exists M9\(and (L3 I8 J8 M9) (I8 M9))) S B1) A,program A]]

   proj(3,2,1):  L3 <- W26\X26\Y26\(X26 (T24 W26 X26 Y26))

[[I1 Q24\R24\(exists S24\(and (R24 (T24 Q24 R24 S24)) (Q24 S24))) S run
(T24 run (I1 U24\V24\(exists W24\(and (V24 (T24 U24 V24 W24)) (U24 W24))) S
run) A),program A],
 [P run (I1 P S run),all A\(implies (program A) (run A))],
 [I1 X24\Y24\(exists Z24\(and (Y24 (T24 X24 Y24 Z24)) (X24 Z24))) V run
(T24 run (I1 A25\B25\(exists C25\(and (B25 (T24 A25 B25 C25)) (A25 C25))) V
run) A),computer A],
 [B1 A,halt A],
 [I1 D25\E25\(exists F25\(and (E25 (T24 D25 E25 F25)) (D25 F25))) S B1 (T24
B1 (I1 G25\H25\(exists I25\(and (H25 (T24 G25 H25 I25)) (G25 I25))) S B1)
A),program A]]

   proj(3,2,0):  I1 <- T26\U26\V26\U26

[[S (T24 run S A),program A],
 [P run S,all A\(implies (program A) (run A))],
 [V (T24 run V A),computer A],
 [B1 A,halt A],
 [S (T24 B1 S A),program A]]

   imitation(1):  S <- S26\(program (J25 S26))

[[J25 (T24 run K25\(program (J25 K25)) A),A],
 [P run L25\(program (J25 L25)),all A\(implies (program A) (run A))],
 [V (T24 run V A),computer A],
```

```
 [B1 A,halt A],
 [J25 (T24 B1 M25\(program (J25 M25)) A),A]]

   proj(1,1,0):  J25 <- R26\R26

[[T24 run program A,A],
 [P run program,all A\(implies (program A) (run A))],
 [V (T24 run V A),computer A],
 [B1 A,halt A],
 [T24 B1 program A,A]]

   proj(3,1,0):  T24 <- O26\P26\Q26\Q26

[[P run program,all A\(implies (program A) (run A))],
 [V A,computer A],
 [B1 A,halt A]]

   imitation(2):  P <- M26\N26\(all (N25 M26 N26))

[[N25 run program,A\(implies (program A) (run A))],
 [V A,computer A],
 [B1 A,halt A]]

   imitation(3):  N25 <- K26\L26\A\(implies (P25 K26 L26 A) (O25 K26 L26 A))

[[O25 run program A,run A],
 [P25 run program A,program A],
 [V A,computer A],
 [B1 A,halt A]]

   proj(3,3,1):  O25 <- H26\I26\J26\(H26 (Q25 H26 I26 J26))

[[Q25 run program A,A],
 [P25 run program A,program A],
 [V A,computer A],
 [B1 A,halt A]]

   proj(3,1,0):  Q25 <- E26\F26\G26\G26

[[P25 run program A,program A],
 [V A,computer A],
 [B1 A,halt A]]

   proj(3,2,1):  P25 <- B26\C26\D26\(C26 (R25 B26 C26 D26))
```

```
[[R25 run program A,A],
 [V A,computer A],
 [B1 A,halt A]]

   proj(3,1,0):  R25 <- Y25\Z25\A26\A26

[[V A,computer A],
 [B1 A,halt A]]

   imitation(1):  V <- X25\(computer (S25 X25))

[[S25 A,A],
 [B1 A,halt A]]

   proj(1,1,0):  S25 <- W25\W25

[[B1 A,halt A]]

   imitation(1):  B1 <- V25\(halt (T25 V25))

[[T25 A,A]]

   proj(1,1,0):  T25 <- U25\U25

[]

Final substitutions:

      A = T27\U27\(T27 U27)
      G = Q27\R27\S27\(Q27 S27 R27)
      M = O27\P27\(exists W27\(and (P27 W27) (O27 W27)))
      Q1 = M27\N27\A\(and (N27 A) (M27 A))
      H3 = J27\K27\L27\(J27 L27)
      G1 = G27\H27\I27\I27
      F1 = E27\F27\F27
      Y = run
      T23 = C27\C27
      B10 = Z26\A27\B27\B27
      L3 = W26\X26\Y26\(X26 Y26)
      I1 = T26\U26\V26\U26
      S = program
      J25 = R26\R26
      T24 = O26\P26\Q26\Q26
```

```
P = M26\N26\(all V27\(implies (N26 V27) (M26 V27)))
N25 = K26\L26\A\(implies (L26 A) (K26 A))
O25 = H26\I26\J26\(H26 J26)
Q25 = E26\F26\G26\G26
P25 = B26\C26\D26\(C26 D26)
R25 = Y25\Z25\A26\A26
V = computer
S25 = W25\W25
B1 = halt
T25 = U25\U25
```

With combintation rules:

```
s(A B C) --> np(B),iv(C).
np(G H I) --> det(H),n(I).
det(M) --> [a].
det(P) --> [every].
n(S) --> [program].
n(V) --> [computer].
iv(Y) --> [runs].
iv(B1) --> [halts].

[[A (G M S) Y,exists E1\(and (program E1) (run E1))],
 [A (G P S) Y,all E1\(and (program E1) (run E1))],
 [A (G M V) Y,exists E1\(and (computer E1) (run E1))],
 [A (G M S) B1,exists E1\(and (program E1) (halt E1))]]

   cproj(2,2,1):  A <- A2\B2\(A2 B2)

[[G M S Y,exists A\(and (program A) (run A))],
 [G P S Y,all A\(and (program A) (run A))],
 [G M V Y,exists A\(and (computer A) (run A))],
 [G M S B1,exists A\(and (program A) (halt A))]]

   cproj(3,3,2):  G <- X1\Y1\Z1\(X1 Z1 Y1)

[[M Y S,exists A\(and (program A) (run A))],
 [P Y S,all A\(and (program A) (run A))],
 [M Y V,exists A\(and (computer A) (run A))],
 [M B1 S,exists A\(and (program A) (halt A))]]

   imitation(2):  M <- V1\W1\(exists (F1 V1 W1))

[[F1 Y S,A\(and (program A) (run A))],
```

```
 [P Y S,all A\(and (program A) (run A))],
 [F1 Y V,A\(and (computer A) (run A))],
 [F1 B1 S,A\(and (program A) (halt A))]]

   imit_proj(2):  F1 <- T1\U1\A\(and (U1 A) (T1 A))

[[Y A,run A],
 [S A,program A],
 [P Y S,all A\(and (program A) (run A))],
 [Y A,run A],
 [V A,computer A],
 [B1 A,halt A],
 [S A,program A]]

   imit_proj(1):  Y <- S1\(run S1)

[[S A,program A],
 [P G1\(run G1) S,all A\(and (program A) (run A))],
 [V A,computer A],
 [B1 A,halt A],
 [S A,program A]]

   imit_proj(1):  S <- R1\(program R1)

[[P G1\(run G1) H1\(program H1),all A\(and (program A) (run A))],
 [V A,computer A],
 [B1 A,halt A]]

   imitation(2):  P <- P1\Q1\(all (I1 P1 Q1))

[[I1 J1\(run J1) K1\(program K1),A\(and (program A) (run A))],
 [V A,computer A],
 [B1 A,halt A]]

   imit_proj(2):  I1 <- N1\O1\A\(and (O1 A) (N1 A))

[[V A,computer A],
 [B1 A,halt A]]

   imit_proj(1):  V <- M1\(computer M1)

[[B1 A,halt A]]

   imit_proj(1):  B1 <- L1\(halt L1)
```

```
[]
      A = A2\B2\(A2 B2)
      G = X1\Y1\Z1\(X1 Z1 Y1)
      M = V1\W1\(exists A\(and (W1 A) (V1 A)))
      F1 = T1\U1\A\(and (U1 A) (T1 A))
      Y = S1\(run S1)
      S = R1\(program R1)
      P = P1\Q1\(all A\(and (Q1 A) (P1 A)))
      I1 = N1\O1\A\(and (O1 A) (N1 A))
      V = M1\(computer M1)
      B1 = L1\(halt L1)
```

E: Further examples of DCG synthesis

E.1. Learning arithmetic functions using Church numerals

E.1.1. Addition function

Usually the task of grammars for natural and other languages is to analyze and give structure to a list of tokens (words), and generate a "semantic representation" which is essentially an alternative representation of what is expressed by the list of words (sentence). Such semantic representations are more suitable for processing by computer applications, like data bases or knowledge bases. However, it is also possible, as shown below, to construct semantic representations in such a way that mathematical computations are performed while parsing mathematical expressions. This in turn means that the kind of system discussed in this dissertation can effectively learn mathematical functions given CFGs which parse mathematical expressions appropriately.

The successor function has already been discussed in section 5.2. The grammar in the next example accepts expressions that are interpreted as additions of natural numbers, also using the Church numerals. The syntax is specified by the following CFG:

```
s    --> n.
s    --> n, op_s.
op_s --> [+], s.
n    --> [0].
n    --> [succ], n.
```

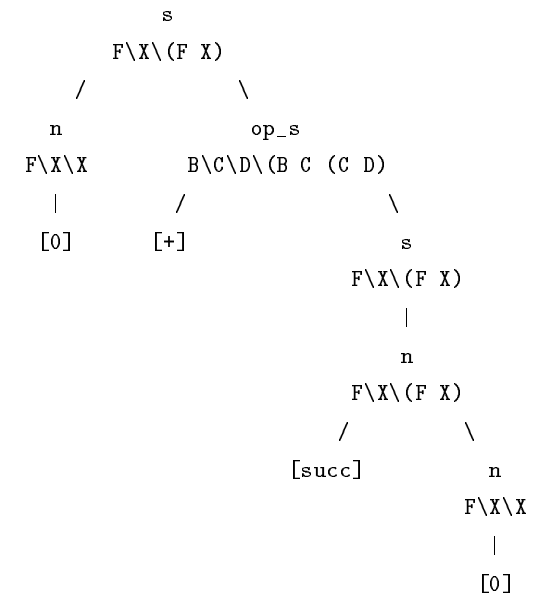
It is trained using these examples:

```
Sentence                 Semantic representation
----------------         -----------------------
[0]                      F\X\X
[succ,0]                 F\X\(F X)
[succ,succ,0]            F\X\(F (F X))
[0,+,succ,0]             F\X\(F X)
[succ,0,+,succ,succ,0]   F\X\(F (F (F X)))
[succ,succ,0,+,0]        F\X\(F (F X))
```

The higher-order DCG derived by the system is

```
s(A\A B) --> n(B).
s(A\B\(B A) C D) --> n(C),op_s(D).
op_s(A\B\C\D\(B C (A C D)) E) --> [+],s(E).
n(A\B\B) --> [0].
n(A\B\C\(A B (B C)) D) --> [succ],n(D).
```

where the term `A\B\C\D\(B C (A C D))` in the third rule essentially performs the additions, and `A\B\C\(A B (B C))` in the last rule increments numbers. For example, consider the parsing of the expression `[0 + succ 0]`:



In this parse tree the semantic representations computed for each rule are listed underneath the corresponding grammar symbols. Terminal symbols are denoted by `[...]`'s. The partially executed version of this DCG is:

```
s(A) --> n(A).
s(A) --> n(D),op_s(D\A).
op_s((A\B\C)\A\D\C) --> [+], s(A\D\B).
n(A\B\B) --> [0].
n(A\B\C) --> [succ], n(A\(A B)\C).
```

Using this DCG, a typical query would be:

```
| ?- s(L, [s,s,0,+,s,0,+,s,s,s,0,+,0], []).
```

where the semantic representation `L` would obtain the following binding:

```
L = A\B\(A (A (A (A (A (A B))))))
```

If the semantic representation is instantiated in a query, and a variable is used for the sentence argument, the system will compute all expressions whose value is equal to that semantic representation. For example, the query

```
| ?- s(A\B\(A (A (A (A B)))), S, []).
```

will produce the following instantiations for `S` (commas are omitted for easier reading):

```
S = [s s s s 0]
S = [s s s s 0 + 0]
S = [s s s 0 + s 0]
S = [s s 0 + s s 0]
S = [s 0 + s s s 0]
S = [0 + s s s s 0]
S = [s s s s 0 + 0 + 0]
S = [s s s 0 + s 0 + 0]
S = [s s 0 + s s 0 + 0]
S = [s 0 + s s s 0 + 0]
S = [0 + s s s s 0 + 0]
etc.
```

E.1.2. Modulo functions

Given a particular semantics (as implied by the training instances), the grammar rules specifying the syntax of a language must be structured appropriately, in order for a solution to exist. Essentially, the compositionality of the language must be expressed by the rule structure (see section 5.3. for further discussion). The following examples illustrate this point.

In the first example a modulo 2 function is computed, where the symbol `succ` representing the successor function is used for the input "sentence", zero is represented by the term `X\Y\X`, and one is represented by `X\Y\Y`. The syntax is again defined by this CFG:

```
s --> [0].
s --> [succ], s.
```

The training instances now are:

```
Sentence            Semantic representation
------------------- -----------------------
[0]                 X\Y\X
[succ,0]            X\Y\Y
[succ,succ,0]       X\Y\X
[succ,succ,succ,0]  X\Y\Y
```

Which leads to the following higher-order DCG:

```
s(A\B\A) --> [0].
s(A\B\C\(A C B) D) --> [succ],s(D).
```

And the partially executed version is:

```
s(A\B\A) --> [0].
s(A\B\C) --> [succ],s(B\A\C).
```

This DCG can now computed this modulo two function for all (infinitely many) sentences accepted by it. As can be easily seen in the partially executed version, this is done simply exchanging two prefix variables (`A` and `B`) on each iteration.

The next example also computes the modulo 2 function, but now using Church numerals. The two CFG rules defining the syntax in the previous example are now inappropriate to express the compositionality. Even though a consistent augmentation was found by the system, it does not appear very natural and is hard to understand:
CFG:

```
s --> [0].
s --> [succ], s.
```

Training instances:

Sentence	Semantic representation
[0]	F\X\X
[succ,0]	F\X\(F X)
[succ,succ,0]	F\X\X
[succ,succ,succ,0]	F\X\(F X)
[succ,succ,succ,succ,0]	F\X\X

Higher-order DCG:

```
s(A\B\B) --> [0].
s(A\B\C\(A D\C (B C)) E) --> [succ],s(E).
```

This DCG computes the correct semantics for all sentences because if the term returned by `E` in the second rule is `F\X\X`, it discards the argument `D\C` , which is used to discard the argument `(B C)` if the term returned by `E` is `F\X\(F X)`. The is quite a clever solution, but is unlikely to be generalizable to other modulo functions. Another interesting point about it is that it cannot be (easily) partially executed. The problem is that the application in `A\B\(A B)` sometimes is supposed to be reduced and sometimes not. The only way to solve this problem would be to require that all applications are partially executed if possible, including those occurring in the final representations (like in the Church numerals). See chapter 7. for further discussion of this problem.

The following CFG defines the language for a modulo 2 function in a more natural way, and allows to easily infer the desired semantics. Instead of just one base case as in the previous example, two base cases are now used:

```
s --> [0].
s --> [succ,0].
s --> [succ,succ], s.
```

The training instances are again:

```
Sentence                  Semantic representation
------------------        ----------------------
[0]                       F\X\X
[succ,0]                  F\X\(F X)
[succ,succ,0]             F\X\X
[succ,succ,succ,0]        F\X\(F X)
```

The following higher-order DCG is obtained:

```
s(A\B\B) --> [0].
s(A\B\(A B)) --> [succ],[0].
s(A\B\C\(A B C) D) --> [succ],[succ],s(D).
```

which can easily be partially executed to:

```
s(A\B\B) --> [0].
s(A\B\(A B)) --> [succ],[0].
s(A\B\C) --> [succ],[succ],s(A\B\C).
```

That is, the third rule simply copies the semantics returned by the subgoal on the right-hand side, which makes sense since every other number has the same semantic representation.

In the following example the modulo 3 function is inferred. The number zero is now represented as `X\Y\Z\X`, the number one as `X\Y\Z\Y`, and the number two as `X\Y\Z\Z`. Syntax is specified again by:

```
s    --> [0].
s    --> [succ], s.
```

The training instances are:

```
Sentence                  Semantic representation
------------------        ----------------------
[0]                       X\Y\Z\X
[succ,0]                  X\Y\Z\Y
[succ,succ,0]             X\Y\Z\Z
[succ,succ,succ,0]        X\Y\Z\X
```

The following higher-order DCG is derived:

```
s(A\B\C\A) --> [0].
s(A\B\C\D\(A C D B) E) --> [succ],s(E).
```

And partially executed:

```
s(A\B\C\A) --> [0].
s(A\B\C\D) --> [succ],s(B\C\A\D).
```

As in the corresponding modulo 2 example above, this DCG can compute the intended semantics for all sentences by simply rotating the prefix variables of the term returned by the right-hand side of the second rule. Obviously, this scheme can easily be generalized to all modulo n functions, where n is any natural number.

In the next example the system infers the modulo 3 function for numbers represented by Church numerals. The language is now defined by the following CFG which used three base cases:

```
s --> [0].
s --> [succ,0].
s --> [succ,succ,0].
s --> [succ,succ,succ], s.
```

Five training instances are needed to ensure correct augmentation of this grammar:

```
Sentence                      Semantic representation
----------------------        ----------------------
[0]                           F\X\X
[succ,0]                      F\X\(F X)
[succ,succ,0]                 F\X\(F (F X))
[succ,succ,succ,0]            F\X\X
[succ,succ,succ,succ,0]       F\X\(F X)
```

Using the above grammar and training instances the system derived this higher-order DCG:

```
s(A\B\B) --> [0].
s(A\B\(A B)) --> [succ],[0].
s(A\B\(A (A B))) --> [succ],[succ],[0].
s((A\B\C\(A B C) D)) --> [succ],[succ],[succ],s(D).
```

and this partially executed DCG:

```
s(A\B\B) --> [0].
s(A\B\(A B)) --> [s],[0].
s(A\B\(A (A B))) --> [s],[s],[0].
s(A\B\C) --> [s],[s],[s],s(A\B\C).
```

As in the corresponding modulo 2 case, only the last rule is actually partially executed. The applications occurring in the base cases are those that occur in the final semantic representations and therefore should not be reduced. Here are some sample executions of this DCG:

```
Input                 Output
---------------       --------------
s s s 0               A\B\B
s s s s 0             A\B\(A B)
s s s s s 0           A\B\(A (A B))
s s s s s s 0         A\B\B
s s s s s s s 0       A\B\(A B)
```

Computing and inferring modulo n functions using Church numerals in this way can be generalized to any natural number n in the obvious way by providing n base cases, and by providing one recursive rule which reduces the input number by n on each iteration and simply copies the corresponding semantic representation.

E.2. Learning to parenthesize arithmetic expressions

In the following example the system infers a DCG that parses simple arithmetic expressions and returns enhanced versions of those expressions by parenthesizing them according to operator precedences implicitly specified by the grammar defining the expression language. The language is defined by the following CFG:

```
s --> prod.
s --> prod, con_sum.
con_sum --> add_op, s.
prod --> number.
prod --> number, con_prod.
```

```
con_prod --> mult_op, prod.
add_op --> [+].
add_op --> [-].
mult_op --> [*].
mult_op --> [/].
number --> [1].
number --> [3].
number --> [5].
```

The training instances below are sufficient to ensure correct augmentation of the grammar:

```
Sentence          Semantic representation
--------          -----------------------
[1]               1
[3]               3
[1,+,5]           ('+' 1 5)
[1,-,5]           ('-' 1 5)
[1,*,5]           ('*' 1 5)
[1,/,5]           ('/' 1 5)
```

The higher-order DCG inferred by the system is:

```
s(A\A B) --> prod(B).
s(A\B\(B A) C D) --> prod(C),con_sum(D).
con_sum(A\B\C\(A C B) D E) --> add_op(D),s(E).
prod(A\A B) --> number(B).
prod(A\B\(B A) C D) --> number(C),con_prod(D).
con_prod(A\B\C\(A C B) D E) --> mult_op(D),prod(E).
add_op(A\B\(+ A B)) --> [+].
add_op(A\B\(- A B)) --> [-].
mult_op(A\B\(* A B)) --> [*].
mult_op(A\B\(/ A B)) --> [/].
number(1) --> [1].
number(3) --> [3].
number(5) --> [5].
```

And the partially executed version is:

```
s(A) --> prod(A).
s(A) --> prod(D),con_sum(D\A).
con_sum(A\B) --> add_op(A\E\B),s(E).
prod(A) --> number(A).
prod(A) --> number(D),con_prod(D\A).
```

```
con_prod(A\B) --> mult_op(A\E\B),prod(E).
add_op(A\B\(+ A B)) --> [+].
add_op(A\B\(- A B)) --> [-].
mult_op(A\B\(* A B)) --> [*].
mult_op(A\B\(/ A B)) --> [/].
number(1) --> [1].
number(3) --> [3].
number(5) --> [5].
```

The following test runs of this partially executed DCG demonstrate that it computes the correct semantics for all accepted sentences.

```
| ?- s(L,[1,+,3],[]).
                                    L = (+ 1 3)

| ?- s(L,[1,+,3,*,5],[]).
                                    L = (+ 1 (* 3 5))

| ?- s(L,[1,+,3,*,5,*,1],[]).
                                    L = (+ 1 (* 3 (* 5 1)))

| ?- s(L,[1,+,3,*,5,*,1,+,3],[]).
                                    L = (+ 1 (+ (* 3 (* 5 1)) 3))
```

E.3. Boolean functions

In this section I demonstrate the inference of the boolean **and** and **or** function. The true values can be represented in the following way:

true = `X\Y\X`
false = `X\Y\Y`

Given these representations, the functions **and** and **or** can be implemented like this:

and = `A\B\(A B X\Y\Y)`
or = `A\B\(A A B)`

In the *typed* λ-calculus the type of `true` would be of the form $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3$. Now, assume we want to perform the following computation:

(**and** **true** **true**) =
`(X\Y\(X Y M\N\N)  X\Y\X  X\Y\X)`

```
(Y\(A\B\A Y M\N\N)  X\Y\X)
(A\B\A C\D\C M\N\N)  X\Y\X
...
```

That is, `true` is applied to `true`: `(A\B\A  C\D\C)`. This implies that the type of the first argument of `true` must have the same type as `true` itself, which is impossible. Therefore the untyped λ-calculus must be used for such cases.

OR-function:

The following CFG defines the language of simple boolean expressions involving only the logical **or** and the boolean values **true** and **false**:

```
s --> a.
s --> a, b.
b --> [or], s.
a --> [true].
a --> [false].
```

The system is trained on the following examples:

```
Sentence              Semantic representation
----------------      -----------------------
[true]                A\B\A
[false]               A\B\B
[false,or,false]      A\B\B
[true,or,false]       A\B\A
[false,or,true]       A\B\A
```

And derives this higher-order DCG:

```
s(A\A B) --> a(B).
s(A\(A A) B C) --> a(B),b(C).
b(A\A B) --> [or],s(B).
a(A\B\A) --> [true].
a(A\B\B) --> [false].
```

The **or** function is implemented essentially by the term `A\(A A)` in the second rule. The system derived this higher-order DCG using the sequence of substitutions given below. First the grammar is augmented with function variables in the usual way:

```
s(F1 A) --> a(A).
s(F2 A B) --> a(A),b(B).
b(F3 A) --> [or],s(A).
a(F4) --> [true].
a(F5) --> [false].
```

Executing this augmented grammar for each training sentence leads to the following set of higher-order equations:

```
[[F1 F4,K\L\K],
 [F1 F5,K\L\L],
 [F2 F5 (F3 (F1 F5)),K\L\L],
 [F2 F4 (F3 (F1 F5)),K\L\K],
 [F2 F5 (F3 (F1 F4)),K\L\K]]
```

Below is the sequence of substitutions and the resulting equations leading to the solution:

```
   proj(1,1,0):  F1 <- K\K

[[F4,K\L\K],
 [F5,K\L\L],
 [F2 F5 (F3 F5),K\L\L],
 [F2 F4 (F3 F5),K\L\K],
 [F2 F5 (F3 F4),K\L\K]]

   proj(2,2,0):  F4 <- K\L\K

[[F5,K\L\L],
 [F2 F5 (F3 F5),K\L\L],
 [F2 K\L\K (F3 F5),K\L\K],
 [F2 F5 (F3 K\L\K),K\L\K]]

   proj(2,1,0):  F5 <- K\L\L

[[F2 K\L\L (F3 K\L\L),K\L\L],
 [F2 K\L\K (F3 K\L\L),K\L\K],
 [F2 K\L\L (F3 K\L\K),K\L\K]]

   proj(1,1,1):  F2 <- K\(K (H1 K))

[[F3 K\L\L,K\L\L],
 [H1 K\L\K,K\L\K],
 [F3 K\L\K,K\L\K]]

   proj(1,1,0):  F3 <- K\K

[[H1 K\L\K,K\L\K]]

   proj(1,1,0):  H1 <- K\K
```

```
[]

      F1 <- K\K
      F4 <- K\L\K
      F5 <- K\L\L
      F2 <- K\(K K)
      F3 <- K\K
      H1 <- K\K
```

The following sample executions of the inferred higher-order DCG demonstrate that it computes the correct result for all sentences of the language.

Input	Output
[true]	A\B\A
[true,or,true]	A\B\A
[false,or,true]	A\B\A
[false,or,false,or,false]	A\B\B
[true,or,false,or,false,or,true,or,false]	A\B\A

AND-function:

The corresponding CFG for the **and** function is:

```
s --> a.
s --> a, b.
b --> [and], s.
a --> [true].
a --> [false].
```

The following training instances are used:

Sentence	Semantic representation
[true]	A\B\A
[false]	A\B\B
[true,and,true]	A\B\A
[true,and,false]	A\B\B
[false,and,true]	A\B\B

The first solution the system came up with is given below. Since in general there are many (equivalent) solutions, it depends on the search strategy used by the higher-order unification procedure which one is found first. This solution appears to be a rather complex but is correct nevertheless as demonstrated by the sample runs further below.

```
s(A\A B) --> a(B).
s(A\(A (A B\B (C A)) (A (D A) E\A)) F G) --> a(F),b(G).
b(A\A B) --> [and],s(B).
a(A\B\A) --> [true].
a(A\B\B) --> [false].
```

This higher-order DCG was derived in the following way:

```
s(F1 A) --> a(A).
s(F2 A B) --> a(A),b(B).
b(F3 A) --> [and],s(A).
a(F4) --> [true].
a(F5) --> [false].

[[F1 F4,K\L\K],
 [F1 F5,K\L\L],
 [F2 F4 (F3 (F1 F4)),K\L\K],
 [F2 F4 (F3 (F1 F5)),K\L\L],
 [F2 F5 (F3 (F1 F4)),K\L\L]]

   proj(1,1,0):  F1 <- K\K

[[F4,K\L\K],
 [F5,K\L\L],
 [F2 F4 (F3 F4),K\L\K],
 [F2 F4 (F3 F5),K\L\L],
 [F2 F5 (F3 F4),K\L\L]]

   proj(2,2,0):  F4 <- K\L\K

[[F5,K\L\L],
 [F2 K\L\K (F3 K\L\K),K\L\K],
 [F2 K\L\K (F3 F5),K\L\L],
 [F2 F5 (F3 K\L\K),K\L\L]]

   proj(2,1,0):  F5 <- K\L\L

[[F2 K\L\K (F3 K\L\K),K\L\K],
 [F2 K\L\K (F3 K\L\L),K\L\L],
 [F2 K\L\L (F3 K\L\K),K\L\L]]

   proj(1,1,2):  F2 <- K\(K (H1 K) (H2 K))

[[H1 K\L\K (F3 K\L\K),K\L\K],
```

```
 [H1 K\L\K (F3 K\L\L),K\L\L],
 [H2 K\L\L (F3 K\L\K),K\L\L]]

   proj(1,1,2):  H1 <- K\(K (H3 K) (H5 K))

[[H3 K\L\K (F3 K\L\K),K\L\K],
 [H3 K\L\K (F3 K\L\L),K\L\L],
 [H2 K\L\L (F3 K\L\K),K\L\L]]

   proj(2,1,0):  H3 <- K\L\L

[[F3 K\L\K,K\L\K],
 [F3 K\L\L,K\L\L],
 [H2 K\L\L (F3 K\L\K),K\L\L]]

   proj(1,1,0):  F3 <- K\K

[[H2 K\L\L K\L\K,K\L\L]]

   proj(1,1,2):  H2 <- K\(K (H6 K) (H4 K))

[[H4 K\L\L K\L\K,K\L\L]]

   proj(2,2,0):  H4 <- K\L\K

[]

      F1 <- K\K
      F4 <- K\L\K
      F5 <- K\L\L
      F2 <- K\(K (K L\L (H5 K)) (K (H6 K) L\K))
      H1 <- K\(K L\L (H5 K))
      H3 <- K\L\L
      F3 <- K\K
      H2 <- K\(K (H6 K) L\K)
      H4 <- K\L\K
```

The above higher-order DCG computes the correct result for all sentences of the language as can be seen from the following sample executions:

```
Input                                                Output
---------------------------------------------------- ------
[true,and,true]                                      A\B\A
[true,and,true,and,false]                            A\B\B
```

`[true,and,true,and,true,and,true,and,true]`	`A\B\A`
`[true,and,true,and,true,and,false,and,true,and,true]`	`A\B\B`
`[false,and,false,and,true,and,false]`	`A\B\B`