We discuss the mechanical transformation of an unambiguous context-free grammar (CFG) into a definite-clause grammar (DCG) using a finite set of examples, each of which is a pair \((s, m)\), where \(s\) is a sentence belonging to the language defined by the CFG and \(m\) is the semantic representation (meaning) of \(s\). The resulting DCG would be such that it could be executed to compute the semantics for every sentence of the original DCG.

The motivation for our work comes from the observation that it is not easy to manually augment a CFG with semantic attributes to obtain a DCG because the task of building a correct and efficient DCG requires a fair amount of search, especially when the semantic representations involve quantified terms, as in natural languages. Our proposed approach is based upon two key assumptions: (1) the semantic representation language is the simply typed \(\lambda\)-calculus, and (2) the semantic representation of a sentence is a function (expressed in the typed \(\lambda\)-calculus) of the semantic representations of its parts (compositionality). With these assumptions, we show how a higher-order DCG can be systematically constructed using a matching procedure for simply typed \(\lambda\)-terms. We then show how to translate the constructed higher-order DCG into a first-order DCG by a partial-execution procedure. We have applied our methodology to the synthesis of the semantics of a small query language, and we believe that this methodology could be a useful tool for generating natural query language front-ends for various applications. © Elsevier Science Inc., 1997
1. INTRODUCTION

This paper is concerned with the problem of inferring semantics of a language from examples, assuming that we are already given its syntax. More precisely, we 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. Our goal is to develop a system that will take as input an unambiguous context-free grammar (CFG) and a finite set of pairs \((s, m)\), where \(s\) is a sentence belonging to the language defined by the CFG and \(m\) is the semantic representation (meaning) of \(s\), and will produce as output a definite clause grammar (DCG) [18] capable of computing the semantic representations for all sentences of the CFG. We will clarify further below the precise sense in which this problem can be solved and the research issues it raises, but first we briefly 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?

The motivation for our work comes from the observation that it is not easy to manually augment a CFG with semantic attributes 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 this paper, it is not obvious what the semantic attributes 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 is natural to seek a mechanical procedure that will compute the semantics of all sentences of a given CFG on the basis of a representative set of sentence-meaning pairs. Our proposed methodology could facilitate rapid prototyping of natural language interfaces for database systems or customizing such interfaces for specific applications [21] since the interface could be obtained merely by defining the grammar and typical sentence-meaning pairs. In general, the conversion of the natural language query into this representation and the conversion from this representation back into natural language could be handled by the generated interface; the latter operation could be achieved by applying the definite-clause grammar in the reverse direction to the semantic representations.

To appreciate our proposed approach, we first note that 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: (1) the semantic representation language is the simply typed \(\lambda\)-calculus [2], (2) 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 Montague for treating quantification in English [5]. To illustrate, consider the following CFG rule,

\[
\text{sentence} \rightarrow \text{nounphrase, verbphrase}
\]

which specifies that a sentence consists of a noun phrase followed by a verb phrase (\textit{sentence, nounphrase, and verbphrase} are nonterminals). A key idea of our
approach is to express the compositionality principle to enhance the rule as follows:

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

where uppercase letters are variables. That is, if variables $X$ and $Y$ represent, respectively, the meanings of the nonterminals $\text{nounphrase}$ and $\text{verbphrase}$, then the meaning of nonterminal $\text{sentence}$ is obtained by applying some function $F$ to $X$ and $Y$.\footnote{This expression of the compositionality principle effectively means that we are permitting only “synthesized” attributes, as opposed to “inherited” attributes, in the terminology of attribute grammars. It is possible to extend our approach to other forms of compositionality, but this issue is beyond the scope of this paper.} The function variable $F$ is a term in the simply typed $\lambda$-calculus, and must be determined by the system based upon the finite set of input examples. Each grammar rule thus introduces one new function variable. Briefly, our technique is to generate a finite set of sentences of the CFG (the selection strategy will be discussed later), obtain from the user the semantic representation of each of the generated sentences, and formulate a set of equations, where each equation relates a user-supplied semantic representation with a term composed of the function variables. The solutions for all function variables serve to augment the original CFG in order to derive the final DCG. For example, if the solution for $F$ was $\lambda a.\lambda b.(a \ b)$, then the grammar rule would become

$$\text{sentence}( (\lambda a.\lambda b.(a \ b) \ X \ Y) ) \rightarrow \text{nounphrase}(X), \text{verbphrase}(Y)$$

which is equivalent to $\text{sentence}( (X \ Y) ) \rightarrow \text{nounphrase}(X), \text{verbphrase}(Y)$.

The typed $\lambda$-calculus is particularly suitable for analyzing and synthesizing semantic representations. It effectively allows us to reduce the generalization problem to a matching problem over simply typed terms. This matching problem is called higher-order matching because variables may range over functions. To solve this problem, we adapt the unification procedure for simply typed $\lambda$-terms \cite{11}. Because this procedure is only a partial decision procedure, our 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. We will see that using the simply typed $\lambda$-calculus as the semantic representation language drastically reduces the search space of allowable solutions.

While higher-order logic is useful for reasoning about and synthesizing programs, it is not as amenable to efficient execution as first-order logic. 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 efficient reverse execution than a higher-order DCG. We have developed a partial execution technique that effectively replaces $\lambda$-terms by first-order terms, and therefore replaces higher-order unification by (the more efficient) first-order unification. Such a scheme is possible because the mechanically generated higher-order DCGs have a very simple and uniform structure. The use of first-order unification to simulate certain cases of $\beta$-reduction was first introduced by Colmerauer \cite{3}, 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 et al. \cite{17}. We 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.

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 sample sentences can be used to determine which application terms should be partially executed; and the semantic representation for a variable may have to be copied if the variable occurs more than once in a rule. Even though one can construct pathological grammars and semantic representations where this scheme of partial execution fails, it appears to be applicable for the most common cases, and we have shown its correctness in those cases. Higher-order DCGs for which we could not find a satisfactory solution are those where a particular application term is reduced for some sentences, but not for others. It appears, however, that such grammars can be rewritten into a more natural form that avoids this problem.

Finally, we delimit the scope of this paper. We would like to first note that natural languages are of interest in our work since they are good examples of languages whose semantics require the use of quantified terms, and hence the full use of the typed λ-calculus. However, our work is not directly concerned with devising suitable semantics for natural language sentences; it 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 whose meanings have no ambiguity. However, our proposed techniques also work for certain forms of context-sensitive grammars, and they can be extended to certain forms of ambiguous grammars. There are several other issues that we do not explore in this paper: exploring the effect of different sentences and their order on the efficiency of synthesis, exploring other forms of the compositionality principle, and showing the effectiveness of our methodology for the intended applications. We refer the reader to [7] for a discussion of some of these issues.

The remainder of this paper is organized as follows. Section 2 describes in detail our approach to synthesizing a higher-order DCG from examples; Section 3 illustrates this technique with two examples; Section 4 describes our procedure for partial execution and its correctness; and Section 5 presents the current status of the work and related work. Henceforth, we assume the reader has some familiarity with DCGs [1, 18] and unification of typed λ-terms [11].

2. FROM A CFG TO A HIGHER-ORDER DCG USING EXAMPLES

We begin with a brief review of the simply typed λ-calculus. 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 $\rightarrow$: $\alpha, \beta \in T \Rightarrow (\alpha \rightarrow \beta) \in T$. There are four kinds of terms in the simply typed λ-calculus: variables, constants, abstractions, and applications. Variables and constants are also referred to as atoms. Every term in the simply typed λ-calculus has a type: constants and binder variables of abstractions must be explicitly assigned a type. We will refer to the type of a term $t$ by $\tau(t)$. An abstraction $\lambda v.e$ has type $t_1 \rightarrow t_2$ if $\tau(v) = t_1$.
and \( \tau(e) = t_2 \). An application term \((e_1 \ e_2)\) has type \( t_2 \) if \( \tau(e_1) = t_1 \rightarrow t_2 \) and \( \tau(e_2) = t_1 \).

We assume familiarity with standard \( \lambda \)-calculus terminology: bound and free occurrences of variables, substitutions, conversion rules (\( \alpha \), \( \beta \), and \( \eta \)), and normal forms. We will say that a term is closed if it has no free occurrences of variables. As is customary, we will omit parentheses and use the shorthand \((f \ e_1 \ldots e_k)\) for \((\ldots (f \ e_1) \ldots e_k)\).

Some additional terminology is needed to discuss the matching of typed \( \lambda \)-terms: we will represent all terms in long-normal form, i.e., \( \lambda x_1 \ldots \lambda x_n \) \((@e_1 \ldots e_p)\), where \( n \geq 0 \), \( p \geq 0 \), and \( @ \) is a constant or a variable of type \( \alpha_1 \rightarrow \cdots \rightarrow \alpha_p \rightarrow \beta \). We will refer to \( @ \) as the head of \( e \). We will say that \( e \) is rigid if \( @ \) is a constant or is a member of \( \{x_1, \ldots, x_n\} \); otherwise, we will say that \( e \) is flexible.

Unlike the untyped \( \lambda \)-calculus, the typed \( \lambda \)-calculus has the strong normalization property, i.e., every reduction sequence from every term is finite (terminates in a normal form).

### 2.1. Basic Technique

We now describe the synthesis technique in terms of four procedures: \textit{SYNTH}, \textit{SOLVE}, \textit{SUBST}, and \textit{DECOMP}. \textit{SYNTH} is the top-level procedure; it collects the sample sentence-meaning pairs, and passes on to \textit{SOLVE} a resulting set \( E \) of higher-order equations. \textit{SOLVE} selects an equation from \( E \) and passes it on to \textit{SUBST}, which determines a solution, \( \sigma \), for the equation if one exists. \textit{SOLVE} then passes on the instantiated equation-set \( E \sigma \) to \textit{DECOMP}, which simplifies the equations where possible. \textit{SOLVE} repeats these two steps until the equation-set becomes empty.

For simplicity of presentation, we assume that a CFG rule has either a single terminal on its right-hand side (r.h.s.) or a sequence of one or more nonterminals; in practice, we permit both terminals and nonterminals on the r.h.s. As in Prolog DCGs, nonterminals are identifiers beginning with a lowercase letter, and terminals are such identifiers surrounded by \( \texttt{[and]} \). A higher-order DCG [13] is similar in structure to a first-order DCG, except that typed \( \lambda \)-terms take the place of first-order terms. It can be converted into a higher-order Horn clause program [16] in a manner similar to the first-order case: by adding two extra arguments to each nonterminal symbol, for the input list and remainder list, respectively.

#### Procedure \textit{SYNTH}(G).

1. Let \( G \) be an unambiguous CFG having \( n \) rules, with start symbol \( \texttt{start} \), and let \( \mathcal{L}(G) \) be the language generated by \( G \). The type of the resulting term of each nonterminal must also be supplied.
2. Construct the higher-order DCG as follows:
   - If the \( i \)-th CFG rule is \( a_i \rightarrow b_{i1} \ldots b_{ik_i} \), the \( i \)-th DCG rule will be
     \[
     a_i((F_i V_1 \ldots V_{k_i})) \rightarrow b_{i1}(V_1), \ldots, b_{ik_i}(V_{k_i}),
     \]
     where each \( F_i \) is existentially quantified over the entire DCG, and each \( V_i \) is universally quantified over the given rule.
   - If the \( i \)-th CFG rule is \( a_i \rightarrow [t] \), the \( i \)-th DCG rule will be
     \[
     a_i(F_i) \rightarrow [t].
     \]


3. Generate a finite set of sentences \( S = \{ s_i : s_i \in L(G) \land 1 \leq i \leq k \} \), for some \( k \) (see discussion in next subsection for selection strategy).

4. Determine the values for the function variables \( F_i \) in the above DCG as follows.

\[
E \leftarrow \phi;
\]

\( \text{for } i = 1 \ldots k \) do

a. Query the user for the semantic representation \( n_i \) of each \( s_i \in S \). We assume \( n_i \) to be a closed, simply typed \( \lambda \)-term.

b. Execute the goal \( \text{start}(M, s_i, []) \) using the constructed DCG of step 2, i.e., using the higher-order Horn clause program corresponding to the higher-order DCG of step 2. Let \( m_i \) be the computed term for \( M \).

c. \( E \leftarrow E \cup \{(m_i, n_i)\} \)

end for

5. Call \( \text{SOLVE}(E) \) to solve for the function variables \( F_i \). In general, \( E \) may have zero or more maximally general solutions. \( \text{SOLVE} \) returns these solutions one at a time, and each solution is used to instantiate the DCG of step 2, and the resulting DCG is printed out.

Procedure \( \text{SOLVE}(E_0) \). Procedure \( \text{SOLVE} \) tries to solve the set of higher-order equations \( E_0 \) by attempting to find substitutions for the function variables occurring in it.

1. Let \( E \leftarrow E_0 \), and \( \sigma \leftarrow \emptyset \), the empty substitution.

2. while \( E \neq \emptyset \) do

a. Select an equation \( eqn \) from \( E \), and call \( \text{SUBST}(eqn) \). If \( \text{SUBST} \) succeeds, it returns a substitution term \( t \) for the variable \( v \) at the head of the left-hand side of \( eqn \).

b. \( \sigma \leftarrow \sigma \{(v, t)\} \) (composition of substitutions)

c. \( E \leftarrow \text{DECOMP}(E \sigma) \) (see below)

end while

3. Return \( \sigma \upharpoonright F \) (the restriction of \( \sigma \) to \( F \), the set of function variables appearing in \( E_0 \)).

Procedure \( \text{SUBST}(eqn) \). Procedure \( \text{SUBST} \) determines a substitution for the head of the left-hand side term of \( eqn \) as follows. Let \( eqn = (e_1, e_2) \), where, in general, \( e_1 \) is a flexible term, \( e_2 \) is a closed rigid term, and \( \tau(e_1) = \tau(e_2) \). If \( e_1 \) is a variable, say \( v \), return the substitution \( v \leftarrow e_2 \). Otherwise, \( e_1 \) and \( e_2 \) are of the form

\[
e_1 = \lambda u_1, \ldots, \lambda u_m. (f \ c_1 \ldots c_p)
e_2 = \lambda v_1, \ldots, \lambda v_m. (@ \ d_1 \ldots d_q)
\]

where \( f \) is a function variable and \( \tau(f) = \alpha_1 \rightarrow \cdots \rightarrow \alpha_p \rightarrow \beta \), but \( @ \) is a constant or a binder variable and \( \tau(@) = \delta_1 \rightarrow \cdots \rightarrow \delta_q \rightarrow \beta \). Nondeterministically select and return one of the following substitutions:

a. Projection substitutions:

\[
f \leftarrow \lambda w_1 \ldots \lambda w_p.(w_1 (h_1 \ldots w_p) \ldots (h_l w_1 \ldots w_p)) \quad (1 \leq i \leq p),
\]

where \( \tau(w_i) = \tau(e_i) = \alpha_i = \gamma_1 \rightarrow \cdots \rightarrow \gamma_l \rightarrow \beta \), for \( 1 \leq i \leq p \), and each \( h_j \), for \( 1 \leq j \leq l \), is a new function variable of type \( \alpha_1 \rightarrow \cdots \rightarrow \alpha_p \rightarrow \gamma_j \).
b. *Imitation* substitution (applicable only if @ is a constant):

\[ f \leftarrow \lambda w_1 \ldots \lambda w_p. (\@ \leq \text{(} h_1 \leq \text{w}_1 \ldots \text{w}_p \ldots \text{h}_q \text{w}_1 \ldots \text{w}_p\text{)}), \]

where each \( h_j \), for \( 1 \leq j \leq q \), is a new function variable of type \( \alpha_1 \rightarrow \ldots \rightarrow \alpha_p \rightarrow \delta_j \).

**Function **DECOMP\( (E) **.** The input set of equations \( E = \{ \langle t_i, u_i \rangle \mid 1 \leq i \leq n \} \), for some \( n \), where each term in the set \( \{ u_i \mid 1 \leq i \leq n \} \) is rigid. In the following definition of **DECOMP**, which is adapted from [15], the notation \( x \) stands for a sequence of binder variables.

1. If \( E = \emptyset \) then return \( \emptyset \).
2. If \( E = \{ \langle t_1, u_1 \rangle \} \) then
   - if \( t_1 \) is flexible then return \( E \)
   - else (\( t_1 \) and \( u_1 \) are both rigid) let \( t_1 = \lambda x. (\@_1 \leq l_1 \ldots l_m) \) and \( u_1 = \lambda x. (\@_2 \leq r_1 \ldots r_m) \). If \( \@_1 \neq \@_2 \) then fail else return \( \text{DECOMP}(\{ \langle \lambda x.l_i, \lambda x.r_i \rangle \mid 1 \leq i \leq m \}) \).
3. Else let \( E = \{ \langle t_i, u_i \rangle \mid 1 \leq i \leq n \} \), for some \( n \geq 2 \):
   - if \( \text{DECOMP}(\{ \langle t_i, u_i \rangle \}) \) fails for some \( i \), then \( \text{DECOMP}(E) \) fails
   - else return \( \bigcup_{i=1}^{n} \text{DECOMP}(\{ \langle t_i, u_i \rangle \}) \).

**2.2. Discussion**

We discuss below the main features of the synthesis procedures; we continue this discussion in Section 3.3 after presenting examples.

1. **Compositionality.** The compositionality principle is expressed in step 2 of procedure **SYNTH** by assuming that, in a CFG rule \( a \rightarrow b_1 \ldots b_k \), the meaning of the nonterminal \( a \) is some function \( F \) of the meanings of the nonterminals \( b_1 \ldots b_k \), where \( F \) is some term in the simply typed \( \lambda \)-calculus.\(^3\) It has been generally recognized that compositionality plays an important role in language semantics; 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 [20] and later by Zadrozny [22], if there are no restrictions imposed on the kinds of functions being used for computing the meaning of an expression from the meanings of its parts, such functions always exist no matter what the meanings of the whole expression and its parts are. Meaningful restrictions would be, for example, allowing only polynomial functions of a certain degree, or functions that can be expressed in the typed \( \lambda \)-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).

\(^3\)When terminal symbols are present along with one or more nonterminals on the r.h.s. of a rule, our methodology assumes that the meaning is independent of these terminal symbols; if the semantics of any such terminal \([t]\) is to be taken into account, it should be replaced by a new nonterminal \( n \), and a new rule \( n \rightarrow [t] \) added to the CFG.
2. Higher-Order Matching. The procedures SUBST and DECOMP are adaptations of the unification procedure for the typed-\(\lambda\) calculus with \(\eta\) equality [11]. In our case, all terms on the right-hand sides of equations will be closed, and hence all we need is a matching procedure. The important special case when an equation is of the form \(v = t\), where \(v\) is a variable, can be solved with the substitution \((v, t)\) without a rigid-path check [11]. When it is known that terms are of second-order type, the matching procedure will terminate [12]; however, it is common to have terms of higher type, as our examples in the next section illustrate. While the decidability of higher-order matching is an open problem—third-order matching was shown to be decidable recently [14]—a decision procedure alone is not sufficient in our context since we are interested in enumerating substitutions. Since the matching procedure (even for third-order types) could give rise to infinitely many maximally general matching substitutions, the matching procedure does not appear to be substantially less complex than Huet's unification procedure [11].

3. Multiple Solutions and Termination. Unlike first-order matching, the matching of simply typed \(\lambda\)-terms can yield more than one solution. However, these solutions do not necessarily result in DCGs that implement different sentence-meaning functions (see example in Section 3.3). 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. There are three possible outcomes from invoking SOLVE: success, failure, and nontermination. By the completeness of Huet's procedure [11], every solution to the set of equations can be found by SOLVE. (In our implementation, the search space is explored using depth-first iterative deepening.) Conversely, in case SOLVE fails, there is no higher-order DCG satisfying the given examples. Note that a matching procedure based upon Huet's substitution rules [11] may sometimes proceed indefinitely when there is no solution to the equations. This is the only way by which nontermination can occur in this system.

4. Sample Sentences. It is desirable to use as few examples as are necessary to guarantee a unique solution. Haas [7] presents 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. First, we assume that the CFG does not have any redundant nonterminal, i.e., one defined by single production rule of the form \(N \rightarrow M\), where \(M\) is a nonterminal. Such nonterminals are first eliminated through a preprocessing step. Our basic technique is to change one word of a sentence at a time, so that it can be uniquely determined which words contribute which subterms of the semantic representation. This rule, however, need not be strictly followed. Since a nonterminal often appears on the r.h.s. of multiple grammar rules, it is sufficient to exercise the nonterminal in the simplest possible surrounding context. Our experience with this methodology for simple natural-language grammars indicates that the number of examples needed to get the intended solution tends to be roughly the same as the number of production rules of the grammar. Further performance improvements can be achieved...
by presenting shorter sample sentences before longer ones. The equations corresponding to shorter sample sentences are easier to solve, and the constraints introduced by them reduce the search for substitutions of subsequent equations.

3. TWO EXAMPLES OF DCG SYNTHESIS

In showing the derivation of the higher-order DCG, we will follow the convention of λProlog [15] and write λXE for λX.E. This notation is, in fact, used by our implementation.

3.1. The Successor Function

We deliberately choose our first example to be a very simple one; its sole purpose is to illustrate the steps of the synthesis procedures.

The input CFG is as shown below, where the type of the term returned by nonterminal s is \((i \rightarrow i) \rightarrow i \rightarrow i\):

\[
s \rightarrow [a]. \\
\]
\[
s \rightarrow [a], s. \\
\]

The skeletal DCG obtained from step 2 of SYNTH is as follows (as noted earlier, when terminal symbols are present on the r.h.s. of a rule, we assume the meaning is independent of these symbols):

\[
s(F1) \rightarrow [a]. \\
\]
\[
s(F2 \ A) \rightarrow [a], s(A). \\
\]

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 to encode these numbers: \(0 = F\backslash X\backslash X, 1 = F\backslash X\backslash (F \ X), 2 = F\backslash X\backslash (F (F \ X)), \) etc. In this example, the type for all Church numerals is \((i \rightarrow i) \rightarrow i \rightarrow i\), where \(i\) is a primitive type. We will see that the desired DCG can be obtained with just three examples: \([a], [a, a], [a, a, a]\).

The user-supplied semantic representations are the Church numerals for 0, 1, and 2, respectively.

In step 4 of SYNTH, executing the skeletal DCG on the sentence \([a]\), the constructed semantic representation will be \(F1\).

The equation

\[F1 = F\backslash X\backslash X\]

is added to \(E\), where the type of \(F1\) is \((i \rightarrow i) \rightarrow i \rightarrow i\). Similarly, during subsequent iterations of the for-loop is step 4 of SYNTH, the following equations are added to \(E\):

\[F2 F1 = F\backslash X\backslash (F \ X)\]
\[(F2 \ (F2 \ F1)) = F\backslash X\backslash (F (F \ X))\]

where the type of \(F2\) is \(((i \rightarrow i) \rightarrow i \rightarrow i) \rightarrow ((i \rightarrow i) \rightarrow i \rightarrow i)\).
The procedure \textit{SOLVE} is called next. Given the set of equation \( E \), a direct assignment solves the first equation:

\[
F_1 \leftarrow K\backslash L\backslash L.
\]

The remaining equations to be solved are

\[
\{(F_2 K\backslash L\backslash L) = F\backslash X\backslash (F \ X), \quad (F_2 (F_2 K\backslash L\backslash L)) = F\backslash X\backslash (F \ (F \ X))\}.
\]

The following projection substitution is next attempted for \( F_2 \):

\[
F_2 \leftarrow K\backslash L\backslash M\backslash (K \ (H_2 K \ L \ M) \ (H_1 K \ L \ M))
\]

where \( H_1 \) and \( H_2 \) are of appropriate types. This yields the following reduced equation set:

\[
\{A\backslash B\backslash (H_1 K\backslash L\backslash L \ A \ B) = F\backslash X\backslash (F \ X), \\
A\backslash B\backslash (H_1 K\backslash L\backslash L \ (H_2 K\backslash L\backslash (H_1 M\backslash N \ K \ L) \ A \ B) \ (H_1 K\backslash L\backslash (H_1 M\backslash N \ K \ L) \ A \ B)) = F\backslash X\backslash (F \ (F \ X))\}.
\]

Next, the variable \( H_1 \) is solved for by the projection substitution \( H_1 \leftarrow K\backslash L\backslash L \), so that the only remaining equation is

\[
\{A\backslash B\backslash (H_2 K\backslash L\backslash (K \ L) \ A \ B \ (A \ B)) = F\backslash X\backslash (F \ (F \ X))\}.
\]

The substitution for \( H_2 \) now is \( H_2 \leftarrow K\backslash L\backslash M\backslash N\backslash (L \ (H_3 K \ L \ M \ N)) \), which yields the equation

\[
\{A\backslash B\backslash (H_3 K\backslash L\backslash (K \ L) \ A \ B \ (A \ B)) = F\backslash X\backslash (F \ X)\}.
\]

The substitution \( H_3 \leftarrow K\backslash L\backslash M\backslash N \) solves this equation. After substituting and reducing all terms, the final set of substitutions is

\begin{align*}
F_1 & \leftarrow K\backslash L\backslash L \\
F_2 & \leftarrow K\backslash L\backslash M\backslash (K \ L \ (L \ M)) \\
H_1 & \leftarrow K\backslash L\backslash L \\
H_2 & \leftarrow K\backslash L\backslash M\backslash N\backslash (L \ N) \\
H_3 & \leftarrow K\backslash L\backslash M\backslash N.
\end{align*}

Procedure \textit{SOLVE} returns the substitutions for \( F_1 \) and \( F_2 \) to procedure \textit{SYNTH}. The resulting higher-order DCG produced by step 5 of \textit{SYNTH} is

\begin{align*}
\text{s}(\text{A}\backslash \text{B}\backslash \text{B}) & \rightarrow [\text{a}], \\
\text{s}(\text{A}\backslash \text{B}\backslash \text{C}\backslash (\text{A} \ \text{B} \ (\text{B} \ \text{C})) \ \text{D}) & \rightarrow [\text{a}], \ \text{s}(\text{D}).
\end{align*}

where the term \( \text{A}\backslash \text{B}\backslash \text{C}\backslash (\text{A} \ \text{B} \ (\text{B} \ \text{C})) \) in the second rule essentially performs the successor operation. For example, in order to parse the sentence \([\text{a}, \text{a}]\), the second rule is invoked first, which then calls the first rule instantiating \( \text{D} \) to \( \text{A}\backslash \text{B}\backslash \text{B} \). The derivation of the result is shown below:

\[
\begin{align*}
(\text{A}\backslash \text{B}\backslash \text{C}\backslash (\text{A} \ \text{B} \ (\text{B} \ \text{C})) \ \text{A}\backslash \text{B}\backslash \text{B}) &= \text{B}\backslash \text{C}\backslash (\text{D}\backslash \text{E}\backslash \text{B} \ (\text{B} \ \text{C})) \\
&= \text{B}\backslash \text{C}\backslash (\text{B} \ \text{C}).
\end{align*}
\]

This example shows how the semantics for an infinite language can be inferred from just a few examples. In the above case, only three input–output pairs are needed to

obtain a unique answer. This example has the flavor of a number-series problem (i.e.,
guessing a number series from a finite portion of it), but the constraints imposed
by the grammar and the typed λ-calculus are very strong and limit which series
can be inferred from examples.

3.2. Simple Natural Language Grammar

Our next example is a more realistic use of our proposed methodology; it also illus-
trates additional aspects of the synthesis technique—the use of imitation substitu-
tions and how the constraints from multiple examples help prune the search space
by eliminating unproductive substitutions quickly. Below on the left is the input
CFG, and on the right, the DCG generated after step 2.

\[
\begin{align*}
    s &\rightarrow np, iv. \\
    np &\rightarrow det, n. \\
    det &\rightarrow [a]. \\
    det &\rightarrow [every]. \\
    n &\rightarrow [program]. \\
    n &\rightarrow [computer]. \\
    iv &\rightarrow [runs]. \\
    iv &\rightarrow [halts]. \\
    s((F_1 V W)) &\rightarrow np(V), iv(W). \\
    np((F_2 V W)) &\rightarrow det(V), n(W). \\
    det(F_3) &\rightarrow [a]. \\
    det(F_4) &\rightarrow [every]. \\
    n(F_5) &\rightarrow [program]. \\
    n(F_6) &\rightarrow [computer]. \\
    iv(F_7) &\rightarrow [runs]. \\
    iv(F_8) &\rightarrow [halts]. \\
\end{align*}
\]

Assuming \(o\) is the type of propositions and \(i\) is the type of individuals, the types
for the various nonterminals are as follows: \(\tau(s) = o\), \(\tau(np) = (i \rightarrow o) \rightarrow o\),
\(\tau(det) = (i \rightarrow o) \rightarrow (i \rightarrow o) \rightarrow o\), \(\tau(n) = i \rightarrow o\), and \(\tau(iv) = i \rightarrow o\). Suppose that
the sentences and their user-supplied semantic representations in step 3a of SYNTH
are as follows (constants \(\text{exists, all, and, implies, prog, comp, run, halt}\)
are assumed to be given suitable types):

\[
\begin{align*}
    [a,program,runs] &\rightarrow (exists X\langle\text{and} (prog X) (run X)\rangle) \\
    [every,program,runs] &\rightarrow (all X\langle\text{implies} (prog X) (run X)\rangle) \\
    [a,computer,runs] &\rightarrow (exists X\langle\text{and} (comp X) (run X)\rangle) \\
    [a,program,halts] &\rightarrow (exists X\langle\text{and} (prog X) (halt X)\rangle). \\
\end{align*}
\]

The equation-set \(E\) after step 4 of SYNTH would be

\[
\begin{align*}
    (F_1 (F_2 F_3 F_5) F_7) &\rightarrow (exists X\langle\text{and} (prog X) (run X)\rangle) \\
    (F_1 (F_2 F_4 F_5) F_7) &\rightarrow (all X\langle\text{and} (prog X) (run X)\rangle) \\
    (F_1 (F_2 F_3 F_6) F_7) &\rightarrow (exists X\langle\text{and} (comp X) (run X)\rangle) \\
    (F_1 (F_2 F_3 F_5) F_8) &\rightarrow (exists X\langle\text{and} (prog X) (halt X)\rangle). \\
\end{align*}
\]

\(SOLVE\) obtains first an imitation substitution from \(SUBST\) for the function variable
\(F_1\), as follows: \(F_1 \leftarrow X\langle\exists (\text{G X Y})\rangle\). However, this choice is immediately
eliminated by \(DECOMP\) when \(F_1\) is substituted for in the second equation. Hence, \(SOLVE\)
obtains the following projection substitution:

\[
F_1 \leftarrow K\langle (H_1 K L) \rangle
\]

where \(H_1\) is a function variable of type \((i \rightarrow o) \rightarrow (i \rightarrow o) \rightarrow (i \rightarrow o)\). Under
this substitution for \(F_1\), the above equations reduce to the following set:

\[
\begin{align*}
    ((F_2 F_3 F_5 (H_1 (F_2 F_3 F_5) F_7)) &\rightarrow (exists K\langle\text{and} (prog K) (run K)\rangle), \\
    (F_2 F_4 F_5 (H_1 (F_2 F_4 F_5) F_7)) &\rightarrow (all K\langle\text{implies} (prog K) (run K)\rangle), \\
    (F_2 F_3 F_6 (H_1 (F_2 F_3 F_6) F_7)) &\rightarrow (exists K\langle\text{and} (comp K) (run K)\rangle), \\
    (F_2 F_3 F_5 (H_1 (F_2 F_3 F_5) F_8)) &\rightarrow (exists K\langle\text{and} (prog K) (halt K)\rangle). \\
\end{align*}
\]
Once again, an imitation substitution can be seen to fail, and a projection substitution must be used. The derivation is continued in this manner; the complete set of variable bindings, including those for the auxiliary function variables introduced during the derivation, is as follows:

\[
\begin{align*}
F_2 & \leftarrow K \backslash L \backslash M (K (H_2 K L M) (H_3 K L M)) \\
F_3 & \leftarrow K \backslash L \backslash (exists (H_4 K L)) \\
H_4 & \leftarrow K \backslash L \backslash M (and (H_6 K L M) (H_5 K L M)) \\
H_5 & \leftarrow K \backslash L \backslash M (K (H_7 K L M)) \\
H_2 & \leftarrow K \backslash L \backslash M \\
H_1 & \leftarrow K \backslash L \backslash L \\
F_7 & \leftarrow K \backslash (run (H_8 K)) \\
H_8 & \leftarrow K \backslash K \\
H_7 & \leftarrow K \backslash L \backslash M \\
H_6 & \leftarrow K \backslash L \backslash M (L (H_9 K L M)) \\
H_3 & \leftarrow K \backslash L \backslash M \backslash L \\
F_5 & \leftarrow K \backslash (prog (H_{10} K)) \\
H_{10} & \leftarrow K \backslash K \\
H_9 & \leftarrow K \backslash L \backslash M \\
F_4 & \leftarrow K \backslash L \backslash (all (H_{11} K L)) \\
H_{11} & \leftarrow K \backslash L \backslash M (implies (H_{13} K L M) (H_{12} K L M)) \\
H_{12} & \leftarrow K \backslash L \backslash M (K (H_{14} K L M)) \\
H_{14} & \leftarrow K \backslash L \backslash M \\
H_{13} & \leftarrow K \backslash L \backslash M (L (H_{15} K L M)) \\
H_{15} & \leftarrow K \backslash L \backslash M \\
F_6 & \leftarrow K \backslash (comp (H_{16} K)) \\
H_{16} & \leftarrow K \backslash K \\
F_8 & \leftarrow K \backslash (halt (H_{17} K)) \\
H_{17} & \leftarrow K \backslash K.
\end{align*}
\]

Thus the constructed higher-order DCG would be

\[
\begin{align*}
s((A \ B)) & \rightarrow np(A), iv(B). \\
np(A \backslash (B \ A \ C)) & \rightarrow det(B), n(C). \\
det(A \backslash B \backslash (exists C \backslash (and (B \ C) (A \ C)))) & \rightarrow [a]. \\
det(A \backslash B \backslash (all C \backslash (implies (B \ C) (A \ C)))) & \rightarrow [every]. \\
n(A \backslash (prog \ A)) & \rightarrow [program]. \\
n(A \backslash (comp \ A)) & \rightarrow [computer]. \\
iv(A \backslash (run \ A)) & \rightarrow [runs]. \\
iv(A \backslash (halt \ A)) & \rightarrow [halts].
\end{align*}
\]

This example also illustrates our point that it can be easy to give the CFG and the semantic representations of typical sentences, but it is not so easy to construct the resulting DCG manually.

### 3.3. Multiple and Equivalent Solutions

In general, the set of higher-order equations generated from a particular CFG and a set of sample sentence-meaning pairs 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 sentences accepted by the grammar. Consider the following CFG and sentence-meaning pairs:

\[
\begin{align*}
\text{s} & \rightarrow \text{pn}, \text{vp}. \\
\text{vp} & \rightarrow \text{tv}, \text{pn}. \\
\text{pn} & \rightarrow [\text{mike}]. \\
\text{pn} & \rightarrow [\text{mary}]. \\
\text{pn} & \rightarrow [\text{john}]. \\
\text{tv} & \rightarrow [\text{saw}]. \\
\text{tv} & \rightarrow [\text{visited}].
\end{align*}
\]

<table>
<thead>
<tr>
<th>Sentence</th>
<th>Semantic representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[mike,saw,mary]</td>
<td>(saw mike mary)</td>
</tr>
<tr>
<td>[john,saw,mary]</td>
<td>(saw john mary)</td>
</tr>
<tr>
<td>[mike,visited,mary]</td>
<td>(visited mike mary)</td>
</tr>
<tr>
<td>[mike,saw,john]</td>
<td>(saw mike john)</td>
</tr>
</tbody>
</table>

The following two DCGs can be derived from the above CFG and sample sentences:

1. \[
\begin{align*}
\text{s}((\text{D C})) & \rightarrow \text{pn(C)}, \text{vp(D)}. \\
\text{vp}((\text{D C E})) & \rightarrow \text{tv(D)}, \text{pn(E)}. \\
\text{pn(mike)} & \rightarrow [\text{mike}]. \\
\text{pn(mary)} & \rightarrow [\text{mary}]. \\
\text{pn(john)} & \rightarrow [\text{john}]. \\
\text{tv}(\text{A}\text{B}\text{\{saw A B\}}) & \rightarrow [\text{saw}]. \\
\text{tv}(\text{A}\text{B}\text{\{visited A B\}}) & \rightarrow [\text{visited}].
\end{align*}
\]

2. \[
\begin{align*}
\text{s}((\text{D C})) & \rightarrow \text{pn(C)}, \text{vp(D)}. \\
\text{vp}((\text{B C})) & \rightarrow \text{tv(B)}, \text{pn(C)}. \\
\text{pn(mike)} & \rightarrow [\text{mike}]. \\
\text{pn(mary)} & \rightarrow [\text{mary}]. \\
\text{pn(john)} & \rightarrow [\text{john}]. \\
\text{tv}(\text{A}\text{B}\text{\{saw B A\}}) & \rightarrow [\text{saw}]. \\
\text{tv}(\text{A}\text{B}\text{\{visited B A\}}) & \rightarrow [\text{visited}].
\end{align*}
\]

The difference between the two DCGs is that the arguments of the semantic representations of verbs are in a different order. This is compensated for by appropriately modifying the semantics of the verb-phrase rule. However, if the problem is underconstrained, that is, if insufficient sample sentences 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.

4. FROM HIGHER-ORDER TO FIRST-ORDER DCGs

The higher-order DCGs constructed in Section 2 are not as efficient as equivalent first-order DCGs since \(\lambda\)-terms are generally more complicated to process than first-order terms, which do not have any binder variables. However, it turns out that for many common cases, higher-order DCGs can be converted into first-order DCGs by precompiling all \(\beta\)-reductions involved in the execution of the DCGs. This conversion can be considered a form of partial execution. Below, we describe a 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 $\beta$-reduction by first-order unification for "forward execution," i.e., computing the semantic representation of a given sentence. For "reverse execution" of the DCG, a simple constraint of treating distinct binder variables as distinct constants yields a correct procedure. Thus, 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 matching by first-order unification.

We present our partial-execution procedure in stages in order to motivate the need for each capability. Section 4.1 gives the basic procedure for partial execution and discusses its correctness and limitations, Section 4.2 shows how to relax the restrictions of the basic partial execution procedure, and Section 4.3 shows how reversibility can be achieved.

4.1. Basic Partial Execution

The input to the partial execution procedure is a higher-order DCG, i.e., the output of procedure SYNTH of Section 2. 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 4.1. All application terms are reduced in computing the semantic representation of the sentence being parsed (unless, of course, the head of the application in the DCG rule is a constant).

Assumption 4.2. In any term $x \backslash t$, the binder variable $x$ occurs at most once in $t$.

Procedure for Basic Partial Execution.

1. Rename variables so that all binder variables within every rule are distinct.
2. foreach higher-order DCG rule $r$ do
   
   foreach application term $(t_1 \ t_2)$ in $r$ where $t_1$ is a variable do
   
   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 $t_2$.

In a partially executed grammar, the symbol $\backslash$ is simply an infix binary constructor, and therefore can now take structured terms in both of its argument positions.

Example 4.1.1. Partial execution is illustrated for the following DCG:

\begin{itemize}
  \item (r1) $s((A \ B)) \rightarrow np(A), \ vp(B)$.
  \item (r2) $np(Y \backslash (Y \ A)) \rightarrow pn(A)$.
  \item (r3) $vp(Z \backslash (B \ (A \ Z))) \rightarrow tv(A), \ np(B)$.
  \item (r4) $pn(mike) \rightarrow [mike]$.
  \item (r5) $pn(mary) \rightarrow [mary]$.
  \item (r6) $pn(john) \rightarrow [john]$.
  \item (r7) $tv(A \backslash B \backslash (saw \ A \ B)) \rightarrow [saw]$.
  \item (r8) $tv(A \backslash B \backslash (visited \ A \ B)) \rightarrow [visited]$.
\end{itemize}
Rule (r1) is partially executed as follows:

\[ s((A \ B)) \rightarrow np(A), \ vp(B). \]
\[ \Rightarrow s((C\ D \ B)) \rightarrow np(C\ D), \ vp(B). \]
\[ \Rightarrow s(D) \rightarrow np(B\ D), \ vp(B). \]

Rule (r2) illustrates how abstractions are partially executed:

\[ np(Y\ (Y \ A)) \rightarrow pn(A). \]
\[ \Rightarrow np((B\ C\ (B\ C\ A)) \rightarrow pn(A). \]
\[ \Rightarrow np((A\ C\ C) \rightarrow pn(A). \]

Likewise, rule (r3):

\[ vp(Z\ (B\ (A \ Z))) \rightarrow tv(A), \ np(B). \]
\[ \Rightarrow vp(Z\ (B\ (C\ D \ Z))) \rightarrow tv(C\ D), \ np(B). \]
\[ \Rightarrow vp(Z\ (B\ D)) \rightarrow tv(Z\ D), \ np(B). \]
\[ \Rightarrow vp(Z\ (E\ F\ D)) \rightarrow tv(Z\ D), \ np(E\ F). \]
\[ \Rightarrow vp(Z\ F) \rightarrow tv(Z\ D), \ np(D\ F). \]

As there are no more applications satisfying Assumption 4.1, the following first-order DCG is obtained (note that \ is a first-order, infix constructor):

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

We will show informally that, under the two assumptions (4.1 and 4.2) 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. For this purpose, we first show when \-terms and the substitution operation in the \-calculus can be correctly simulated by first-order terms and first-order substitution. Let \( t[x \leftarrow u] \) abbreviate the operation of substitution. In a first-order language, it refers to the result of textually replacing all occurrences of variable \( x \) in term \( t \) by term \( u \). In \-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 conditions under which \-terms may be simulated by first-order terms (in which all binder variables are treated as logical variables) and \-calculus substitution simulated by first-order substitution (where renaming is absent):

1. all binder variables have distinct names; and
2. each binder variable occurs at most once in the body of the \-term (linearity).

To see what can go wrong without distinct binder variables, consider the result of \( X\ (foo Y) [Y \leftarrow X\ X] \). The resulting term, \( X\ (foo X\ X) \), when viewed as a first-order term, is not a satisfactory representation of the result of the substitution operation because the occurrences of \( X \) in the two binder positions stand for the same variable. To see what can go wrong without linearity, consider the result of
The result, $Z(\text{foo } Z X X X \times X X)$, when viewed as a first-order term, is again not satisfactory.

**Proposition 1.** Assuming all binder variables in a $\lambda$-term $t$ have distinct names and each binder variable occurs at most once in the body of the term, $\beta$-reduction of $t$ can be correctly simulated by first-order substitution.

**Proof.** Consider a $\beta$-redex in $t$ of the form $(x \langle t_1 t_2 \rangle)$. By the given assumption, all binder variables are distinct in this redex. Since $x$ occurs at most once in $t_1$, the result of $\beta$-reducing this redex is either $t_1$ (in case $x$ does not occur in $t_1$) or else it is $t_1[x \leftarrow t_2]$ by a textual, first-order substitution. (Note that none of the free variables in $t_2$ can be "captured" by a binder in $t_1$ by the assumption of distinct binders.) In either case, the result of reducing $t$ is a strictly smaller term satisfying the assumptions of the proposition. Hence, the term can be repeatedly reduced by textual substitution to derive its normal form. □

**Theorem 1.** Under Assumptions 4.1 and 4.2, the partially executed DCG computes the same semantic representations for all sentences as the corresponding higher-order DCG.

**Proof.** Given a higher-order DCG, we first show that the two requirements for correct simulation of $\beta$-reduction by first-order substitution are met. Distinct binder variables are guaranteed through a combination of compile-time renaming of binder variables and the use of distinct variants of clauses at each backchaining step. That is, if each individual DCG rule is linear to start with (Assumption 4.2), linearity is guaranteed during rule instantiation because the terms returned by different nonterminals on the right-hand side of a rule cannot have any variables in common. Thus, the requirements for simulating $\beta$-reduction by first-order substitution are satisfied. The partial-execution procedure replaces every application term $(x \langle t_2 \rangle)$, where $x$ is a variable, by a new variable $y$, and it replaces all occurrences of $x$ by $t_2 \langle y \rangle$. By Assumption 4.1, every such application term must be reduced, and by the above proposition, the result of the reduction is correct. □

**4.2. Improved Partial Execution Procedure**

We now discuss how the basic partial execution procedure can be improved, by illustrating how Assumptions 4.1 and 4.2 can be relaxed.

**4.2.1. Relaxing Assumption 4.1.** We now consider the case where not all application terms need to be reduced. An illustration is provided in Example 4.2.1, where the application terms in certain Church numerals are not reduced. The solution to this problem is to trace the $\beta$-reductions performed in the higher-order DCG for the sample sentences to see which applications actually need to be reduced. For each application $(A \langle B \rangle)$ occurring in a higher-order DCG rule, we may 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 $(A \langle B \rangle)$ will be reduced eventually. Assuming that such an application $(A \langle B \rangle)$ is either reduced in all sentences or is never reduced, one can distinguish accordingly which applications can be partially executed and which cannot (we reconsider this assumption in Example 4.2.2).
Example 4.2.1. Consider the following CFG and sample sentence-meaning pairs:

\[
\begin{align*}
  s & \rightarrow [0]. \\
  s & \rightarrow [0,0]. \\
  s & \rightarrow [0,0], s.
\end{align*}
\]

<table>
<thead>
<tr>
<th>Sentence</th>
<th>Semantic representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>F\X\X</td>
</tr>
<tr>
<td>[0,0]</td>
<td>F\X\X(F X)</td>
</tr>
<tr>
<td>[0,0,0]</td>
<td>F\X\X</td>
</tr>
<tr>
<td>[0,0,0,0]</td>
<td>F\X\X(F X)</td>
</tr>
</tbody>
</table>

The following higher-order DCG is produced by procedure \textit{SYNTH} (Section 2):

\[
\begin{align*}
  s(A\B\B) & \rightarrow [0]. \\
  s(A\B(A\ B)) & \rightarrow [0],[0]. \\
  s(A\B\C(A\ B\ C) D) & \rightarrow [0],[0],s(D).
\end{align*}
\]

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

\[
\begin{align*}
  (A\B\C(A\ B\ C) F\X\X) \\
  & = B\C(F\X\X B C) \\
  & = B\C(X\X C) \\
  & = B\C C.
\end{align*}
\]

Therefore, the third rule can be partially executed accordingly:

\[
\begin{align*}
  s(A\B\C(A\ B\ C) D) & \rightarrow [0],[0],s(D). \\
  \Rightarrow s(B\C(D B C)) & \rightarrow [0],[0],s(D). \\
  \Rightarrow s(B\C(KL B C)) & \rightarrow [0],[0],s(KL). \\
  \Rightarrow s(B\C(L C)) & \rightarrow [0],[0],s(B\L). \\
  \Rightarrow s(B\C(F)) & \rightarrow [0],[0],s(B\C F).
\end{align*}
\]

The reader may verify that the application term in the second rule is never reduced in deriving the meaning of any sentence.

Example 4.2.2. In certain cases, a particular application term is reduced in parsing certain sentences, but not others. Consider the following DCG:

\[
\begin{align*}
  s(A\B\B) & \rightarrow [0]. \\
  s(A\B(A\ B)) & \rightarrow [0],[0]. \\
  s(A\B\B) & \rightarrow [0],[0],[0]. \\
  s(A\B\C(A\ D\C (B\ C) E) & \rightarrow [0],[0],[0],s(E).
\end{align*}
\]
This higher-order DCG cannot be partially executed with any of the schemes discussed so far. This is because the application in $A \backslash B \backslash (A \ B)$ 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 $A \backslash B \backslash (A \ B)$ would reduce to $(B \backslash D) \backslash B \backslash D$, and the rules of the above grammar would have to be changed accordingly:

\[
\begin{align*}
    s(A \backslash B \backslash B) & \rightarrow [0]. \\
    s((B \backslash D) \backslash B \backslash D) & \rightarrow [0],[0]. \\
    s(A \backslash B \backslash B) & \rightarrow [0],[0],[0]. \\
    s((C \backslash H) \backslash C \backslash J) & \rightarrow [0],[0],[0],s((D \backslash C) \backslash H \backslash J)).
\end{align*}
\]

If such a completely reduced DCG is used, not all generated semantic terms would be legal terms of the $\lambda$-calculus. Therefore, partial execution has to be reversed for some terms, e.g., $(B \backslash D) \backslash B \backslash D$ has to be converted back to $A \backslash B \backslash (A \ B)$, and $(D \backslash B) \backslash D \backslash D$, which is generated for the sentences $[0, 0, 0, 0, 0]$, has to be converted to $E \backslash D \backslash D$. A partially executed term in converted back to a legal $\lambda$-term by replacing any binder term of the form $(A \backslash B)$ by a new variable $C$, and then replacing all occurrences of $B$ by $(C \ A)$. The conversion of $(C \backslash B) \backslash C \backslash B$ to $A \backslash B \backslash (A \ B)$ is done by replacing $(C \backslash B)$ with the new variable $A$, and then replacing $B$ with $(A \ B)$.

4.2.2. Relaxing Assumption 4.2. We now consider relaxing the linearity assumption. First, we note that there are certain forms of nonlinear rules that do not require any special treatment, e.g.,

\[
\begin{align*}
    \text{det}(A \backslash B \backslash (\exists C (\text{and } (B \ C) (A \ C)))) & \rightarrow [a]. \\
    \text{det}(A \backslash B \backslash (\forall C (\text{implies } (B \ C) (A \ C)))) & \rightarrow [\text{every}].
\end{align*}
\]

In the above rules, the variable $C$ is of a primitive type $i$, and hence can never be bound to an abstraction. Hence, partially executing these rules by the basic partial execution procedure is correct:

\[
\begin{align*}
    \text{det}((C \backslash M) \backslash (C \backslash L) \backslash (\forall C (\text{implies } L \ M))) & \rightarrow [\text{every}]. \\
    \text{det}((C \backslash M) \backslash (C \backslash L) \backslash (\text{some } C (\text{and } L \ M))) & \rightarrow [a].
\end{align*}
\]

When nonlinearity arises because of repeated occurrences of a variable that has a functional type, the basic partial execution procedure cannot work correctly. For example, consider the rule

\[
\begin{align*}
    a((B (B C))) & \rightarrow b(B), c(C).
\end{align*}
\]

Here, $B$ is of a functional type. To illustrate the problem, suppose we replace $B$ with $X \ Y$; we would obtain

\[
\begin{align*}
    a((X \ Y (X \ Y C))) & \rightarrow b(X \ Y), c(C).
\end{align*}
\]

Now, $X$ must be equated to $(X \ Y C)$ and also 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 $B$. In the example above, we would have

\[
\begin{align*}
    a((B1 (B2 C))) & \rightarrow b(B1), \text{copy}(B1,B2), c(C).
\end{align*}
\]

\[\text{A copy of term } B \text{ leaves constants unchanged, but variables are consistently renamed, e.g., foo}(C,D,D) \text{ would be a copy of foo}(A,B,B).\]
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:

\[ a((X1\Y1 (B2 C))) \rightarrow b(X1\Y1), \text{copy}(X1\Y1,B2), c(C). \]

Reducing the applications gives

\[ a(Y1) \rightarrow b((B2 C)\Y1), \text{copy}(((B2 C)\Y1),B2), c(C). \]

Next, B2 is replaced by X2\Y2:

\[ a(Y1) \rightarrow b((X2\Y2 C)\Y1), \text{copy}(((X2\Y2 C)\Y1,X2\Y2), c(C). \]

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

\[ a(Y1) \rightarrow b(Y2\Y1), \text{copy}(Y2\Y1,C\Y2), c(C). \]

Example 4.2.3. 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:

\[ F\X(F (F X)) \]

\[ = F\X(F1 (F2 X)), \text{copy}(F, F1), \text{copy}(F, F2) \]

\[ = F\X(A1\B1 (A2\B2 X)), \text{copy}(F, A1\B1), \text{copy}(F, A2\B2) \]

\[ = F\X(A1\B1 B2), \text{copy}(F, A1\B1), \text{copy}(F, X\B2) \]

\[ = F\X(B1, \text{copy}(F, B2\B1), \text{copy}(F, X\B2). \]

The reduced form is converted back into a regular \( A \)-term as follows:

\[ F\X(B1, \text{copy}(F, B2\B1), \text{copy}(F, X\B2) \]

\[ = F\X(F1 B2), \text{copy}(F, F1), \text{copy}(F, X\B2) \]

\[ = F\X(F1 (F2 X)), \text{copy}(F, F1), \text{copy}(F, F2) \]

\[ = F\X(F (F X)). \]

4.3. Reversibility

Dymetman et al. [6] 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. The main benefit of reversibility is that a separate DCG is not needed for generation. Reversibility also makes it easier to check that a DCG neither overgenerates nor undergenerates, i.e., it generates or accepts all and only correct sentences for a particular semantic representation.

The higher-order DCGs of Section 2 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 \( s((F A B)) \rightarrow \text{np}(A), \text{vp}(B) \) is used for parsing, the semantics, \( A \), for \text{np}, and \( B \), for \text{vp}, are computed first, and then \( F \) is applied to \( A \) and \( B \). If, however, the rule is used for generation, \( A \) and \( B \) would have to be assigned nondeterministically using higher-order matching. The partially executed grammar supports efficient reverse execution of DCGs because first-order unification (rather than higher-order matching) is needed. However, in certain cases, reverse execution might give incorrect answers, as illustrated by the following example. The grammar

\[ s(X\Y\Y) \rightarrow [0]. \]

\[ s(X\Y\X) \rightarrow [1]. \]
will be expanded to the following clauses:

\[ s(X \backslash Y \backslash Y, [OIT], T). \]
\[ s(X \backslash Y \backslash X, [lIT], [1IT], T). \]

When used in the parsing mode, this grammar correctly computes \( X \backslash Y \backslash Y \) for \([0]\), and \( X \backslash Y \backslash X \) for \([1]\). However, consider its use in the generation mode:

\[- s(A \backslash B \backslash B, Sent, []). \]

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.

We now discuss how to obtain correct reverse execution of a partially executed DCG. It has been shown in the previous section that a partially executed DCG \( G_\pi \) 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 for a particular semantic representation. In order to ensure correctness when using the partially executed DCG in the reverse direction, we “freeze” the input semantic representation; that is, we convert all of its variables into constants (different constants are used for different variables).

**Theorem 2.** Assuming that an input semantic representation \( m \) is closed, reverse execution of a partially executed DCG \( D \) from the frozen form of \( m \) computes exactly the set of sentences whose semantics is \( m \).

**Proof.** Let \( s_1, \ldots, s_k \) be all the sentences with semantics \( m \). By the correctness of forward execution (from \( s_1, \ldots, s_k \)) 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 of the variables in \( m \) (since they 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, i.e., that reverse execution does not compute any incorrect sentence from \( n \). All we need to show is 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. Hence, reverse execution from the frozen form of a term \( m \) computes exactly the set of sentences whose semantics is \( m \). \( \square \)

5. CONCLUSIONS AND RELATED WORK

We have shown that it is possible, under reasonable assumptions, to mechanically transform an unambiguous 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 our knowledge, the problem has not been addressed in the literature. The key idea needed to solve this problem was to adopt the simply typed \( \lambda \)-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 in this way is not only natural, it seems to be sufficient for the class of natural query languages. With these assumptions, we showed that the problem of generalization from examples can be cast as a unification problem over simply typed $\lambda$-terms [11].

Higher-order logics (or typed $\lambda$-calculi) are useful for synthesizing and manipulating formulae since the latter can be viewed as (data) terms, 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, it was necessary to implement the unification procedure so that the constraints from several examples are enforced simultaneously. To further improve performance, we 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. This conversion is based on the observation that, for most of the mechanically generated higher-order DCGs, the needed forms of $\beta$-reduction can be simulated by first-order substitution. This aspect of our work is similar in spirit to the motivation underlying the language $L_\lambda$ [14], although the respective technical approaches are different. With the aid of a simple "trick" of freezing binder variables, the resulting partially executed DCG is also capable of correct and efficient reverse execution. The idea of freezing variables in order to obtain correct reverse execution has also been used in semantic-head-driven generation [19].

Haas has implemented a system that incorporates the above ideas, and has tested it on the synthesis of a variant of the CHAT-80 natural query language [7]. In applying the system to practical grammars, we have found that the (semantic) function variables introduced for rules that derive terminal symbols, e.g., $N \rightarrow [t]$, can be solved by using an imitation substitution or by a direct assignment, i.e., projection substitutions are not needed. Incorporating this knowledge into the system helps curtail the search considerably. We have also found that, 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 paper allow one to associate semantics with grammar rules or terminals if their semantics is known or has already been inferred. In this way, 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. Our implemented system also does not require the types to be supplied for all nonterminals. When type information is missing, the system will try to infer types through type inference, and will also generate additional projection substitutions through type enumeration (see [8] for more details).

While we are not aware of any research that solves our stated problem, research in program synthesis by examples and machine learning is closely related. The analogy between DCG synthesis and program synthesis is the following: the context-free grammar can be viewed as a program schema, the sample sentence-meaning pairs can be viewed as the sample input–output pairs of the desired program, and the unknown function variables of the DCG of step 2 of SYNTH correspond to the
unknown function variables of the program schema. However, program synthesis is the harder of the two problems since DCG synthesis starts with the knowledge of the context-free grammar, whereas program synthesis also involves the determination of the right schema. Recently, Hagiya showed the use of the simply typed $\lambda$-calculus and higher-order unification for program synthesis from schemas and examples [9, 10]. A noteworthy technical difference from Hagiya's work is that he encodes schemas using a special kind of term, and provides an extended higher-order unification procedure for an extended simply typed $\lambda$-calculus, whereas we maintain a sharp difference between the grammar (or schema) and the typed $\lambda$-terms.

Our DCG synthesis work can be considered as an example of inductive learning since the semantics of all sentences of a grammar is induced from a finite number of examples. In this connection, it would be interesting to characterize the class of DCGs that can be learned from a finite number of examples using our synthesis technique. As noted in the Introduction, there are several possible areas of further work, both of a theoretical and practical nature. Of particular interest in considering the synthesis of DCGs for larger fragments of natural language rather than natural query languages is the necessity of working with ambiguous grammars. Certain forms of ambiguous grammars can be accommodated in our scheme with only minor extensions: essentially, the user provides one semantic representation for each different parse of an ambiguous sentence; and SYNT draws up multiple equations for such a sentence—the pairing-up of each user-supplied semantic representation with the computed semantic representation contributes an additional source of nondeterminism. We are investigating how this scheme can be made more efficient.

This research was supported in part by NSF Grant CCR 9004357. We thank Dale Miller, Fernando Pereira, and the anonymous referees for their comments and suggestions.

REFERENCES


