Some Uses of Higher-Order Logic in Computational Linguistics

Dale A. Miller and Gopalan Nadathur
Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104 – 3897
April 1986

Abstract

Consideration of the question of meaning in the framework of linguistics often requires an allusion to sets and other higher-order notions. The traditional approach to representing and reasoning about meaning in a computational setting has been to use knowledge representation systems that are either based on first-order logic or that use mechanisms whose formal justifications are to be provided after the fact. In this paper we shall consider the use of a higher-order logic for this task. We first present a version of definite clauses (positive Horn clauses) that is based on this logic. Predicate and function variables may occur in such clauses and the terms in the language are the typed λ-terms. Such term structures have a richness that may be exploited in representing meanings. We also describe a higher-order logic programming language, called λProlog, which represents programs as higher-order definite clauses and interprets them using a depth-first interpreter. A virtue of this language is that it is possible to write programs in it that integrate syntactic and semantic analyses into one computational paradigm. This is to be contrasted with the more common practice of using two entirely different computation paradigms, such as DCGs or ATNs for parsing and frames or semantic nets for semantic processing. We illustrate such an integration in this language by considering a simple example, and we claim that its use makes the task of providing formal justifications for the computations specified much more direct.

This work has been supported by NSF grants MCS-82-19196-CER, MCS-82-07294, AI Center grants MCS-83-05221, US Army Research Office grant ARO-DAA29-84-9-0027, and DARPA N000-14-85-K-0018.

1. Introduction

The representation of meaning, and the use of such a representation to draw inferences, is an issue of central concern in natural language understanding systems. A theoretical understanding of meaning is generally based on logic, and it has been recognized that a higher-order logic is particularly well suited to this task. Montague, for example, used such a logic to provide a compositional semantics for simple English sentences. In the computational framework, knowledge representation systems are given the task of representing the semantical notions that are needed in natural language understanding programs. While the formal justifications that are provided for such systems are usually logical, the actual formalisms used are often distantly related to logic. Our approach in this paper is to represent meanings directly by using logical expressions, and to describe the process of inference by specifying manipulations on such expressions. As it turns out, most programming languages are poorly suited for an approach such as ours. Prolog, for instance, permits the representation and the examination of the structure of first-order terms, but it is not easy to use such terms to represent first-order formulas which contain quantification. Lisp on the other hand allows the construction of lambda expressions which could encode the binding operations of quantifiers, but does not provide logical primitives for studying the internal structure of such expressions. A language that is based on a higher-order logic seems to be the most natural vehicle for an approach such as ours, and in the first part of this paper we shall describe such a language. We shall then use this language to describe computations of a kind that is needed in a natural language understanding system.

Before we embark on this task, however, we need to consider the arguments that are often made against the computational use of a higher-order logic. Indeed, several authors in the current literature on computational linguistics and knowledge representation have presented reasons for preferring first-order logic over higher-order logic in natural language understanding systems, and amongst these the following three appear frequently.

(1) Gödel showed that second-order logic is essentially incomplete, *i.e.* true second-order logic statements are not recursively enumerable. Hence, theorem provers for this logic cannot be, even theoretically,

complete.

(2) Higher-order objects like functions and predicates can themselves be considered to be first-order objects of some sort. Hence, a sorted first-order logic can be used to encode higher-order objects.

(3) Little research on theorem proving in higher-order logics has been done. Moreover, there is reason to believe that theorem proving in such a logic is extremely difficult.

These facts are often used to conclude that a higher-order logic should not be used to formalize systems if such formalizations are to be computationally meaningful. While there is some truth in each of these observations, we feel that they do not warrant the conclusion that is drawn from it. We discuss our reasons for this belief below.

The point regarding the essential undecidability of second-order logic has actually little import on the computational uses of higher-order logic. This is because the second-order logic as it is construed in this observation, is not a proof system but rather a truth system of a very particular kind. Roughly put, the second-order logic in question is not so much a logic as it is a branch of mathematics which is interested in properties about the integers. There are higher-order logics that have been provided which contain the formulas of second-order logic but which do not assume the same notion of models (*i.e.* the integers). These logics, in fact, have *general models*, including the standard, integer model, as well as other non-standard models, and with respect to this semantics, the logic has a sound and complete proof system.

From a theoretical point-of-view, the second observation is important. Indeed, any system which could not be encoded into first-order logic would be more powerful than Turing machines and, hence, would be rather unsatisfactory computationally! The existence of such an encoding has little significance, however, with regard to the appropriateness of one language over another for a given set of computational tasks. Clearly, all general purpose programming languages can be encoded onto first-order logic, but this has little significance with regard to the suitability of a given programming language for certain applications.

Although less work has been done on theorem proving in higher-order logic than in first-order logic as claimed in the last point, the nature of proofs in higher-order logic is far from mysterious. For example, higher-order resolution [1] and unification [8] has been developed, and based on these principles, several theorem provers for various higher-order logics (see [2] and its references) have been built and tested. The experience with such systems shows that theorem proving in such a logic is difficult. It is not clear, however, that the difficulty is inherent in the language chosen to express a theorem rather than in the theorem itself. In fact, expressing a higher-order theorem (as we will claim many statements about meaning are) in a higher-order logic makes its logical structure more explicit than an encoding into first-order logic does. Consequently, it is reasonable to expect that the higher-order representation should actually simplify the process of finding proofs. In a more specific sense, there are sublogics of a higher-order logic in which the process of constructing proofs is not much more complicated than in similar sublogics of first-order logic. An example of such a case is the higher-order version of definite clauses that we shall consider shortly.

In this paper, we present a higher-order version of definite clauses that may be used to specify computations, and we describe a logic programming language, λProlog, that is based on this specification language. We claim that λProlog has several linguistically meaningful applications. To bolster this claim we shall show how the syntactic and semantic processing used within a simple parser of natural language can be smoothly integrated into one logical and computational process. We shall first present a definite clause grammar that analyses the syntactic structure of simple English sentences to produce logical forms in much the same way as is done in the Montague framework. We shall then show how semantic analyses may be specified via operations on such logical forms. Finally, we shall illustrate interactions between these two kinds of analyses by considering an example of determining pronoun reference.

2. Higher-Order Logic

The higher-order logic we study here, called $\mathcal{T}$, can be thought of as being a subsystem of either Church's Simple Theory of Types [5] or of Montague's intensional logic IL [6]. Unlike Church's or Montague's logics, $\mathcal{T}$ is very weak because it assumes no axioms regarding extensionality, definite descriptions, infinity, choice, or possible worlds. $\mathcal{T}$ encompasses only the most primitive logical notions, and generalizes first-order logic by introducing stronger notions of variables and substitutions. Our use of $\mathcal{T}$ is not driven by a desire to capture the meaning of linguistic objects, as was the hope of Montague. It is our hope that programs written in $\mathcal{T}$ will do that.

The language of $\mathcal{T}$ is a *typed* language. The typing mechanism provides for the usual notion of sorts often used in first-order logic and also for the notion of *functional types*. We take as primitive types (*i.e.* sorts) o for booleans and i for (first-order) individuals, adding others as needed. Functional types are written as $\alpha \rightarrow \beta$, where α and β are types. This type is intended to denote the type of functions whose domains are α and whose codomains are β. For example, $i \rightarrow i$ denotes the type of functions which map individuals to individuals, and $(i \rightarrow i) \rightarrow o$ denotes the type of functions from that domain to the booleans. In reading such expressions we use the convention that $\rightarrow$ is right associative, *i.e.* we read $\alpha \rightarrow \beta \rightarrow \gamma$ as $\alpha \rightarrow (\beta \rightarrow \gamma)$.

The terms or formulas of $\mathcal{T}$ are specified along with their respective types by the following simple rules: We start with denumerable sets of constants and variables at each type. A constant or variable in any of these sets is considered to be a formula of the corresponding type. Then, if A is of type $\alpha \rightarrow \beta$ and B is of type α, the function application (AB) is a formula of type β. Application associates to the left. Finally, if x is a variable of type α and C is a term of type β, the *function abstraction* $\lambda x C$ is a formula of type $\alpha \rightarrow \beta$.

We assume that the following symbols, called the logical constants, are included in the set of constants of the corresponding type: *true* of type o, $\sim$ of type $o \rightarrow o$, $\wedge$, $\vee$, and $\supset$ each of type $o \rightarrow o \rightarrow o$ and Π and Σ of type $(A \rightarrow o) \rightarrow o$ for each type A. All these symbols except the last two correspond to the normal propositional connectives. The symbols Π and Σ are used in conjunction with the abstraction operation to represent universal and existential quantification: $\forall x\ P$ is an abbreviation for $\Pi(\lambda x\ P)$ and $\exists x\ P$ is an abbreviation for $\Sigma(\lambda x\ P)$. Π and Σ are examples of what are often called *generalized quantifiers*.

The type o has a special role in this language. A formula with a function type of the form $t_1 \rightarrow \ldots \rightarrow t_n \rightarrow o$ is called a *predicate of* n *arguments*. The i^{th} argument of such a predicate is of type t_i. Predicates are to be thought of as representing sets and relations. Thus a predicate of type $i \rightarrow o$ represents a set of individuals, a predicate of type $(i \rightarrow o) \rightarrow o$ represents a set of sets of individuals, and a predicate of type $i \rightarrow (i \rightarrow o) \rightarrow o$ represents a binary relation between individuals and sets of individuals. Formulas of type o are called *propositions*. Although predicates are essentially functions, we shall generally use the term function to denote a formula that does not have the type of a predicate.

Derivability in $\mathcal{T}$, denoted by $\vdash_{\mathcal{T}}$, is defined in the following (simplified) fashion. The axioms of $\mathcal{T}$ are the propositional tautologies, the formula $\forall x\ Bx \supset Bt$, and the formula $\forall x\ (Px \wedge Q) \supset \forall x\ Px \wedge Q$. The rules of inference of the system are *Modus Ponens*, *Universal Generalization*, *Substitution*, and *λ-conversion*. The rules of λ-conversion that we assume here are α-conversion (change of bound variables), β-conversion (contraction), and η-conversion (replace A with $\lambda z(Az)$ and vice versa if A has type $\alpha \rightarrow \beta$, z has type α, and z is not free in A). λ-conversion is essentially the only rule in $\mathcal{T}$ that is not in first-order logic, but combined with the richer syntax of formulas in $\mathcal{T}$ it makes more complex inferences possible.

In general, we shall consider two terms to be equal if they are each convertible to the other; further distinctions can be made between formulas in this sense by omitting the rule for η-conversion, but we feel that such distinctions are not important in our context. We say that a formula is a *λ-normal formula* if it has the form

$$\lambda x_1 \ldots \lambda x_n\ (h\ t_1\ \ldots\ t_m) \qquad \text{where } n, m \geq 0,$$

where h is a constant or variable, $(h\ t_1\ \ldots\ t_m)$ has a primitive type, and, for $1 \leq i \leq m$, t_i also has the same form. We call the list of variables $x_1, \ldots, x_n$ the *binder*, h the *head*, and the formulas $t_1, \ldots, t_m$ the arguments of such a formula. It is well known that every formula, A, can be converted to a λ-normal formula that is unique up to α-conversions. We call such a formula a λ-normal form of A and we use $\lambda norm(A)$ to denote any of these alphabetic variants. Notice that a proposition in λ-normal form must have an empty binder and contain either a constant or free variable as its head. A proposition in λ-normal form which has a non-logical constant as its head is called *atomic*.

Our purpose in this paper is not merely to use a logic as a representational device, but also to think of it as a device for specifying computations. It turns out that $\mathcal{T}$ is too complex for the latter purpose. We shall therefore restrict our attention to what may be thought of as a higher-order analogue of positive Horn clauses. We define these below.

We shall henceforth assume that we have a fixed set of nonlogical constants. The *positive Herbrand Universe* is identified in this context to be the set of all the λ-normal formulas that can be constructed via function application and abstraction using the nonlogical constants and the logical constants *true*, $\wedge$, $\vee$ and Σ; the omission here is of the symbols $\sim$, $\supset$, and Π. We shall use the symbol $\mathcal{H}^+$ to denote this set of terms. Propositions in this set are of special interest to us. Let G and A be propositions in $\mathcal{H}^+$ such that A is atomic. A *(higher-order) definite clause* then is the universal closure of a formula of the form $G \supset A$, *i.e.* the formula $\forall \bar{x}\ (G \supset A)$ where $\bar{x}$ is an arbitrary listing of all the free variables in G and A, some of which may be function and predicate variables. These formulas are our generalization of positive Horn clauses for first-order logic. The formula on the left of the $\supset$ in a higher-order definite clause may contain nested disjunctions and existential quantification. This generalization may be dispensed within the first-order case because of the existence of appropriate normal forms. For the higher-order case, it is more natural to retain the embedded disjunctions and existential quantifications since substitutions for predicate variables have the potential for re-introducing them. Illustrations of this aspect appear in Section 4.

Deductions from higher-order definite clauses are very similar to deductions from positive Horn clauses in first-order logic. Substitution, unification, and backchaining can be combined to build a theorem prover in either case. However, unification in the higher-order setting is complicated by the presence of λ-conversion: two terms t and s are unifiable if there exists some substitution φ such that φs and φt are equal *modulo* λ-conversions. Since β-conversion is a very complex process, determining this kind of equality is difficult. The unification of typed λ-terms is, in general, not decidable, and when unifiers do exist, there need not exist a single most general unifier. Nevertheless, it is possible to systematically search for unifiers in this setting [8] and an interpreter for higher-order definite clauses can be built around this procedure. The resulting interpreter can be made to resemble Prolog except that it must account for the extra degree of nondeterminism which arises from higher-order unification. Although there are several important issues regarding the search for higher-order unifiers, we shall ignore them here since all the unification problems which arise in this paper can be solved by even a simple-minded implementation of the procedure described in [8].

3. λProlog

We have used higher-order definite clauses and a depth-first interpreter to describe a logic programming language called λProlog. We present below a brief exposition of the higher-order features of this language that we shall use in the examples in the later sections. A fuller description of the language and of the logical considerations underlying it may be found in [9].

Programs in λProlog are essentially higher-order definite clauses. The following set of clauses that define certain standard list operations serve to illustrate some of the syntactic features of our language.

```
append nil K K.
append (cons X L) K (cons X M) :- append L K M.
member X (cons X L).
member X (cons Y L) :- member X L.
```

As should be apparent from these clauses, the syntax of λProlog borrows a great deal from that of Prolog. Symbols that begin with capital letters represent variables. All other symbols represent constants. Clauses are written backwards and the symbol `:-` is used for $\subset$. There are, however, some differences. We have adopted a curried notation for terms, rather than the notation normally used in a first-order language. Since the language is a typed one, types must be associated with each term. This is done by either explicitly defining the type of a constant or a variable, or by inferring such a type by a process very similar to that used in the language ML [7]. The type expressions that are attached to symbols may contain variables which provide a form of polymorphism. As an example `cons` and `nil` above are assumed to have the types `A -> (list A) -> (list A)` and `(list A)` respectively; they serve to define lists of different kinds, but each list being such that all its elements have a common type. (For the convenience of expression, we shall actually use Prolog's notation for lists in the remainder of this paper, *i.e.* we shall write (cons X L) as [X|L]). In the examples in this paper, we shall occasionally provide type associations, but in general we shall assume that the reader can infer them from context when it is important. We need to represent λ-abstraction in our language, and we use the symbol \ for this purpose; *i.e.* $\lambda X\ A$ is written in λProlog as X \ A.

The following program, which defines the operation of mapping a function over a list, illustrates a use of function variables in our language.

```
mapfun F [X|L] [(F X)|K] :- mapfun F L K.
mapfun F [] [].
```

Given these clauses, (mapfun F L1 L2) is provable only if L2 is a list that results from applying F to each element of L1. The interpreter for λProlog would therefore evaluate the goal (mapfun (X\(g X X)) [a, b] L) by returning the value [(g a a),(g b b)] for L.

The logical considerations underlying the language permit functions to be treated as first-class, logic programming variables. In other words, the values of such variables can be computed through unification. For example, consider the query

```
(mapfun F [a, b] [(g a a), (g a b)]).
```

There is exactly one substitution for F, namely X\(g a X), that makes the above query provable. In searching for such higher-order substitutions, the interpreter for λProlog would need to backtrack over choices of substitutions. For example, if the interpreter attempted to prove the above goal by attempting to unify (F a) with (g a a), it would need to consider the following four possible substitutions for F:

```
X\(g X X)  X\(g a X)  X\(g X a)  X\(g a a).
```

If it chooses any of these other than the second, the interpreter would fail in unifying (F b) with (g a b), and would therefore have to backtrack over that choice.

It is important to notice that the set of functions that are representable using the typed λ-terms of λProlog is not the set of all computable functions. The set of functions that are so representable are in fact much weaker than those representable in, for example, a functional programming language like Lisp. Consider the goal

```
(mapfun F [a, b] [c, d]).
```

There is clearly a Lisp function which maps a to c and b to d, namely,

```
(lambda (x) (if (eq x 'a) 'b
                (if (eq x 'c) 'd 'e)))
```

Such a function is, however, not representable using our typed λ-terms since these do not contain any constants representing conditionals (or fixed point operators needed for recursive definitions). It is actually this restriction to our term structures that makes the determination of function values through unification a reasonable computational operation.

The provision of function variables and higher-order unification has several uses, some of which we shall examine in later sections. Before doing that we consider briefly certain kinds of function terms that have a special status in the logic programming context, namely predicate terms.

4. Predicates as Values

From a logical point of view, predicates are not much different from other functions; essentially they are functions that have a type of the form $\alpha_1 \to \ldots \to \alpha_n \to o$. In a logic programming language, however, variables of this type may play a different and more interesting role than non-predicate variables. This is because such variables may appear *inside* the terms of a goal as well as the *head* of a goal. In a sense, they can be used intensionally and extensionally (or nominally and saturated). When they appear intensionally, predicates can be determined through unification just as functions. When they appear extensionally, they are essentially "executed."

An example of these mixed uses of predicate variables is provided by the following set of clauses; the logical connectives $\wedge$ and $\vee$ are represented in λProlog by the symbols , and ;, *true* is represented by true and Σ is represented by the symbol sigma that has the polymorphic type (A -> o) -> o.

```
sublist P [X|L] [X|K] :- P X, sublist P L K.
sublist P [X|L] K :- sublist P L K.
sublist P [] [].
have_age L K :-
```

```
      sublist Z\(sigma X\(age Z X)) L K.
same_age L K :- sublist Z\(age Z A) L K.
age bob 23.
age sue 24.
age ned 23.
```

The first three clauses define the predicate `sublist` whose first argument is a predicate and is such that `(sublist P L K)` is provable if `K` is some sublist of `L` and all the members in `K` satisfy the property expressed by the predicate `P`. The fourth clause uses `sublist` to define the predicate `have_age` which is such that `(have_age L K)` is provable if `K` is a sublist of the objects in `L` which have an age. In the definition of `have_age` a predicate term that contains an explicit quantifier is used to instantiate the predicate argument of `sublist`; the predicate `(Z\(sigma X\(age Z X)))`, which may be written in logic as $\lambda z\ \exists x\ age(z,x)$, is true of an individual if that individual has an age. This predicate term needs to be executed in the course of evaluating, for example, the query `(have_age [bob,sue,ned] K)`. The predicate `same_age` whose definition is obtained by dropping the quantifier from the predicate term defines a different property; `(same_age L K)` is true only when the objects in `K` have the same age.

Another example is provided by the following set of clauses that define the operation of mapping a predicate over a list.

```
mappred P [X|L] [Y|K] :- P X Y, mappred P L K.
mappred P [] [].
```

This set of clauses may be used, for example, to evaluate the following query:

```
      mappred (X\Y\(age Y X)) [23,24] L.
```

This query essentially asks for a list of two people, the first of which is 23 years old while the second is 24 years old. Given the clauses that appear in the previous example, this query has two different answers: `[bob, sue]` and `[ned, sue]`. Clearly the mapping operation defined here is much stronger than a similar operation considered earlier, namely that of mapping a function over a list. In evaluating a query that uses this set of clauses a new goal, *i.e.* `(P X Y)`, is formed whose evaluation may require arbitrary computations to be performed. As opposed to this, in the earlier case only λ-reductions are performed. Thus, `mappred` is more like the mapping operations found in Lisp than `mapfun` is.

In the cases considered above, predicate variables that appeared as the heads of goals were fully instantiated before the goal was invoked. This kind of use of predicate variables is similar to the use of apply and lambda terms in Lisp: λ-contraction followed by the goal invocation simulates the apply operation in the Prolog context. However, the variable head of a goal may not always be fully instantiated when the goal has to be evaluated. In such cases there is a question as to what substitutions should be attempted. Consider, for example, the query `(P bob 23)`. One value that may be returned for `P` is `X\Y\(age X Y)`, and this may seem to be the most "natural" value. There are, however, many more substitutions for `P` which also satisfy this goal: `X\Y\(X = bob, Y = 23)`, `X\Y\(Y = 23)`, `X\Y\(age sue 24)`, *etc.* are all terms that could be picked, since if they were substituted for `P` in the query they would result in a provable goal. There are, clearly, too many substitutions to pick from and perhaps backtrack over. Furthermore several of these may have little to do with the original intention of the query. A better strategy may be to pick the one substitution that has the largest "extension" in such cases; in the case considered here, such a substitution for `P` would be the term `X\Y\true`. It is possible to make such a choice without adding to the incompleteness of an interpreter.

Picking such a substitution does not necessarily trivialize the use of predicate variables. If a predicate occurs intensionally as well as extensionally in a goal, this kind of a trivial substitution may not be possible. To illustrate this let us consider the following set of clauses:

```
primrel father.
primrel mother.
primrel wife.
primrel husband.
rel R :- primrel R.
rel X\Y\(sigma Z\(R X Z, S Z Y)) :-
     primrel R, primrel S.
```

The first four clauses identify four primitive relations between individuals (`primrel` has type `(i -> i -> o) -> o`). These are then used to define other relations that are a result of "joining" primitive relations. Now if `(mother jane mary)` and `(wife john jane)` are provided as additional clauses, then the query `(rel R, R john mary)` would yield the substitution `X\Y\(sigma Z\(wife X Z, mother Z Y))` for `R`. This query asks for a relation (in the sense of `rel`) between `john` and `mary`. The answer substitution provides the relation mother-in-law.

We have been able to show (Theorem 1 [9]) that any proof in $\mathcal{T}$ of a goal formula from a set of definite clauses which uses a predicate term containing the logical connectives $\sim$, $\supset$, or $\forall$, can be converted into another proof in which only predicate terms from $\mathcal{H}^+$ are used. Thus, it is not possible for a term such as

$$\lambda x\ (person(x) \wedge \forall y\ (child(x,y) \supset doctor(y)))$$

to be specified by a λProlog program, *i.e.* be the unique substitution which makes some goal provable from some set of definite clauses. This is because a consequence of our theorem is that if this term is an answer substitution then there is also another λ-term that does not use implications or universal quantification that can be used to satisfy the given goal. If an understanding of a richer set of predicate constructions is desired, then one course is to leave definite clause logic for a stronger logic. An alternative approach, which we use in Section 6, is to represent predicates as function terms whose types do not involve `o`. This, of course, means that such predicate constructions could not be the head of goals. Hence, additional definite clauses would be needed to interpret the meaning of these encoded predicates.

5. A Simple Parsing Example

The enriched term structure of λProlog provides two facilities that are useful in certain contexts. The notion of λ-abstraction allows the representation of binding a variable over a certain expression, and the notion of application together with λ-contraction captures the idea of substitution. A situation where this might be useful is in representing expressions in first-order logic as terms, and in describing logical manipulations on them. Consider, for example, the task of representing the formula $\forall x \exists y (P(x,y) \supset Q(y,x))$ as a term. Fragments of this formula may be encoded into first-order terms, but there is a genuine problem with representing the quantification. We need to represent the variable being quantified as a genuine variable, since, for instance, instantiating the quantifier involves substituting for the variable. At the same time we desire to distinguish between occurences of a variable within the scope of the quantifier from occurences outside of it. The mechanism of λ-abstraction provides the tool needed to make such distinctions. To illustrate this let us consider how the formula above may be encoded as a λ-term. Let the primitive type `b` be the type of terms that represent first-order formulas. Further let us assume we have the constants `&` and `=>` of type `b -> b -> b`, and `all` and `some` of type `(i -> b) -> b`. These latter two constants have the type of generalized quantifiers and are in fact used to represent quantifiers. The λ-term `(all X\(some Y\(p X Y => q Y X)))` may be used to represent the above formula. The type `b` should be thought of as a term-level encoding of the boolean type `o`.

A more complete illustration of the facilities alluded to above may be provided by considering the task of translating simple English sentences into logical forms. As an example, consider translating the sentence "Every man loves a woman" to the logical form

$$\forall x(man(x) \supset \exists y(woman(y) \wedge loves(x,y)))$$

which in our context will be represented by the λ-term

```
(all X\(man X =>
       (some Y\(woman Y & loves X Y))))
```

A higher-order version of a DCG [10] for performing this task is provided below. This DCG draws on the spirit of Montague Grammars. (See [11] for a similar example.)

```
sentence (P1 P2)    --> np P1, vp P2, [.].
np (P1 P2)          --> determ P1, nom P2.
np P\(P X)          --> propernoun X.
nom P               --> noun P.
nom X\(P1 X & P2 X) --> noun P1, relcl P2.
vp X\(P2 (P1 X))    --> transverb P1, np P2.
vp P                --> intransverb P.
relcl P             --> [that], vp P.
determ P1\P2\(all X\(P1 X => P2 X)) -->
   [every].
determ P1\P2\(P2 (iota P1)) --> [the].
determ P1\P2\(some X\(P1 X & P2 X)) --> [a].

noun man          --> [man].
noun woman        --> [woman].
propernoun john   --> [john].
propernoun mary   --> [mary].
transverb loves   --> [loves].
transverb likes   --> [likes].
intransverb lives --> [lives].
```

We use above the type `token` for English words; the DCG translates a list of such `tokens` to a term of some corresponding type. In the last few clauses certain constants are used in an overloaded manner. Thus the constant `man` corresponds to two distinct constants, one of type `token` and another of type `i -> b`. We have also used the symbol `iota` that has type `(i -> b) -> i`. This constant plays the role of a definite description operator; it picks out an individual given a description of a set of individuals. Thus, parsing the sentence "The woman that loves john likes mary" produces the term `(likes (iota X\(woman X & loves X john)) mary)`, the intended meaning of which is the predication of the relationship of liking between an object that is picked out by the description `X\(woman X & loves X john))` and `mary`.

Using this DCG to parse a sentence illustrates the role that abstraction and application play in realizing the notion of substitution. It is interesting to compare this DCG with the one in Prolog that is presented in [10]. The first thing to note is that the two will parse a sentence in nearly identical fashions. In the first-order version, however, there is a need to explicitly encode the process of substitution, and considerable ingenuity must be exercised in devising grammar rules that take care of this process. In contrast in λProlog the process of substitution and the process of parsing are handled by two distinct mechanisms, and consequently the resulting DCG is more perspicuous and so also easier to extend.

The DCG presented above may also be used to solve the inverse problem, namely that of obtaining a sentence given a logical form, and this illustrates the use of higher-order unification. Consider the task of obtaining a sentence from the logical form `(all X\(man X => (some Y\(woman Y & loves X Y))))`. This involves unifying the above form with the expression `(P1 P2)`. One of the unifiers for this is

```
P1 --> P\(all X\(man X => P X))
P2 --> X\(some Y\(woman Y & loves X Y).
```

Once this unifier is picked, the task then breaks into that of obtaining a noun phrase from `P\(all X\(man X => P X))` and a verb phrase from `X\(some Y\(woman Y & loves X Y)`. The use of higher-order unification thus seems to provide a top-down decomposition in the search for a solution. This view turns out to be a little simplistic however, since unification permits more structural decompositions than are warranted in this context. Thus, another unifier for the pair considered above is

```
P1 --> Z\(all Z)
P2 --> X\(man X =>
         (some Y\(woman Y & loves X Y)))
```

which does not correspond to a meaningful decomposition in the context of the rest of the rules. It is possible to prevent such decompositions by anticipating the rest of the grammar rules. Alternatively decompositions may be eschewed altogether; a logical form may be constructed bottom-up and compared with the given one. The first alternative detracts from the clarity, or the specificational nature, of the solution. The latter involves an exhaustive search over the space of all sentences. The DCG considered here, together with higher-order unification, seems to provide a balance between clarity and efficiency.

The final point to be noted is that the terms that are produced at intermediate stages in the parsing process are logically meaningful terms, and computations on such terms may be encoded in other clauses in our language. In Section 7, we show how some of these terms can be directly interpreted as frame-like objects.

6. Knowledge Representation

We now consider the question of how a higher-order logic might be used for the task of representing knowledge. Traditionally, certain network based formalisms, such as KL-ONE [4], have been described for this purpose. Such formalisms use nodes and arcs in a network to encode knowledge, and provide algorithms that operate on this network in order to perform inferences on the knowledge so represented. The nature of the information represented in the network may be clarified with reference to a logic, and the correctness of the algorithms is often proved by showing that they perform certain kinds of logical inference on the underlying information. Our approach here is to encode the relevant notions by using λ-terms that directly correspond to their logical nature, and to use definite clauses to specify logical inferences on these notions. We demonstrate this approach below through a few examples.

A key notion in knowledge representation is that of a *concept*. KL-ONE provides the ability to define *primitive* roles and concepts and a mechanism to put these together to define more complex concepts. The intended interpretation of a role is a two place relation, and of a concept is a set of objects characterized by some defining property. An appropriate logical view of a concept, therefore, is to identify it with a one-place predicate. A particularly apt way of modeling the connection between a concept and a predicate is to use λ-terms of a certain kind to denote concepts. The following set of clauses that are used to define concepts modelled after examples in [4] serves to make this clear.

```
prim_role recipient.
prim_role sender.
prim_role supervisor.
prim_concept person.
prim_concept crew.
prim_concept commander.
prim_concept message.
prim_concept important_message.

role R :- prim_role R.
concept C :- prim_concept C.
concept (X\(C1 X & C2 X)) :-
   concept C1, concept C2.
concept (X\(all Y\(R X Y => C1 Y))) :-
   concept C1, role R.
```

The type of `prim_role` and `role` in the above example is `(i -> i -> b) -> o` and of `prim_concept` and `concept` is `(i -> b) -> o`. Any term that can be substituted for `R` so as to make `(role R)` provable from these clauses is considered a *role*. Similarly, any term that can be substituted for `C` so as to make `(concept C)` provable is considered a *concept*. The first three clauses serve to define primitive roles in this sense, and the next five clauses define primitive concepts. The remaining clauses describe a mechanism for constructing further roles and concepts. As can be readily seen, all roles are primitive roles. An example of a complex concept is provided by the term

```
(X\(message X &(all Y\(sender X Y => crew Y))))
```

which may be described by the noun phrase "messages all of whose senders are crew members."

One of the purposes for providing a representation for concepts is so that inferences that involve them can be described. One kind of inference that is of particular interest is that of determining *subsumption*. A concept C_1 is said to subsume another concept C_2 if every element of the set described by C_2 is a member of the set described by C_1. Given our representation of concepts, the question of whether C_1 subsumes C_2 reduces to the question of whether $\forall x(C_2(x) \supset C_1(x))$ is valid (*i.e.* provable). Such an inference may be based either on certain primitive containment relations, or on an analysis of the structure of the terms used to denote concepts. The following set of clauses make these ideas precise:

```
subsume person crew.
subsume (X\(all Y\(sender X Y => person Y)))
        message.
subsume (X\(all Y\(recipient X Y => crew Y)))
        message.
subsume message important_message.
subsume (X\(all Y\(sender X Y => commander Y)))
        important_message.

subsume C C.
subsume A B :- subsume A C, subsume C B.
subsume (Z\(A Z & B Z)) C :-
    subsume A C, subsume B C.
subsume A (Z\(B Z & C Z)) :- subsume A B.
subsume A (Z\(B Z & C Z)) :- subsume A C.
subsume (Z\(all (Y\(R Z Y => A Y))))
        (Z\(all (Y\(R Z Y => B Y)))) :-
    subsume A B.
```

The first few clauses specify certain primitive containment relations; thus the first clause states that the set described by `crew` is contained in the set described by `person`. The later clauses specify subsumption relations based on these primitive ones and on the logical structure of the terms describing the concepts. One of the virtues of our representation now becomes clear: It is easy to see that the above set of clauses correctly specifies the relation of subsumption. If `A` and `B` are two terms that represent concepts, then rather elementary proof-theoretic arguments may be employed to show that `(subsumes A B)` is provable from the above clauses if and only if the first-order term `(all X\(B X => A X))` is logically entailed by the primitive subsumption relations. Furthermore, any sound and complete interpreter for λProlog (such as one searching breath-first) may be used together with these clauses to provide a sound and complete subsumption algorithm.

Another kind of inference that is often of interest is that of determining whether an object `a` is in the set of objects denoted by a concept `C`. This question reduces to whether `(C a)` is a theorem. This inference may be encoded in definite clauses in the manner illustrated below:

```
fact (important_message m1).
fact (sender m1 kirk).
fact (recipient m1 scotty).

interp A :- fact A.
interp (A & B) :- interp A, interp B.
interp (C U) :-
    subsume (X\(all Y\ (R X Y => C Y))) D,
    fact (R V U), interp (D V).
interp (C U) :- subsume C D, interp (D U).
```

In the clauses above, `fact` and `interp` are predicates of type `b -> o`. The first few clauses state which formulas of type `b` should be considered true; `(fact X)` may be read as an assertion that `X` is true. The last few clauses define `interp` to be a theorem-prover that uses `subsume` and `fact` to deduce additional formulas of type `b`. The only clause that may need to be explained here is the third one pertaining to `interp`. This clause may be explained as follows. Let `(D V)` and `(subsume (X\(all Y\ (R X Y => C Y))) D)` be true. By virtue of the meaning of subsumption, `((X\(all Y\ (R X Y => C Y))) V)`, i.e. `(all Y\ (R V Y => C Y))`, is true. From this it follows that for any `U` if `(R V U)` is true then so is `(C U)`. Given the clauses in this section, some of the inferences that are possible are the following: `kirk` is a `person` and a `commander`, and `scotty` is a `crew` and a `person`. That is, `(interp (person kirk))`, for example, is provable from these definite clauses.

7. Syntax and Semantics in Parsing

In Section 5, we showed how sentences and phrases could be translated into logical forms that correspond to their meaning. Such logical forms are well defined objects in our language and in Section 6 we illustrated the possibility of defining logical inferences on such objects. There are parsing problems which require semantical analysis as well as syntactic analysis and our language provides the ability to combine such analyses in one computational framework. A common approach in natural language understanding systems is to use one computational paradigm for syntactic analysis (*e.g.* DCGs, ATNs) and another one for semantic analysis (*e.g.* frames, semantic nets). An integration of these two paradigms is often difficult to explain in a formal sense. Using the approach that we suggest here also results in the syntactic and semantic processing being done at two different levels: one is first-order and the other is higher-order. Bridging these two levels, however, can be very natural. For example, the query (see Section 4)

```
rel R, R john mary
```

mixes both aspects. The process of determining a suitable instantiation for `R` is second-order, while the process of determining whether or not `(R john mary)` is provable is first-order.

The problem of determining referents for pronouns provides an example where such an intermixing of levels is necessary, since possible referents for a pronoun must be checked for membership in the `male` or `female` concepts. For example, consider the following sentences: "John likes Mary. She loves him." The problem here is that of identifying "she" with Mary and "him" with John. This processing could be done in the following fashion: First, a DCG similar to the one in Section 5 could be written which returns not only the logical form corresponding to a sentence but also a list of possible referents for pronouns that occur later. In this example, the list of proper nouns `[john, mary]` would be returned. When pronouns are encountered, the DCG would substitute some male or female elements from this list, depending on the gender of the pronoun. The process of selecting an appropriate referent may be accomplished with the following clauses:

```
prim_concept male.
prim_concept female.
fact (female mary).
fact (male john).
select G X [X|L] :- interp (G X).
select G X [Y|L] :- select G X L.
```

A call to the goal `(select female X [john, mary])` would result in picking `mary` as a female from the set of proper nouns. This is, of course, a very simple example. This framework, however, supports the following extension.

Let sentences contain definite descriptions. Consider the following sentences: "The uncle whose children are all doctors likes Mary. She loves him." Here, "him" clearly refers to the uncle whose children are all doctors. In order to modify our above program we need to make only a few additions. First, we need to be able to take a concept, such as "uncle whose children are all doctors" and encode the (unique) individual within it. To do this, we use the definite description operator described in Section 5. Hence, after parsing the first sentence, the list

```
[(iota (X\(uncle X &
           (all Y\(child X Y => doctor Y))))),
 mary]
```

would be returned as the list of possible pronoun references. Consider the following additional definite clauses.

```
prim_concept man.
prim_concept uncle.
prim_concept doctor.
prim_relation child.
subsume male man.
subsume man uncle.
interp (P (iota Q)) :- subsume P Q.
```

The first six clauses give properties to some of the lexical items in this sentence. Only the last clause is an addition to our actual program. This clause, however, is very important since it is one of those simple and elegant ways in which the different logical levels can be related. A term of the form `(iota Q)` represents a first-order individual (*i.e.* some object), but it does so by carrying with it a *description* of that object (the concept `Q`). This description can be invoked by the following inference: the `Q` is a `P` if all `Q`s are `P`s. Hence, checking membership in a concept is transformed into a check for subsumption.

To find a referent for "him" in our example sentences, the goal

```
(select male X
   [(iota (X\(uncle X &
       (all Y\(child X Y => doctor Y))))),
    mary])
```

would be used to pick the male from the list of possible pronoun references. (Notice here that `X` occurs both free and bound in this query.) In attempting to satisfy this goal, the goal

```
(interp
  (male (iota (X\(uncle X &
           (all Y\(child X Y => doctor Y)))))))
```

and then the goal

```
(subsume male
   (X\(uncle X &
       (all Y\(child X Y => doctor Y)))))
```

would be attempted. This last goal is clearly satisfied providing a suitable referent for the pronoun "him."

8. Compiling into First-Order Logic

We have suggested that higher-order logic can be used to provide a formal specification and justification of certain computations involving meanings and parsing. We have been concerned with *explaining* a logic programming approach to integrating syntactic and semantic processing. Higher-order logic is, of course, not *needed* to perform such computations. In fact, once we have specified algorithms in this higher-order setting, it is occasionally the case that a first-order reimplementation is possible. For example, all the specifications in Section 6 can be transformed or "compiled" into first-order definite clauses. One way of performing such a compilation is to define the following constants to be the corresponding λ-terms:

```
   and     C\D\X\(C X & D X)
 restr     R\C\X\(all Y\(R X Y => C Y))
```

Using these definitions, the clauses for `role`, `concept`, and `subsume` may be rewritten as the following:

```
role R :- prim_role R.
concept C :- prim_concept C.
concept (and C1 C2) :- concept C1, concept C2.
concept (restr R C1) :- concept C1, role R.

subsume C C.
subsume A B :-
    subsume A C, subsume C B.
subsume (and A B) C :-
    subsume A C, subsume B C.
subsume A (and B C) :- subsume A B.
subsume A (and B C) :- subsume A C.
subsume (restr R A) (restr R B) :-
    subsume A B.
```

Introducing the notion of an element of a concept is less straightforward. In order to do this, we need to first differentiate between a fact that states membership in a concept and a fact that states a relationship between two elements. We do this by making the following additional definitions:

```
    is_a     C\X\(fact (C X))
 related     R\X\Y\(fact (R X Y))
```

If we assume that `interp` is only used to decide membership in concepts, then we may replace `(interp (C`

X)) by (is_a C X). The remaining clauses in Section 6 can be translated into the following:

```
is_a important_message m1.
related sender m1 kirk.
related recipient m1 scotty.

is_a (and A B) X :- is_a A X, is_a B X.
is_a C U :- subsume (restr R C) D,
            related R V U, is_a D V.
is_a C U :- subsume C D, is_a D U.
```

The resulting first-order program is isomorphic to the original, higher-order program. The subsumption algorithm in [3] is essentially the one specified by the clauses that define subsume. There are two important points to make regarding this program, however. First, to correctly specify its meaning, one needs to develop the machinery of the higher-order program which we first presented. Second, this latter program represents a *compilation* of the first program. This compilation relys on simplifing the representation of concepts and roles to a point where their logical structure is no longer apparent. As a result, it would be harder to extend this program with new forms of concepts, roles and inferences that involves them. The original program, however, is easy to extend.

Another way to see this comparison is to say that the higher-order program *is* the formal semantics of the first-order program. This way of looking at semantics is very similar to the denotational approach to specifying program language semantics. There, the correct understanding of very simple, low level programming features might involve constructions which are higher-order and functional in nature.

9. Conclusions

Our goal in this paper was to argue that higher-order logic has a meaningful role to play in computational linguistics. Towards this end, we have described a version of definite clauses based on higher-order logic and presented several examples that illustrate their possible use in a natural language understanding system. We have built an experimental, depth-first interpreter for λProlog on which we have tested all the programs that appear in this paper (and many others). We are currently working on the design and implemention of an efficient interpreter for this programming language.

References

[1] Peter B. Andrews, "Resolution in Type Theory," Journal of Symbolic Logic **36** (1971), 414 – 432.

[2] Peter B. Andrews, Dale A. Miller, Eve Longini Cohen, Frank Pfenning, "Automating Higher-Order Logic" in *Automated Theorem Proving: After 25 Years*, AMS Contemporary Mathematics Series **29** (1984).

[3] Ronald J. Brachman, Hector J. Levesque, "The Tractability of Subsumption in Frame-based Description Languages" in the Proceedings of the National Conference on Artificial Intelligence, AAAI 1984, 34 – 37.

[4] Ronald J. Brachman, James G. Schmolze, "An Overview of the KL-ONE Knowledge Representation System," Cognitive Science **9** (1985), 171 – 216.

[5] Alonzo Church, "A Formulation of the Simple Theory of Types," Journal of Symbolic Logic **5** (1940), 56 – 68.

[6] David R. Dowty, Robert E. Wall, Stanley Peters, *Introduction to Montague Semantics*, D. Reidel Publishing Co., 1981.

[7] Michael J. Gordon, Arthur J. Milner, Christopher P. Wadsworth, *Edinburgh LCF*, Springer-Verlag Lecture Notes in Computer Science No. 78, 1979.

[8] Gérard P. Huet, "A Unification Algorithm for Typed λ-calculus," Theoretical Computer Science **1** (1975), 27 – 57.

[9] Dale A. Miller, Gopalan Nadathur, "Higher-order Logic Programming," in the Proceedings of the Third International Logic Programming Conference, Imperial College, London England, July 1986.

[10] F. C. N. Pereira, D. H. D. Warren, "Definite Clause Grammars for Language Analysis – A Survey of the Formalism and a Comparison with Augmented Transition Networks" in Artificial Intelligence **13** (1980).

[11] David Scott Warren, "Using λ-Calculus to Represent Meaning in Logic Grammars" in the Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics, June 1983, 51 – 56.