# HILOG: A FOUNDATION FOR HIGHER-ORDER LOGIC PROGRAMMING

WEIDONG CHEN,* MICHAEL KIFER, AND DAVID S. WARREN

▷ We describe a novel logic, called HiLog, and show that it provides a more suitable basis for logic programming than does traditional predicate logic. HiLog has a higher-order syntax and allows arbitrary terms to appear in places where predicates, functions, and atomic formulas occur in predicate calculus. But its semantics is first-order and admits a sound and complete proof procedure. Applications of HiLog are discussed, including DCG grammars, higher-order and modular logic programming, and deductive databases. ◁

## 1. PREFACE

Manipulating predicates, functions, and even atomic formulas is commonplace in logic programming. For example, Prolog combines predicate calculus, higher-order, and meta-level programming in one working system, allowing programmers routine use of generic predicate definitions (e.g., transitive closure, sorting) where predicates can be passed as parameters and returned as values [7]. Another well-known useful feature is the "call" meta-predicate of Prolog. Applications of higher-order constructs in the database context have been pointed out in many works, including [24, 29, 41].

Although predicate calculus provides the basis for Prolog, it does not have the wherewithal to support any of the above features, which, consequently, have an ad hoc status in logic programming. In this paper, we investigate the fundamental principles underlying higher-order logic programming and, in particular, shed new light on why and how these Prolog features appear to work in practice. We propose

*Present address: Computer Science and Engineering, Southern Methodist University, Dallas, TX 75275.

*Address correspondence to* Michael Kifer and David Warren, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794.

Received May 1992.

*THE JOURNAL OF LOGIC PROGRAMMING*

a novel logic, called HiLog, which provides a clean declarative semantics to much of the higher-order logic programming.

From the outset, even the terminology of "higher-orderness" seems ill-defined. A number of works have proposed various higher-order constructs in the logic framework [1, 5, 13, 7, 23, 24, 30, 31, 42, 45], but with such a diversity of syntax and semantics, it is not always clear what kind of higher-orderness is being claimed. In our opinion, there are at least two different facets to the issue: a higher-order *syntax* and a higher-order *semantics*. Logicians seem to have been long aware of this distinction and, for example, Enderton [16, p. 282] describes a translation of the syntactically second-order predicate calculus into a first-order multisorted logic that admits a first-order semantics.

Informally, by higher-order *syntax*, logicians mean a language in which variables are allowed to appear in places where normally predicate and/or function symbols do. In contrast, higher-order *semantics* is manifested by semantic structures in which variables may range over domains of relations and functions constructed out of the domains of individuals. In a first-order semantics, variables can only range over domains of individuals or over the names of predicates and functions.

The third component in this picture is the equality theory that is built into the semantics of the logic. As explained below, an equality theory may bring the first-order and higher-order semantics closer to each other. In the extreme case, this theory may imply a 1–1 correspondence between the domain of individuals and the domains of higher-order constructs. In this case, higher-order and first-order semantics become virtually indistinguishable.

We note that the classification based upon semantics has no implication whatsoever regarding the "intrinsic higher-orderness," that is, whether there exists a sound and complete proof theory for a higher-order semantics. It is quite possible to replace a higher-order semantics for some languages by an entailment-equivalent first-order semantics. On the other hand, it is well known that some semantics (e.g., the standard semantics of second-order predicate calculus) are inherently higher-order, and no equivalent first-order substitute exists for the corresponding languages.

Predicate calculus is the primary example of a logic where syntax and semantics are both first-order. There are logics that have a higher-order syntax, but a first-order semantics. F-logic [24, 25] is one example; HiLog, to be discussed in this paper, also falls into this category. Examples of logics with higher-order syntax and semantics include COL [1] and Church's simple theory of types (under the standard and Henkin's semantics) [13, 23]. On the other hand, LDL [5, 46] is a language with a first-order syntax and a higher-order semantics.

Let us examine the equational theories underlying various logics more closely. Under a higher-order semantics, an equation between predicate (or function) symbols, e.g., $p = q$, is true if and only if these symbols are interpreted via the same relation (resp., function). Another way of saying this is that logics with a higher-order semantics have a built-in *extensional* equality theory of predicates and functions. In contrast, in HiLog and F-logic, predicates and other higher-order syntactic objects are not equal unless they (i.e., their names) are equated explicitly. Thus, it is possible for two predicate symbols, say $p$ and $q$, to be interpreted by the same relation and yet the equality $p = q$ be false.

The above examples represent the two extreme cases, in a sense. There are higher-order languages with a first-order semantics that embody a nontrivial equality theory. An example is a subset of λProlog [42, 45] that has no type variables and whose syntax is essentially the same as Church's simple theory of types [13]. Equality in λProlog corresponds to λ-equivalence and is not extensional: there may exist predicates that are not λ-equivalent, but are still extensionally equal.

To better see the role of equational theories on the classification of logical theories, note that there are two different aspects associated with an expression that denotes a predicate or a function. For instance, in λ-Prolog, one aspect is the *meaning* of the expression as a λ-term (or, more precisely, as an equivalence class of λ-terms), which we call the *intension*; the other aspect is the relation or a function associated with the expression, which we call the *extension*. These two aspects become indistinguishable if extensionality axioms are built into the logic. For instance, in Henkin's semantics of Church's type theory [23], it makes no difference whether predicate variables are considered to range over the domain of interpretation for predicate names (predicate intensions) or over relations (predicate extensions), since each extension now becomes associated with exactly one intension, and vice versa.

The distinction between intensions and extensions is important. It is known that extensions can be notoriously difficult to handle in an efficient manner. Separating intensions from extensions makes it possible to have an equational theory over predicate and function names that is separate from the extensional equality of relations and functions. A logic that avoids an overly strong equational theory of intensions can have a simple first-order semantics, a decidable unification problem, and at the same time, a higher-order syntax.

In a type-free logic, the same term may appear in different contexts as predicates or functions of different arities and even as atomic formulas. Thus, the same intension can be associated with different extensions in different contexts. For instance, in Lambda Calculus [40], a λ-term is considered a function or an object, depending on its syntactic position. In HiLog, the same symbol may denote a predicate, a function, or an atomic formula. Semantics of a type-free logic has to maintain the distinctions between various extensions associated with the same intension. Since the same variable may appear in different contexts and have different extensions, it makes better sense for variables to range over intensions rather than extensions because only intensions remain the same across different contexts.

In this paper, we present a simple logical framework in which predicates, functions, and atomic formulas can be manipulated, as first-class entities. It is quite clear that in order to support such manipulation naturally enough, the syntax has to be higher-order. As explained earlier, this leaves open two possibilities for the semantics. Under higher-order semantics, predicates and functions are identified by their extensions, that is, two predicates represent the same thing if and only if their extensions coincide in every appropriate semantic structure. Unfortunately, extensional equality of predicates and functions is not decidable, in general, which carries over to the unification problem. For genuine second-order theories (e.g., second-order predicate calculus and Church's simple theory of types, both under the standard semantics), extensional equality is not even semi-decidable.

In contrast, under first-order semantics, predicates and functions have intensions that can be manipulated directly. Furthermore, depending on the context, the same intension may assume different roles, acting as a relation, a function, or even a proposition. The logical consequence relation under a first-order semantics is likely to be semi-decidable, unless a too strong equality theory destroys this property. This observation motivates our choice for HiLog.[1] The basic idea is to construct a logic that distinguishes between intensional and extensional aspects of the semantics and embodies only a trivial equality theory of intensions. Intensions can be thought of as names of abstract or concrete entities, while extensions correspond to various roles these entities play in the real world. It has also been argued by Maida and Shapiro [36, 37] that knowledge representation is part of the conceptual structure of cognitive agents, and therefore should not (and even cannot) contain extensions. The reason is that cognitive agents do not have direct access to the world, but only to one of its representations. Our approach is in the same spirit and, consequently, extensions of predicates and functions are not available for direct manipulation. On the other hand, intensions of higher-order entities such as predicates, functions, and even atomic formulas can be freely manipulated. Their extensions come into play only when the respective expressions need to be evaluated. Thus, HiLog combines advantages of higher-order syntax with the simplicity of first-order semantics, benefiting from both.

The rest of the paper is organized as follows. After a brief motivational discussion, we formally present the syntax and semantics of HiLog. Then we give several examples of applying HiLog to higher-order logic programming, DCG grammars, and modular logic programming. After that, we discuss the relationship of HiLog to predicate calculus; its relationship to some of the recently proposed database languages is discussed in Section 5. A resolution-based proof theory of HiLog is described in Section 6.

## 2. SYNTAX AND SEMANTICS OF HILOG

### 2.1. Motivation

Prolog syntax is quite flexible; it allows symbols to assume different roles depending upon their context (e.g., to have different arities, or to be viewed as a constant, a function, or a predicate). For instance, in the clause

$$\mathsf{r(X) \leftarrow p(X,f(a)),q(p(X,f(a)),f(p,b)).}$$

the symbol f occurs both as a unary and as a binary function, and p appears as a binary predicate as well as a binary function symbol. Furthermore, the same syntactic object, p(X, f(a)), is an atomic formula in the first literal of the clause-body and an individual term in the second.

The support for multiple roles for nonlogical parameters (constants, functions, and predicates) is a handy feature of Prolog, but unfortunately, this support is provided in a rather ad hoc way. For instance, while in the above example different occurrences of the same symbol can be semantically disambiguated simply by

[1]This rule of thumb has exceptions, though. For instance, type theory under Henkin's semantics has a proof theory despite the fact that Henkin's semantics is higher-order. In contrast, for certain strains of first-order annotated logics [26, 27], complete proof procedures may not exist, in general.

renaming the occurrences of f and p, this cannot be done in the following rule:

```
p(X,Y) ← q(Y),X.
```

Here, the individual variable X occurs as a first-order term and as an atomic formula, and renaming its different occurrences will, intuitively, yield a semantically different statement.

HiLog supports multiple roles for parameter symbols in a much more general and elegant manner. Parameters are arityless, and the distinction among predicate, function, and constant symbols is eliminated. In particular, a HiLog term can be constructed from *any* logical symbol followed by *any* finite number of arguments. Different occurrences of the same parameter are related to the *same object* characterized by the *same intension*. Associated with such an intension are several different *extensions* that capture the different roles the symbol may assume in different contexts.

The same view is extended to arbitrary terms. HiLog allows complex terms (not just parameter symbols) to be viewed as functions, predicates, and atomic formulas. For example, a *generic* transitive closure predicate can be defined as follows:

```
closure(R)(X,Y) ← R(X,Y).
closure(R)(X,Y) ← R(X,Z), closure(R)(Z,Y).
```

Here, closure is (syntactically) a second-order function which, given any relation R, returns its transitive closure closure(R). Generic definitions can be used in various ways. For instance,

```
parent(john, bill).
parent(bill, bob).
manager(john, mary).
manager(mary, kathy).
relation(parent).
relation(manager).
reports_to(Person)(Superior) ← relation(Relname),
        closure(Relname)(Person, Superior).
```

will return {bill, mary, bob, kathy} in response to the query ?- reports_to(john)(X), which is the set of john's ancestors and bosses.

Often, various applications of Prolog require term traversal. In HiLog, binary terms can be traversed as follows:

```
traverse(X(Y,Z)) ← traverse(Y), traverse(Z).
```

Notice that X is a variable that ranges over functions of the language, and therefore this program is independent of the alphabet of that language. In contrast, in Prolog, one would have to either specify such a rule for every functor in the alphabet of the program, e.g.,

```
traverse(f(Y,Z)) ← traverse(Y), traverse(Z).
traverse(g(Y,Z)) ← traverse(Y), traverse(Z).
...   ...   ...
```

or use a built-in predicate arg/3, as in

```
traverse(Tree) ←
        arg(1, Tree, Left), arg(2, Tree, Right),
        traverse(Left), traverse(Right).
```

In the latter, the structure of Tree is less visible and even is not enforced (unless we add another subgoal for checking the total number of arguments).

In databases, schema browsing is common [41]. In HiLog, browsing can be performed through the same query language as the one used for data retrieval. For instance,

```
relations(Y)(X) ← X(Y, Z).
relations(Z)(X) ← X(Y, Z).
?- relations(john)(X).
```

will return the names of all binary predicates whose extensions (under Herbrand interpretations) contain one or more tuples with the token *john*. HiLog shares this browsing capability with another recently proposed language, called F-logic [24, 25].

The ability to use sets is very important in logic programming. Prolog supports the use of sets by means of the setof construct, which has only an operational semantics. Although newer logic languages, such as LDL [5, 46] or COL [1], do provide semantics for sets, they have to severely restrict their use to avoid logical paradoxes and/or computational intractability. In HiLog, sets can be represented naturally by parameterized predicates mentioned earlier. For instance, the following HiLog rule defines groups of satisfied employees working for each manager. An employee is satisfied with the job if and only if he earns more than his boss:

```
sat_empl(Boss)(Empl) ← supervises(Boss, Empl),
                       sal(Boss, B_sal), salary(Empl, E_sal),
                       E_sal > B_sal.
```

The set-term, sat_empl(Boss) is akin to the setof and the grouping constructs of Prolog and LDL, respectively, although HiLog sets are not constructed or used the same way. Having defined sets intensionally, via terms, we can go on and use these sets in other relationships, as in the following fragment that associates packages of benefits with groups of satisfied employees:

```
package1(health_ins).
package1(life_ins).
package2(free_car).
package2(long_vacations).
benefits(package1, sat_empl(john)).
benefits(package2, sat_empl(bob)).
```

The following is a query regarding benefit packages that bob enjoys as an employee in the above enterprise:

```
?- benefits(X, Y), Y(bob).
```

Note that the above information about sets can be also encoded in Prolog, but less naturally. A more detailed discussion of these issues appears in Sections 4, 5, and 8.

As seen from the earlier examples, HiLog terms are also atomic formulas. In this capacity, their semantics is indirectly captured by the truth value assigned to each ground term. For instance, instead of saying that a pair $\langle a, b\rangle$ is in the relation for a predicate $p$, we say that the term $p(a, b)$ denotes a *true proposition*. Formal details are provided in the next two subsections.

## 2.2. Language

In addition to parentheses, connectives, and quantifiers, the alphabet of a language $\mathcal{L}$ of HiLog contains a countably infinite set $\mathcal{V}$ of variables and a countable set $\mathcal{S}$ of parameter symbols. We assume that $\mathcal{V}$ and $\mathcal{S}$ are disjoint.

The set $\mathcal{T}$ of HiLog *terms* of $L$ is a minimal set of strings over the alphabet satisfying the following conditions:

- $\mathcal{V} \cup \mathcal{S} \subseteq \mathcal{T}$;
- if $t, t_1, \dots, t_n$ are in $\mathcal{T}$, then $t(t_1, \dots, t_n) \in \mathcal{T}$, where $n \geq 1$.

Notice that according to this definition, a term can be applied to any finite number of terms, and parameter symbols are arityless.

An *atomic formula* is a term. More complex formulas are built from atomic formulas in the usual way, by means of connectives $\vee$, $\wedge$ (also written as &), $\neg$, and quantifiers $\exists$ and $\forall$.

In addition, we use $\leftarrow$ and $\subset$ to denote the implication, where $(\phi \leftarrow \psi) \equiv (\phi \subset \psi) \equiv (\phi \vee \neg \psi)$. Likewise, $\phi \rightarrow \psi$ and $\phi \supset \psi$ will stand for $\neg \phi \vee \psi$. The bidirectional implication "$\leftrightarrow$" also has the standard definition.

A *literal* is either an atomic formula, a *positive* literal, or the negation $\neg A$ of an atomic formula $A$, a *negative* literal. A *clause* is of the form $\forall X_1 \dots \forall X_k$ $(L_1 \vee \cdots \vee L_n)$, where $L_1, \dots, L_n$ are literals and $X_1, \dots, X_k$ are all the variables occurring in $L_1, \dots, L_n$. It is usually written simply as $L_1 \vee \cdots \vee L_n$. When $n = 0$, a clause is called the *empty* clause. A *Horn clause* is a clause that contains at most one positive literal. A *definite clause* is a Horn clause of the form $A \vee \neg B_1 \vee \cdots \vee \neg B_n$, where $A, B_1, \dots, B_n$ are atomic formulas; definite clauses are usually written as $A \leftarrow B_1, \dots, B_n$, where $A$ is the *head* and the conjunction $B_1, \dots, B_n$ is the *body* of the clause.[2] A *negative* clause is a clause containing no positive literals. A *goal* is a conjunction of literals, and a *query* is the negation of a goal, which can be written as a negative clause. For practical applications, we consider *definite logic programs* that consist of a finite number of definite clauses. Section 4.1 briefly deals with logic programs with negation.

## 2.3. Semantic Structures

Since—at first glance—the semantics of HiLog may seem a bit unusual, we precede the formal description by an informal discussion that shows how the semantics of the first-order predicate calculus has evolved to support higher-orderness of HiLog. First, we define the Herbrand semantics because it is especially close to the Herbrand semantics of predicate calculus.

The *Herbrand universe* for a HiLog language, $\mathcal{L}$, is the set of all *ground* (i.e., variable-free) HiLog terms in $\mathcal{T}$. Since terms in HiLog are also atomic formulas, it

[2]As usual in logic programming literature, we use "," to denote conjunction of literals in the clause body.

follows that Herbrand universe is identical to *Herbrand base* (cf. [35]). This is not a coincidence: if we want to manipulate atomic formulas as terms, the former must be elements of the universe. Now, a *Herbrand interpretation* is simply a subset of the Herbrand base ($\equiv$ universe), the definition of Herbrand interpretation commonly used in logic programming [35]. The definition of formula satisfaction in such interpretations simply repeats the corresponding definition in [35].

Although Herbrand interpretations suffice for many of the needs of logic programming, there are at least two important reasons to be interested in a more general notion of semantic structure [16]: programs with equality and negation-as-failure.

Recall that, in first-order predicate calculus, a semantic structure **M** for a language $\mathcal{L}$ is a pair $\langle U, I \rangle$, where $U$ is called the domain of **M** and $I$ is a function that interprets *parameters* of $\mathcal{L}$ (i.e., constants, function symbols, and predicate symbols). $I$ consists of two different mappings, $\mathcal{F}$ and $\mathcal{P}$. The former, $\mathcal{F}$, interprets each $k$-ary ($k \geq 0$) function symbol of $\mathcal{L}$ by a function $U^k \mapsto U$; the mapping $\mathcal{P}$ maps each $n$-ary predicate symbol to an $n$-ary relation over $U$. We consider constants as 0-ary function symbols, so $\mathcal{F}$ takes care of constants as well. In what follows, whenever $f$ is a function symbol, $I(f)$ will stand for $\mathcal{F}(f)$; when $p$ is a predicate name, $I(p)$ will stand for $\mathcal{P}(p)$. Note that the set of function symbols and the set of predicate symbols are disjoint.

In HiLog, one of our original goals was to eliminate the syntactic distinction among constants, function symbols, and predicates, so one cannot tell a constant from a predicate or a function symbol.[3] Therefore, $\mathcal{F}$ and $\mathcal{P}$ must map *each* parameter symbol to all of the following: an element of $U$, a function over $U$, *and* a relation on $U$. Another of our goals was to make symbols arityless. So, each symbol must be mapped into an *infinite tuple* of functions and relations, one for each arity.

There is a subtlety, however. How should we interpret a formula like $X(a)$, where $X$ is a variable? In predicate calculus, such formulas are interpreted with respect to a variable assignment, which is a mapping $v$: $\mathcal{V} \mapsto U$. However, $v(X)$ is an element of the semantic domain $U$, and in order to interpret $X(a)$ with respect to $v$, we must associate relations with elements of $U$ rather than parameters of $\mathcal{L}$. Thus, the interpreting mappings $\mathcal{F}$ and $\mathcal{P}$ must associate functions and relations with every element in $U$, including $I(s)$, for each $s \in \mathcal{S}$, not with the symbols in $\mathcal{S}$.

Following the usual development in predicate calculus, we can extend a variable assignment $v$ recursively to the set $\mathcal{T}$ of all terms as follows:

- $v(s) = I(s)$ for every parameter symbol in $\mathcal{S}$;
- $v(t(t_1, \ldots, t_n)) = \mathcal{F}(v(t))(v(t_1), \ldots, v(t_n))$.

However, the truth of an atomic formula $t(t_1, \ldots, t_n)$ can be determined either with respect to the $n$-ary relation that $\mathcal{P}$ associates with $v(t)$, that is, $\mathcal{P}(v(t))(v(t_1), \ldots, v(t_n))$, or with respect to the 0-ary relation (i.e., true or false) which it associates with $v(t(t_1, \ldots, t_n))$, that is, $\mathcal{P}(v(t(t_1, \ldots, t_n)))$. Both alternatives seem reasonable, but the second one results in a more uniform treatment of HiLog

[3]This is not to say that we dismiss the utility of typing. To the contrary, we distinguish between well-formedness and typing, which we feel are orthogonal concepts. Our goal here is to relax the stringent well-formedness requirements of predicate calculus, while preserving the ability to talk about type correctness. An attempt to introduce types into HiLog is described in [9].

terms. Furthermore, it turns out that under certain natural assumptions, these two choices lead to the same semantics.

It is easy to see now that adopting the second alternative obviates the need for $\mathcal{P}$. Indeed, all we need is to classify elements of the domain $U$ into the intensions of true and false propositions. A simple way to do this is by introducing a subset $U_{true} \subseteq U$ that designates propositions that are true in **M**. Coming back to Herbrand interpretations discussed at the beginning of this section, we see that—in the Herbrand case—$U_{true}$ is precisely that subset of the Herbrand base that determines the interpretation.

After this informal introduction, we are ready for the formal development. A *semantic structure* for HiLog, **M**, is a quadruple $\langle U, U_{true}, I, \mathcal{F} \rangle$, where

- $U$ is a nonempty set of intensions for the domain of **M**;
- $U_{true}$ is a subset of $U$ that specifies which of the elements in $U$ are intensions of true propositions;
- $I$: $\mathcal{S} \mapsto U$ is a function that associates an intension with each logical symbol;
- $\mathcal{F}$: $U \mapsto \prod_{k=1}^{\infty}[U^k \mapsto U]$ is a function, such that for every $u \in U$ and $k \geq 1$, the $k$th projection of $\mathcal{F}(u)$, also denoted by $u_{\mathcal{F}}^{(k)}$, is a function in $[U^k \mapsto U]$. Here, $\prod$ denotes the Cartesian product of sets and $[U^k \mapsto U]$ is the set of all functions that map $U^k$ to $U$.

Given a semantic structure **M** and a variable assignment $v$: $\mathcal{V} \mapsto U$, we extend it recursively to the set $\mathcal{T}$ of terms as follows:

- $v(s) = I(s)$ for every $s \in \mathcal{S}$;
- $v(t(t_1, \ldots, t_n)) = (v(t))_{\mathcal{F}}^{(n)}(v(t_1), \ldots, v(t_n))$.

Let $\phi$ be an atomic formula. It is *satisfied* by **M** under $v$, denoted $\mathbf{M} \models_v \phi$, if and only if $v(\phi) \in U_{true}$.

We thus see that a HiLog term may represent an individual, a function, a predicate, or an atomic formula in different contexts. The *intension* of a term is its associated element in the universe $U$. *Extensional* aspects of terms are captured as follows. The extensional aspect of a term, viewed as a proposition, is its truth value defined by its membership in $U_{true}$, while the extensional meaning of a term in its capacity as a function is captured by $\mathcal{F}$. For example, if a term $t$ appears as a $k$-ary function, e.g., in $t(t_1, \ldots, t_k)$, its extensional meaning under a variable assignment $v$ is the $k$th projection of the vector of functions associated with the intension of $t$, that is, $(v(t))_{\mathcal{F}}^{(k)}$. On the other hand, the role of the above term as a proposition is indirectly captured through $\mathcal{F}$ and $U_{true}$, so that $t(t_1, \ldots, t_k)$ is true if and only if $v(t(t_1, \ldots, t_k)) \in U_{true}$.

Finally, the meaning of complex formulas is defined in the standard way:

- $\mathbf{M} \models_v (\phi \wedge \psi)$ if and only if $\mathbf{M} \models_v \phi$ and $\mathbf{M} \models_v \psi$;
- $\mathbf{M} \models_v (\phi \vee \psi)$ if and only if $\mathbf{M} \models_v \phi$ or $\mathbf{M} \models_v \psi$;
- $\mathbf{M} \models_v (\neg \phi)$ if and only if $\mathbf{M} \not\models_v \phi$;
- $\mathbf{M} \models_v (\forall X)\phi$ if and only if for *every* variable assignment, $\mu$, that may differ from $v$ only on $X$, $\mathbf{M} \models_\mu \phi$;
- $\mathbf{M} \models_v (\exists X)\phi$ if and only if for *some* variable assignment, $\mu$, that may differ from $v$ only on $X$, $\mathbf{M} \models_\mu \phi$;

For *closed* formulas (i.e., formulas all of whose variables are quantified), $\mathbf{M} \models_v \phi$ does not depend on $v$, and we can write simply $\mathbf{M} \models \phi$.

Interpreted symbols, such as " = ," "true," and "false," can be incorporated into HiLog by requiring that interpretations satisfy the following restrictions:

- $I(\text{true}) \in U_{true}$;
- $I(\text{false}) \notin U_{true}$;
- $I(=)_{\mathcal{F}}^{(k)}(u_1, \ldots, u_k) \in U_{true}$ if and only if $u_1 = u_2 = \cdots = u_k$ (i.e., if all the $u_i$'s denote the same element of $U$).

Compared with other higher-order logics, such as Church's theory of types [13] and λProlog [42, 45], HiLog treats only atomic formulas as first-class objects. Formulas that contain connectives, such as "and," or," or "not," could be build into HiLog by making these connectives into constants and requiring every semantic structure to satisfy the following conditions:

- $I(\text{and})_{\mathcal{F}}^{(2)}(d_1, d_2) \in U_{true}$ if and only if both $d_1$ and $d_2$ are in $U_{true}$, for every $d_1, d_2 \in U$;
- $I(\text{or})_{\mathcal{F}}^{(2)}(d_1, d_2) \in U_{true}$ if and only if $d_1$ or $d_2$ are in $U_{true}$, for every $d_1, d_2 \in U$;
- $I(\text{not})_{\mathcal{F}}^{(1)}(d) \in U_{true}$ if and only if $d$ is not in $U_{true}$, for every $d \in U$.

Alternatively, the above connectives can be defined using HiLog clauses, e.g., and(X, Y):-X, Y. Encoding formulas with quantified variables would require the introduction of λ-abstraction into HiLog, which also can be done, but is beyond the scope of this paper.

## 3. RELATIONSHIP TO PROLOG AND PREDICATE CALCULUS

In this section, we show from various perspectives that HiLog is, in a well-defined sense, a faithful extension of predicate calculus. In the sequel, we will use "PC" as an abbreviation for "predicate calculus." First, we compare HiLog to what we call *contextual* predicate calculus. This is an extension of ordinary, "pure" predicate calculus, and is inspired by the way Prolog permits the use of the same symbol in different contexts (e.g., as a predicate symbol or as a function symbol of different arities). Syntactically, formulas in contextual PC are also well-formed in HiLog, and therefore there is a question of whether the validity problems for such formulas under the semantics of HiLog and contextual PC coincide. We show that this is not always the case, but under certain conditions, the two validity problems do coincide. In the second subsection, we consider formulas in pure predicate calculus. Such formulas are also well-formed in contextual PC, and therefore also in HiLog. We then establish that the validity problem in pure PC coincides with the same problem in HiLog for an even larger class of formulas than in the case of contextual PC. The third subsection shows that *every* HiLog formula can be encoded as a formula in pure PC.

### *3.1. Validity in Contextual Predicate Calculus and HiLog*

In Prolog, the same symbol may appear in different contexts as a predicate, a constant, or a function symbol; the same symbol can even occur with different arities in different parts of the program. The exact role played by an occurrence of a symbol depends on the context in which this symbol occurs. Such Prolog programs are normally understood as formulas in predicate calculus in which different occurrences of the same symbol are replaced by different function or predicate symbols (whichever is appropriate) of suitable arities. This ad hoc use of parameter symbols in multiple roles motivates a simple extension of PC, which we call *contextual* predicate calculus.

A language $\mathcal{L}$ of contextual PC contains a set of parameter symbols, $\mathcal{S}$, and a set of variables, $\mathcal{V}$. A term is either a variable, a parameter symbol, or an expression of the form $s(t_1,\ldots,t_n)$, where $s \in \mathcal{S}$, $n > 0$, and $t_1,\ldots,t_n$ are terms. An *atomic formula* is any term other than a variable, and the rest of the definitions are as usual.

Clearly, contextual PC is an extension of PC in the direction of HiLog, but is not as radical. It stops short of introducing a higher-order syntax (because $s$ above is an element of $\mathcal{S}$, not a variable). Every formula in contextual PC is also a formula in HiLog. On the other hand, $X$ and $X(a)(b)$ are well-formed formulas in HiLog, but not in contextual PC. Similarly, every PC formula is also a formula in contextual PC, but not vice versa, e.g., $s(s,s)$ is a formula in contextual PC, but not in PC.

Semantics of contextual PC is also a middle ground between HiLog and pure PC. A semantic structure for a contextual PC language is a tuple $\mathbf{M} = \langle U, I_{\mathcal{F}}, I_{\mathcal{P}} \rangle$. For each symbol $s \in \mathcal{S}$, the function $I_{\mathcal{F}}$ associates an infinite tuple $\langle f_0, f_1, f_2, \ldots \rangle$, where $f_0$ is an element of $U$, and for $i > 0, f_i$ is a function $U^i \mapsto U$. Likewise, $I_{\mathcal{P}}(s)$ is a tuple $\langle p_0, p_1, p_2, \ldots \rangle$, where $p_0$ is a 0-ary relation, $p_1$ is a unary relation, etc. We will denote the $k$th components of the above infinite tuples by $I_{\mathcal{F}}^{(k)}(s)$ and $I_{\mathcal{P}}^{(k)}(s)$, respectively. (Recall that a 0-ary relation is a set of 0-ary tuples, and that there is only one such tuple—the empty tuple $\langle\ \rangle$. Therefore, there are only two 0-ary relations: the empty one and the relation $\{\langle\ \rangle\}$. The former is used to interpret false propositions, while the latter interprets true propositions.)

Given a variable assignment $v$: $\mathcal{V} \mapsto U$, we define $v(s) = s$, whenever $s \in \mathcal{S}$, and $v(s(t_1,\ldots,t_n)) = I_{\mathcal{F}}^{(n)}(s)(v(t_1),\ldots,v(t_n))$. If $s(t_1,\ldots,t_n)$ is an atomic formula, $\mathbf{M} \models_v s(t_1,\ldots,t_n)$ if and only if $\langle v(t_1),\ldots,v(t_n)\rangle \in I_{\mathcal{P}}^{(n)}(s)$. The rest of the definition of $\models$ is the same as for PC and HiLog.

Although there are contextual PC formulas that are not well-formed in PC, any such formula, $\phi$, can be *transformed* into a formula $\phi'$ in PC by replacing occurrences of each parameter symbol as follows. All occurrences of $s$ as a $k$-ary function are replaced by a new $k$-ary function symbol $s_{\mathcal{F}}^k$; all occurrences of $s$ in the position of a $k$-ary predicate are replaced by a new predicate symbol $s_{\mathcal{P}}^k$. For instance, under this transformation, $s(s, t(t))$ becomes $s_{\mathcal{P}}^2(s_{\mathcal{F}}^0, t_{\mathcal{F}}^1(t_{\mathcal{F}}^0))$. This transformation is explicit in Prolog where parameter symbols can assume multiple roles. A justification for using this transformation stems from the following easy fact.

*Lemma 3.1.*

1) *A formula in contextual PC is valid if and only if its transformation into pure PC is valid.*
2) *A formula in PC is valid if and only if it is also valid when considered under the semantics of contextual PC.*

PROOF. Let $\mathcal{L}$ be a language of contextual PC, and let $\mathcal{L}'$ be the corresponding language of pure PC in which the transformed formulas are expressed. That is, if $\mathcal{S}$ is a set of parameters of $\mathcal{L}$, then the set of function symbols of $\mathcal{L}'$ is $\{s_{\mathcal{F}}^k | k \geq 0,$

$s \in \mathcal{S}\}$, where the superscript indicates the arity; $\{s^k_{\mathcal{P}} | k \geq 0,\ s \in \mathcal{S}\}$, is the set of predicate symbols of $\mathcal{L}'$.

The proof of 1) is then carried out by constructing an isomorphism $\mathcal{M}$ between the sets of semantic structures for $\mathcal{L}$ and $\mathcal{L}'$ such that if $\mathbf{M} = \langle U, I_{\mathcal{F}}, I_{\mathcal{P}} \rangle$ is a semantic structure for $\mathcal{L}$, $\phi$ is a formula in $\mathcal{L}$, and $\phi'$ is its translation to $\mathcal{L}'$, then $\mathbf{M} \models \phi$ if and only if $\mathcal{M}(\mathbf{M}) \models \phi'$. $\mathcal{M}(\mathbf{M}) = \langle U, I \rangle$ is constructed in an obvious way: $I(s^k_{\mathcal{P}}) = I^{(k)}_{\mathcal{P}}(s)$, and $I(s^k_{\mathcal{F}}) = I^{(k)}_{\mathcal{F}}(s)$.

To prove 2), consider $\mathcal{L}'$, a language of PC. Let $\mathcal{L}''$ be a language of contextual PC with the same sets of parameters and variables. Let $\mathbf{M}' = \langle U, I \rangle$ be a semantic structure of $\mathcal{L}'$. The corresponding semantic structure $\mathbf{M}'' = \langle U, I_{\mathcal{F}}, I_{\mathcal{P}} \rangle$ is constructed thus: if $p$ is a $k$-ary predicate symbol in $\mathcal{L}'$, then $I^{(k)}_{\mathcal{P}}(p) = I(p)$; if $f$ is a $k$-ary function symbol in $\mathcal{L}'$, then $I^{(k)}_{\mathcal{F}}(f) = I(f)$. The rest of $I_{\mathcal{P}}$ and $I_{\mathcal{F}}$ can be arbitrary. It is easy to see that a PC formula $\phi$ in $\mathcal{L}'$ is true in $\mathbf{M}'$ if and only if it is true in $\mathbf{M}''$. Thus, if such a formula is valid when considered as a formula in $\mathcal{L}'$, then it is valid as a formula in $\mathcal{L}'$.

By reversing the above construction, for every semantic structure $\mathbf{M}''$ of the language $\mathcal{L}''$ of contextual PC, one can construct a PC structure $\mathbf{M}'$ for $\mathcal{L}'$ such that a PC formula $\phi$ in $\mathcal{L}'$ is true in $\mathbf{M}''$ if and only if it is true in $\mathbf{M}'$. Thus, if $\phi$ is valid as a PC formula in $\mathcal{L}'$, it is valid as a formula in the contextual PC language $\mathcal{L}''$. □

The above lemma essentially says that contextual PC is only a minor extension of PC. However, because of the greater similarity between the synax of contextual PC and HiLog, it is instructive to investigate the relationship between the validity problems in these two logics. Suppose $\mathcal{S}$ and $\mathcal{V}$ is a set of parameters and variables, respectively. Let $\mathcal{L}_H$ be a language of HiLog, and let $\mathcal{L}_P$ be the sublanguage of contextual PC. In view of Lemma 3.1, it is tempting to think that every formula in $\mathcal{L}_P$ is valid under the semantics of contextual PC if and only if it is valid under the semantics of HiLog. It turns out that the difference between these semantics is more fundamental than in the case of the two flavors of PC. Consider the following formula that is valid in HiLog:

$$(\mathsf{s(a)} = \mathsf{r(b,c)}) \supset (\mathsf{s(a)} \leftrightarrow \mathsf{r(b,c)}) \qquad (1)$$

It is easy to see that this formula is not valid in contextual PC, for the latter can assign relations to the parameters s and r that are not related in any way to the meaning assigned to the terms s(a) and r(b, c). This stands in sharp contrast with HiLog where truth value is assigned to intensions of atoms, and according to the premise of (1), the terms s(a) and r(b, c) do have the same intension; hence, as atomic formulas, they also must have the same truth value.

Nevertheless, there is a correspondence between valid formulas of $\mathcal{L}_P$ in HiLog and contextual PC in the important cases described in Theorem 3.1 below.

*Definition 3.1.* Given a language $\mathcal{L}$ of contextual PC, a semantic structure $\mathbf{M} = \langle U, I_{\mathcal{F}}, I_{\mathcal{P}} \rangle$ is **free** if and only if for every pair of parameter symbols $f$ and $g$ and for any pair of arities $n \geq 0$ and $m \geq 0$, the following holds:

- for all $u_1, \ldots, u_n$ and $v_1, \ldots, v_m$ in $U$, $I^{(n)}_{\mathcal{F}}(f)(u_1, \ldots, u_n) = I^{(m)}_{\mathcal{F}}(g)(v_1, \ldots, v_m)$ if and only if $f$ and $g$ are identical symbols, $n = m$, and $u_i, v_i$ are identical elements in $U$, for all $i$ $(1 \leq i \leq n)$.

*Definition 3.2.* A sentence $\phi$ in contextual PC is **freely-interpretable** if it has the following property:

- if $\phi$ is satisfied by every free semantic structure, then it is satisfied by all semantic structures.

*Theorem 3.1. Let, as before, $\mathcal{L}_H$ be a language of HiLog, and let $\mathcal{L}_P$ be the corresponding sublanguage of contextual PC (i.e., they share the same set $\mathcal{S}$ of parameters and the set $\mathcal{V}$ of variables). Let $\phi$ be a formula in $\mathcal{L}_P$. Then*

1) *if $\models^{cpc} \phi$, then $\models^{hilog} \phi$;*
2) *if $\phi$ is freely-interpretable, then $\models^{hilog} \phi$ if and only if $\models^{cpc} \phi$.*

*Here, $\models^{cpc}$ and $\models^{hilog}$ denote the logical implication relation with respect to the semantics of contextual PC and HiLog, respectively.*

PROOF. We prove both directions of 2). The proof of the "if" direction of 2) is also a proof of 1). To prove 2), suppose that $\models^{hilog} \phi$ holds. We show that $\models^{cpc} \phi$ holds as well. Given a *free* semantic structure $\mathbf{M}_P = \langle U, I_{\mathcal{F}}, I_{\mathcal{P}} \rangle$ for $\mathcal{L}_P$, we construct a semantic structure $\mathbf{M}_H = \langle U, U_{true}, I, \mathcal{F} \rangle$ of $\mathcal{L}_H$ as follows:

- the domain, $U$, is the same as in $\mathbf{M}_P$;
- $I(s) = I_{\mathcal{F}}^{(0)}(s)$ for every symbol $s$ of $\mathcal{S}$;
- for every $u \in U$ and every $k \geq 1$, if $u = I_{\mathcal{F}}^{(0)}(s)$ for some $s \in \mathcal{S}$, then $u_{\mathcal{F}}^{(k)} = I_{\mathcal{F}}^{(k)}(s)$; otherwise, define $u_{\mathcal{F}}^{(k)}(u_1, \ldots, u_k) = d$ for some fixed $d \in U$;
- $u \in U_{true}$ if and only if one of the following conditions holds:
  —$u = I_{\mathcal{F}}^{(0)}(s)$ for some $s \in \mathcal{S}$ such that $I_{\mathcal{P}}^{(0)}(s)$ is a nonempty 0-ary relation[4] (equivalently, if $\mathbf{M}_P \models s$); or
  —$u = I_{\mathcal{F}}^{(k)}(s)(u_1, \ldots, u_k)$ for some $s \in \mathcal{S}$ and $u_i \in U$ $(1 \leq i \leq k)$, such that $\langle u_1, \ldots, u_k \rangle \in I_{\mathcal{P}}^{(k)}(s)$.

The semantic structure $\mathbf{M}_H$ is well-defined since $\mathbf{M}_P$ is free. It can be shown by induction that for any formula $\phi$ of $\mathcal{L}_P$ and any variable assignment $v, \mathbf{M}_P \models_v \phi$ if and only if $\mathbf{M}_H \models_v \phi$. Therefore, if $\models^{hilog} \phi$, then $\phi$ is satisfied in every free semantic structure for $\mathcal{L}_P$. Since $\phi$ is freely-interpretable, it is satisfied in every semantic structure for $\mathcal{L}_P$, that is, $\models^{cpc} \phi$ holds.

To prove the other direction of 2), let us assume that $\models^{cpc} \phi$ holds. We need to show that $\models^{hilog} \phi$ holds as well. Given a semantic structure $\mathbf{M}_H = \langle U, U_{true}, I_H, \mathcal{F} \rangle$ for $\mathcal{L}_H$, we construct a semantic structure $\mathbf{M}_P = \langle U, I_{\mathcal{F}}, I_{\mathcal{P}} \rangle$ for $\mathcal{L}_P$ as follows:

- the domain, $U$, is the same as in $\mathbf{M}_H$;
- for every symbol $s$ of $\mathcal{L}_H$,
  —$I_{\mathcal{F}}^{(0)}(s) = I_H(s)$;
  —$I_{\mathcal{F}}^{(k)}(s) = (I_H(s))_{\mathcal{F}}^{(k)}$, for every $k \geq 1$;
  —$I_{\mathcal{P}}^{(0)}(s)$ is a nonempty 0-ary relation (equivalently, $M_P \models s$) if and only if $I_H(s) \in U_{true}$;
  $\langle u_1, \ldots, u_k \rangle \in I_{\mathcal{P}}^{(k)}(s)$, $k \geq 1$, if and only if $(I_H(s))_{\mathcal{F}}^{(k)}(u_1, \ldots, u_k) \in U_{true}$.

Again it can be shown by induction that for any formula $\phi$ of $\mathcal{L}_P$ and any variable

[4]Recall that the nonempty 0-ary relation interprets true propositions, while the empty relation interprets false propositions.

assignment $v$, $\mathbf{M}_P \models_v \phi$ if and only if $\mathbf{M}_H \models_v \phi$. Therefore, if there exists some $\mathbf{M}_H$ such that $\mathbf{M}_H \not\models_v \phi$, a semantic structure $\mathbf{M}_P$ of $\mathcal{L}_P$ can be constructed such that $\mathbf{M}_P \not\models \phi$, contrary to the assumption. Thus $\models^{hilog} \phi$ holds. □

Although the condition in Theorem 3.1 is not syntactic, it encompasses important classes of formulas. The following lemma shows that all equality-free formulas of predicate calculus are freely-interpretable. The other interesting class that is particularly important for logic programming consists of sets of definite clauses *with* equality, but such that the equality is restricted to clause bodies. If $G$ is a query and is a negative clause and $P$ is a definite logic program (possibly with equality in clause bodies), then evaluating the query amounts to showing that the set $S = G \cup P$ is unsatisfiable. Since Theorem 3.1 concerns validity (which is a contrapositive of unsatisfiability), the class of formulas to consider in this theorem is $\{\neg S | S$ is a conjunction of Horn clauses free of equality in the head$\}$.

*Lemma 3.2. The following classes of contextual PC formulas are freely-interpretable:*

1) *Sets of equality-free sentences.*
2) *Formulas of the form* $\neg S$, *where* $S$ *is a conjunction of Horn clauses free of "$=$" in clause-heads.*

PROOF. Consider a language $\mathcal{L}$ of contextual PC with a set of parameters $\mathcal{S}$. Let $\phi$ be a formula in $\mathcal{L}$ that has one of the forms 1) or 2) above. Assume that for some semantic structure $\mathbf{M} = \langle U, I_{\mathcal{F}}, I_{\mathcal{P}} \rangle$ for $\mathcal{L}$, $\mathbf{M} \not\models \phi$. We show that there exists a *free* semantic structure $\mathbf{M}' = \langle U', I'_{\mathcal{F}}, I'_{\mathcal{P}} \rangle$ such that $\mathbf{M}' \not\models \phi$.

The domain $U'$ of $\mathbf{M}'$ is a free algebra built out of the elements of $U$ and the parameter symbols of $\mathcal{S}$. In other words, it is a minimal set satisfying the following conditions:

- $U \cup \mathcal{S} \subseteq U'$;
- if $d_1, \ldots, d_k$ are in $U'$ and $s \in \mathcal{S}$, then the abstract symbol $s(d_1, \ldots, d_k)$ is in $U'$ for every $k \geq 0$.

For every $s \in \mathcal{S}$, define $I'^{(0)}_{\mathcal{F}}(s) = s$, and for each $k \geq 1$ define $I'^{(k)}_{\mathcal{F}}(s)(d_1, \ldots, d_k) = s(d_1, \ldots, d_k)$ for all $d_1, \ldots, d_k$ in $U'$.

There is a natural mapping $*$: $U' \mapsto U$ defined thus: for every element $u$ in $U'$, the corresponding element $u^*$ in $U$ is such that:

- $u^* = u$, if $u \in U$;
- $s^* = I^{(0)}_{\mathcal{F}}(s)$, if $s \in \mathcal{S}$; and
- $(s(u_1, \ldots, u_k))^* = I^{(k)}_{\mathcal{F}}(s)(u_1^*, \ldots, u_k^*)$.

Next, define $I'^{(k)}_{\mathcal{P}}(s)$ as follows: $\langle u_1, \ldots, u_k \rangle \in I'^{(k)}_{\mathcal{P}}(s)$ if and only if $\langle u_1^*, \ldots, u_k^* \rangle \in I^{(k)}_{\mathcal{P}}(p)$.

By the definition, $\mathbf{M}'$ is a free semantic structure, and the mapping $*$: $U' \mapsto U$ is a *homorphism* of $\mathbf{M}'$ *onto* $\mathbf{M}$ [16]. By the Homomorphism Theorem [16],[5] the

[5]In [16], the Homomorphism Theorem was stated for PC. Its validity for CPC is immediate from Lemma 3.1.

following properties hold:

i) If $v\colon \mathscr{V} \mapsto U'$ is a variable assignment with respect to $\mathbf{M}'$, then there is a variable assignment $v^*\colon \mathscr{V} \mapsto U$ with respect to $\mathbf{M}$, defined as follows: for each variable $X$ of $\mathscr{L}$, $v^*(X) = (v(X))^*$;
ii) For any term $t$ of $\mathscr{L}$, $v^*(t) = (v(t))^*$, where $v^*(t)$ is evaluated with respect to $\mathbf{M}$ while $v(t)$ is evaluated with respect to $\mathbf{M}'$;
iii) The mapping $v \mapsto v^*$ is an epimorphism of variable assignments, i.e., for every variable assignment $\mu$ for $\mathbf{M}$, there is an assignment $v$ for $\mathbf{M}'$ such that $v^* = \mu$;
iv) For every equality-free formula $\phi$, $\mathbf{M}' \models_v \phi$ if and only if $\mathbf{M} \models_{v^*} \phi$.

Obviously, 1) follows from iii) and iv).

To show 2), let $S$ be a conjunction of Horn clauses free of equality in clause heads. To show that $\neg S$ is freely-interpretable, we prove that $\mathbf{M} \not\models_{v^*} \neg S$ implies $\mathbf{M}' \not\models_v \neg S$. The latter is the same as saying that $\mathbf{M} \models_{v^*} S$ implies $\mathbf{M}' \models_v S$, and since $S$ is a conjunction of clauses, we only need to establish this fact for the case of a single clause, $r$, that is, to show that $\mathbf{M} \models_{v^*} r$ implies $\mathbf{M}' \models_v r$.

Let $r$ be a clause of the form $L_1 \vee \cdots \vee L'_n$. Assume that $\mathbf{M} \models_{v^*} r$. Then there exists some literal $L_i$ such that $\mathbf{M} \models_{v^*} L_i$. If $L_i$ is not an equality literal, then $\mathbf{M}' \models_v L_i$, by iii), and thus $\mathbf{M}' \models_v r$. If $L_i$ is an equality literal, then $L_i$ must be of the form $\neg(t_1 = t_2)$, by the assumption. It follows from the definition of the mapping "$*$" that if $v(t_1) = v(t_2)$, then also $v^*(t_1) = v^*(t_2)$ (but the opposite may not be true). Hence, $\mathbf{M} \models_{v^*} L_i$ implies $\mathbf{M}' \models_v L_i$, and also $\mathbf{M}' \models_v r$. $\square$

It is interesting to note that although formulas of the form $\neg S$ in the above lemma are freely-interpretable, $S$ itself may not be freely-interpretable. A simple example is the formula $\mathsf{c} \rightarrow \mathsf{a} = \mathsf{b}$ that is true in every free interpretation, but is not a valid formula in general.

## 3.2. *Validity in Pure Predicate Calculus and HiLog*

Since formulas in pure PC are also formulas in contextual PC, the results of the previous subsection hold for PC formulas as well. However, since the language of PC is a subset of contextual PC, we can obtain a stronger result than the specialization of Theorem 3.1 to PC.

*Definition 3.3.* Let $\mathbf{P}$ be a set of formulas in a PC or a contextual PC language $\mathscr{L}$. Let $\gamma$ denote the cardinality of the set of parameters of $\mathscr{L}$. We say that $\mathbf{P}$ is cardinal with respect to $\mathscr{L}$ if the following property holds:

- If $\mathbf{P}$ is true in every semantic structure $\mathbf{M}$ such that the cardinality of the domain of $\mathbf{M}$ is at least $\gamma$, then $\mathbf{P}$ is true in every semantic structure of $\mathscr{L}$.[6]

*Lemma 3.3. Every freely-interpretable formula in contextual PC is cardinal. In particular, the formulas of the form described in Lemma* 3.2 *are cardinal.*

[6]Implicit here is the assumption that whenever $\mathbf{P}$ is viewed as a PC formula, then $\mathbf{M}$ is a PC structure, and when $\mathbf{P}$ is viewed as a CPC formula, then $\mathbf{M}$ is a CPC structure.

PROOF. For simplicity, we will prove the lemma for languages with finite sets of parameters. It suffices to show that every free semantic structure for a language $\mathcal{L}$ of contextual PC has an infinite domain. Then, if $\phi$ is freely-interpretable, suppose that it is true in every semantic structure of cardinality at least $\gamma$. Then it holds in every infinite and, in particular, every free semantic structure. Hence, since $\phi$ is freely-interpretable, it holds in every semantic structure.

Let $\mathbf{M} = \langle U, I_{\mathcal{F}}, \mathcal{P} \rangle$ be a semantic structure for $\mathcal{L}$, and let $s$ be a parameter in $\mathcal{L}$. Such an $s$ always exists since any language of CPC contains at least one parameter symbol (or else no well-formed formulas can be constructed in this language). Let $u_0 = I(s)$, $u_1 = I_{\mathcal{F}}^{(1)}(s)(u_0)$, $u_2 = I_{\mathcal{F}}^{(1)}(s)(u_1)$, and so on. Then, by the definition of free structures, $u_0$, $u_1$, $u_2$, and so on, are all distinct elements of $U$. $\square$

It is easy to see that the class of cardinal formulas strictly contains the class of freely-interpretable formulas. Lemma 3.3 provides for the inclusion. To show that this inclusion is strict, note that every propositional Horn rule (in PC or CPC) with a nontrivial equality in the body [e.g., $p \leftarrow (a = b)$] is cardinal, but not freely-interpretable.

*Theorem 3.2. Let $\phi$ be a set of formulas in PC (and hence in HiLog). Then*

- *if $\phi$ is valid in PC, then it is valid in HiLog;*
- *if $\phi$ is* cardinal, *then it is valid in HiLog if and only if it is valid in PC.*

Before presenting the proof, we show that the second part of this theorem may not hold if $\phi$ is not cardinal, or it is a CPC but not a PC formula. Consider the following formula:

$$(\mathsf{q}(\mathsf{a}) \leftrightarrow \mathsf{r}(\mathsf{a})) \leftarrow \forall \mathsf{X} \forall \mathsf{Y}(\mathsf{X} = \mathsf{Y}).$$

Clearly, this is a well-formed formula, both in predicate calculus and in HiLog. It is also a valid HiLog formula because whenever the right side of the formula is true in a semantic structure $\mathbf{M} = \langle U, U_{true}, I, \mathcal{F} \rangle$, the domain of this structure must be a singleton element. So, in $\mathbf{M}$, $\mathsf{q}$ and $\mathsf{r}$ are mapped to the same element of $U$. Therefore, $I(\mathsf{q}(\mathsf{a})) = I(\mathsf{r}(\mathsf{a}))$, and the left-hand side of the formula holds true. However, this formula is not valid in predicate calculus, since it is falsified by every semantic structure that has a one-element domain and interprets $\mathsf{q}$ by an empty relation while $\mathsf{r}$ by a nonempty one. One reason for this discrepancy between HiLog and predicate calculus is that the domains of quantification are different: in HiLog, this domain contains $\mathsf{r}$ and $\mathsf{q}$, while in predicate calculus, it does not. Therefore, predicate calculus has, in a sense, more interpretations than HiLog. Another reason is that the truth of atomic formulas is defined via intensions of these formulas in the domain of interpretation rather than via relations associated with predicate symbols.

The second part of Theorem 3.2 hinges in an essential way on the assumption that $\phi$ is a well-formed formula in predicate calculus. It is not true, for example, for formulas in contextual predicate calculus. Consider

$$(\mathsf{s}(\mathsf{a}) \leftrightarrow \mathsf{r}(\mathsf{b})) \leftarrow (\mathsf{s}(\mathsf{a}) = \mathsf{r}(\mathsf{b})). \qquad (2)$$

This formula is not well-formed in PC ($\mathsf{s}$ and $\mathsf{r}$ play multiple roles here), but it is well-formed in contextual PC. It is also a cardinal formula. It is easy to see that (2) is valid in HiLog, but not in contextual PC (the latter is because semantic structures in contextual PC can assign different truth values to the atoms s(a) and r(b), even if the terms s(a) and r(b) have the same intension). Another observation is that (2) is cardinal, but not freely-interpretable (it holds in every free structure because the premise is always false; however, as noted above, this formula is not valid in contextual PC).

Since the class of cardinal formulas in PC is strictly larger than the class of freely-interpretable formulas, Theorem 3.2 strengthens the result of Theorem 3.1 for the class of PC formulas. In fact, since cardinal PC formulas that are valid in HiLog are also valid in PC (Theorem 3.2), it follows that they are also freely interpretable (as are all valid formulas). Thus, Theorems 3.1 and 3.2 together imply that all cardinal, yet nonfreely-interpretable PC formulas are not valid in HiLog and in PC. (This means that these theorems identify the same class of valid PC formulas that are also valid in HiLog. However, Theorem 3.2 identifies a strictly larger class of nonvalid PC formulas that are also nonvalid in HiLog.)

PROOF. Let $\mathscr{L}_P$ be a language of PC with a set of parameters $\mathscr{S}$ and a set of variables $\mathscr{V}$. Let $\mathscr{L}_H$ be a language of HiLog with the same sets of parameters and variables.[7]

We define a pair of mappings:

- *HtoP*: $\mathscr{L}_H$-Structures $\mapsto \mathscr{L}_P$-Structures
- *PtoH*: $\mathscr{L}_P$-Structures $\mapsto \mathscr{L}_H$-Structures

such that for any predicate calculus formula over $\mathscr{L}_P$, the following holds, where $\models^{pc}$ and $\models^{hilog}$ denote logical entailment in predicate calculus and HiLog, respectively.

*Property 1.* If $\mathbf{M}_H$ is a semantic structure for $\mathscr{L}_H$ and *HtoP* $(\mathbf{M}_H) \models^{pc} \phi$, then $\mathbf{M}_H \models^{hilog} \phi$.

*Property 2.* If $\mathbf{M}_P$ is a semantic structure for $\mathscr{L}_P$ such that the cardinality of the domain of $\mathbf{M}$ is at least that of $\mathscr{S}$ and *PtoH* $(\mathbf{M}_P) \models^{hilog} \phi$, then $\mathbf{M}_P \models^{pc} \phi$.

The theorem immediately follows from the existence of the mappings with these two properties.

To construct *HtoP*, let $\mathbf{M}_H = \langle U, U_{true}, I_H, \mathscr{F} \rangle$ be a semantic structure for $\mathscr{L}_H$. Define the corresponding structure for $\mathscr{L}_P$ as follows: *HtoP* $(\mathbf{M}_H) = \langle U, I_P \rangle$ has the same domain, $U$, as $\mathbf{M}_H$, and $I_P$ is defined below:

- $I_P(c) = I_H(c)$, if $c$ is a constant of $\mathscr{L}_P$;
- $I_P(f) = (I_H(f))_{\mathscr{F}}^{(k)}$, if $f$ is a $k$-ary function symbol of $\mathscr{L}_P$; and
- $I_P(p) = \{(u_1, \ldots, u_m) | (I_H(p))_{\mathscr{F}}^{(m)}(u_1, \ldots, u_m) \in U_{true}\}$.

Now, Property 1 follows straightforwardly, by induction on the structure of $\phi$. Note that here we did not impose any restrictions on semantic structures.

To prove Property 2, we define *PtoH* as follows. Let $\mathbf{M}_P = \langle U, I_P \rangle$ be a semantic structure for $\mathscr{L}_P$, where the cardinality of $U$ is at least as high as that of $\mathscr{S}$.

[7] It is important for the following construction that HiLog interpreted parameters, such as "=," "true," and "false," are never used as function symbols in the PC language $\mathscr{L}_P$.

Because of the cardinality assumption, there is a mapping $I_H$ from $\mathcal{S}$ to $U$ such that

- $I_H(c) = I_P(c)$ for every symbol $c$ in $\mathcal{S}$ that corresponds to a constant symbol $c$ of $\mathcal{L}_P$; and
- $I_H(f) \neq I_H(g)$ for every pair of distinct symbols $f$ and $g$ in $\mathcal{S}$ that correspond to function symbols (of arity > 0) or predicate symbols of $\mathcal{L}_P$.

We can now define $PtoH(\mathbf{M}_P) = \mathbf{M}_H = \langle U, U_{true}, I_H, \mathcal{F} \rangle$, where

- the universe, $U$, of $\mathbf{M}_H$ is the same as that of $\mathbf{M}_P$;
- the mapping $I_H$ from the parameters $\mathcal{S}$ of $\mathcal{L}_H$ to $U$ is the one defined in the previous paragraph; and
- $\mathcal{F}$ is defined thus:
  —If $f \in \mathcal{S}$ corresponds to a $k$-ary function symbol of $\mathcal{L}_P$, where $k \geq 1$, then $(I_H(f))_{\mathcal{F}}^{(k)}(u_1, \ldots, u_k) = I_P(f)(u_1, \ldots, u_k)$; for other arities $r$ $(r \neq k)$, $(I_H(f))_{\mathcal{F}}^{(r)}$ is defined arbitrarily.
  —If $p \in S$ is an $m$-ary predicate symbol, where $m \geq 0$, then $(I_H(p))_{\mathcal{F}}^{(m)}$ $(u_1, \ldots, u_m)$ is in $U_{true}$ if and only if $(u_1, \ldots, u_m) \in I_P(p)$; there are no other requirements to the definition of $(I_H(p))_{\mathcal{F}}^{(m)}$. Neither are there any requirements to $(I_H(p))_{\mathcal{F}}^{(s)}$ for other arities $s$, where $s \neq m$.
  —On other elements of $U$, $\mathcal{F}$ is defined in an arbitrary way.

Since $I_H$ maps distinct function and predicate symbols of $\mathcal{S}$ into different elements of $U$, the mappings $(I_H(f))_{\mathcal{F}}^{(k)}$ and $(I_H(p))_{\mathcal{F}}^{(m)}$ are well-defined.

Let $v$ be a variable assignment. Since $\mathcal{L}_H$ and $\mathcal{L}_P$ share the same set of variables and $PtoH(\mathbf{M}_P)$ shares the domain $U$ with $\mathbf{M}_P$, $v$ is a variable assignment for both $PtoH(\mathbf{M}_P)$ and $\mathbf{M}_P$. Let $v_H$ and $v_P$ denote the extensions of $v$ to the set of all terms in $\mathcal{L}_H$ and $\mathcal{L}_P$, respectively. The following properties can be proved by induction:

- for any term $t$ of $\mathcal{L}_P$ in PC (which is also a term in $\mathcal{L}_H$), $v_H(t) = v_P(t)$;
- for any formula $\phi$ of $\mathcal{L}_P$ in PC (which is also a formula in $\mathcal{L}_H$), $PtoH(\mathbf{M}_P)$ $\models_v^{hilog} \phi$ if and only if $\mathbf{M}_P \models_v^{pc} \phi$.

Therefore, if $\phi$ is valid in HiLog, $\phi$ is valid in all semantic structures $\mathbf{M}_P$ such that the cardinality of the domain of $\mathbf{M}_P$ is at least that of $\mathcal{S}$. If $\phi$ is also cardinal, this implies that $\phi$ is true in all semantic structures of $\mathcal{L}_P$. □

## 3.3. *Encoding HiLog in Predicate Calculus*

Since HiLog syntax is richer than that of predicate calculus, it may seem that HiLog is a more expressive logic. It turns out, however, that every HiLog formula can be *encoded* in predicate calculus, and therefore these two logics are equally expressive. It should be borne in mind, though, that our objective is not to devise a more expressive logic, but a logic whose syntax is more suitable for logic programming. In this respect, one can compare programming in HiLog versus Prolog to programming in Prolog versus Horn logic. It is well-known that Horn logic has the computational power of a Turing machine and, therefore, is sufficient for all computational needs. However, the programmer's convenience and the need to simplify problem specification call for a richer syntax (negation, etc.), and this is why Prolog has the additional constructs it is notorious for. More discussion of this issue appears in Section 8.

The following encoding of HiLog in predicate calculus was suggested by Wu [59]. Given a HiLog language $\mathscr{L}_H$ with a set of variables $\mathscr{V}$ and parameters $\mathscr{S}$, we define $\mathscr{L}_P^{encode}$ to be a language of predicate calculus with the set of variables $\mathscr{V}$, constant symbols $\mathscr{S}$, a unique predicate symbol "call," and for each $n \geq 1$, an $(n+1)$-ary function symbol $\mathsf{apply}_{n+1}$. Given a HiLog formula $\phi$, its encoding in predicate calculus, $\phi^*$, is determined by the following recursive transformation rules. In these rules, $\mathsf{encode}_a$ is a transformation that encodes HiLog terms that appear in contexts where they are interpreted as atomic formulas, and $\mathsf{encode}_t$ encodes these terms in all other contexts.

- $\mathsf{encode}_t(X) = X$, for each variable $X \in \mathscr{V}$;
- $\mathsf{encode}_t(s) = s$, for each logic symbol $s \in \mathscr{S}$;
- $\mathsf{encode}_t(t(t_1, \ldots, t_n)) = \mathsf{apply}_{n+1}(\mathsf{encode}_t(t), \mathsf{encode}_t(t_1), \ldots, \mathsf{encode}_t(t_n))$;
- $\mathsf{encode}_a(A) = \mathsf{call}(\mathsf{encode}_t(A))$, where $A$ is a HiLog atomic formula;
- $\mathsf{encode}_a(A \vee B) = \mathsf{encode}_a(A) \vee \mathsf{encode}_a(B)$;
- $\mathsf{encode}_a(A \wedge B) = \mathsf{encode}_a(A) \wedge \mathsf{encode}_a(B)$;
- $\mathsf{encode}_a(\neg A) = \neg \mathsf{encode}_a(A)$;
- $\mathsf{encode}_a((QX)A) = (QX)\mathsf{encode}_a(A)$, where $Q$ is either $\exists$ or $\forall$.

Given a HiLog semantic structure $\mathbf{M} = \langle U, U_{true}, I_H, \mathscr{F} \rangle$, the corresponding predicate calculus structure, $\mathsf{encode}(\mathbf{M}) = \langle U, I_P \rangle$, is defined as follows:

- $I_P(c) = I_H(c)$, for each $c \in \mathscr{S}$;
- $I_P(\mathsf{apply}_{n+1})(u, u_1, \ldots, u_n) = (u)_{\mathscr{F}}^{(n)}(u_1, \ldots, u_n)$;
- $I_P(\mathsf{call}) = U_{true}$;
- the equality predicate " = " has the standard interpretation in $\mathsf{encode}(\mathbf{M})$ (i.e., $I_P(=) \overset{\mathrm{def}}{=} \{\langle u, u \rangle | u \in U\}$).

The following result and its proof are due to Wu [59]. The reason for the special treatment of " = " in the above definitions will become apparent in Section 6.4.

*Theorem 3.3 (Encoding Theorem). Let $\phi$ be a HiLog formula and $\mathbf{M}$ a semantic structure. Let $v$ be a variable assignment for the free variables in $\phi$. Then*

$$\mathbf{M} \models_v \phi \textit{ if and only if } \mathsf{encode}(\mathbf{M}) \models_v \mathsf{encode}_a(\phi).$$

PROOF. By structural induction, $v(t) = v(\mathsf{encode}_t(t))$, for every HiLog term $t$. Consider now a HiLog atomic formula A, other than an equation. By the definition, $\mathsf{encode}_a(\mathsf{A}) = \mathsf{call}(\mathsf{encode}_t(\mathsf{A}))$. Therefore, $\mathbf{M} \models_v \mathsf{A}$

if and only if $v(\mathsf{A}) \in U_{true}$;
if and only if $v(\mathsf{encode}_t(\mathsf{A})) \in I_P(\mathsf{call})$;
if and only if $\mathsf{encode}(\mathbf{M}) \models_v \mathsf{call}(\mathsf{encode}_t(\mathsf{A}))$;
if and only if $\mathsf{encode}(\mathbf{M}) \models_v \mathsf{encode}_a(\mathsf{A})$.

The above derivation holds for the equality predicate as well because of the special interpretation assigned to " = " in Section 2.3.

We have thus proved the claim for atomic formulas. The rest of the proof is an easy induction on the structure of HiLog formulas. □

Let $\alpha$ denote the following sentence: $(\forall X, Y)\mathsf{call}(\mathsf{apply}_3(=, X, Y)) \leftrightarrow X = Y$. Then we have the following corollary that relates the validity of a formula in HiLog and of its encoding in PC.

*Corollary 3.1. Let $\phi$ be a HiLog formula. Then $\phi$ is valid if and only if $\alpha \supset$ $\mathsf{encode}_a(\phi)$ is valid in predicate calculus.*

PROOF. Let $\mathscr{L}_H$ be a HiLog language. It is easy to see that the function encode is a 1–1 and onto mapping from HiLog semantic structures of $\mathscr{L}_H$ to PC semantic structures of $\mathscr{L}_P^{encode}$ that satisfy the aforementioned formula $\phi$ (the language $\mathscr{L}_P^{encode}$ was defined earlier, before the definition of the encoding of HiLog in PC). Indeed, we could use equations for encode "in reverse" to define a function decode such that decode(encode(**M**)) = **M**. The claim now follows from Theorem 3.3. □

We conclude this section with a few remarks that shed some light on the relationship between HiLog and the second-order predicate calculus under the standard semantics [16]. Clearly, one should expect certain similarities because, for example, quantification over predicates and function symbols is allowed in both logics. On the other hand, the first-order nature of HiLog semantics suggests that there must be very significant differences.

For a similarity, it is easy to see that the language of HiLog is rich enough to express so-called *extensionality axioms*. For instance, for arity 2, we can write

$$\forall P \forall Q(\mathsf{eq2}(P, Q) \leftrightarrow \forall X \forall Y(P(X, Y) \leftrightarrow Q(X, Y))) \qquad (3)$$

According to (3), eq2(P, Q) is true if and only if P and Q are the same when viewed as binary predicates. This formula is identical to the extensionality axiom for binary predicates in second-order logic with the standard high-order semantics. Thus, HiLog syntax appears to be rich enough to capture both the intensional and extensional aspects of functions and predicates.

However, a critical difference between the semantics of HiLog and the standard semantics for second-order logic is the domain of variables. In HiLog, variables range over a nonempty set of intensions. So, in (3), P and Q range only over the *intensions* of binary relations, but there also may be relations without the corresponding intensions (the latter do not affect the meaning of (3) in HiLog). In contrast, in the standard second-order semantics, the domain of predicate variables P and Q is the set of *all* binary relations over the individual domain. Therefore, the extensionality axiom of standard second-order logic talks about all possible relations. Although this distinction does not appear to have impact on the meaning of (3) in HiLog versus second-order PC, it does make a difference for the so-called *relation comprehension formulas* [16]:

$$\exists P \forall X_1 \ldots \forall X_n(P(X_1 \ldots, X_n) \leftrightarrow \phi) \qquad (4)$$

where $\phi$ is some formula in which $P$ does not occur free, and the only free variables are $X_1, \ldots, X_n$. It is well-known that these formulas are valid in the standard second-order predicate calculus. It turns out however, that they are not valid in HiLog. For instance, the formula:

$$\exists \mathsf{P} \forall \mathsf{X} \forall \mathsf{Y}(\mathsf{P}(\mathsf{X},\mathsf{Y}) \leftrightarrow \neg \mathsf{p}(\mathsf{X},\mathsf{Y})) \tag{5}$$

is not valid in HiLog. One HiLog semantic structure which falsifies it is $\langle \{d\}, \{d\}, I, \mathcal{F} \rangle$, where $I(p) = d$, and $d_{\mathcal{F}}^{(k)}(d, \ldots, d) = d$ for all $k (k \geq 1)$. Intuitively, this happens because domains of HiLog interpretations do not necessarily have intensions for complements of relations associated with predicate names. Whether this effect is desirable or not depends on the intended meaning of quantification. In HiLog, the quantification is over intensions, and since there is no name for $\neg p$ in $\mathcal{F}$, (5) is not valid.

## 4. APPLICATIONS OF HILOG

As explained earlier, although the semantics of HiLog is first-order, a term can be viewed as an individual, a function, or a predicate, depending on the context in which it appears. When functions or predicates are treated as objects, they are manipulated as terms through their intensions: when applied to arguments, they are evaluated as functions or relations through their extensions. By distinguishing between intensional and extensional aspects of functions and predicates, HiLog preserves the advantages of higher-order logic and avoids the computational difficulties with extensions introduced by higher-order semantics. In this section, we review a subset of HiLog that is suitable for logic programming, and then show its uses in higher-order logic programming, definite clause grammars in natural language processing, and other areas.

### *4.1. Logic Programming in HiLog*

For practical applications, we consider logic programming instead of general theorem proving with HiLog. By a *logic program*, we mean a finite set of formulas of the form

$$\forall X_1 \ldots \forall X_n (A \leftarrow_1 \wedge \cdots \wedge L_m)$$

where $m \geq 0$, $A$ is an atomic formula, $L_1, \ldots, L_m$ are literals, and $X_1, \ldots, X_n$ are all the variables occurring in $A, L_1, \ldots, L_m$ [35]. Each universal formula in a program can be written as a general clause. However, in logic programming, such clauses are often written as:

$$A \leftarrow L_1, \ldots, L_m.$$

If $L_1, \ldots, L_m$ are atomic formulas, such clauses are called *definite*. Horn clauses, and a logic program consisting of only definite clauses is called a *definite* logic program.

For definite logic programs, standard logic programming techniques [54, 4, 35] can be used to define the declarative and procedural semantics. The only difference is the form of atomic formulas and unification (unification is discussed in Section 6.3). For instance, for definite HiLog programs, the model intersection property holds and, therefore, each such program has a unique minimal (with respect to set inclusion) Herbrand model. The $T_P$ operator is also defined as usual: if **H** is a Herbrand interpretation and **P** is a definite program, $T_{\mathbf{P}}(\mathbf{H})$ is the set of all literals that are heads of ground clauses of **P** whose body is satisfied in **H**. The standard results about fixpoint also hold. These facts are easy to verify directly or with the use of Theorem 3.3.

For logic programs with negation, semantics is defined by choosing one or several of the minimal Herbrand models. For instance, the definitions of well-founded semantics [50, 52, 55, 56] and stable model semantics [20] are independent of the notions of atomic formulas and Herbrand bases, and thus the same definitions can be applied to HiLog. Some results on negation in HiLog can be found in [53].

## *4.2. Higher-Order and Modular Logic Programming*

Higher-order constructs have been found very useful in programming practice. An example is the *maplist* of Lisp. It can be defined in HiLog either as a higher-order predicate

```
maplist(F, [], []).
maplist(F, [X|R], [Y|Z]) ← F(X, Y), maplist(F, R, Z).
```

or as a generic predicate

```
maplist(F)([], []).
maplist(F)([X|R], [Y|Z]) ← F(X, Y), maplist(F)(R, Z).
```

The latter is possible since HiLog allows complex terms such as maplist(F) to appear as predicates.

The example in Section 2.1 shows the usefulness of generic view definitions, such as closure, in databases. Generic transitive closure can also be defined in Prolog:

```
closure(R, X, Y) ← C =..[R, X, Y], call(C).
closure(R, X, Y) ← C =..[R, X, Z], call(C), closure(R, Z, Y).
```

However, this is obviously inelegant compared to HiLog (see Section 2.1), since this involves both constructing a term out of a list and reflecting this term into an atomic formula using "call." The point of this example is that the lack of theoretical foundations for higher-order constructs in Prolog resulted in an obscure syntax, which partially explains why Prolog programs involving those constructs are notoriously hard to understand.

It turns out that since variables may be instantiated to HiLog terms which in turn have a propositional meaning, there is no need for the infamous "call" predicate that is built into Prolog. The latter is naturally defined in HiLog as

```
call(X) ← X.
```

which has the intended semantics.

Higher-order constructs have been also used in database languages, such as COL [1] and LDL [5, 46], for modeling complex objects containing sets. The original semantics of these languages, as described in [1, 5, 46], is higher-order, which leads to certain semantic and computational difficulties. In Section 5, we propose an alternative semantics for COL and LDL using HiLog, and argue that the latter is computationally more tractable and practically more convenient.

Modular logic programming is another application where higher-order logic can be employed. In [7, 8], a theory of modules is developed based upon standard second-order logic. The semantics of a module is a second-order relation over first-order predicates interpreted as first-order relations. The relational view of modules is a natural extension of logic programming. However, standard second-order logic is not recursively axiomatizable, and the practical usage of modules does not need the full power of second-order predicate calculus.

HiLog can be used to provide a simple alternative semantics for modular logic programming which, unlike [7, 8], has a sound and complete proof theory. Consider the following program fragment (which is well-formed both in predicate calculus and in HiLog):

```
trans(X, Y) ← edge(X, Y).
trans(X, Y) ← edge(X, Z), trans(Z, Y).
```

To turn it into a module definition, we need to determine its interface with other modules. This program fragment contains a definition for trans in terms of edge. Suppose that predicate trans is exported and predicate edge is a parameter which can be instantiated to any binary predicate of the user's choice. To obtain a module definition, we replace each nonglobal predicate symbol by a predicate variable of the same arity, where different predicate names are replaced by distinct variables. This is done to ensure the local scope of those symbols, since only variables have local scopes in logic. In the concrete syntax of [7], the module definition for transitive closure may look like this:

```
mod_trans(In, Out) {
        Out(X, Y) ← In(X, Y).
        Out(X, Y) ← In(X, Z), Out(Z, Y).
}.
```

The head of a module definition defines module interface. It consists of exported predicate parameters (like Out) and input predicate parameters (like In); the body of a module definition is the "implementation" of the module. Individual variables are still universally quantified with respect to each clause in the body (unless they are also module interface parameters, which is discussed in [7, 8]). Noninterface predicate variables in the body, if any, represent *private* predicates that are not to be seen outside the module. The quantification of predicate variables is such that all input predicate parameters are universally quantified and all other predicates (private and exported ones) are *existentially* quantified inside the scope of the universal quantification of the input predicate variables. It is precisely due to this existential quantification of predicate variables that the local data in the module are shielded from outside.

The above module definition not only defines first-order predicates in the body of the module, but also the second-order predicate corresponding to the module name, namely, mod_trans. The above module definition can be given meaning via the following formula:

```
∀In∃Out(mod_trans(In, Out) ∧
        ∀X∀Y(Out(X, Y) ← In(X, Y)) ∧
        ∀X∀Y∀Z(Out(X, Y) ← In(X, Z) ∧ Out(Z, Y))
       )
```

Under the second-order semantics of [7, 8], mod_trans is treated as a second-order predicate symbol. Instead of interpreting the above formula under the standard second-order logic, we can interpret it as a formula in HiLog, getting a more tractable semantics for modules. Under the HiLog interpretation, mod_trans is simply a first-order parameter symbol, just like link. In an implementation, existential variables can be eliminated by Skolemization, which preserves unsatisfiability in refutational theorem proving (e.g., SLD-resolution).

After a module is defined, it can be used just as any other predicate, except that it may also take predicates as arguments. Consider the query

```
link(a, b). link(b, d). link(d, e). link(a, c).
?-mod_trans(link, Closure), Closure(a, X).
```

where Closure is a predicate variable and X is an individual variable.

Under the second-order semantics, Closure will be bound to a *relation* corresponding to the transitive closure of the link predicate. Therefore, X can be bound to any node reachable from a, and the answer to this query will be the set of all such nodes.

Under the HiLog semantics, Closure will be bound to an *intension*, $c$, in the domain of quantification, rather than a relation. Nevertheless, the binary relation associated to $c$ by HiLog semantics (that is, $(c_{\mathscr{F}}^{(2)})^{-1}(U_{true})$) will be the same as under the second-order semantics—the transitive closure of link. Therefore, once again, $X$ can be bound to any node reachable from a, and the set of answers to the above query will be the same as before.

The HiLog semantics for modules yields the same result as [7] in most cases, but disagrees with [7] in marginal situations when the inherent difference between the intensional treatment of predicates in HiLog and the extensional treatment in [7, 8] becomes essential. For instance, when two different predicates $p$ and $q$ that are extensionally equivalent are to be unified, they are unifiable in the standard second-order semantics, but not in HiLog.

Another difference is that HiLog modules have features not found in [7]. For instance, modules in [7], being second-order predicates, cannot be imported into other modules, since second-order predicates cannot be arguments to other second-order predicates. Therefore, manipulating modules requires more than just a second-order PC. In contrast, there are no such difficulties in HiLog, and module names, being merely first-order terms in HiLog, can be passed to other modules as parameters.

## *4.3. Definite Clause Grammars in HiLog*

Definite clause grammars (DCGs), developed for processing natural language [47], extend the context-free grammars by adding arguments for checking the agreement among linguistic structures, and by allowing arbitrary computation in the body of a rule. They can be translated into Prolog by adding an additional pair of arguments for representing the string of words being parsed. As an example, consider the following DCG rules:

```
s(sent(Subj, Verb)) → np(Subj, Num), vp(Verb, Num).
np(Word, Num) → [Word], {isproper_name(Word, Num)}.
vp(Word, Num) → [Word], {isverb(Word, Num)}.
```

The standard transformation yields the following Prolog program:

```
s(sent(Subj, Verb), L0, L) ← np(Subj, Num, L0, L1), vp(Verb, Num, L1, L).
np(Word, Num, L0, L) ← connect(Word, L0, L), isproper_name(Word, Num).
vp(Word, Num, L0, L) ← connect(Word, L0, L), isverb(Word, Num).
connect(Word, [Word|L], L).
```

Suppose we have the following facts:

```
isproper_name(john, singular).
isverb(walks, singular).
```

Then, parsing a natural language sentence such as

::= john walks.

will be reduced to evaluating a logical query

```
?-s(X, [john, walks], []).
```

Although Prolog is adequate for many languages defined by DCG grammars, there are cases in which *generic* grammatical rules become necessary. Optionality of a nonterminal symbol and the occurrence of an arbitrary symbol zero or more times are some of the examples [2]. Such rules can be specified as follows (adapted from [2]):

```
option(X) → X.
option(X) → [].

seq(X) → X, seq(X).
seq(X) → [].
nonempty_seq(X) → X, seq(X).
```

Unfortunately, the above translation into Prolog no longer works, since Prolog does not allow variable predicates. Indeed, blindly following this recipe, we would get

```
nonempty_seq(X)(L0, L) ← X(L0, L1), seq(X)(L1, L).
```

which is a HiLog, but not a Prolog rule. To overcome this problem, we could try to turn X(L0, L) into something like NewX =.. [X, L0, L], call(NewX). However, even this patch does not work all the time. Because of the syntactic limitations of Prolog, special mechanisms for translating such grammars into Prolog are needed, and papers have been written on that subject (e.g., [2]). In contrast, as we have just seen, translation of generic grammars into HiLog is immediate, and does not require any special machinery: we simply go on (as suggested by the standard algorithm) and replace each nonterminal, N, by a HiLog atom N(L1, L2). Regardless of whether N is a predicate symbol, a function symbol, a variable, or a term, HiLog accommodates them all, due to the richness of its syntax.

As a matter of fact, even the original DCG-to-Prolog translation could be made more natural in HiLog. DCG nonterminals with parameters are readily transformed into parameterized predicates in HiLog. For example, the first DCG rule above:

```
s(sent(Subj, Verb, L0, L)) → np(Subj, Num, L0, L1), vp(Verb, Num, L1, L).
```

would become

```
s(sent(Subj, Verb))(L0, L) ← np(Subj, Num)(L0, L1), vp(Verb, Num)(L1, L).
```

## 4.4. *Query Evaluator*

Consider relational expressions composed from, say, binary relations connected by the relational operators minus, union, and the like. Suppose that the parser has already produced a parse tree (parsing is easy using Horn clauses) of the expression in the form of a term, say, *minus*(*union*($p, q$), *inter*($q, r$)), or similar. As the next step, we would like to write an evaluator for such expressions, which in HiLog looks as follows:

```
minus(P, Q)(X, Y) ← P(X, Y), ¬Q(X, Y).
union(P, Q)(X, Y) ← P(X, Y).
union(P, Q)(X, Y) ← Q(X, Y).
...   ...   ...
```

For comparison, we present an analog of the above program in Prolog. The simplest approach to this problem seems to be to define a translation predicate, tr, that converts parse trees into Prolog goals, and then use call. The rules for tr are as follows:

```
tr(minus(P, Q), X, Y, (G1, not (G2))) ← tr(P, X, Y, G1), tr(Q, X, Y, G2).
tr(union(P, Q), X, Y, (G1; G2)) ← tr(P, X, Y, G1), tr(Q, X, Y, G2).
...   ...   ...
tr(P, X, Y, Atom) ← Atom =..[P, X, Y].
```

The first observation about this Prolog program is that it is clumsy compared to its HiLog counterpart (notice that the arguments X, Y in tr are essential for the program to run correctly). Second, the last rule is intended to capture the situation where P is instantiated to a predicate symbol. However, this restriction is only implicit in the built-in predicate "=.." which will give an error if P is not a predicate symbol. One way to get around this is to list all the facts such as tr(p, X, Y, P(X, Y)) (for each predicate symbol) in advance. However, this is particularly inconvenient in the database environment when the user may create or delete new relations, since the above program would have to be updated each time.

The ease of writing the above program in HiLog stems from the ability to represent intermediate results of query evaluation in a natural way. For instance, minus(p, q) can be viewed as the name of an intermediate relation for the result of subtracting $Q$ from $P$. However, it should be clear that in order to take full advantage of HiLog, arities of all relations must be known at compile time, since we must know how many variables should appear in various places in rules. Therefore, rules for the relational operators that do not change the arities of relations (like the ones above) look particularly attractive in HiLog. On the other hand, operators such as join or Cartesian product require a heavier machinery, such as *functor* and *arg*. Still, this program would be much more elegant in HiLog than in Prolog.

### 4.5. *Type Checking*

Recently, Thom Fruehwirth pointed out to us that the monomorphic type checking system of [17] can be naturally extended to a polymorphic type checking system for HiLog [18], while expressing general polymorphism in predicate calculus is usually quite cumbersome. The use of HiLog as a language for type specification is further explored in [9].

## 5. HILOG AS A DATABASE PROGRAMMING LANGUAGE

In this section, we show that HiLog provides an alternative (first-order) semantics to some of the well-known database languages with higher-order semantics, thereby eliminating some of their problems. Specifically, we will focus on COL [1] and LDL [5, 46]. After that, we will discuss various applications of HiLog to object-oriented databases.

COL [1] is a logic-based language for complex objects. One notable feature of COL is that, in addition to finite set construction, it also provides so-called "data functions." These functions may take sets and individuals as arguments, and return sets as results. Data functions can be defined by either facts or rules. For example, some of the operations over sets can be defined as data functions as follows [1]:

```
X ∈ intersection(S1, S2) ← X ∈ S1, X ∈ S2.
X ∈ union(S1, S2) ← X ∈ S1.
X ∈ union(S1, S2) ← X ∈ S2.
X ∈ difference(S1, S2) ← X ∈ S1, not (X ∈ S2).
```

Notice that variables S1, S2 range over a domain of sets, and thus, according to our classification, the semantics of COL is higher-order. This higher-orderness presents certain semantic problems for logic programs in COL. Consider the following example adapted from [1]:

```
person(peter, {bridge}).
person(thom, {chess, tennis}).
person(thom, hobby(peter)).
Y ∈ hobby(X) ← person(X, Z), Y ∈ Z.
```

This program is unstratified in COL, since hobby and person mutually depend on each other and, therefore, the following perfectly legal queries that request all of Thom's hobbies and inquire whether Thom plays bridge, will be rejected:

```
?- X ∈ hobby(thom).
?- person(thom, {chess, tennis}).
```

To cope with this problem, [1] proposes to use an analog of the notion of local stratification adapted from logic programming [49].

For simplicity, we restrict our attention to COL programs without tuple constructs, and such that their finite set-constructs have no variables. Thus, the forms such as {a, b} *are* allowed, but {a, b, X} *are not*. Later we will show how to extend our results to the general case. Now, the restricted COL programs can be transformed into HiLog programs as follows:

1) Replace every finite set construction $\{a_1, \dots, a_n\}$, that appears in the COL

program by a *new* HiLog logical symbol, *sym*, and add the following facts to the program: $sym(a_1), \ldots, sym(a_n)$. Identical finite set constructions must be consistently replaced by the same symbol. For instance, person(thom, {chess, tennis}) will be replaced by person(thom, hobby_thom), and the following pair of facts:

```
hobby_thom(chess).
hobby_thom(tennis).
```

2) Replace COL-formulas of the form X ∈ f(...) by HiLog formulas f(...)(X).

Then the above program is transformed into the following well-formed HiLog program:

```
person(peter, hobby_peter).
person(thom, hobby_thom).
person(thom, hobby(peter)).
hobby_peter(bridge).
hobby_thom(chess).
hobby_thom(tennis).
hobby(X)(Y) ← person(X, Z), Z(Y).
```

and the corresponding queries become

```
?- hobby(thom)(X).
?- person(thom, hobby_thom).
```

These queries yield the same results as their counterparts in COL (provided that the above unstratified COL program is given the intended semantics). For another example, consider the COL query

```
?- person(thom, {chess}).
```

to the above database (with the last rule deleted, to ensure stratification). This query fails because {chess} and {chess, tennis} are two different sets. The corresponding HiLog query is

```
?- person(thom, another_hobby_thom), another_hobby_thom(chess).
```

[and the database must contain another_hobby_thom(chess)]. This query also fails since, similarly to COL, another_hobby_thom represents a different set of hobbies than hobby_thom. However, the query

```
?- person(thom, hobby_thom), hobby_thom(chess).
```

succeeds, since it is asking whether Thom plays chess, while the previous query inquired whether {chess} is one of Thom's hobby sets. Notice that the semantics of COL treats Thom as if he had different sets of hobbies in different "frames of mind," and we have been able to capture this aspect of COL pretty adequately. In contrast, in COL, there is no easy way (without introducing additional rules) to ask whether Thom plays chess, which demonstrates greater flexibility of HiLog compared to COL.

There is, however, a significant difference between the semantics of COL and that of HiLog. For instance, suppose that instead of the previous clause for hobby, the hobby-function is defined as follows:

```
chess ∈ hobby(peter).
bridge ∈ hobby(peter).
```

Then the query

```
?- person(thom, {bridge, chess}).
```

returns the answer "true" in COL, while the corresponding query

```
?- person(thom, yet_another_hobby_thom).
```

in HiLog returns "false."

This difference stems from the fact that in COL, the domain of set variables consists of sets (extensions), while the domain of variables in HiLog is a set of objects (intensions). Thus, in COL, person(thom, hobby(peter)) is evaluated to person(thom, {bridge, chess}), which allows the above COL query to succeed. Our contention is, however, that the separation of intensional and extensional aspects in HiLog makes it more flexible: the extensional semantics of COL can be captured in HiLog rather easily, by defining extensional equality of intensions. For instance, the following formula

$$\forall P \forall Q(\mathrm{eq1}(P,Q) \leftrightarrow \forall X(P(X) \leftrightarrow Q(X)))$$

says that P and Q always represent the same unary relations extensionally; P and Q may be intensionally different, though, and nothing is said about, say, binary relations represented by P and Q. However, the above axiom does not quite capture our intentions. What we really need is the extensional equality of predicates with respect to the *intended* semantics of logic programs. For the purpose of this section, we adopt the perfect model semantics of locally stratified programs, which can be defined for HiLog along the lines of [49].[8] The intended extensional equality can now be captured via the following rules with negation:

```
eq1(P, Q) ← subset(P, Q), subset(Q, P).
subset(P, Q) ← not not_subset(P, Q).
not_subset(P, Q) ← P(X), not (Q(X).
```

Now, if desirable, HiLog can simulate the extensional semantics of COL for the above query as follows:

```
?- person(thom, X), eq1(X, yet_another_hobby_thom).
```

Recall that according to the first transformation rule, the database would contain the facts

```
yet_another_hobby_thom(bridge).
yet_another_hobby_thom(chess).
```

Since person(thom, hobby(peter)) is one of the original facts in the database, the above HiLog query succeeds with X = hobby(peter).

[8][53] defines a well-founded semantics of HiLog, which is analogous to [56].

So, to simulate the semantics of COL more closely (but still not exactly), we must add one more transformation rule, which must be applied after the first two:

3) Replace every literal $p(\ldots,t,\ldots)$ in the body of a rule (including the queries), whether $t$ is either a term involving a data function or a constant that replaces a finite set construction (introduced by the first transformation rule), by the following conjunction: $p(\ldots,X,\ldots), eq1(X,t)$. For instance, in the above, the query

   ```
   ?- person(thom, yet_another_hobby_thom).
   ```

   was replaced by

   ```
   ?- person(thom, X), eq1(X, yet_another_hobby_thom).
   ```

We complete our discussion of COL by showing how to extend the translation described above to the general case. Capturing the tuple construct is easy. We just reserve a parameter symbol, say tuple, for the specific purpose of representing COL tuples; e.g., [a, b] and [c, d, e] become tuple(a, b) and tuple(c, d, e), respectively. For the set construct, we reserve a special symbol, setc. Now, every finite set construct, e.g., {a, b, X, d }, should be replaced by a HiLog term, setc(a, b, X, d), and the appropriate facts must be added to the database. In our example, these would be

```
setc(a, b, X, d)(a).
setc(a, b, X, d)(b).
setc(a, b, X, d)(X).
setc(a, b, X, d)(d).
```

The transformation rule 2) requires no change; in rule 3), we only need to replace the phrase "constant replacing a finite set construct" by the phrase "set term replacing a finite set construct."

LDL [5, 46] is another influential logic database language with a higher-order semantics. In addition to finite set construction, it provides the so-called grouping construct, which is even more general than data functions of COL. Again, LDL programs can be translated into HiLog with an alternative, first-order semantics. Similarly to COL, certain programs that are not well-formed in LDL can be given satisfactory semantics by such a translation.

Finite set constructs of LDL are translated into HiLog the same way as in the case of COL. For the grouping construct (which in LDL can occur only in rule heads), we can do the following:

Replace every LDL rule of the form $\mathrm{p}(\vec{X}, \langle\vec{Y}\rangle) \leftarrow$ body, where $\vec{X}$, $\vec{Y}$ are vectors of variables, by the pair of HiLog rules:

$\mathrm{p}(\vec{X}, \mathrm{sym}(\vec{X})) \leftarrow$ body.
$\mathrm{sym}(\vec{X})(\vec{Y}) \leftarrow$ body.

where sym is a new HiLog symbol.

Consider the following LDL program:

$\mathrm{p(X)}$.
$\mathrm{p}(\langle \mathrm{X} \rangle) \leftarrow \mathrm{p(X)}$.

Intuitively, this says that p is true of all elements of the domain, and also of some other element of the domain that, in some sense, is the set of all domain elements. However, in LDL, sets are members of the domain on a par with individuals. As a result, the above program becomes tantamount to the famous set theoretic paradox, since p has to contain the set of all sets as an element. The corresponding HiLog program is

```
p(X).
p(a) ← p(X).
a(X) ← p(X).
```

It defines a predicate which is true of every element in the domain. It also defines *a* as a symbol that *represents* the entire domain, whenever this symbol appears as a constant within an atomic formula. This corresponds to the aforementioned intuitive meaning of the above LDL program, which favors the HiLog semantics over LDL. This also shows that, in general, the translation of LDL into HiLog does not capture all the power of the grouping construct.[9] However, in most "normal" cases of grouping in LDL, the corresponding HiLog program gives the same result. For instance, consider the set-intersection example from [5]:

```
intersect(S, T, ⟨X⟩) ← member(X, S), member(X, T).
```

The corresponding HiLog program is

```
intersect(S, T, inter(S, T)) ← member(X, S), member(X, T).
inter(S, T)(X) ← member(X, S), member(X, T).
```

which gives the same result. Note, however, that in HiLog, set intersection has a more concise and elegant representation:

```
inter(S, T) (X) ← S(X), T(X).
```

We thus see that HiLog provides an alternative (we believe, computationally more attractive) semantics to LDL. Unlike COL, where it seems that HiLog can closely mimic the semantics, there is a more dramatic difference between HiLog and LDL because the latter is so expressive that even logical paradoxes can be represented without much difficulty. Such paradoxes are not representable in HiLog since the latter has a first-order semantics, and the translation of paradoxical LDL rules yields rather benign HiLog programs.

To conclude this comparison, we mention that the alternative semantics for sets in LDL and COL described above is in the same spirit as the semantics described in C-logic [12] and O-logic [28], although the latter are first-order (object-oriented) languages. It appears, thus, that higher-orderness of the syntax of HiLog is inessential in order to simulate sets. However, it is essential for other applications described earlier. It seems like an interesting observation, therefore, that changing the philosophy behind logical languages can—in some cases—eliminate the need for higher-orderness either in semantics, or in syntax, or in both.

Another promising application of HiLog in the database field is its use as an implementation vehicle for object-oriented languages recently proposed in [12, 28, 24, 25]. C-logic [12] is the simplest of the three; it supports complex objects, object

[9] In our opinion, the above discussion suggests that capturing the semantics of grouping construct *exactly* is not necessary and even may be harmful.

ids, sets, and classes. For instance, the "hobby" example discussed earlier can be represented in C-logic as follows:

```
person: peter[hobby → {bridge}].
person: thom[hobby → {chess, tennis}].
thom[hobby → {X}] ← peter[hobby → {X}].
```

yielding the same results as those produced by the corresponding HiLog representation.

C-logic admits a natural translation into predicate calculus by viewing each class symbol (e.g., person) as a unary predicate, and each set-valued attribute (e.g., hobby) as a binary predicate [12]. O-logic [28] extends C-logic by allowing single-valued attributes and by introducing a lattice structure over object ids, which helps to localize the effect of data inconsistency. Translation of O-logic into predicate calculus is just a shade more complex; it additionally requires the axiomatization of functional attributes and data inconsistency. F-logic [24, 25] takes object-oriented logics to a new dimension by introducing higher-orderness with a first-order semantics, in the same spirit as HiLog. Although it can be encoded in predicate calculus, this encoding is neither natural nor suggestive of an efficient implementation. However, an extension of the algorithm from [12] would translate F-logic into HiLog quite naturally.

This opens up a possibility of using HiLog for fast prototyping of object-oriented logic languages. Compared to the direct implementation of object-oriented databases, the advantage of HiLog as an implementation platform is that it can be relatively easily implemented using one of the already available technologies developed for Prolog [58, 43] or LDL.

## 6. PROOF THEORY OF HILOG

Because of the encoding in Section 3, one can use a proof theory for predicate calculus in order to prove theorems for HiLog. However, a proof theory stated in terms of this encoding is neither intuitive nor suggestive of a possible efficient implementation. Furthermore, extensions of HiLog will be developed in order to support more features such as lambda abstractions and dynamic updates, and these extensions may not have a direct translation into predicate calculus. A direct proof theory of HiLog is therefore important in its own right, and it may provide additional insights for further investigations of HiLog extensions.

In this section, we present a resolution-based proof theory given directly in terms of HiLog. The following issues are examined: Skolem and Herbrand theorems, unification, and resolution. The discussion follows the development of resolution-based proof theory for predicate calculus [6].

### 6.1. Skolemization

Given a sentence $\phi$ in HiLog, it can be transformed into an equivalent formula $\phi'$ in prenex normal form $Q_1 X_1 \ldots Q_n X_n \psi$, where $Q_i$ $(1 \le i \le n)$ is either $\forall$ or $\exists$, $\psi$ is a quantifier-free formula in conjunctive normal form, and $X_i$ $(1 \le i \le n)$ are all variables occurring in $\psi$. The rules for this transformation are identical to those for the ordinary predicate calculus, since the usual De Morgan's laws apply due to the fact that the definition of the logical entailment relation $\models_v$ in Section 2.3 is identical to that in predicate calculus (except for the notion of atomic formulas).

Because of the similarities between HiLog and predicate calculus, one might think that by eliminating all existential quantifiers in $\phi'$ using the usual Skolemization process in predicate calculus, we will obtain a Skolem standard form $\phi*$ of $\phi$, which is unsatisfiable if and only if $\phi$ is too. However, the unsatisfiability of $\phi*$ does not entail the unsatisfiability of $\phi$.[10] To see this, consider

$$\phi \equiv \forall \mathsf{X} \exists \mathsf{Y} \mathsf{p}(\mathsf{X},\mathsf{Y}) \wedge \forall \mathsf{F} \exists \mathsf{Z} \neg \mathsf{p}(\mathsf{Z},\mathsf{F}(\mathsf{Z})).$$

Converting $\phi$ into prenex normal form and then Skolemizing X and Y in the ordinary manner (using the new function symbols g and h) yields

$$\phi* \equiv \forall \mathsf{X} \forall \mathsf{F}(\mathsf{p}(\mathsf{X},\mathsf{g}(\mathsf{X})) \wedge \neg \mathsf{p}(\mathsf{h}(\mathsf{F}),\mathsf{F}(\mathsf{h}(\mathsf{F})))).$$

It is easy to see that $\phi$ is satisfiable in HiLog (but not in the second-order predicate calculus!), which can be verified directly by constructing a semantic structure $\mathbf{M} = \langle U, U_{true}, I, \mathcal{F} \rangle$ in which for every $u \in U$, the function $u_{\mathcal{F}}^{(1)}$ is different from all the functions $\lambda\colon U \mapsto U$ defined by the expression $\forall \mathsf{X} \exists \mathsf{Y} \mathsf{p}(\mathsf{X},\mathsf{Y})$. In contrast, $\phi*$ is unsatisfiable, since

$$\mathsf{p}(\mathsf{h}(\mathsf{g}),\mathsf{g}(\mathsf{h}(\mathsf{g}))) \wedge \neg \mathsf{p}(\mathsf{h}(\mathsf{g}),\mathsf{g}(\mathsf{h}(\mathsf{g})))$$

is an instance of $\phi*$.

Intuitively, the reason why the usual Skolemization process misbehaves is that in HiLog, it is not enough to choose a new symbol to represent a new Skolem function: such a symbol also needs to be assigned a *new intension*. Indeed, we have to assign an appropriate Skolem function as an extension of the symbol introduced by Skolemization. In predicate calculus, extensions are assigned directly to function symbols and, therefore, by choosing a new function symbol, we can construct a semantic structure for any desired Skolem function. In contrast, HiLog assigns extensions to *intensions* of function symbols, and simply choosing a new symbol is no longer enough: we should be able to assign a new intension to such a symbol, which—as the above example shows—is not always possible. In fact, as we shall see, choosing a new symbol is not that crucial for Skolemization in HiLog.

One way to overcome this problem is to modify Skolemization so as to avoid the need to assign new intensions to Skolem functions. Instead of introducing a new Skolem symbol, we will use an *unused arity* of one of the *old* symbols, patching the extra argument positions. For the formula $\phi$ above, we could use, say, symbol p with the arity 3 for Y and with the arity 4 for Z, obtaining the following Skolemized form:

$$\forall \mathsf{X} \mathsf{p}(\mathsf{X},\mathsf{p}(\mathsf{X},\mathsf{p},\mathsf{p})) \wedge \forall \mathsf{F} \neg \mathsf{p}(\mathsf{p}(\mathsf{F},\mathsf{p},\mathsf{p},\mathsf{p}),\mathsf{F}(\mathsf{p}(\mathsf{F},\mathsf{p},\mathsf{p}))).$$

Notice how the symbol p was used to fill in the argument positions of the terms p(X, p, p) and p(F, p, p, p).

Let $\mathcal{L}$ be a language of HiLog that contains at least one parameter symbol. Given a formula $\phi_0$ of $\mathcal{L}$ in prenex normal form $Q_1 X_1 \ldots Q_n X_n \psi$, where $Q_i (1 \leq i \leq n)$ is either $\forall$ or $\exists$. $\psi$ is a quantifier-free formula in conjunctive normal form, and $X_i (1 \leq i \leq n)$ are all variables occurring in $\psi$. Suppose that $Q_i$ is the leftmost

[10] We are grateful to one of the referees for suggesting this example and pointing out a mistake in an earlier draft of this paper.

existential quantifier in $\phi$. Let $k$ be the maximum arity that has been used in $\phi$, which exists since $\phi$ is finite. We obtain a new formula $\phi_1$ by eliminating the leftmost existential quantifier $Q_i$ in $\phi$ and replacing every occurrence of $X_i$ in $\psi$ with $p(X_1,\dots,X_{i-1},p,\dots,p)$ which has $max(k+1,i-1)$ arguments. Assume that there are $m$ existential quantifiers in $\phi_0$. We repeat this process for every $\phi_j (j < m)$, and finally derive a formula $\phi_m$ without any existential quantifiers. The formula $\phi_m$ is called the *Skolem standard form* of $\phi_0$.

*Lemma 6.1. A sentence $\phi$ is unsatisfiable if and only if its Skolem standard form $\phi^*$ is unsatisfiable.*

PROOF. Without loss of generality, we assume that $\phi$ is already in prenex normal form. Let $\phi_0$ be $\phi$ and let $m$ be the number of existential quantifiers in $\phi$. Then $\phi^*$ is $\phi_m$, where each $\phi_j$ $(1 \le j \le m)$ is obtained from $\phi_{j-1}$ by the above Skolemization process. We show that $\phi_j$ is unsatisfiable if and only if $\phi_{j-1}$ is unsatisfiable.

Let $\phi_{j-1}$ be in a prenex normal form $\forall X_1 \dots \forall X_{i-1} \exists X_i Q_{i+1} X_{i+1} \dots Q_n X_n$ $\psi[X_1,\dots,X_n]$, where $X_i$ is the leftmost existential variable, $\psi[X_1,\dots,X_n]$ is a formula in conjunctive form, and $X_1,\dots,X_n$ are all the variables occurring in it. Suppose that $k$ is the maximum arity that has been used in $\phi_{j-1}$, and $f$ is the parameter symbol used in the Skolemization of $X_i$. Then $\phi_j$ is of the form

$$\forall X_1 \dots \forall X_{i-1} Q_{i+1} X_{i+1} \dots Q_n X_n$$
$$\psi[X_1,\dots,X_{i-1},f(X_1,\dots,X_{i-1},f,\dots,f),X_{i+1},\dots,X_n],$$

where $f(X_1,\dots,X_{i-1},f,\dots,f)$ has $max(k+1,i-1)$ arguments.

Suppose that $\phi_{j-1}$ is true in a semantic structure $\mathbf{M} = \langle U, U_{true}, I, \mathcal{F} \rangle$, but $\phi_j$ is unsatisfiable. Then for every $x_1,\dots,x_{i-1} \in U$, there exists $x_i \in U$ such that

$$\mathbf{M} \models_v Q_{i+1} X_{i+1} \dots Q_n X_n \psi[X_1,\dots,X_n], \tag{6}$$

for every variable assignment $v$ that maps $X_1,\dots,X_i$ to $x_1,\dots,x_i$, respectively. We construct a new semantic structure $\mathbf{M}' = \langle U, U_{true}, I, \mathcal{F}' \rangle$, where $\mathcal{F}'$ is exactly the same as $\mathcal{F}$ except that for every $x_1,\dots,x_{i-1} \in U$, $I(f)^{k+1}_{\mathcal{F}'}(x_1,\dots,x_{i-1},I(f),\dots,I(f)) = x_i$, where $x_i$ is chosen as explained in (6) above. Since the arity $k+1$ is not used in $\phi_{j-1}$, the function $I(f)^{k+1}_{\mathcal{F}}$ is not used to determine whether (6) holds. This, together with the fact that the domains of $\mathbf{M}$ and $\mathbf{M}'$ coincide, yields $\mathbf{M}' \models_v \phi_j$. Since $v$ was chosen arbitrarily for $X_1,\dots,X_{i-1}$ and $X_i$ does not occur in $\phi_j$, it follows that $\mathbf{M}' \models \phi_j$, contrary to the assumption that $\phi_j$ is unsatisfiable. Therefore, $\phi_{j-1}$ must be unsatisfiable.

For the reverse direction, assume (in the above notation) that $\phi_j$ is true in a semantic structure $\mathbf{M}$, but $\phi_{j-1}$ is unsatisfiable. Following the definition, it can be verified that $\phi_{j-1}$ is true in $\mathbf{M}$—a contradiction. Thus, $\phi_j$ must be unsatisfiable. □

Note that Skolemization in HiLog, as presented above, works for finite sets of formulas only. A solution that also works for infinite sets is proposed in [9]. The idea is to introduce sorted symbols into HiLog and then use classical Skolemization, which does not rely on the existence of unused arities. This, however, requires an extension to HiLog and will not be discussed here (see [9]).

### *6.2. Herbrand's Theorem*

Once Skolemized, we can restrict our attention to *universal* formulas, i.e., to formulas in prenex normal form whose prefix contains universal quantification only. Furthermore, transforming such formulas into conjunctive normal form yields HiLog formulas in the *clausal* form. As usual, we can then drop quantification altogether with the understanding that clauses are implicitly universally quantified. Variable-free formulas are called *ground*. The rest of the formulas are nonground.

Herbrand interpretations for HiLog were defined in Section 2.3. Recall that given a HiLog language $\mathscr{L}$, its Herbrand universe, $HU(\mathscr{L})$, coincides with its Herbrand base, and is defined as the set of all ground (i.e., variable-free) HiLog terms. However, when the equality predicate is taken into account, the definition of Herbrand interpretations needs adjustment.

*Definition 6.1.* Let $\mathscr{L}$ be a HiLog language and let $\mathbf{H}$ be a subset of its Herbrand base. Then $\mathbf{H}$ is called a **Herbrand interpretation** if it is closed under the congruence:[11]

- $(\alpha = \alpha) \in \mathbf{H}$;
- if $(\alpha = \beta) \in \mathbf{H}$, then $(\beta = \alpha) \in \mathbf{H}$;
- if $(\alpha = \beta)$, $(\beta = \gamma) \in \mathbf{H}$, then $(\alpha = \gamma) \in \mathbf{H}$;
- if $L$, $(\alpha = \beta) \in \mathbf{H}$ and $L'$ is the result of replacing an occurrence of $\alpha$ in $L$ by $\beta$, then $L' \in \mathbf{H}$.

Given a Herbrand interpretation $\mathbf{H}$, we can define satisfaction of formulas in $\mathbf{H}$ similarly to the classic case:

- If $a$ is a ground HiLog atomic formula, then $\mathbf{H} \models a$ if and only if $a \in \mathbf{H}$.
- If $\phi$ and $\phi$ are ground HiLog formulas, then
  — $\mathbf{H} \models (\phi \vee \psi)$ if and only if $\mathbf{H} \models \phi$ or $\mathbf{H} \models \psi$;
  — $\mathbf{H} \models (\phi \wedge \psi)$ if and only if $\mathbf{H} \models \phi$ and $\mathbf{H} \models \psi$;
  — $\mathbf{H} \models \neg\, \phi$ if and only if it is not the case that $\mathbf{H} \models \phi$.
- If $\phi$ is a universal formula, then $\mathbf{H} \models \phi$ if and only if $\mathbf{H} \models \phi'$ for every ground instance $\phi'$ of $\phi$.

To avoid possible misunderstanding, recall that in the presence of equality, we only consider those semantic structures that assign the intended meaning to the equality predicate (see Section 2.3). To emphasize this fact, we will call such semantic structures *E-structures*. We can then talk about *E-unsatisfiable* and *E-valid* formulas, i.e., formulas that are false (resp., true) in all $E$-structures.

*Lemma 6.2. Let $\mathscr{L}$ be a HiLog language. For every E-structure* $\mathbf{M}$, *there is a Herbrand interpretation* $\mathbf{H}_\mathbf{M}$ *such that for every universal formula* $\phi$, $\mathbf{M} \models \phi$ *if and only if* $\mathbf{H}_\mathbf{M} \models \phi$. *Likewise, for every Herbrand interpretation* $\mathbf{H}$, *there is an E-structure* $\mathbf{M}_\mathbf{H}$ *such that* $\mathbf{H} \models \phi$ *if and only if* $\mathbf{M}_\mathbf{H} \models \phi$, *for every universal formula* $\phi$.

Proof. If $\mathbf{M}$ is an $E$-structure, then the corresponding Herbrand interpretation $\mathbf{H}_\mathbf{M}$ is simply the set of all ground atomic formulas of $\mathscr{L}$ satisfied by $\mathbf{M}$.

[11]The two formulas in the middle follow from the other two.

Conversely, let **H** be a Herbrand interpretation. The corresponding $E$-structure $\mathbf{M}_\mathbf{H} = \langle U, U_{true}, I, \mathcal{F} \rangle$ is defined thus:

1) $U \overset{\text{def}}{=} HU(\mathcal{L})/=$, the factor-space of the Herbrand universe, $HU(\mathcal{L})$, with respect to the equivalence relation induced by the equality atoms in **H**. If $s \in HU(\mathcal{L})$, its corresponding element in $U$ will be denoted by $[s]$.
2) $U_{true} \overset{\text{def}}{=} \mathbf{H}(\mathcal{L})/=$, the factor-space of **H** with respect to the aforementioned equivalence relation.
3) $I$: $\mathcal{S} \mapsto U$ is obtained from the identity embedding $Id$: $\mathcal{S} \mapsto HU(\mathcal{L})$ by composing $Id$ with the natural epimorphism $HU(\mathcal{L}) \mapsto U$, i.e., for each $s \in \mathcal{S}$, $I(s) = [s]$.
4) $\mathcal{F}$ is defined thus: for every $[s] \in U$ and $k \geq 0$, $[s]_{\mathcal{F}}^{(k)}([t_1], \ldots, [t_k]) = [s(t_1, \ldots, t_k)]$.

It is left to the reader to verify that **M** and $\mathbf{H}_\mathbf{M}$ (resp., **H** and $\mathbf{M}_\mathbf{H}$) stand in the desired relationship to each other. The proof is carried out by structural induction on HiLog formulas, similar to the classic case. □

As a consequence of the above lemma, a universal formula is $E$-unsatisfiable if and only if it has no Herbrand model:

*Corollary 6.1. A set* S *of HiLog clauses is E-unsatisfiable if and only if* S *is false under all Herbrand interpretations of* S.

*Theorem 6.1 (cf. Herbrand's Theorem). A set* S *of HiLog clauses is E-unsatisfiable if and only if there is a* finite *E-unsatisfiable set* S′ *of ground instances of clauses of* S.

Proof. It is easy to verify that for any set of HiLog clauses S, the encoding of Section 2.3 establishes a 1–1 correspondence between Herbrand interpretations of S and those of $\mathsf{encode}_a$(S). The only subtlety here is that, in HiLog, the Herbrand universe and the Herbrand base coincide, while in predicate calculus, they are distinct. For this reason, the Herbrand universe of $\mathsf{encode}_a$(S) is obtained by applying $\mathsf{encode}_t$ to the Herbrand universe of S, while the Herbrand base of $\mathsf{encode}_a$(S) is obtained from that of S via $\mathsf{encode}_a$.

Herbrand's theorem for HiLog now directly follows from Theorem 3.3 and the analogous result in predicate calculus [6]. □

## *6.3. Unification*

Although HiLog allows arbitrary terms to appear in places where only predicates and functions are usually allowed in predicate calculus, unification in HiLog is a very simple extension of unification in predicate calculus and is decidable. This is mainly because HiLog separates intensions from extensions and does not have any nontrivial built-in equality theory over intensions.

Except for the different notion of terms in HiLog, the definitions of substitutions, applications of substitutions to terms, and substitution composition in HiLog are the same as those in predicate calculus [6].

*Definition 6.2.* A *substitution* is a finite set of the form $\{t_1/X_1, \ldots, t_n/X_n\}$, where $X_1, \ldots, X_n$ are distinct variables, and every term $t_i$ is different from $X_i$, where $1 \leq i \leq n$.

*Definition 6.3.* A *unifier* for a set of HiLog terms $\{e_1,\ldots,e_k\}$ is a substitution $\theta$ such that $e_1\theta = \cdots = e_k\theta$. A set $\{e_1,\ldots,e_k\}$ is *unifiable* if it has a unifier.

*Definition 6.4.* A unifier $\sigma$ for a set $\{e_1,\ldots,e_k\}$ of expressions is *most general* if and only if for each unifier $\theta$ for this set, there is a substitution $\lambda$ such that $\theta = \sigma\lambda$.

Following [39], we can derive an efficient unification algorithm by solving equations. An *equation* is of the form $t_1 = t_2$, where $t_1, t_2$ are terms. An equation set (possibly empty) is *solved* if it has the form $\{X_1 = t_1,\ldots,X_n = t_n\}$ and the $X_i$'s are distinct variables which do not occur in any $t_j$ $(1 \le j \le n)$. Notice that a finite equation set in solved form can be naturally viewed as a substitution.

A *solution* of the equation set $\{t_1 = s_1,\ldots,t_n = s_n\}$ is a substitution $\theta$ such that $t_i\theta \equiv s_i\theta$ $(1 \le i \le n)$. An equation set is *solvable* if it has a solution. A solution $\sigma$ for an equation set $E$ is *most general* if and only if for each solution $\theta$ of $E$, there is a substitution $\lambda$ such that $\theta = \sigma\lambda$.

Given a finite equation set, the unification algorithm nondeterministically chooses an equation from the equation set to which it applies one of the following transformations according to the form of the selected equation:

1) For $t(t_1,\ldots,t_n) = s(s_1,\ldots,s_m)$, where $n \neq m$, halt with failure.
2) For $t(t_1,\ldots,t_n) = s(s_1,\ldots,s_n)$, replace the equation by $t = s, t_1 = s_1,\ldots,t_n = s_n$.
3) For $f = g$, delete the equation if $f$ and $g$ are identical parameter symbols in $\mathcal{S}$; otherwise, halt with failure.
4) For $X = X$, where $X$ is a variable, delete the equation.
5) For $t = X$, where $t$ is not a variable and $X$ is a variable, replace the equation by $X = t$.
6) For $X = t$, where $X$ is a variable and $t$ is a term different from $X$, if $X$ appears in $t$, then halt with failure; otherwise, replace $X$ by $t$ wherever it occurs in the equations.

The algorithm terminates when no further transformation can be applied or when failure is reported.

*Theorem 6.2 (Unification Theorem). The unification algorithm applied to a finite set of equations E returns a finite set E* of equations in solved form if and only if E is solvable. It returns failure otherwise. The returned equation set E* viewed as a substitution is a most general solution of E if E is solvable.*

PROOF. It is easy to see that for any pair of HiLog terms, $s$ and $r$, they are unifiable if and only if so are $\mathsf{encode}_t(s)$ and $\mathsf{encode}_t(r)$, if and only if $\mathsf{encode}_a(s)$ and $\mathsf{encode}_a(r)$ are unifiable. It is also easy to see that the encoding of HiLog in PC transforms the above unification algorithm for HiLog into the corresponding algorithm for PC, described in [39]. The theorem now follows from these two facts. A direct proof can be also obtained as a simple adaptation of the proof in [39]. □

According to the above theorem, HiLog unification is decidable. The reader may wonder about the result reported in [21], where it is shown that unification becomes undecidable once variables are allowed in places where normally only function symbols are permitted to appear. There is no contradiction, however, with our result, for the following reasons. Informally, for a sufficiently rich logic with a higher-order syntax to have a decidable unification problem, two conditions are sufficient. First, the semantics of the logic should be first-order. That is, predicate and function variables should not range over relations or functions. Instead, they must range over intensions or names of predicates and functions. Second, the logic should not have an undecidable built-in equality theory over names of predicates and functions. HiLog satisfies both conditions by separating intensions from extensions (thereby avoiding extensional unification) and by embodying only a trivial equality theory of intensions. In contrast, Goldfarb [21] essentially works with second-order $\lambda$-terms that embody an undecidable equality theory, and this is what makes unification undecidable in this case.

## 6.4. *Refutation*

*Definition 6.5.* If two or more literals of the same polarity (i.e., both positive or both negative) in a clause $C$ have a most general unifier $\sigma$, then $C\sigma$ is called a **factor** of $C$.

*Definition 6.6.* Let $C_1 \equiv L[t] \vee C_1'$, and $C_2 \equiv (r = s) \vee C_2'$ be two clauses with no variables in common, where $l[t]$ is a literal with an occurrence of the term $t$ (including the case when $L[t] \equiv t$). If $t$ and $r$ have a most general unifier $\sigma$, then

$$L\sigma[s\sigma] \vee C_1'\sigma \vee C_2'\sigma$$

s a **binary paramodulant** of $C_1$ and $C_2$. In this paramodulant, $L\sigma[s\sigma]$ denotes the result of replacing one single occurrence of $t\sigma$ in $L\sigma$ by $s\sigma$.

It is important to realize that since $L[t]$ in the above definition may be the same as $t$, in HiLog one needs to paramodulate not only on the terms occurring strictly inside atomic formulas, as is the case in predicate calculus, but also on atomic formulas themselves. This is an interesting distinction with respect to PC which stems from the fact that HiLog is a higher-order language in which atomic formulas are not distinguished from terms. An example when atomic formulas must be paramodulated upon is presented below.

*Definition 6.7.* Let $\mathrm{C}_1 = \mathrm{L}_1 \vee \mathrm{C}_1'$ and $\mathrm{C}_2 = \neg \mathrm{L}_2 \vee \mathrm{C}_2'$ be a pair of clauses with no variables in common. If $L_1$ and $L_2$ have a most general unifier $\sigma$, then the clause

$$C_1'\sigma \vee C_s'\sigma$$

is called a **binary resolvent** of $C_1$ and $C_2$.

A *deduction* of a clause $C$ from a set of clauses S is a finite sequence of clauses $D_1, \ldots, D_n$ such that $D_n = C$, and for $1 \leq k \leq n$ each $D_k$ is either a member of S, a factor of $D_i$ $(i < k)$, a paramodulant of some $D_i$ and $D_j$ $(i, j < k)$, or a binary resolvent of $D_i$ and $D_j$ $(i, j < k)$. A deduction ending with the empty clause is called a *refutation*. If $C$ is deducible from S, we write $\mathrm{S} \vdash C$.

One way to prove the completeness theorem is to adapt the corresponding proof for PC [6]. A shorter proof, however, results from the use of Theorem 3.3:

*Theorem 6.3 (Soundness and Completeness of Resolution). A set* S *of clauses is $E$-unsatisfiable if and only if there exists a refutation of* $\mathrm{S} \cup \{X = X\}$.

Proof. Under the encoding of Section 3, given a set of HiLog clauses S, any application of the factorization, paramodulation, or the resolution rules to clauses in S corresponds to an application of the respective rule to encode(S), and vice versa.[12] Thus, it follows from the analogous result in predicate calculus [6] that the combination of resolution, factorization, and paramodulation is a complete set of inference rules for refuting $E$-unsatisfiable sets of HiLog clauses. Note that, unlike [6], we do not include functional reflexivity axioms $f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$ in the formulation of Theorem 6.3. This is possible due to a result in [48] that shows that these axioms are not needed for completeness of paramodulation.[13]

We also note that if S contains no equality atoms, then "$E$-unsatisfiability" can be replaced by "unsatisfiability" and the axiom $X = X$ would not be needed. □

Consider an example. Recall from Section 3 that the formula

$$(\mathsf{s(a)} = \mathsf{r(b,c)}) \supset (\mathsf{s(a)} \leftrightarrow \mathsf{r(b,c)})$$

is valid in HiLog. Therefore, its negation should be $E$-unsatisfiable. The negation of the above formula can be transformed into the following three clauses:

1) s(a) = r(b, c).
2) s(a) ∨ r(b, c).
3) ¬s(a) ∨ ¬r(b, c).

By paramodulation, we can derive from 2) and 1)

4) r(b, c).

and from 3) and 1)

5) ¬r(b, c).

Then, by resolution, we can derive the empty clause from 4) and 5), which completes the refutation.

## 7. HILOG IMPLEMENTATION ISSUES

A straightforward implementation of a Horn clause logic programming language based on HiLog is possible using the encoding of HiLog into PC, described in Section 3. Since this encoding preserves Horn clauses, we could simply use a standard Prolog compiler to execute the result of such encoding. While this is theoretically sound and may even be practical for prototype programs, for large programs there are some efficiency issues that deserve further consideration. These have to do with fast access to the appropriate clauses at predicate invocation, and efficient representation of data structures. Also, built-in nonlogical predicates,

[12] It is noted that the special treatment of " = " in the definition of encode in Section 3.3 was essential to ensure the above correspondence in case of the paramodulation rule.

[13] This result was pointed out to us by Leo Bachmair.

dynamic changes to the database, and control constructs, such as the "cut," lead to new issues in the HiLog framework.

Consider a simple HiLog Horn clause:

```
p(X,Y) ← q(X,Z),p(Z,Y).
```

this clause also happens to be a predicate logic Horn clause. Its encoding in predicate logic is

```
call(apply(p,X,Y)) ← call(apply(q,X,Z)), call(apply(p,Z,Y)).
```

With every clause encoded in this way, each HiLog program, no matter how large, defines exactly one Prolog predicate, `call/1`. For Prolog-like efficiency, each predicate invocation must quickly locate the clauses whose heads might possibly unify. In Prolog systems, this is achieved first by branching directly to clauses with the same predicate, and then using indexing, normally on the main functor symbol of the first argument, to further refine the search. Notice that for the encoding of a HiLog program, these two ways of refining the search are not very effective: Prolog sees only one predicate, `call/1`, and the main functors of all those heads are `apply/n`. Thus, the only filtering would come from the arity of "`apply`." One approach would be to use partial evaluation techniques to optimize the direct translation [22]. A problem with this approach is how to treat nonlogical constructs, such as "assert." Another way to improve clause access would be to improve the indexing strategy used by the underlying Prolog system, perhaps as proposed in [51]. An alternative is to constrain the HiLog programs in such a way as to allow simple compile-time optimizations. One way we are exploring is to restrict the clauses to those whose heads have a nonvariable symbol in the leftmost position, that is, clauses whose heads are *rigid* terms. In this case, all rigid calls in the bodies of clauses can be compiled to branch directly to the appropriate clauses.

Another issue is the representation of HiLog terms. In Prolog, complex terms are normally represented as linked record structures. An $n$-ary term is represented using $n + 1$ consecutive words; the first word indicates the functor symbol and the arity, $n$, and the following $n$ words represent the $n$ arguments. To represent a HiLog term, another arity is required. The most natural way uses $n + 2$ words: one for the arity, one for (a pointer to) the functor term, and $n$ for the argument terms. While this might seem somewhat wasteful of space, using an entire word for a relatively small arity, we suspect that this will turn out to be the most efficient representation. It is actually a regularization of Prolog's representation, and will make for slightly simpler traversal algorithms.

A realistic logic programming system must support built-ins, which have nonlogical behavior. A decision must be made as to whether these built-ins can be accessed by a most general call. For example, should the query `← X(Y)` be able to call the `read/1` predicate? The most reasonable choice seems to be to exclude this possibility. A desired separation of built-in predicates from the rest can be achieved by introducing *sorts* into HiLog [9], that is, by splitting $\mathcal{S}$ and $\mathcal{V}$ into disjoint subsets and then proceeding to define HiLog terms of different sorts. In this way, "normal" variables will not be instantiable by the symbols that represent built-in predicates. If the user wishes to manipulate such symbols, he would have to use variables of an appropriate sort explicitly.

The issue of the "cut" is also complicated. In Prolog, a cut eliminates the alternative clauses remaining for the current predicate. Since in HiLog, different calls can access various subsets of clauses, the notion of "predicate" is not so well-defined. These and other issues are explored in more detail in [19].

## 8. CONCLUSION: DO WE NEED YET ANOTHER LOGIC?

Whenever a new logic is proposed, the question is (and should be) raised as to whether yet another logic is needed. Why not just stay with the logics we know and understand? This is an especially cogent point when the new logic has a simple and reasonably direct translation into the granddaddy of all logics, first-order predicate calculus, as, in our case, HiLog does. We believe, however, that in the case of HiLog, there are at least three reasons why we should seriously entertain the idea of this new logic as a basis for logic programming.

Firstly, programming in HiLog makes more logic programs logical. We all admonish Prolog programmers to make their programs as pure as possible and to eschew the evils of Prolog's nonlogical constructs. In Prolog, the intermixing of predicate and function symbols, in particular in the predicate, call/1, is nonlogical, whereas in HiLog, it is completely logical and is a first-class citizen. So in HiLog, programmers need not avoid using call/1, and thus have more flexibility in their task of writing pure logic programs.

Secondly, even though one might say that HiLog is simply a syntactic variant of Prolog, syntax is important when one is doing meta-programming. Since in meta-programming the syntax determines the data structures to be manipulated, a simpler syntax means that meta-programs can be much simpler. We saw this in the example of translating DCGs to HiLog, to the extent that a meta-program already written (the DCG translator) would automatically work in a more complicated situation, and so did not need to be changed at all.

Thirdly, and perhaps most importantly, a different logic encourages in the programmer a different way of thinking. Certainly, the translation of any HiLog program could be programmed directly in Prolog, but would it be? The DCG example, we believe, shows not. Not only is there an issue of efficiency in that the translation might not execute as efficiently as the programmer desires, but also that the programmer would not even think of that Prolog program. HiLog encourages the programmer to think about and use parameterized predicates; Prolog does not. HiLog programmers would be more likely than Prolog programmers to modularize their programs along the lines suggested in Section 4.2 above. The language influences the way programmers think, and the programs they write.

Finally, we would like to point out that HiLog is not the only logic which can be encoded in predicate calculus. The classic first-order modal logics provide a rich source of examples when a different syntax (motivated by a different philosophy of logic) inspired important and useful studies both in Logic and Artificial Intelligence.

In summary, we believe that HiLog offers significant advantages over Prolog, and deserves serious consideration as the basis for a new logic programming language.

We are grateful to Leo Bachmair, Thom Fruehwirth, Sanjay Manchanda, Shuky Sagiv, Rodney Topor, and James Wu for their comments on the contents of this paper. We especially thank James Wu for suggesting the encoding described in Section 3. Suggestions of the referees are also appreciated. This paper is an expanded version of the work previously reported in [10, 11]. The work of M. Kifer was supported in part by NSF Grant IRI-8903507.

## REFERENCES

1. Abiteboul, S. and Grumbach, S., COL: A Logic-Based Language for Complex Objects, in: *Proc. Workshop on Database Programming Languages*, Roscoff, France, Sept. 1987, pp. 253–276.
2. Abramson, H., Metarules and an Approach to Conjunction in Definite Clause Translation Grammars: Some Aspects of Grammatical Metaprogramming, in: *Proceedings of the 5th International Conference and Symposium on Logic Programming*, R. A. Kowalski and K. A. Bowen (eds.), Seattle, WA, Aug. 1988, pp. 233–248.
3. Apt, K. R., Blair, H., and Walker, A., Towards a Theory of Declarative Knowledge, in: *Foundations of Deductive Databases and Logic Programming*, J. Minker (ed.), Morgan Kaufmann, Los Altos, CA, 1988, pp. 89–148.
4. Apt, K. R. and Van Emden, M. H., Contributions to the Theory of Logic Programming, *JACM* 29:841–862 (1982).
5. Beeri, C., Naqvi, S., Shmueli, O., and Tsur, S., Sets and Negations in a Logic Database Language (LDL), MCC Technical Report, 1987.
6. Chang, C. L. and Lee, R. C. T., *Symbolic Logic and Mechanical Theorem Proving*, New York, Academic, 1973.
7. Chen, W., A Theory of Modules Based on Second-Order Logic, in: *Proceedings of IEEE 1987 Symposium on Logic Programming*, San Francisco, CA, Sept. 1987, pp. 24–33.
8. Chen, W., Modules for Logic Programming, Research Report. Dept. of Computer Science, SUNY at Stony Brook, 1989.
9. Chen, W. and Kifer, M., Sorts, Types and Polymorphism in Higher-Order Logic Programming, Technical Report 92-CSE-7, Dept. of Computer Science and Engineering, Southern Methodist University, Dallas, TX, Mar. 1992.
10. Chen, W., Kifer, M., and Warren, D. S., HiLog as a Platform for Database Language (or Why Predicate Calculus is not Enough), in: *2nd Intl. Workshop on Database Programming Languages*, Oregon Coast, OR, June 1989, pp. 315–329.
11. Chen, W., Kifer, M., and Warren, D. S., HiLog: A First-Order Semantics of Higher-Order Logic Programming Constructs, in: *Proceedings of North American Conf. On Logic Programming*, 1989, pp. 1090–1114.
12. Chen, W. and Warren, D. S., C-logic for Complex Objects, in: *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Mar. 1989, pp. 369–378.
13. Church, A., A Formulation of the Simple Theory of Types, *Journal of Symbolic Logic* 5:56–68 (1940).
14. Clocksin, W. F. and Mellish, C. S., *Programming in Prolog*, Springer Verlag, 1981.
15. Dershowitz, N. and Manna, Z., Proving Termination with Multiset Orderings *CACM* 22(8) (1979).
16. Enderton, H. B., *A Mathematical Introduction to Logic*, Academic, New York, 1972.
17. Fruehwirth, T., Type Inference by Program Transformation and Partial Evaluation, in: *IEEE Intl. Conf. on Computer Languages*, Miami Beach, FL, 1988, pp. 347–355.
18. Fruehwirth, T., Polymorphic Type Checking for Prolog in HiLog, in: *6th Israel Conference on Artificial Intelligence and Computer Vision*, Tel Aviv, Dec. 1989.
19. Fruehwirth, T. and Warren, D. S., Putting HiLog to Work, manuscript, SUNY Stony Brook, 1990.
20. Gelfond, M. and Lifschitz, V., The Stable Model Semantics for Logic Programming, in: *Logic Programming: Proceedings of the 5th Conference and Symposium*, MIT Press, 1988, pp. 1070–1080.

21. Goldfarb, W. D., The Undecidability of the Second-Order Unification Problem, *Theoretical Computer Science* 13:225–230 (1981).
22. Smith, D. A. and Hickey, T., private communication, 1990.
23. Henkin, L., Completeness in the Theory of Types, *Journal of Symbolic Logic* 15:81–91 (1950).
24. Kifer, M. and Lausen, G., F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme, in: *ACM SIGMOD Conf. on Management of Data*, May 1989, pp. 134–146.
25. Kifer, M., Lausen, G., and Wu, J., Logical Foundations of Object-Oriented and Frame-Based Languages, Technical Report 90/14, Dept. of Computer Science, SUNY at Stony Brook, July 1990; to appear in *J. ACM.*
26. Kifer, M. and Lozinskii, E. L., A Logic for Reasoning with Inconsistency, *Journal of Automated Reasoning* vol. 9, no. 2, 1992.
27. Kifer, M. and Subrahmanian, V. S., On the Expressive Power of Annotated Logic Programs, in: *North American Conference on Logic Programming*, 1989, pp. 1069–1089.
28. Kifer, M. and Wu, J., A Logic for Object-Oriented Logic Programming (Maier's O-logic Revisited), in: *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Mar. 1989, pp. 379–393.
29. Krishnamurthy, R. and Naqvi, S., Towards a Real Horn Clause Language, MCC Report ACA-ST-077-88, Austin, TX, 1988.
30. Kuper, G., Logic Programming with Sets, in: *Proc. 6th ACM Conf. on PODS*, San Diego, CA, 1987, pp. 11–20.
31. Kuper, G., An Extension of LPS to Arbitrary Sets, IBM Research Report, 1987.
32. Kuper, G. and Vardi, M. Y., A New Approach to Database Logic, in: *Proc. ACM PODS*, 1984.
33. Lassez, J.-L., Maher, M. J., and Marriott, K., Unification Revisited, in: *Foundations of Deductive Databases and Logic Programming*, J. Minker (ed.), Morgan Kaufmann, Los Altos, CA, 1988, pp. 587-625.
34. Lifschitz, V., On the Declarative Semantics of Logic Programs with Negation, in: *Foundations of Deductive Databases and Logic Programming*, J. Minker (ed.), Morgan Kaufmann, Los Altos, CA, 1988, pp. 177–192.
35. Lloyd, J. W., *Foundations of Logic Programming* (2nd edition), Springer Verlag, New York, 1987.
36. Maida, A. and Shapiro, S. Intensional Concepts in Propositional Semantic Networks, *Cognitive Science* 6(4):291–330 (1982).
37. Maida, A., Knowing Intensional Individuals, and Reasoning about Knowing Intensional Individuals, in: *Proc. 8th IJCAI*, Germany, 1983, pp. 382–384.
38. Maier, D., A Logic for Objects, in: *Preprints of Workshop on Foundations on Deductive Database and Logic Programming*, Minker (ed.), Washington, DC, Aug. 1986.
39. Martelli, A. and Montanari, U., An Efficient Unification Algorithm, *ACM Trans. on Progr. Lang. and Systems* 4(2):258–282 (1982).
40. Meyer, A. R., What is a Model of the Lambda Calculus, *Information and Control* 52:87–122 (1982).
41. Motro, A., BAROQUE: A Browser for Relational Databases, *ACM Trans. on Office Information Systems* 4(2):164–181 (1986).
42. Miller, D. A. and Nadathur, G., Higher-Order Logic Programming, in: *Proceedings of the 3rd International Conference on Logic Programming*, London, England, July 1986, pp. 448–462.
43. Maier, D. and Warren, D. S., *Computing with Logic*. Benjamin/Cummings, 1988.
44. Montague, R., The Proper Treatment of Quantification in English, in: *Approaches to Natural Languages*, K. J. J. Hintikka et al. (eds.), Dordrecht, 1973, pp. 221–242.
45. Nadathur, G., *A Higher-Order Logic as the Basis for Logic Programming*, Ph.D. dissertation, University of Pennsylvania, Philadelphia, June 1987.

46. Naqvi, S. and Tsur, S., *A Logical Language for Data and Knowledge Bases*, Computer Science Press, Rockville, MD, 1989.
47. Pereira, F. C. N. and Warren, D. H. D., Definite Clause Grammars for Language Analysis: A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence* 13:231–278 (1980).
48. Peterson, G. E., A Technique for Establishing Completeness Results in Theorem Proving with Equality, *SIAM Journal of Computing* 12(1):82–100 (Feb. 1983).
49. Przymusinski, T. C., On the Semantics of Stratified Deductive Databases, in: *Foundations of Deductive Databases and Logic Programming*, J. Minker (ed.), Morgan Kaufmann, Los Altos, CA, 1988, pp. 193–216.
50. Przymusinski, T. C., Every Logic Program has a Natural Stratification and an Iterated Least Fixed Point Model, in: *Proc. ACM PODS*, 1989.
51. Ramesh, R., Ramakrishnan, I. V., and Warren, D. S., Automata-Driven Indexing of Prolog Clauses, in: *Proceedings of POPL*, Jan. 1990.
52. Ross, A., A Procedural Semantics for Well Founded Negation in Logic Programs, in: *Proc. ACM PODS*, 1989.
53. Ross, A., On Negation in HiLog, in: *Proc. ACM PODS*, Denver, CO, May 1991, pp. 206–215.
54. Van Emden, M. H. and Kowalski, R. A., The Semantics of Predicate Logic as a Programming Language *JACM* 23:733–742 (1976).
55. Van Gelder, A., The Alternating Fixpoint of Logic Programs with Negation, in: *Proc. ACM PODS*, 1989, pp. 1–10.
56. Van Gelder, A., Ross, K., and Schlipf, J. S., Unfounded Sets and Well-Founded Semantics for General Logic Programs, in: *Proc. ACM PODS*, 1988, pp. 221–230.
57. Warren, D. H. D., Higher-Order Extensions to Prolog: Are They Needed? *Machine Intelligence* 10:441–454 (1982).
58. Warren, D. H. D., An Abstract Prolog Instruction Set, Report 309, AI Center, SRI International, Menlo Park, CA, Oct. 1983.
59. Wu, J., private communication, 1989.