Sorted HiLog: Sorts in Higher-Order Logic Data Languages

Weidong Chen*
Computer Science and Engineering
Southern Methodist University
Dallas, Texas 75275-0122, U.S.A.
wchen@seas.smu.edu

Michael Kifer†
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, U.S.A.
kifer@cs.sunysb.edu

June 8, 1994

Abstract

HiLog enhances the modeling capabilities of deductive databases and logic programming with higher-order and meta-data constructs, complex objects, and schema browsing. Its distinctive feature, a higher-order syntax with a first-order semantics, allows for efficient implementation with speeds comparable to Prolog. In fact, HiLog implementation in XSB [30, 26] together with tabulated query evaluation offers impressive performance with negligible penalty for higher-order syntax, thereby bringing the modeling capabilities of HiLog to practical realization.

The lack of sorts in HiLog, however, is somewhat of a problem in database applications, which led to a number of HiLog dialects such as DataHiLog [24]. This paper develops a comprehensive theory of sorts for HiLog. It supports HiLog's flexible higher-order syntax via a *polymorphic* and *recursive* sort structure, and it offers an easy and convenient mechanism to control the rules of well-formedness. By varying the sort structure we obtain a full spectrum of languages, ranging from classical predicate logic to the original (non-sorted) HiLog. In between, there is a number of interesting higher-order extensions of Datalog with various degrees of control over the syntax, including second-order predicate calculus with Henkin-style semantics, as described in [10]. We also discuss the benefits of using Sorted HiLog for modeling complex objects and for meta programming. Finally, Sorted HiLog can be easily incorporated into XSB, which makes its practical realization feasible.

*Work supported in part by the NSF grant IRI-9212074.

†Work supported in part by the NSF grant CCR-9102159 and a grant from New York Science and Technology Foundation RDG90173.

1 Introduction

HiLog [6] is a higher-order language for deductive databases and logic programming. It not only expands the limits of first-order logic programming and obviates the need for several non-logical features of Prolog, but also provides important features for databases, including schema browsing and nested and higher-order relations similar to those in COL [1] and LDL [2]. We refer the reader to [6] for the details of these applications. HiLog has been used by many researchers for various ends, such as for specifying types in logic programming [11, 34], for database query languages (*e.g.*, in the Glue-Nail! project [23]), and for object-oriented databases [19]. HiLog has been implemented as part of the XSB system with tabulated query evaluation [30],[1] and it runs at a very impressive speed compared to other deductive databases, such as LDL or Coral [26]. The on-going implementation [27] of C-logic and F-logic [16, 7] in XSB and its integration with HiLog will offer the ability to reason with objects and schema.

The main reason for the popularity of HiLog is its flexible syntax, the simplicity of its semantics, and the fact that its logical entailment is upward-compatible with classical logic. However, at the same time, it was felt that the syntax of HiLog is much too flexible, sometimes making it necessary to impose unwelcome restrictions on the range of logical variables in the program clauses.

Another problem is that HiLog does not have higher-order counterparts for various tractable sublogics of classical logic, such as Datalog, that are all-important in deductive databases. The Herbrand universe (which is the same as the Herbrand base) in HiLog is always infinite due to term application. One unpleasant off-shot of this is that the usual semi-naive bottom-up computation may not terminate, and even proper formulation of complexity results (analogous to those for Datalog) becomes an issue. As a result every query has to be analyzed for "finiteness" before it can be evaluated, even for programs with no applications of function symbols. To overcome this drawback, some researchers attempted to extract useful specialized sublogics out of HiLog. One example of this is DataHiLog proposed in [24]. However, strictly speaking, DataHiLog is not a sublanguage of HiLog in the sense in which Datalog is a sublanguage of classical Horn logic.

The third problem concerns the proof theory. Although HiLog has a sound and complete proof theory, the *direct* resolution-based proof theory of [6] has limitations, which are caused by the fact that Skolemization is not possible in some cases (see [6] for details).

In this paper, we show that all these problems can be rectified with a single mechanism, a *sorted* logic. A superposition of the idea of sorts and HiLog results in what we call *Sorted HiLog*. The idea of using sorts to control syntax is, of course, not new and one may even feel skeptical that applying this idea to HiLog may yield something original. However, as it turns out, developing a theory of sorts for HiLog is not a trivial matter. Sorts, as they are known in classical logic, are too limited when it comes to supporting the syntax of HiLog. Even the more elaborate theories [9, 14, 29] do not meet the requirements, as they were designed to address different problems. The requisite theory of sorts for HiLog should provide for more control over the syntax and, at the same time, be able to support those features of the syntax that make HiLog an attractive language.

The sort structure proposed in this paper is designed to accommodate both of these (seemingly conflicting) goals. The proposed sort structure is *polymorphic* and *recursive*. The logic itself is independent of the particular choice of a sort structure, and sorts can be viewed as a parameter to the logic. By varying the sort structure, we obtain a "continuum" of logic languages, ranging from ordinary HiLog to classical predicate calculus, with various decidable and higher-order extensions of Datalog in between. DataHiLog [24], mentioned earlier, is one of the special cases of Sorted HiLog and so is the second-order predicate calculus with Henkin-style semantics, described in [10, Section 4.4].

Before going into technical details of this paper, it may be useful to give a brief overview of the notions

[1] XSB and HiLog can be obtained via the anonymous FTP to *cs.sunysb.edu* in *pub/XSB/*.

of *sorts* and *types* as they apply to deductive languages.

Historically, *sorts* came from logic, where they were used to separate symbols into (usually disjoint) subdomains. Although sorts do not increase the expressive power of the logic, they may lead to clearer and more concise specifications; they also have been used to speed up automatic proofs [32].

Types, too, originate in logic [8]. However their introduction into logic-based programming languages is primarily due to the influence of functional and object-oriented programming, where it has been shown that sufficiently rich polymorphic type systems would allow the user to write interesting programs and, at the same time, guard against common programming errors.

In principle, logical sorts can be used in a similar way, since an ill-formed term in a program would certainly indicate a programming error. However, sorts lie at the very bottom of any logic — they are part of the very definition of what constitutes syntactically correct formulas in the language of the logic. As such they impose more generic constraints on the well-formedness of terms. For example, an individual term in predicate calculus may not appear as an atomic formula or be applied to other terms.

The work on type systems for logic programs follows two main approaches. One adopts the thesis that the semantics of typed logic programs should be based upon a typed logic [13, 15, 21, 22, 28]. Most of the proposals are designed mainly for predicate calculus like languages and cannot accommodate the flexible syntax of HiLog. The other approach is meta-theoretic in the sense that types are essentially constraints over type-free logic programs [20, 33, 17, 16]. A logic program may have a type-free logical semantics even though it may be ill-typed.

In the meta-logical setting, introducing sorts may be useful for several reasons. First, verifying well-formedness can be a "first cut" at ill-typed programs, since checking for well-formedness with respect to sorts is usually much cheaper than verifying well-typedness with respect to type systems, because the latter are usually much richer. Second, a non-trivial sort structure may significantly improve the efficiency of unification, thereby speeding up query execution. Finally, sorts lead to more natural and concise programs.

In accordance with this philosophy, the sort structure of HiLog does not support such essential elements of a viable type system as parametric and inclusion polymorphism. This is relegated to a richer, meta-level type system [4]. However, our sort system is arity-polymorphic and recursive, and despite its sophistication, well-formedness of HiLog formulas with respect to this sort system can be checked using a linear number of elementary operations such as retrieving the sort declaration of a variable. It should be noted, however, that the framework presented here can be easily extended to include parametric sorts. In contrast, support for inclusion polymorphism (*i.e.*, subsorts) is harder to provide because of complications with unification.[2]

This paper is organized as follows. Section 2 briefly sketches the original HiLog, as described in [6]. Section 3 introduces Sorted HiLog. Section 4 discusses several applications of Sorted HiLog. Section 5 describes the proof theory, and Section 6 concludes the paper.

2 Overview of HiLog

We assume some familiarity with HiLog, as it has been fairly well-researched in the literature [6, 23, 34, 25, 24, 19]. However, for easier reference, we provide a brief sketch of the syntax and semantics of HiLog.

HiLog is a higher-order logic that allows arbitrary terms to appear in contexts where only predicates and functions may occur in predicate calculus. As a result, higher-order predicates and functions can be defined with ease and, furthermore, higher-order constructs can be parameterized. This, for example,

[2] Problems also arise from the interaction of parametric sorts and subsorts. See [12, 4] for some work related to these issues.

allows the programmer to define generic predicates that accept other predicates as parameters and whose contents depend on these parameters. Despite the fact that HiLog treats predicates and functions as first-class entities, it maintains the semantic simplicity that is characteristic of predicate calculus.

The alphabet of a HiLog language consists of a countably infinite set of variables, $\mathcal{V}$, and a countable set of intensional parameters, $\mathcal{S}$, which is disjoint from $\mathcal{V}$. As usual in logic programming, we adopt the convention by which variables will be denoted via symbols that start with a capital letter. The set of HiLog *terms* is the smallest set that contains variables and intensional parameters, and that is closed under *term application*, *i.e.*, $t(t_1, ..., t_n)$ is a term if $t, t_1, ..., t_n$ $(n \geq 1)$ are terms. For instance, $p(X(p), b)(p(p))$ is a term.

Note that the above recursive definition of HiLog terms implies, in particular, that any symbol can be used with different arities (as in most Prologs) and that variables can occur in places that normally are reserved for function symbols.

Atomic formulas in HiLog are just the same as HiLog terms. Therefore, any symbol from $\mathcal{S}$ (and, in fact, any term) may occur in a context where predicates would be expected in classical logic.

Complex formulas are constructed out of the atomic ones using connectives and quantifiers in the standard manner, *i.e.*, $\phi \wedge \psi$, $\neg\eta$, and $(\forall X)(\phi \longleftarrow \psi)$ are formulas, provided that so are ϕ, ψ, and η. (The implication, "$\longleftarrow$", is defined as in classical logic: $\phi \longleftarrow \psi \equiv \phi \vee \neg\psi$.) For instance, the following is legitimate in HiLog:

$$\begin{array}{lll} \mathsf{call(X)} & \longleftarrow & \mathsf{X} \\ \mathsf{closure(R)(X,Y)} & \longleftarrow & \mathsf{R(X,Y)} \\ \mathsf{closure(R)(X,Y)} & \longleftarrow & \mathsf{R(X,Z)} \wedge \mathsf{closure(R)(Z,Y)} \end{array} \tag{1}$$

The first clause in (1) defines the familiar Prolog meta-predicate, call, and the other two rules define a parametric predicate, closure(R). Here, the term closure(R) is used in a predicate position, and the symbol closure can be viewed as a higher-order function that applies to binary relations. When it is supplied with an argument, r, it computes the transitive closure of r under the name closure(r).

The semantics of HiLog is designed to capture the different roles an object can play in different contexts. A semantic structure $\mathbf{M}$ is a quadruple $\langle U, U_{true}, \mathcal{F}, \mathcal{I} \rangle$, where:

- U is a nonempty set, called the domain of $\mathbf{M}$;
- $U_{true} \subseteq U$;
- $\mathcal{F}$ associates with each $d \in U$ and each $k > 0$ a k-ary function $U^k \longrightarrow U$, denoted by $d_{\mathcal{F}}^{(k)}$;
- $\mathcal{I}$ associates with each intensional parameter, $a \in \mathcal{S}$, an element in U.

Let ν be a variable assignment that associates to each variable, X, an element $\nu(X) \in U$. This assignment is extended to all terms as follows:

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

Let A be an atomic formula. Then $\mathbf{M} \models_\nu A$ holds precisely when $\nu(A) \in U_{true}$. Satisfaction of composite formulas is defined as in predicate calculus. For instance:

- $\mathbf{M} \models_\nu \phi \wedge \psi$ if and only if $\mathbf{M} \models_\nu \phi$ and $\mathbf{M} \models_\nu \psi$;
- $\mathbf{M} \models_\nu \neg\phi$ if and only if it is not true that $\mathbf{M} \models_\nu \phi$;

- $\mathbf{M} \models_\nu (\forall X)\phi$ if and only if for every other variable assignment, μ, that is identical to ν everywhere except on X, $\mathbf{M} \models_\mu \phi$ holds; etc.

One off-shot of the above intensional semantics is that two relations, say p and q, are considered equal (when their names occur in terms) if and only if the equality $p = q$ can be derived. Thus, it is possible for relations to be unequal even though they consist of the same tuples. An extensive discussion of the merits and demerits of intensional and extensional semantics appears in [6]. Here we will only mention that equality of relations and sets can be expressed in HiLog via additional axioms [6].

3 Sorted HiLog

In classical logic, it is common to distinguish between different categories of objects via the notion of sorts. In this section we describe Sorted HiLog, an extension of HiLog with a sort structure. Various applications of this enhanced version of HiLog are described in subsequent sections.

Recursive, Arity-Polymorphic Sorts

A traditional approach to sorts is to introduce a set of primitive sort names and then define functional sorts using these primitive sorts as building blocks. Each parameter and each variable in the logic language is then assigned a sort. The difficulty in extending this approach to a language like HiLog is two-fold. First, HiLog symbols must be poly-sorted, i.e., they must be acceptable in several different syntactic contexts. For instance, $p(a)$, $q(p(a))$, and $p(p, a)$ should all be considered well-formed, given a suitable sort for p. At the same time, by changing the sort structure we should be able to outlaw some of these contexts, say $q(p(a))$, if desired. Second, HiLog semantics treats every term as a constructor that can be used to build other terms. For instance, $p(a)$ is a term and so are $p(a)(a)$ and $p(a)(a)(a)$. Thus, if $s \to s'$ is the sort for p and s is the sort for a then $p(a)$ is of sort s'. To enable $p(a)$ to act as a constructor in $p(a)(a)$, the sort s' itself has to have internal structure that somehow includes the sort $s \to s'$. This leads us to a realization that a sort-scheme suitable for HiLog must be *recursive*. A formal development follows next.

Let Δ be a set of *sort names*. An *arrow expression* has the form $s_1 \times \cdots \times s_n \to s$, where $s_1, ..., s_n, s$, $n \geq 1$, are sort names. This expression is said to have *arity* $s_1 \times \cdots \times s_n$ (but sometimes we will simply say that the arity is n). The sort names $s_1, ..., s_n$ are *argument* sorts and s is the *target* sort.

A *sort* defined over Δ is any expression of the form $s\sigma$ where $s \in \Delta$ is the *name* of the sort and σ is a (possibly infinite) set of arrow expressions, called the *signature* of that sort. Empty signatures will be omitted, for brevity. A signature, σ, may be infinite, but it is assumed to satisfy the following *uniqueness* and *effectiveness* assumptions:

- *Uniqueness:* For every arity $s_1 \times \cdots \times s_n$, σ has at most one arrow expression of the form $s_1 \times \cdots \times s_n \to s$. (Note, that σ can have several arities, *i.e.*, expressions of the form $s_1 \times \cdots \times s_n$, for any given n.)

- *Effectiveness:* There is an effective "arrow-fetching" procedure that, for every arity, $s_1 \times \cdots \times s_n$, returns the arrow expression $s_1 \times \cdots \times s_n \to s$, if such an expression is in σ (in which case it is unique, by the *uniqueness* property); if σ contains no such expression, the procedure returns some agreed upon symbol, *e.g.*, *nil*.

The idea behind sorts with complex internal structure is that if f is a parameter symbol of sort $s\{a_1, \ldots, a_k, \ldots\}$ then, *as an individual,* it belongs to the domain of s and, as *a term constructor,* it can occur only in the contexts specified by the arrow expressions $a_1, ..., a_k, ...$

Note that if $s\sigma$ is a sort and s appears in an arrow expression in σ, then the definition of s acquires recursive flavor. The ability to define recursive sorts is necessary for supporting one important feature of HiLog syntax—terms with several levels of parentheses, such as in (1) above.

For instance, according to the well-formedness rules, below, if the symbols *closure*, X, Y, and P all have the same recursive sort, $s\{s \rightarrow s,\ s \times s \rightarrow s\}$, then the terms $closure(P)(X,Y)$ and $closure(P)(X)(Y)(P)$, will be well-formed. Informally, well-formedness holds by the following argument: $closure(P)$ is well-formed and has the sort s because so do *closure* and P separately, and because the signature of s has the arrow expression $s \rightarrow s$. Therefore, $closure(P)(X,Y)$ is also well-formed and has the sort s, because the signature of s (which is also the sort of $closure(P)$) has an arrow expression $s \times s \rightarrow s$, and because X and Y are variables of the sort s. The well-formedness of $closure(P)(X)(Y)(P)$ is established similarly.

On the other hand, $closure(P)(X,Y)(X,Y,Y)$ is not well-formed because, although the term $closure(P)(X,Y)$ is well-formed and has sort s, this sort does not possess an arrow expression that would allow this term to take three arguments.

Let Σ be a set of sorts over Δ. We say that Σ is *coherent* if different elements of Σ have different names (but elements of Σ having different names may have identical signatures).

We are now ready to define the language of Sorted HiLog. The alphabet of a Sorted HiLog language, $\mathcal{L}$, consists of:

- Δ — a set of sort names.
- Σ — a (possibly infinite) coherent set of sorts defined over Δ.
- For each sort $s \in \Sigma$:
  - $\mathcal{V}_s$ — a set of variables, which must be either empty or countably infinite.
  - $\mathcal{S}_s$ — a countable (empty, finite, or infinite) set of intensional parameters.

Since, according to the coherence requirement, different elements in Σ must have different names, a symbol of any sort, $s\sigma$, can unambiguously be said to have the sort s. Furthermore, without loss of information we can drop the name of any sort (leaving just the signature) if this name is not mentioned inside σ or in some other signature of Σ. This name can even be dropped from Δ, if we assume that the names of these "anonymous" sorts are unique new symbols, different from those mentioned in Δ.

Terms of each sort are defined inductively as follows:

- A variable or an intensional parameter of sort s is a term of sort s.
- If $t_1, ..., t_n$, where $n > 0$, are terms of sorts $s_1, ..., s_n$, respectively, and t is a term of sort $\bar{s}\{..., s_1 \times ... \times s_n \rightarrow s, ...\}$, then $t(t_1, ..., t_n)$ is a term of sort s.

It follows from the above that every term has a unique sort. However, since sorts encode *sets* of arrow expressions, a term can be applicable in many different contexts (even for the same number of arguments there can be several different contexts).

We will also need some control over syntactic formation of atomic formulas. For this purpose we introduce a subset of distinguished sorts, $\Delta_{\texttt{atomic}} \subseteq \Delta$, that designates certain sorts as being appropriate for atomic formulas. In other words, for a term to be counted as an *atomic formula*, it must have a sort, `atm`, such that $\texttt{atm} \in \Delta_{\texttt{atomic}}$. Complex formulas are built out of atomic ones in the standard manner using connectives and quantifiers.

Complexity of Checking Well-Formedness

The following result shows that, despite the polymorphic and recursive nature of sorts in Sorted HiLog, well-formedness of HiLog terms is easy to verify.

Proposition 3.1 (Complexity of Well-Formedness) *Consider a term, T, in ordinary, unsorted HiLog, and let $\mathcal{L}$ be a language of Sorted HiLog. Whether or not T is well-formed in $\mathcal{L}$ can be checked using a number of elementary operations that is linear in the size of T, where arrow-fetching and retrieval of the sort declaration of variables and intensional parameters (in Sorted HiLog) are considered to be elementary operations.*[3]

Proof: Consider a HiLog term, $S(R_1, ..., R_n)$, occurring inside T such that:

- The sort of S has already been determined;
- The sort of each argument, R_i, has been determined; and
- Each argument R_i has been marked as "processed."

We shall call such terms *eligible*. In the beginning, the only eligible terms are the symbols from $\mathcal{S}$ that occur in T. For instance, in $f(a, b(c)(d))(X(Y), e)$, such terms would be f, a, b, c, d, X, Y, and e. Note that, say, $X(Y)$ is not eligible, even though the sorts of X and Y are known from $\mathcal{L}$. This is because, in the beginning, Y is not yet marked as "processed." Likewise, $b(c)$ is not yet eligible at the first stage. However, at the next stage, $b(c)$ and $X(Y)$ become eligible, since their arguments, c and Y, are now "processed." We can now determine the sort of $b(c)$ and $X(Y)$, mark them as "processed" arguments, and go on to the next stage.

Formally, suppose $R_1, ..., R_n$ have been determined to have sorts s_1, ..., s_n, respectively. By the uniqueness assumption, the sort of S must have at most one arrow expression of the form $s_1 \times \cdots \times s_n \rightarrow \cdots$. If it has no such expression at all, then $S(R_1, ..., R_n)$ is ill-formed and then so is T. If there is such an arrow expression, say $s_1 \times \cdots \times s_n \rightarrow s$, then we conclude that $S(R_1, ..., R_n)$ has sort s and mark this term as "processed." This process continues in such a bottom-up manner until T is either declared ill-formed or is assigned a sort (and declared to be well-formed).

Clearly, this requires a linear number of arrow-fetching operations and operations that retrieve sort declarations for HiLog variables and parameters. □

Semantics of Sorted HiLog

The semantics for Sorted HiLog is a refinement of the semantics of ordinary HiLog, which is sketched in Section 2. A semantic structure, $\mathbf{M}$, is a quadruple $\langle U, U_{true}, \mathcal{F}, \mathcal{I} \rangle$, where

- U is the domain of $\mathbf{M}$; it has the structure of the union $\bigcup_{s \in \Delta} U_s$, where each U_s is *nonempty* and represents the subdomain corresponding to the sort name s;
- U_{true} is a subset of $\cup_{\mathtt{atm} \in \Delta_{\mathtt{atomic}}} U_{\mathtt{atm}}$;[4]

[3] It should be noted, however, that, for some sort structures, these may not be constant-time operations. For instance, retrieval of sort declaration may take $log(n)$ time, where n is the size of Σ, and arrow fetching may be arbitrarily complex. This is not the case, however, for the useful logics considered in this paper.

[4] Strictly speaking, it suffices to require only that $U_{true} \subseteq U$, because the elements of $U_{true} - \cup_{\mathtt{atm} \in \Delta_{\mathtt{atomic}}} U_{\mathtt{atm}}$ are intensions of terms that are *not* atomic formulas, and so they have no truth value, anyway.

- For each $\bar{s} \in \Delta$, $\mathcal{F}$ associates with each $d \in U_{\bar{s}}$ and each $k \geq 1$ a k-ary function $\mathcal{F}^{(k)}(d) : U^k \longrightarrow U$, denoted by $d^{(k)}_{\mathcal{F}}$. This function is subject to the restriction that if $s_1 \times \cdots \times s_k \to s$ is in the signature of $\bar{s}$ then $d^{(k)}_{\mathcal{F}}$ maps $U_{s_1} \times \ldots \times U_{s_k}$ into U_s;
- $\mathcal{I}$ associates with each intensional parameter, a, of sort $s \in \Delta$ an element $\mathcal{I}(a)$ in U_s.

Intensional equality in Sorted HiLog can be represented by the intensional parameter "=" whose sort may depend on the specific needs. The general theme is, however, that "=" must have signatures composed of the arrow expressions of the form $s \times \cdots \times s \to \mathtt{atm}$, where $s \in \Delta$ and $\mathtt{atm} \in \Delta_{\mathtt{atomic}}$. The equality symbol has fixed interpretation under which $(\mathcal{I}(=))^{(k)}_{\mathcal{F}}(u_1, \ldots, u_n) \in U_{true}$ if and only if all $u_1, \ldots, u_n$ coincide in U.

The semantics of terms and formulas is now defined as in HiLog (Section 2) with the only addition that the variable assignments have to respect sorts, i.e., for each $s \in \Sigma$ they must map V_s — the variables of sort s — into U_s, the domain of s. Given a sort-preserving variable assignment, ν, and an atomic formula A, we write $\mathbf{M} \models_\nu A$ precisely when $\nu(A) \in U_{true}$. Satisfaction of complex formulas is defined as in ordinary HiLog.

A model of a formula, ϕ, is any semantic structure, $\mathbf{M}$, such that $\mathbf{M} \models_\nu \phi$, for all ν. If ϕ is closed, then the truth (or falsehood) of $\mathbf{M} \models_\nu \phi$ does not depend on ν, and we can simply write $\mathbf{M} \models \phi$. The logical entailment relation, $\phi \models \psi$, is also defined as is customary in first-order logic: it holds if and only if every model of ϕ is also a model of ψ.

4 Applications

As explained earlier, the main drive behind the introduction of sorts was to provide a way to control the gap between the rigid well-formedness rules of classical predicate calculus and the sometimes-too-flexible syntax of ordinary HiLog, thereby enabling HiLog to better suit practical needs. As we shall see in this section, both predicate calculus and HiLog are special cases of Sorted HiLog—its two extremes, in a sense. We shall also describe several other sort structures with interesting rules of well-formedness, notably, various higher-order extensions of Datalog.

4.1 A Sort Structure for Ordinary HiLog

In ordinary HiLog, any term can be applied to any arbitrary number of terms. To make such expressions into well-formed terms on a Sorted HiLog, let $\Delta = \Delta_{\mathtt{atomic}} = \{\mathtt{atm}\}$ have exactly one symbol and suppose Σ contains exactly one sort:

$$\mathtt{atm}\{\mathtt{atm} \to \mathtt{atm}, \mathtt{atm} \times \mathtt{atm} \to \mathtt{atm}, \mathtt{atm} \times \mathtt{atm} \times \mathtt{atm} \to \mathtt{atm} \ldots\} \tag{2}$$

For instance, if t, a, and b had the sort $\mathtt{atm}$, the term $t(a)(t, b)$ would be well-formed and have the sort $\mathtt{atm}$ because:

a	is well-formed and has sort $\mathtt{atm}$;
$t(a)$	is well-formed and has the sort $\mathtt{atm}$, because of the arrow $\mathtt{atm} \to \mathtt{atm}$ in the signature of t's sort; and
$t(a)(t, b)$	is well-formed because $t(a)$'s sort, $\mathtt{atm}$, has the arrow $\mathtt{atm} \times \mathtt{atm} \to \mathtt{atm}$ in its signature.

The sort structure in (2) defines precisely the well-formedness rules used in ordinary HiLog, as described in Section 2. Notice that even though HiLog allows formation of terms with several levels of parentheses,

there is no need for highly nested functional sorts. That is, the components of an arrow expression are all primitive sort names, which is possible because of the recursive structure of **atm**.

Proposition 4.1 *The syntax and the semantics of ordinary HiLog of Section 2 and of Sorted HiLog with the sort structure (2) coincide.*

Proof: (Sketch) A language of ordinary HiLog can be viewed as a language of sorted HiLog with the sort structure (2) by assigning sort **atm** to every variable and intensional parameter. All terms of ordinary HiLog become terms of sort **atm** in Sorted HiLog. The domain of a semantic structure of ordinary HiLog corresponds to the domain of Sorted HiLog and so do the other components of semantic structures. The converse also holds for each step. Easy details are left as an exercise. □

4.2 A Sort Structure for Classical Predicate Calculus

The well-formedness rules of classical logic are fairly rigid: Each intensional parameter is designated to be a predicate or a function symbol and, furthermore, each symbol can be applied only to a fixed number of arguments that corresponds to the arity of the symbol. On top of this, arguments in a term can be constructed only out of function symbols. To capture this notion of well-formedness, let Δ contain the sort names $\mathbf{fun}_n$ and $\mathbf{pred}_n$, where $n \geq 0$, for function and predicate symbols, respectively. In addition, Δ has a sort name **trm** for terms and **atm** for atomic formulas. The set of sorts for atomic formulas has only one element: $\Delta_{\mathbf{atomic}} = \{\mathbf{atm}\}$. Suppose, further, that signatures in Σ are defined as follows:

$$\begin{array}{ll} \mathbf{fun}_n\{\times^n \mathbf{trm} \rightarrow \mathbf{trm}\} & \mathbf{atm}\{\,\} \\ \mathbf{pred}_n\{\times^n \mathbf{trm} \rightarrow \mathbf{atm}\} & \mathbf{trm}\{\,\} \end{array} \tag{3}$$

for each $n \geq 1$, where $\times^n s$ denotes $s \times \cdots \times s$ taken n times. The equality parameter, $=$, is given the sort $\mathbf{pred}_2$. Assuming that only **trm** has a nonempty set of variables, the language of Sorted HiLog with (3) as a sort system would become isomorphic to the language of first-order predicate calculus.

As shown in [6], in general, logical entailment in ordinary HiLog is *not* identical to the classical logical entailment, even if we restrict our attention to the subset of classical first-order formulas. Consider the following formula:

$$((q(a) \leftarrow r(a)) \wedge (q(a) \rightarrow r(a))) \leftarrow \forall X \forall Y (X = Y)$$

This is a well-formed formula both in predicate calculus and in HiLog. It is a valid HiLog formula, but not in predicate calculus. Therefore, the result, below, cannot be taken for granted.

Proposition 4.2 *The syntax and the semantics of Sorted HiLog* with the above sort structure *are equivalent to the syntax and the semantics of first-order predicate calculus.*

Proof: The syntactic part is obvious. We shall prove the equivalence of logical entailment in both logics, which is somewhat unexpected due to the aforesaid inequivalence result for ordinary HiLog in [6].

Let $\mathcal{L}_{PC}$ be a language of first-order predicate calculus. The corresponding language for Sorted HiLog, $\mathcal{L}_{SH}$, is derived from $\mathcal{L}_{PC}$ so that variables, n-ary function symbols, and n-ary predicate symbols of $\mathcal{L}_{PC}$ become variables and intensional parameters of sorts **trm**, $\mathbf{fun}_n$, and $\mathbf{pred}_n$, respectively, where $n \geq 0$. The mapping of terms and formulas between $\mathcal{L}_{PC}$ and $\mathcal{L}_{SH}$ is obvious, and goes in both directions.

Let $\mathbf{M}_{PC} = \langle U_{PC}, \mathcal{I}_{\mathcal{F}}, \mathcal{I}_{\mathcal{P}} \rangle$ be a semantic structure for $\mathcal{L}_{PC}$ (in predicate calculus). We construct a semantic structure, $\mathbf{M}_{SH} = \langle U, U_{true}, \mathcal{F}, \mathcal{I} \rangle$, for Sorted HiLog as follows:

- Define U to be the union of $U_{\mathbf{trm}}$, $U_{\mathbf{atm}}$, $U_{\mathbf{fun}_n}$ and $U_{\mathbf{pred}_n}$, for every $n \geq 1$, where
  - $U_{\mathbf{trm}} = U_{PC}$.

- $U_{\mathtt{fun}_n}$ and $U_{\mathtt{pred}_n}$ are the sets of intensional parameters of sort $\mathtt{fun}_n$ and $\mathtt{pred}_n$, respectively.
- $U_{\mathtt{atm}}$ is the set of all expressions of the form $p(d_1, ..., d_n)$, where $p \in U_{\mathtt{pred}_n}$ for some n, $d_1, ..., d_n$ are in $U_{\mathtt{trm}}$.
- If, for some sort s, the domain U_s comes out empty by the above rules, we set U_s to an arbitrary nonempty set, disjoint from everything else.

- Define U_{true} to be the set containing all elements of the form $p(d_1, ..., d_n)$ where p is an n-ary predicate symbol and $\mathcal{I}_{\mathcal{P}}(p)$ contains the tuple $\langle d_1, ..., d_n \rangle$.
- Define $\mathcal{I}(c) = \mathcal{I}_{\mathcal{F}}(c)$ for each c of sort $\mathtt{trm}$ and $\mathcal{I}(f) = f$ for each f of sort $\mathtt{fun}_n$ or $\mathtt{pred}_n$. (Observe the non-uniformity in the definition of $\mathcal{I}$, which stems from the non-uniformity with which predicate calculus treats terms and predicate and function symbols.)
- Define $\mathcal{F}(f) = \langle f_{\mathcal{F}}^{(1)}, f_{\mathcal{F}}^{(2)}, \ldots \rangle$ as follows. For every $f \in \mathtt{fun}_n$, $f_{\mathcal{F}}^{(n)}$ is defined to be $\mathcal{I}_{\mathcal{F}}(f)$; and for every $p \in \mathtt{pred}_n$, $p_{\mathcal{F}}^{(n)}$ is defined to map $d_1, \ldots, d_n$ to $p(d_1, ..., d_n)$, whenever $d_1, ..., d_n \in U_{\mathtt{trm}}$. For any other $u \in U$ and any other arity k, $u_{\mathcal{F}}^{(k)}(f)$ and $u_{\mathcal{F}}^{(k)}(p)$ can be an arbitrary function of k arguments.

It is easy to verify by structural induction that for every formula ϕ in $\mathcal{L}_{PC}$ and every variable assignment ν, $\mathbf{M}_{PC} \models_\nu \phi$ if and only if $\mathbf{M}_{SH} \models_\nu \phi$ (in the latter, we treat ϕ as a formula in $\mathcal{L}_{SH}$ *i.e.*, in Sorted HiLog.).

Similarly, given a semantic structure $\mathbf{M}_{SH}$ of $\mathcal{L}_{SH}$, we can derive a semantic structure $\mathbf{M}_{PC}$ of $\mathcal{L}_{PC}$ as follows:

- $U = U_{\mathtt{trm}}$;
- $\mathcal{I}_{\mathcal{C}}(c) = \mathcal{I}(c)$ for each constant symbol c;
- $\mathcal{I}_{\mathcal{F}}(f)$ is the restriction of $\mathcal{I}(f)_{\mathcal{F}}^{(n)}$ to $\times^n U_{\mathtt{trm}}$, for every n-ary function symbol f;
- $\mathcal{I}_{\mathcal{P}}(p)$ consists of all tuples $\langle d_1, ..., d_n \rangle$ over $U_{\mathtt{trm}}$ such that $\mathcal{I}(p)_{\mathcal{F}}^{(n)}(d_1, ..., d_n) \in U_{true}$, for every n-ary predicate symbol.

It can be verified that the meaning of formulas is preserved under this mapping. □

4.3 Second-Order Calculus with Henkin-style Semantics

If, in the sort structure of the previous subsection, we permit variables of sorts $\mathtt{fun}_n$ and $\mathtt{pred}_n$, we obtain a second-order predicate calculus with a semantics equivalent to Henkin's-style semantics described in [10, Section 4.4] (modulo the extensionality and comprehension axioms, which could be added). Due to space limitation, we shall not prove this result here.

The fact that Henkin-style calculus is a special case of Sorted HiLog is rather unexpected, since the semantics in [10, Section 4.4] seems radically different from the semantics of HiLog and because ordinary HiLog does not properly extend the aforesaid calculus. It is, therefore, interesting to see how a single semantic framework—HiLog and sorts—can model a wide variety of logics in a uniform way.

4.4 Higher-Order Datalog

Datalog is a sublanguage of pure Horn logic that has been extensively studied in the deductive database community (see, *e.g.*, [31]). The distinctive feature of Datalog is that function symbols with positive arities are not allowed. Due to this restriction, the Herbrand universe of every Datalog program is finite and consists of all the constants in the program.

Ordinary HiLog sketched in Section 2 does not support the notion of constants since any intensional parameter can be used as a function of any arity. However, HiLog with a suitably chosen sort structure is equivalent to Datalog. Moreover, by relaxing the sort structure, we can design various versions of Datalog with higher-order variables and predicates and still maintain finiteness of the Herbrand universe.

Let Δ consist of $\mathtt{pred}_n$, for predicate symbols, where $n \geq 1$; $\mathtt{atm}$, for atoms; and $\mathtt{trm}$, for terms. Suppose $\Delta_{\mathtt{atomic}} = \{\mathtt{atm}\}$, i.e., $\mathtt{atm}$ is the only sort for atomic formulas. Let, further, Σ consist of all the sorts in (3), less the sort fun_n, for each $n \geq 1$. Assuming that only the sort $\mathtt{trm}$ has a nonempty set of variables, we get ordinary Datalog. Introduction of variables of the sort $\mathtt{pred}_n$ ($n \geq 0$) and letting them be used as arguments to other predicate symbols yields a higher-order version of Datalog, which was dubbed DataHiLog in [24].

We can go still further and introduce structural parametric predicate symbols, such as closure(R) in (1). Caution must be taken here to preserve the decidability of Datalog. For instance, suppose that closure has the following sort (where cpred stands for "predicate constructor"):

$$\mathsf{cpred}\{\mathtt{pred}_2 \rightarrow \mathtt{pred}_2\} \tag{4}$$

and let r have the sort $\mathtt{pred}_2$. Then, the Herbrand base would no longer be finite, since there would be infinitely many predicates including r, closure(r), closure(closure(r)), ...

However, parametric predicates can still be supported, while maintaining a finite Herbrand universe. To break the recursion in (4), we could introduce a sort, $\mathsf{xpred_n}$, for "complex predicates," which is synonymous to $\mathsf{pred_n}$ in terms of the signature, but has a different sort name:

$$\mathsf{xpred_n}\{\times^{\mathsf{n}}\mathtt{trm} \rightarrow \mathtt{atm}\}, \quad n \geq 0 \tag{5}$$

Then we can modify (4) as follows:

$$\mathsf{cpred_{n,m}}\{\times_{\mathsf{i=1}}^{\mathsf{n}}\alpha_{\mathsf{i}} \rightarrow \mathsf{xpred_m}\}, \quad n, m \geq 0 \tag{6}$$

where each α_i is either $\mathtt{trm}$ or $\mathtt{pred}_k$, for some $k \geq 0$. Now, if closure had the sort $\mathsf{cpred_{1,2}}$ then closure(r)(X,Y) (where r is of sort $\mathtt{pred}_2$) would be a well-formed formula, while closure(closure(r))(X,Y) would be ill-formed, because closure(r) has the sort $\mathsf{xpred_2}$, which cannot be input to another application of closure.

4.5 Complex Objects

Complex objects are an extension of relational databases in which arguments of a relation may be relations themselves. In [6], we showed how HiLog can be used to model complex objects by providing "names" for relations. Consider the following example:

$$\mathsf{person(john, children(john))} \quad \begin{array}{l}\mathsf{children(john)(greg)}\\ \mathsf{children(john)(sarah)}\end{array}$$

Extensional equality of relations can be approximated by additional axioms [6]. Most languages of complex objects, such as COL [1], use a sorted or typed framework. For instance, person is a binary

predicate whose first argument is a term and the second argument is a unary relation, and children is a function, analogous to *data functions* in COL [1], that takes a term as an argument and returns a unary relation. This sort information gets lost when the same program is viewed as a formula in ordinary HiLog.

In contrast, in Sorted HiLog, one can assign sorts so that john, greg, and sarah would have the sort $\mathtt{trm}$; person would have the sort $\{\mathtt{trm} \times \mathtt{pred}_1 \rightarrow \mathtt{atm}\}$; and children would be a data function of the sort $\{\mathtt{trm} \rightarrow \mathtt{pred}_1\}$, where $\mathtt{pred}_1\{\mathtt{trm} \rightarrow \mathtt{atm}\}$.

4.6 Built-in Predicates in HiLog-based Programming Systems

In HiLog, a query such as "$?- \mathsf{P(a)}$" would retrieve all unary predicates, p, such that $\mathsf{p(a)}$ succeeds. However, in practical systems, not all bindings for P may be appropriate. For instance, there are built-in predicates, such as read and write, and without additional restrictions P may become bound to read or write, which is usually undesirable. Sorted HiLog provides a simple solution to these problems by dedicating one or more special sorts to these built-in predicates. For example, P may have the sort

$$\mathsf{pred}_1\{\mathtt{trm} \rightarrow \mathtt{atm}\}$$

while read and write would belong to a sort like

$$\mathsf{syspred}_1\{\mathtt{trm} \rightarrow \mathsf{sysatm}\}$$

where $\mathtt{atm}, \mathsf{sysatm} \in \Delta_{\mathtt{atomic}}$. In this way, instantiations for P will not include any of the system predicates, such as read, or write. Likewise, a query $?- X$, where X has the sort $\mathtt{atm}$, will not return answers of the form read(...). If, on the other hand, the query is about built-in predicates, we would have to use variables of sort pred_1 and sysatm, respectively.

4.7 Encapsulation and Modules

A module in logic programming encapsulates a collection of predicate definitions. There are two problems with developing a logical theory of modules. One is to avoid name clashes between predicates used in different modules. The other is to represent a module definition as a logic formula. The latter requires a higher-order framework since predicates can be passed as parameters and returned as results. The development, below, follows the outline of [3].

A program now consists of a finite set of clauses and a finite set of basic module definitions. Each basic module definition consists of a module interface and a body that contains a finite number of clauses. The concrete syntax of a module definition may be the following:

$$\begin{array}{lll} \mathsf{closure(In, Out)} & \{ & \\ & \quad \mathsf{Out(X,Y)} & \longleftarrow \ \mathsf{In(X,Y)} \\ & \quad \mathsf{Out(X,Y)} & \longleftarrow \ \mathsf{In(X,Z)} \wedge \mathsf{Out(Z,Y)} \\ & \} & \end{array}$$

Here, In is the input predicate variable, and Out is the variable exported by the module; it is instantiated to the transitive closure of In computed by the module. In [3], the above abstract syntax is given meaning using the following formula:

$$\begin{array}{lll} \forall \mathsf{In}\, \exists \mathsf{Out}\, (& \mathsf{closure(In, Out)} & \\ \wedge & \forall \mathsf{X} \forall \mathsf{Y} (\mathsf{Out(X,Y)} \longleftarrow \mathsf{In(Y,Y)}) & \\ \wedge & \forall \mathsf{X} \forall \mathsf{Y} \forall \mathsf{Z} (\mathsf{Out(X,Y)} \longleftarrow \mathsf{In(X,Z)} \wedge \mathsf{Out(Z,Y)} \quad) & \end{array} \tag{7}$$

Notice that encapsulated predicates are represented by existential variables since only variables have local scope in logic and only existentially quantified variables can represent objects inaccessible through other variables. It is precisely this style of quantification that precludes changing the definition of encapsulated predicates from outside the module.

A query or any other clause may use the module closure just as any other predicate, e.g.,

$$?- \ \mathsf{closure(parent,\ \ Ancestor)} \wedge \mathsf{Ancestor(bill, X)} \wedge \mathsf{closure(boss, Mngr)} \wedge \mathsf{Mngr(X, bob)}$$

This query would return all descendants of bill who are managers of bob, provided that parent(a,b) means that a is a parent of b and boss(c,d) stands for "c is a boss of d."

Now, in [3], the expression (7) was understood as a formula in second-order predicate calculus. With the advent of HiLog, it turned out that viewing (7) as a HiLog formula leads to a more tractable semantics of logical modules. If ordinary HiLog gives a satisfactory semantics for modules, then how does Sorted HiLog fit into the picture?

One problem in (7) is that there is an existential quantifier of the kind that cannot be handled directly by most logic programming systems. A natural way to implement modules, then, is to use Skolemization to transform module definitions into ordinary Horn clauses. Since Skolemization preserves unsatisfiability, query answers obtained by refutational proof procedures, such as SLD-resolution, are preserved. Unfortunately, the problem with Skolemization found in ordinary HiLog (see the Section 5.1 later) precludes this natural implementation. In contrast, as we shall see, Skolemization in Sorted HiLog can be performed pretty much as in classical logic and, thus, it appears to be a better vehicle for implementing logical modules.

4.8 Sorted Meta Programming

Prolog applications often rely on meta-programming techniques, which require flexibility of the kind HiLog syntax can provide. For instance, consider the following program:

$$\begin{array}{l}
\mathsf{call(A)} \longleftarrow \mathsf{A} \\
\mathsf{call(P, X)} \longleftarrow \mathsf{ispredicate(P)} \wedge \mathsf{P(X)} \\
\mathsf{call(P, X, Y)} \longleftarrow \mathsf{ispredicate(P)} \wedge \mathsf{P(X, Y)} \\
\mathsf{call(P(X), Y)} \longleftarrow \mathsf{ispredicate(P)} \wedge \mathsf{P(X, Y)} \\
\mathsf{call(P(X), Y, Z)} \longleftarrow \mathsf{ispredicate(P)} \wedge \mathsf{P(X, Y, Z)} \\
\mathsf{ispredicate(ispredicate)} \\
\mathsf{ispredicate(call)} \\
\mathsf{ispredicate(p)} \qquad \% \ \text{for every predicate, p, in the program}
\end{array} \tag{8}$$

Here, call is a meta-predicate[5] that 1) "executes" atomic formulas passed to it as an argument (Clause 1 in (8)); 2) applies predicate symbols to arguments and then executes the resulting atoms (Clauses 2 and 3); and 3) accepts a "partial-load" atoms as a first argument and then applies them to an appropriate number of terms (Clauses 4 and 5).

The problem with the above program is that if call is passed a wrong first argument (that is not an atom, a partial-load atom, or a predicate) the subgoal will simply fail without alerting the user to the problem. Current systems of sorts or types are not expressive enough to handle meta-programs, such as above. However, in Sorted HiLog, ill-formed expressions can be detected at compile time by specifying sorts appropriately. To see this, let Δ contain sort names for:

- function and predicate symbols: $\mathtt{fun}_n$ and $\mathtt{pred}_n$, where $n \geq 0$;

[5] Of course, it is not a meta-predicate in HiLog, but it is in classical logic programming.

- terms: **trm**;
- ordinary and built-in atomic formulas: **atm** and **sysatm**;
- partial-load atomic formulas: **partatm**;
- meta-predicates call and ispredicate: **callsort** and **ispredsort**.

The set of sorts for atomic formulas consists of two elements: $\Delta_{\mathtt{atomic}} = \{\mathtt{atm}, \mathtt{sysatm}\}$. Suppose, further, that signatures in Σ are defined as follows:

$$
\begin{array}{rll}
\mathtt{trm} & \{\ \} & \\
\mathtt{atm} & \{\ \} & \\
\mathtt{sysatm} & \{\ \} & \\
\mathtt{fun}_n & \{\ \times^n \mathtt{trm} \rightarrow \mathtt{trm}\ \} & \text{for all } n \geq 0 \\
\mathtt{ispredsort} & \left\{ \begin{array}{l} \mathtt{pred}_n \rightarrow \mathtt{sysatm} \\ \mathtt{callsort} \rightarrow \mathtt{sysatm} \\ \mathtt{ispredsort} \rightarrow \mathtt{sysatm} \end{array} \right\} & \text{for all } n \geq 0 \\
\mathtt{pred}_n & \left\{ \begin{array}{l} \times^n \mathtt{trm} \rightarrow \mathtt{atm} \\ \times^m \mathtt{trm} \rightarrow \mathtt{partatm} \end{array} \right\} & \text{for all } n \geq 0 \text{ and } 1 \leq m < n \\
\mathtt{callsort} & \left\{ \begin{array}{l} \mathtt{atm} \rightarrow \mathtt{sysatm} \\ \mathtt{sysatm} \rightarrow \mathtt{sysatm} \\ \mathtt{partatm} \times \mathtt{trm} \rightarrow \mathtt{sysatm} \\ \mathtt{partatm} \times \mathtt{trm} \times \mathtt{trm} \rightarrow \mathtt{sysatm} \end{array} \right\} &
\end{array}
\tag{9}
$$

Under this sort structure, the above program is well-formed, provided that the variable A in (8) has the sort **atm** or **sysatm**; P has sort $\mathtt{pred}_1$, $\mathtt{pred}_2$, or $\mathtt{pred}_3$, as appropriate; and the variables X, Y, and Z are of sort **trm**. Moreover, passing a non-atom to call in the first clause in (8) will be impossible, since the resulting term will not be well-formed. Similarly, the variables in other clauses in (8) will have to be bound to appropriate entities in order to comply with the above sort structure.

Although sort systems for meta-programming need further investigation, we believe that the polymorphic and recursive sort structure of Sorted HiLog represents a step in the direction towards achieving this goal. One possible extension here is the incorporation of subsorts, as in order-sorted logics [14, 29]. However, a difficulty arises from the interaction between subsorts and recursive poly-sorts. It seems that a "brute-force" approach to such richer sort systems does not lead to an elegant solution.

5 Proof Theory (Sketch)

A direct, resolution-based proof theory for Sorted HiLog can be developed along the lines of [6]. The outline of this development is as follows. First, all sentences must be converted into the prenex normal form and then Skolemized. The proof procedure itself consists of three inference rules, resolution, factorization, and paramodulation, which are similar to those in ordinary HiLog. Proving soundness and completeness is done by encoding the logic in classical predicate calculus and then establishing an isomorphism between proofs in Sorted HiLog and the corresponding proofs in classical logic.

We shall not reproduce all the details here, which can be found in [6]. Instead, we will concentrate on the adjustments that have to be made. The only significant difference concerns the Skolemization process. There also are differences in the notion of unification and in the encoding process, which now must take sorts into account.

5.1 Skolemization in Sorted HiLog

In Sorted HiLog, Skolemization is performed in an almost standard way. However, what is interesting here is not how it is done, but rather why it didn't work in ordinary HiLog [6].[6] Consider the following example from [6]:

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

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

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

By direct inspection, we verify that ϕ is satisfiable in HiLog, while $\phi*$ is not. The latter holds because $\mathsf{p}(\mathsf{h}(\mathsf{g}), \mathsf{g}(\mathsf{h}(\mathsf{g}))) \wedge \neg \mathsf{p}(\mathsf{h}(\mathsf{g}), \mathsf{g}(\mathsf{h}(\mathsf{g})))$ is an instance of $\phi*$. The reason for this misbehavior is that, in general, HiLog semantic structures may not have sufficient supply of Skolem functions, since each such function must be associated with an *existing* element of the domain. The round-about method of Skolemization described in [6] works for finite sets of formulas only.

Despite the semantic similarity between sorted and ordinary HiLog's (and the fact that the latter is just a special case of the former), the standard Skolemization procedure does not cause problems in the sorted case. The idea is to introduce a *new sort* for each new Skolem function of arity ≥ 1. Skolem constants (arity 0) should keep the sort of the existential variables they came from. For instance, to Skolemize $\psi = (\forall X \forall Y \exists Z)\phi$, where X, Y, and Z have sorts s_1, s_2, and s_3, respectively, we would introduce a new Skolem function, g, of the sort $\bar{s}\{s_1 \times s_2 \rightarrow s_3\}$, where $\bar{s}$ is a new sort name.

If a semantic structure, $\mathbf{M} = \langle U, U_{true}, \mathcal{I}, \mathcal{F} \rangle$, satisfies ψ then we can construct another structure, $\mathbf{M}' = \langle U', U_{true}, \mathcal{I}', \mathcal{F}' \rangle$, with domain $U' = U \cup U_{\bar{s}}$, where $U_{\bar{s}} = \{v\}$ is the subdomain allocated to the new sort $\bar{s}$ and v is a new element. Then $\mathcal{I}'$ would behave exactly as $\mathcal{I}$, except that it maps the new Skolem function, g, into v. (If g happens to be a Skolem constant then $\mathcal{I}'(g)$ is chosen as in predicate calculus, i.e., it is the element whose existence is asserted through the existential quantifier.) The functional meaning of v, $\mathcal{F}(v)$, is chosen as in the standard proof of Skolem's theorem in predicate calculus. Since the original formula ψ has no variables of sort $\bar{s}$, $\mathbf{M}'$ satisfies the Skolemization of ψ, and the aforesaid problem with HiLog does not arise.

5.2 An Encoding of Sorted HiLog in Predicate Calculus

Just as ordinary HiLog, Sorted HiLog can be encoded in predicate calculus, as shown below. The difference between the two encodings has to do mainly with the sorts.

Let $\mathcal{L}_{SH}$ be a Sorted HiLog language with a coherent set Σ of sorts defined over a set Δ of sort names, a set $\mathcal{V}_s$ of variables and a set $\mathcal{S}_s$ of intensional parameters for each $s \in \Sigma$. We define $\mathcal{L}_{PC}^{encode}$ to be a language of predicate calculus with the set of variables $\cup_{s \in \Sigma} \mathcal{V}_s$, the set of constant symbols $\cup_{s \in \Sigma} \mathcal{S}_s$, a predicate symbol s for each sort name $s \in \Delta$, a predicate "call", and, for each $n \geq 1$, an $(n+1)$-ary function symbol apply_{n+1}. Given a Sorted HiLog formula, ϕ, its encoding in predicate calculus, ϕ^*, is determined by the following transformation rules. In these rules, $\mathsf{encode_a}$ is a transformation that encodes Sorted HiLog terms that appear in contexts where they are interpreted as atomic formulas; $\mathsf{encode_t}$ encodes these terms in all other contexts.

- $\mathsf{encode}_t(X) = X$, for each variable $X \in \mathcal{V}_s$.
- $\mathsf{encode}_t(c) = c$, for each intensional parameter $c \in \mathcal{S}_s$.

[6] In [5] it was erroneously claimed that Skolemization can be done in a manner similar to predicate calculus.

- $\mathsf{encode}_t(\, t(t_1, ..., t_n)\,) \;=\; \mathsf{apply}_{n+1}(\, \mathsf{encode}_t(t), \mathsf{encode}_t(t_1), ..., \mathsf{encode}_t(t_n)\,)$.
- $\mathsf{encode}_a(A) \;=\; \mathsf{call}(\, \mathsf{encode}_t(A)\,)$, where A is any HiLog atomic formula.
- $\mathsf{encode}_a(\, A \vee B\,) \;=\; \mathsf{encode}_a(A) \vee \mathsf{encode}_a(B)$.
- $\mathsf{encode}_a(\, A \wedge B\,) \;=\; \mathsf{encode}_a(A) \wedge \mathsf{encode}_a(B)$.
- $\mathsf{encode}_a(\, \neg A\,) \;=\; \neg \mathsf{encode}_a(A)$.
- $\mathsf{encode}_a(\, (\forall X)A\,) \;=\; (\forall X)(s(X) \rightarrow \mathsf{encode}_a(A))$, where X is a variable of the sort s.
- $\mathsf{encode}_a(\, (\exists X)A\,) \;=\; (\exists X)(s(X) \wedge \mathsf{encode}_a(A))$, where X is a variable of the sort s.

Given a semantic structure of Sorted HiLog, $\mathbf{M} =< U, U_{true}, \mathcal{I}_{SH}, \mathcal{F} >$, the corresponding structure in predicate calculus, $\mathsf{encode}(\mathbf{M}) =< U, \mathcal{I}_{\mathcal{F}}, \mathcal{I}_{\mathcal{P}} >$, is defined as follows:

- $\mathcal{I}_{\mathcal{F}}(c) \;=\; \mathcal{I}_{SH}(c)$, for each $c \in \mathcal{S}_s$ and $s \in \Sigma$.
- $\mathcal{I}_{\mathcal{F}}(\mathsf{apply}_{n+1})(u, u_1, ..., u_n) \;=\; (u)^{(n)}_{\mathcal{F}}(u_1, ..., u_n)$.
- $\mathcal{I}_{\mathcal{P}}(\mathsf{call}) \;=\; U_{true}$.
- $\mathcal{I}_{\mathcal{P}}(s) = U_s$, for each $s \in \Delta$.
- The equality predicate, "=", has a standard, diagonal interpretation in $\mathsf{encode}(\mathbf{M})$: $\mathcal{I}_{\mathcal{P}}(=) \stackrel{\text{def}}{=} \{< u, u > |\; u \in U\}$.

The following lemma is an extension of the encoding theorem of HiLog [6]:

Lemma 5.1 (Encoding Lemma) *Let ϕ be a formula and $\mathbf{M}$ be a semantic structure of Sorted HiLog. Let ν be a sort-preserving variable assignment for the free variables in ϕ. Then*

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

Proof: By structural induction, $\nu(\mathsf{t}) = \nu(\mathsf{encode_t(t)})$, for every term, t, of Sorted HiLog. Consider now an atomic formula, A, in Sorted HiLog. By the definition, $\mathsf{encode_a(A)} = \mathsf{call}(\mathsf{encode_t(A)})$. Therefore, $\mathbf{M} \models_\nu \mathsf{A}$

if and only if $\nu(\mathsf{A}) \in U_{true}$;
if and only if $\nu(\mathsf{encode_t(A)}) \in \mathcal{I}_{\mathcal{P}}(\mathsf{call})$;
if and only if $\mathsf{encode}(\mathbf{M}) \models_\nu \mathsf{call}(\mathsf{encode_t(A)})$;
if and only if $\mathsf{encode}(\mathbf{M}) \models_\nu \mathsf{encode_a(A)}$.

Thus, we have proved the claim for atomic formulas. The rest of the proof is an easy induction of the structure of HiLog formulas. □

The standard meaning of "=" in Sorted HiLog is enforced by the following axiom, $\mathcal{E}$, in predicate calculus:

$$\mathsf{call}(\mathsf{apply}_3(=, X, Y)) \equiv (X = Y)$$

This leads to the following result:

Corollary 5.2 *Let ϕ be a sentence (*i.e.*, a formula without free variables) in Sorted HiLog. Then ϕ is valid if and only if the formula $\mathsf{encode}_a(\phi) \longleftarrow \mathcal{E}$ is valid in predicate calculus.*

Proof: Let $\mathcal{L}_{SH}$ be a language of Sorted HiLog. It is easy to see that the function encode is a 1-1 and onto mapping from Sorted HiLog structures over $\mathcal{L}_{SH}$ to classical semantic structures over $\mathcal{L}^{encode}_{PC}$ that satisfy $\mathcal{E}$. Indeed, we could use equations for encode "in reverse" to define a function, decode, such that $\mathsf{decode}(\mathsf{encode}(\mathbf{M})) = \mathbf{M}$. The claim now follows from Lemma 5.1. □

5.3 Unification in Sorted HiLog

A substitution in a sorted logic has to respect the sorts of variables, so that every instance of any well-formed term or formula would also be well-formed. This leads to the following definition. A mapping

$$\theta : \cup_{s \in \Delta} \mathcal{V}_s \longrightarrow \{\text{Sorted HiLog terms}\}$$

is a *substitution* if it has finite domain (i.e., the set $\{ X \mid \theta(X) \neq X \}$ is finite) and respects sorts of variables (i.e., the sorts of X and $\theta(X)$ are the same). Application of a substitution, θ, to a term (or a formula), t, is denoted by $t\theta$; composition of substitutions is defined in the standard manner.

A *unifier* of a set of terms, E, is a substitution, σ, such that for every pair of terms, $t_1, t_2 \in E$, $t_1\sigma$ and $t_2\sigma$ are identical. The set E is *unifiable* if E has a unifier σ. It is a *most general* unifier if for every unifier θ for E there is a substitution, λ, such that $\theta = \sigma\lambda$.

Any substitution, θ, can be represented as a set of equations of the form $X = t$, where X and t have the same sort. Following [18], we 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 of the same sort. 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 that do not occur in any t_j $(1 \leq j \leq n)$.

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

To unify a pair of terms t and s, the unification algorithm first checks if they have the same sort. If so, an equation set $\{t = s\}$ is created; otherwise, it halts with failure. Given a finite equation set E, the algorithm proceeds non-deterministically by choosing an equation $e \in E$ to which it applies the following transformations whenever they are applicable:

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 $t = s$, $t_1 = s_1$, ..., $t_n = s_n$, provided that t, s and t_i, s_i pairwise have the same sort. Otherwise, halt with failure.
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 other equations.

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

Theorem 5.3 (Unification Theorem) *The unification algorithm applied to a finite set of equations, E, returns a finite set of equations, E^*, 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 a pair of Sorted HiLog terms, s and r, is unifiable if and only if so are $\mathsf{encode_t}(s)$ and $\mathsf{encode_t}(r)$; the latter are unifiable 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 sorted HiLog in predicate calculus transforms the above unification algorithm for HiLog into the corresponding algorithm for predicate calculus, described in [18]. The theorem now follows from these two facts. A direct proof can also be obtained as a simple adaptation of the proof in [18]. □

6 Conclusion

We presented *Sorted HiLog*, a logic that enhances ordinary HiLog [6] with recursive poly-sorts. The rules of well-formedness for Sorted HiLog enable it to simulate the syntax of a wide range of logic languages, from predicate calculus to the original version of HiLog, as described in [6]. Within this range, we find a pair of decidable, higher-order extensions of Datalog and also the second-order predicate calculus with Henkin-style semantics (see [10, Section 4.4]).

Applications of deductive databases and logic programming demand more flexible syntax than what is offered by languages based on classical predicate calculus (or on the "pure" subset of Prolog). This need is partly filled by the ordinary HiLog, as described in [6]. However, to facilitate understanding and debugging of programs, expressions are often classified into different sorts and restrictions are imposed to guarantee well-formedness. Sorted HiLog is capable of accommodating both of these seemingly conflicting goals. It also appears to capture some important aspects of well-formedness in meta-programs and, we believe, provides a suitable basis for further studies of sort systems for meta programming and schema manipulation. With efficient implementation as part of the XSB system, Sorted HiLog would allow users to choose sort structures for different database applications—all without sacrificing the syntactic flexibility of HiLog.

References

[1] S. Abiteboul and S. Grumbach. COL: A logic-based language for complex objects. In *Workshop on Database Programming Languages*, pages 253–276, Roscoff, France, September 1987.

[2] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Sets and negation in a logic database language (LDL). Technical report, MCC, 1987.

[3] W. Chen. A theory of modules based on second-order logic. In *IEEE Symposium on Logic Programming (SLP)*, pages 24–33, September 1987.

[4] W. Chen and M. Kifer. Polymorphic types in higher-order logic programming. Technical Report 93/20, Department of Computer Science, SUNY at Stony Brook, December 1993.

[5] W. Chen, M. Kifer, and D.S. Warren. HiLog: A first-order semantics for higher-order logic programming constructs. In *North American Conference on Logic Programming (NACLP)*, October 1989.

[6] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.

[7] W. Chen and D.S. Warren. C-logic for complex objects. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 369–378, March 1989.

[8] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[9] A.G. Cohn. A more expressive formulation of many sorted logic. *Journal of Automated Reasoning*, 3:113–200, 1987.

[10] H.B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[11] T. Fruehwirth. Polymorphic type checking for Prolog in HiLog. In *6th Israel Conference on Artificial Intelligence and Computer Vision*, Tel Aviv, Israel, 1989.

[12] J.A. Goguen and J. Meseguer. Eqlog: Equality, types, and generic modules for Logic Programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*. Prentice Hall, 1986.

[13] M. Hanus. Polymorphic higher-order programming in prolog. In *Intl. Conference on Logic Programming (ICLP)*, pages 382–397, Lisboa, Portugal, 1989. MIT Press.

[14] M. Hanus. Parametric order-sorted types in logic programming. Technical Report 377, Universitaet Dortmund, Fachbereich Informatik, Dortmund, FRG, January 1991.

[15] P. Hill and R. Topor. A semantics for typed logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 1–62. The MIT Press, 1992.

[16] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical Report 93/06 (a revision of 90/14), Department of Computer Science, SUNY at Stony Brook, April 1993. To appear in Journal of ACM. Available in *pub/TechReports/kifer/flogic.ps.Z* by anonymous ftp to *cs.sunysb.edu*.

[17] M. Kifer and J. Wu. A first-order theory of types and polymorphism in logic programming. In *Intl. Symposium on Logic in Computer Science (LICS)*, pages 310–321, Amsterdam, The Netherlands, July 1991. Expanded version: TR 90/23 under the same title, Department of Computer Science, University at Stony Brook, July 1990.

[18] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[19] I.S. Mumick and K.A. Ross. An architecture for declarative object-oriented databases. In *Proceedings of the JICSLP-92 Workshop on Deductive Databases*, pages 21–30, November 1992.

[20] A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[21] G. Nadathur and D. Miller. Higher-order horn clauses. *Journal of ACM*, 37(4):777–814, October 1990.

[22] G. Nadathur and F. Pfenning. Types in higher-order logic programming. In F. Pfenning, editor, *Types in Logic Programming*, pages 245–283. The MIT Press, 1992.

[23] G. Phipps, M.A. Derr, and K.A. Ross. Glue-Nail: A deductive database system. In *ACM SIGMOD Conference on Management of Data*, pages 308–317, 1991.

[24] K.A. Ross. Relations with relation names as arguments: Algebra and calculus. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, May 1992.

[25] K.A. Ross. On negation in HiLog. *Journal of Logic Programming*, 18(1):27–53, January 1994.

[26] K. Sagonas, T. Swift, and D.S. Warren. XSB as an efficient deductive database engine. In *ACM SIGMOD Conference on Management of Data*, pages 442–453, May 1994.

[27] Konstantinos F. Sagonas and David S. Warren. A compilation scheme for HiLog. Submitted for publication, 1994.

[28] G. Smolka. Logic programming with polymorphically order-sorted types. In J. Grabowski, P. Lescanne, and W. Wechler, editors, *Algebraic and Logic Programming*, volume 343 of *Lecture Notes in Computer Science*, pages 53–70. Springer-Verlag, 1988.

[29] G. Smolka, W. Nutt, J.A. Goguen, and J. Meseguer. Order-sorted equational computation. Technical Report SEKI Report SR-87-14, Universität Kaiserslautern, West Germany, December 1987.

[30] T. Swift and D.S. Warren. Compiling OLDT evaluation: Background and overview. Technical report, Department of Computer Science, SUNY at Stony Brook, 1992.

[31] J.F. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 1.* Computer Science Press, 1988.

[32] C. Walther. A mechanical solution of Schubert's Steamroller by many-sorted resolution. *Artificial Intelligence*, 26:217–224, 1985.

[33] J. Xu and D.S. Warren. A type inference system for Prolog. In *Joint Intl. Conference and Symposium on Logic Programming (JICSLP)*, pages 604–619, 1988.

[34] E. Yardeni, T. Fruehwirth, and E. Shapiro. Polymorphically typed logic programs. In *Intl. Conference on Logic Programming (ICLP)*, Paris, France, June 1991.