Efficient Execution of HiLog in WAM-based Prolog implementations

Konstantinos Sagonas **David S. Warren**
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
{kostis,warren}@cs.sunysb.edu

Abstract

In this paper we address the problem of efficiently implementing HiLog, a logic programming language with higher-order syntax and first-order semantics. In contrast to approaches proposed in the literature that modify, or abandon the WAM framework in order to implement HiLog, our approach to the problem stems from a belief that the WAM should be an adequate abstract machine for the execution of any logic language with first-order semantics. To show how to implement HiLog by *staying within the WAM framework*, we identify the reasons for poor performance characteristics of HiLog programs, present requirements for efficient HiLog execution, and propose a complete solution to the problem.

Our proposal, which can be viewed either as a compile-time program specialisation preprocessing step, or as an enhancement to the HiLog encoding in predicate calculus presented by Chen, Kifer, and Warren in [1], allows HiLog to be efficiently implemented on any Prolog system by simply modifying Prolog's input/output predicates to handle terms that are expressed using the flexible higher-order syntax of HiLog. We formally prove that our proposal allows all HiLog programs that do not use any higher-order features to execute at the same speed as Prolog programs. Furthermore, we present performance results showing that generic HiLog predicates when compiled using the compilation scheme execute at least an order of magnitude faster than generic Prolog predicates, and with only minimal overhead compared to non-generic Prolog ones.

Everything should be made as simple
as possible, but not simpler.
— *Albert Einstein*

1 Prologue

In [1], Chen, Kifer, and Warren explored the fundamental principles underlying higher-order logic programming, and, in particular, shed new light on why and how the "higher-order" Prolog features appear to work in practice. The key insight of that work is that what many applications require is *a higher-order syntax and a first-order semantics*. As a result, a novel logic was proposed, called HiLog, that expands the limits of first-order logic programming and provides a clean first-order declarative semantics to much of higher-order logic programming, obviating the need for several non-logical features of Prolog.

Since its first proposal, HiLog has gained noticeable popularity among many researchers in several different areas. Besides the area of logic programming, where it has been proposed as a framework to specify polymorphic types [23] and set abstractions [2], HiLog has also been used as a declarative query language for deductive [6, 16] and object-oriented databases [11, 12]. Despite its popularity, the viability of HiLog as a logic programming language depends on whether it can be implemented efficiently. In particular, the performance of a HiLog implementation must be able to compete with that of modern Prolog systems. Before we can expect Prolog programmers to use HiLog, it is absolutely critical that Prolog programs do not degrade in performance when run under HiLog implementations.

As mentioned, HiLog has a first-order semantics and HiLog programs admit a natural encoding in predicate calculus. Direct use of this encoding provides a naive implementation of HiLog. The remaining issue in claiming an actual implementation of HiLog is the very poor performance of the encoded programs, especially when compiled in a WAM-based Prolog system using standard compilation techniques. The encoded HiLog programs generally contain few predicates with many clauses, and calls to these predicates do not execute efficiently with the discrimination provided by one argument indexing alone. Another problem is that WAM-based execution of HiLog encodings may generate extra overhead due to excessive record copying. The poor performance of the encoded programs has caused several research groups to abandon the idea of encoding HiLog in predicate calculus, and investigate alternative ways of implementing HiLog. The Glue-Nail system from Stanford [6] implements a subset of HiLog by an expensive adaptive optimisation technique that optimises queries to HiLog predicates at run time. An ongoing effort from the University of Cape Town [14] proposes a WAM extension to implement HiLog. The basic proposed change is in the cell representation of structures; the functor position of a structure is stored as a normal cell, as opposed to the WAM, where it is stored as a special functor cell. Also, the arity must be stored separately in an arity cell, thus requiring an extra cell for each structure (at least for WAM implementations that store cells in a single machine word). Besides the space overhead, there is a corresponding increase in the time cost of manipulating structures and choice-point frames. As a result, programs that do not use any higher-order features (i.e. Prolog programs) are unnecessarily penalised by these overheads.

There are a number of reasons why we believe that it is better **not** to modify the underlying WAM implementation in an attempt to implement HiLog. All low-level Prolog optimisations and compilation techniques [5, 8, 20, 22] developed throughout the years would be immediately applicable; Prolog programs would not incur the extra cost of the WAM modifications; and HiLog programs could run on any Prolog implementation. Indeed, in principle, the WAM [21] should be an adequate abstract machine for the execution of any logic language with first-order semantics.

In this paper we propose a simple, yet complete solution to the HiLog compilation problem that satisfies these requirements. To resolve the issue of efficiently implementing HiLog in the WAM we present a *compile-time program transformation algorithm* that specialises partially instantiated calls in HiLog programs. This transformation eliminates the sources of inefficiency from the encoded HiLog programs. More specifically, the main contributions of this paper are as follows:

1. A complete solution to the problem of HiLog implementation, which stays within the WAM framework.
2. A formal proof that HiLog programs that do not use any higher-order features execute at the same speed as Prolog programs, when compiled with the proposed scheme.
3. A completely automated call specialisation algorithm that uses global static

information, but does not require any user supplied annotations, information about the queries, or approximation of the dynamic behaviour of HiLog programs using abstract interpretation [3].

4. Performance results showing that generic HiLog predicates execute much faster than generic Prolog predicates, and with only minimal overhead compared to non-generic Prolog ones.

The syntax of HiLog and the compilation scheme presented in this paper have already been incorporated in XSB [17]. Various versions of XSB have been installed in hundreds of sites for educational, research, and commercial use. Furthermore, the implementation of the HiLog compilation is available through anonymous ftp from `ftp.cs.sunysb.edu` and can be used in any Prolog system as a source-to-source transformation preprocessing step. This allows for implementations of HiLog in any Prolog system by simply changing its input/output predicates to support terms that are expressed using higher-order syntax.

We conclude this section by mentioning that the technique presented in this paper is novel when viewed from the perspective of HiLog compilation. However, it shares ideas with compilation techniques whose purpose is to find specialised, more efficient implementations of programs for some set of known (or derived) initial goals. More specifically, it is in the same spirit as techniques developed in the areas of meta interpretation [7], program specialisation [22], and partial evaluation [9, 10] of logic programs. Although it shares some properties with the above works, (the closest being the work in [7]), there are also considerable differences. Besides the differences mentioned in contribution 3 above, our technique automatically handles impure constructs such as cuts, side-effects and meta calls. Also, since it does not perform any *unfolding* [18], it is independent of the selection strategy, and it is always guaranteed to terminate. Although we believe that the specialisation technique developed in this paper and its formulation are of interest in itself, the emphasis of this paper is not on the technique, but on its properties and the consequences that these properties have for the efficient evaluation of HiLog programs on any WAM-based Prolog implementation. In view of the results in this paper, we claim that HiLog provides a familiar, extremely simple, and promising logical framework for the efficient incorporation of higher-order features in declarative programming languages.

2 HiLog

We begin by briefly reviewing the syntax of HiLog and the encoding of HiLog terms in predicate calculus. We then discuss the reasons for the poor performance characteristics of the encoded programs, and, based on these reasons, we present requirements for efficiently executing HiLog on WAM-based Prolog implementations.

2.1 Syntax and Encoding of HiLog

HiLog provides a higher-order syntax for logic programs, allowing arbitrary terms to occur as the functor of a term (or predicate symbol of an atom). As an example, `X(a,1)` is a well-formed HiLog term. In HiLog, the distinction between individual terms on the one hand and predicates, functions, and atomic formulas on the other is eliminated. Thus, all terms can be manipulated as first-class objects. Without going through all the formal details of HiLog, logic programs in HiLog are defined as follows.

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 logical symbols. It is assumed that sets $\mathcal{V}$ and $\mathcal{S}$ are disjoint. The set of HiLog terms is constructed according to the following definition from [1].

Definition 2.1 (HiLog terms) *The set $\mathcal{T}$ of HiLog terms of language $\mathcal{L}$ is the minimal set of strings over the alphabet satisfying the following conditions:*

- $\mathcal{V} \cup \mathcal{S} \subseteq \mathcal{T}$
- *If $t, t_1, t_2, \ldots, t_n \in \mathcal{T}$, then $t(t_1, t_2, \ldots, t_n) \in \mathcal{T}$, $n \geq 1$.*

In the second case we say that term t is applied to terms $t_1, t_2, \ldots, t_n$.

In actual implementations of HiLog, $\mathcal{S}$ consists of Prolog constants and numbers, while $\mathcal{V}$ consists of Prolog variables.

Though the syntax of HiLog is higher-order, HiLog terms can be encoded into first-order logic using a family of `apply` function and predicate symbols. A full development is given in [1]; we briefly sketch the encoding here. For a HiLog term t, of arity $N \geq 1$, the encoding uses the `apply` symbol of arity $N+1$, where the first argument of `apply/`$(N+1)$ is the encoding of the term in the functor position of t, and the remaining N arguments are the encodings of the N arguments of t. The encoding of a term $t \in \mathcal{V} \cup \mathcal{S}$ is simply t. For example, the HiLog term `X(a,1)` would translate into the term `apply(X,a,1)`. This encoding provides a first-order semantics for HiLog.

XSB [17] implements HiLog using a variant of the above translation. Logically, first order terms are simply a subset of HiLog terms, but operationally, they can be encoded using the somewhat more efficient Prolog term representation. To allow exploitation of this fact, users can partition the set of logical symbols $\mathcal{S}$ into two sets, by using a `hilog` declaration. Symbols not declared as `hilog` are not encoded using the `apply` symbols when appearing in the functor position of a term of arity $N \geq 1$.

Example 2.1 The HiLog program H in figure 1(a) would be encoded as program

```
:- hilog p.
p(a,b).  p(b,c).  p(c,d).  p(d,e).
map(P)([],[]).
map(P)([H1|T1],[H2|T2]) :-
   P(H1,H2), map(P)(T1,T2).
closure(Graph)(X,Y) :-
   Graph(X,Y).
closure(Graph)(X,Y) :-
   Graph(X,Z), closure(Graph)(Z,Y).
```
(a)

```
apply(p,a,b).  apply(p,b,c).
apply(p,c,d).  apply(p,d,e).
apply(map(P),[],[]).
apply(map(P),[H1|T1],[H2|T2]) :-
   apply(P,H1,H2), apply(map(P),T1,T2).
apply(closure(Graph),X,Y) :-
   apply(Graph,X,Y).
apply(closure(Graph),X,Y) :-
   apply(Graph,X,Z), apply(closure(Graph),Z,Y).
```
(b)

Figure 1: HiLog program and its encoding in Prolog.

P in figure 1(b). Note that program P can be compiled in any Prolog system since it is a Prolog program. The only problem may be its efficiency. ■

From here on and unless otherwise specified we use the term HiLog programs to mean HiLog programs that are encoded in predicate calculus using the presented encoding.

2.2 Requirements for efficient HiLog execution

The main advantage of encoding HiLog in predicate calculus is that HiLog programs can execute on exactly the same abstract machine in which Prolog programs execute. Its main disadvantage is that the encoded HiLog programs, when compiled in WAM-based Prolog systems using standard compilation techniques, cannot execute as efficiently as their Prolog counterparts. This is primarily due to the loss of most of the indexing capabilities of the Warren Abstract Machine (WAM). The WAM allows compiled code to be indexed on a predicate basis first, and, within the same predicate, clauses to be indexed on the outer functor of one of their arguments. Traditionally, hash-based indexing on the first argument is used for clauses of the same predicate. The indexing of clauses not only speeds up the process of selecting the appropriate clauses to execute, but often prevents the creation of choice-points, and so results in less trailing. Both of these operations are among the most expensive operations of the WAM.

The issue of efficient clause discrimination is more important for HiLog programs than for typical Prolog programs. This is because HiLog programs generally contain very few predicates with many clauses. Notice that in example 2.1 all atoms are encoded with the same predicate symbol (`apply/3`). As a result, calls to HiLog predicates cannot execute efficiently with the discrimination provided by WAM-style indexing alone.

Even though loss of indexing is a major source of inefficiency, it is not the only one. Due to the encoding of the term in the functor position as the first argument of the `apply` predicate and function symbols, HiLog clauses contain more head unification (`get_*` or `unify_*`) and more argument register (`put_*`) WAM instructions than comparable Prolog clauses. For an instance of this, notice the recursive calls of the `map(P)/2` and `closure(Graph)/2` predicates of example 2.1. During execution on a structure-copying Prolog implementation, these calls build the same structures on the heap over and over again. As a result, more WAM instructions are executed for each recursive step, and heap consumption is increased.[1]

With the risk of oversimplifying the requirements of an optimal HiLog compilation scheme, we expect that in most HiLog programs there will be a fair number of partially instantiated calls. The reason that we expect this property to hold generally, is that the first argument of the `apply` predicates encodes the predicate name of the corresponding Prolog predicate. Most probably, at least all recursive calls to the same predicate will be partially instantiated. Since recursive calls are used in logic-based languages to express loops, we can expect that there is much to be gained by optimising even these calls only. By optimising calls here we mean a compile-time selection of the clauses that might be tried by these calls during execution. For example, a reasonable compilation requirement for the recursive call of the `map(P)/2` predicate is to constrain at compile-time the set of clauses that have to be examined at run-time, and generate an instruction that is more efficient than an "`execute apply/3`" WAM instruction.

To satisfy these requirements, we present in the next section a compile-time program transformation that specialises (partially instantiated) calls in HiLog programs.

---

[1] We note that these problems could be avoided on a structure-sharing Prolog system; however, all WAM-based, and most of modern Prolog implementations are structure-copying systems (see discussion in [19]).

3 Optimising the HiLog encoding through Specialisation

In addition to the standard definitions of logic programming, in our description of the specialisation algorithm we will also need the following definitions. For the first definition, we assume that every clause is labelled by a unique identifier.

Definition 3.1 (Immediately Selected Set) *Let c be an atomic formula with predicate symbol p. We define the set of Horn clauses that are immediately selected by c, as the set of identifiers of clauses whose heads unify with c. We denote this set as $Sel(c)$, and use $Heads(Sel(c))$ to denote the set of atoms in the heads of clauses in $Sel(c)$.*

For every call c, $Sel(c)$ is a safe compile-time approximation of the set of clauses that might be selected by c during run-time for its execution. Run-time instantiations of the call, or cuts in the bodies of the immediately selected clauses can further constrain the $Sel(c)$ set. However, in the absence of any program analysis technique that infers the *context conditions* of c or the *success conditions* [4] of the body literals of the clauses in $Sel(c)$, $Sel(c)$ is the safest approximation of the selected clauses. An important concept for the optimisation of the HiLog encoding is the *most specific generalisation* (or *anti-unifier*) of a set of atoms.

Definition 3.2 (Most Specific Generalisation) *Let T be a non empty set of atoms of predicate symbol p. A generalisation of T is an atom t_g such that for all $t \in T$, t is an instance of t_g. A most specific generalisation (or anti-unifier) of T is a generalisation t_{msg} such that for all other generalisations t_g of T, t_{msg} is an instance of t_g.*

It can be proven [15] that the most specific generalisation of a non empty set of atoms always exists and is unique up to variable renaming. For notational convenience, we write $msg(T) = t_{msg}$, although the most specific generalisation is not a function.

Definition 3.3 (Benefits from Specialisation) *Let P be a program, c be an atom with predicate symbol p defined in P, R be the set of identifiers of the clauses of p, $Sel(c)$ be the set of clauses immediately selected by c, and g be a most specific generalisation of the set $Heads(Sel(c)) \cup \{c\}$. We say that c benefits from specialisation if and only if either of the following conditions is satisfied:*

- *$Sel(c)$ is a proper subset of R.*
- *There exists an argument of g that is not a variable.*

We say that a HiLog program P benefits from specialisation if and only if it contains a call that benefits from specialisation; otherwise we say that P is optimally specialised.

Let C be a set of calls (not necessarily to the same HiLog predicate). We partition C into a finite number of sets $C^{g_1}_{S_1}, C^{g_2}_{S_2}, \ldots, C^{g_k}_{S_k}$ where

$$C^{g_i}_{S_i} = \{c | c \in C \wedge Sel(c) = S_i \wedge msg(Heads(S_i) \cup \{c\}) \text{ is a variant of } g_i\}.$$

Intuitively, calls end up in the same equivalence class if and only if they have the same set of immediately selected clauses S_i, and their most specific generalisations with the heads of the clauses in S_i are identical (modulo variable renaming). For notational convenience we drop the g_i superscript, and use the notation $Sel(C_{S_i})$ to denote the set of immediately selected clauses of the equivalence class C_{S_i}.

Definition 3.4 (Representative) *Let C be an equivalence class of calls to a predicate symbol p and let $S = Sel(C)$ be the set of clauses immediately selected by the calls in C. Provided that S is not empty, a chain rule R of the form $H \leftarrow B$ is a p'-representative of S for the set of calls C if and only if the following conditions are satisfied:*

- *H is the most specific generalisation of the set $C \cup Heads(S)$ where $Heads(S)$ is the set of heads of clauses in S.*
- *The predicate symbol of the atom B is p', and its arguments are the set of variables that appear in H. It is assumed that the predicate symbol p' is of the appropriate arity.*

Whenever there is no confusion regarding the predicate symbol p', R is simply called a representative of S for C.

Note that the notion of a representative of $S \neq \emptyset$ for C is well-defined since H is always non-variable. This happens because all elements of the set $C \cup Heads(S)$ are atoms of the same predicate symbol p.

Definition 3.5 *Let $R = (H \leftarrow B)$ be a p'-representative of a set of immediately selected clauses S for a set of calls C, and let T be an atom that is either a head of a clause in S or a call in C. The p'-difference of T from the head H of R is the resolvent of T with the clause $B \vee \neg H$.*

Example 3.1 For the set of calls

$$C = \{q(a, f(V_3), g(V_3, c)), q(a, f(b), g(b, h(V_4, d)))\}$$

and set of immediately selected clauses

$$Sel(C) = \{q(a, f(b), g(b, V_1)) \leftarrow B_1,\ q(a, f(V_1), g(V_1, V_2)) \leftarrow B_2\}$$

the following two chain rules are the only q'-representatives

$$\begin{aligned} q(a, f(X_1), g(X_1, X_2)) &\leftarrow q'(X_1, X_2). \\ q(a, f(X_1), g(X_1, X_2)) &\leftarrow q'(X_2, X_1). \end{aligned}$$

The q'-difference of the clause head $q(a, f(b), g(b, V_1))$ with the first q'-representative is $q'(b, V_1)$, and the q'-difference of the call $q(a, f(b), g(b, h(V_4, d)))$ with the second q'-representative is $q'(h(V_4, d), b)$. ■

Definition 3.6 (Call Specialisation) *Let c be a call to a HiLog predicate p, C be its equivalence class, and $S = Sel(C)$ the set of immediately selected clauses by c. Provided that $S \neq \emptyset$, let $R = (H \leftarrow B)$ be a p'-representative of S for C. The p'-specialisation of c is defined as the p'-difference of c from H, if $S \neq \emptyset$, and fail otherwise.*

3.1 The specialisation algorithm

Algorithm Specialise in Figure 2 provides a very high-level description of the proposed specialisation of calls in a HiLog program. Given a program P, the algorithm produces a *computationally equivalent residual program* P' in which all partially instantiated calls to predicates that are defined in P and benefit from specialisation are replaced by calls to specialised versions of these predicates. The expectation from this process is that the calls of the residual program can be executed more efficiently than their non-specialised counterparts. This expectation is justified mainly because of the following two basic properties of the algorithm.

Algorithm Specialise($Program$)

1. Collect in C all (partially instantiated) calls to predicates defined in $Program$;
2. For each $c_i \in C$ find and associate with c_i
   i. the set $Sel(c_i)$ of $Program$ clauses that are immediately selected by c_i, and
   ii. the most specific generalisation g_i of the set $Heads(Sel(c_i)) \cup \{c_i\}$;
3. Remove from C all calls that do not benefit from specialisation; let C' be the result;
4. Partition C' into $C_{S_1}, C_{S_2}, \ldots, C_{S_k}$ where
   $C_{S_i} = \{c | c \in C \wedge Sel(c) = S_i \wedge msg(Heads(S_i) \cup \{c\})$ is a variant of $g_i\}$;
5. For each equivalence class of calls C_{S_i} do
   /* Let $Sel(C_{S_i})$ be the set of immediately selected clauses of the calls in C_{S_i},
   p_i be their predicate symbol, and p'_i be a *new* predicate symbol. */
   i. If $(Sel(C_{S_i}) \neq \emptyset)$ then
      α. Choose a p'_i-representative $R_i = (H_i \leftarrow B_i)$ of $Sel(C_{S_i})$ for the calls C_{S_i};
      β. For each clause $Cl_{i_j} = (Head_{i_j} \leftarrow Body_{i_j}) \in Sel(C_{S_i})$ do
         Insert in $Program$ the clause $Cl'_{i_j} = (Head'_{i_j} \leftarrow Body_{i_j})$ where $Head'_{i_j}$
         is the p'_i-difference of $Head_{i_j}$ from the head H_i of the representative R_i;
   ii. For each call c_{i_j} of the equivalence class C_{S_i},
      find its p'-specialisation c'_{i_j} (and associate it with c_{i_j});
6. For each equivalence class of calls C_{S_i} do
   Replace in $Program$ all occurrences of call $c_{i_j} \in C_{S_i}$ by its p'-specialisation c'_{i_j};

Figure 2: The call specialisation algorithm.

Compile-time Clause Selection The specialised calls of the residual program P' directly select (at compile time) a subset containing only the clauses that the corresponding calls of P would otherwise have to examine during their execution (at run time). By doing so, laying down unnecessary choice points is at least partly avoided, and so is the need to select clauses through indexing.

Factoring of Common Subterms Non-variable subterms of the partially instantiated calls that are common with subterms in the heads of the selected clauses are factored out from these terms during the specialisation process. As a result, some head unification (`get_*` or `unify_*`) and some argument register (`put_*`) WAM instructions of the original HiLog program become unnecessary. These instructions are eliminated from both the specialised calls as well as from the specialised versions of the predicates.

Algorithm Specialise begins by collecting all (partially instantiated) calls to predicates that are defined in program P. We allow for *open HiLog programs*, that is HiLog programs for which the definitions of some predicates are missing or are imported from other modules. The second step of the algorithm finds and associates with each call c_i, the set of program clauses $Sel(c_i)$ whose heads unify with c_i. Each of these sets contains the program clauses that have the potential of being selected for the execution of call c_i during run-time. As mentioned in the previous section, in the absence of any information about the context conditions of c_i, or the success conditions of each clause in $Sel(c_i)$, the $Sel(c_i)$ set is the best safe approximation of clauses that might be chosen for the execution of c_i at run-time. Not all collected calls, however, can benefit from specialisation; calls that do not are eliminated in the third step. The filtering is based on definition 3.3, and ensures that all remaining calls satisfy at least one of the previously mentioned properties of the algorithm.

Even if step 1 collects all calls to predicates defined in P, instead of the partially instantiated ones, all calls that are not partially instantiated will be eliminated in this third step. The fourth step groups the remaining calls into equivalence classes based on equality of their associated clause selection, and variance of the most specific generalisations. We note that, in principle, this partitioning is not necessary, or can be based on any other clause containment hierarchy. It is essential, however, for the avoidance of unnecessary code explosion.

The actual specialisation of the calls is performed in the fifth and sixth steps of the algorithm.

The specialisation of partially instantiated calls is performed for one equivalence class of calls, C_{S_i}, at a time. In step 5.$i.\alpha$, a p'_i*-representative* $R_i = (H_i \leftarrow B_i)$ of the selected clauses $Sel(C_{S_i})$ for the equivalence class of calls C_{S_i} is nondeterministically chosen. The head H_i factors out all common ground subterms from the heads of the immediately selected clauses and from the calls that selected these clauses. In the body B_i of the representative, p'_i is a *new* predicate symbol of the appropriate arity that does not appear anywhere either in the HiLog original program, or in the portion of the residual program generated so far. So that the generated residual program P' has the same interface with other modules as P, we require that these new predicate symbols are internal to P', i.e. we require that they are not accessible from other modules or the top-level.

For each clause $Cl_{i_j} = (Head_{i_j} \leftarrow Body_{i_j})$ of the selected set of clauses $Sel(C_{S_i})$, a new clause $Cl'_{i_j} = (Head'_{i_j} \leftarrow Body_{i_j})$ is inserted in the residual program (step 5.$i.\beta$). The head of Cl'_{i_j} is the p'_i*-difference* of the head of Cl_{i_j} from the chosen p'_i*-representative* R_i, while its body is the same as the body of Cl_{i_j}. These new clauses are used for the specialisation of the calls in the equivalence class C_{S_i}. The specialisations c'_{i_j} of each call c_{i_j} in C_{S_i} are computed according to definition 3.6 in step 5.ii. Finally, in the sixth step, all the partially instantiated calls are replaced by their associated specialisations. The resulting program is the residual program P'. We use the notation $\mathsf{Specialise}(P)$ for a residual program of P generated by algorithm Specialise.

Example 3.2 Applying algorithm Specialise on program P of example 2.1 generates the residual program P' of P shown in figure 3. Note that program P' is

```
apply(p,a,b).  apply(p,b,c).  apply(p,c,d).  apply(p,d,e).
apply(map(P),[],[]).
apply(map(P),[H1|T1],[H2|T2]) :- apply(P,H1,H2), apply_map(T1,P,T2).
apply(closure(Graph),X,Y) :- apply(Graph,X,Y).
apply(closure(Graph),X,Y) :- apply(Graph,X,Z), apply_closure(Graph,Z,Y).

apply_map([],P,[]).
apply_map([H1|T1],P,[H2|T2]) :- apply(P,H1,H2), apply_map(T1,P,T2).
apply_closure(Graph,X,Y) :- apply(Graph,X,Y).
apply_closure(Graph,X,Y) :- apply(Graph,X,Z), apply_closure(Graph,Z,Y).
```

Figure 3: Residual program generated by specialising the program of figure 1(b).

computationally equivalent to P, and that the recursive calls of the HiLog predicates `map(P)/2` and `closure(Graph)/2` that were specialised do not need indexing in order to select the appropriate clauses of predicate `apply/3`. Also, note that `apply_map/3` and `apply_closure/3` are Prolog predicate symbols, and WAM-style indexing is sufficient for these predicates. ■

4 Properties of the HiLog Specialisation

In this section we present some properties of algorithm Specialise. The first theorem describes a basic property of the optimisation of the HiLog encoding. It states that for any goal G, the generated residual program P' of P gives exactly the same answers for G as P does; furthermore, P' preserves the finite failure of calls to P.

Theorem 4.1 (Computational equivalence) *Let P be a HiLog program, P' be a residual program of P generated by algorithm* Specialise*, and G be any goal. Then the following hold:*

1. *$P' \cup \{G\}$ has an SLDNF-refutation with computed answer θ if and only if $P \cup \{G\}$ does.*

2. *$P' \cup \{G\}$ has a finitely failed SLDNF-tree if and only if $P \cup \{G\}$ does.*

From the way algorithm Specialise is formalised (as one pass) it follows that it always terminates. However, we can prove an even stronger result. The following theorem shows that applying the same kind of specialisation to a residual program will never result in any further specialisations. Thus, one pass of the algorithm is sufficient to guarantee an optimally specialised residual program (according to definition 3.3).

Theorem 4.2 (Idempotence of the Specialisation Algorithm) *Let P be a HiLog program. Then* $\mathsf{Specialise}(\mathsf{Specialise}(P)) = \mathsf{Specialise}(P)$.

We now present a theorem that relates HiLog encoding of the predicates in a Prolog program and specialisation. We hold that this theorem is very important for HiLog. As pointed out by Nadathur and Miller [13], a challenge for any implementation of a higher-order logic programming language is to be as efficient on programs that do not use any higher-order features as standard Prolog implementations. What the following theorem shows is that calls in a Prolog program whose predicate symbols are encoded in HiLog and specialised by algorithm Specialise never execute slower than calls in the original Prolog program. To prove this, we will need the following definitions.

Definition 4.1 (Identical up to Predicate Renaming) *Let P, P' be programs, and R, R' be the sets of predicate symbols defined in P, P' respectively. We say that programs P and P' are identical up to predicate renaming (and write $P \cong P'$) if and only if there exists a renaming (1-1 and onto mapping) of the predicate symbols in R to the predicate symbols in R' such that each clause in P is a variant of a clause in P'.*

Definition 4.2 (Internally Called Predicate) *Let P be a program. A predicate symbol p is said to be internally called in P if and only if it is defined in P and P contains at least one call to p.*

Given a HiLog program P, and a set of logical symbols S, we use the notation $\mathsf{Encode}(P, S)$ for the program obtained by applying the HiLog encoding to P, treating all symbols of S as symbols to be encoded, when in functor position, using the `apply` predicate and function symbols. We also use the notation $\mathsf{Extract}(P)$ for the program obtained by removing from P all `apply`/N clauses, for every arity $N \geq 2$. Now we can prove the following.

Theorem 4.3 *Let P be a Prolog program, and R be the set of internally called predicate symbols of P. Provided that the sets of predicate and function symbols of P are disjoint, the following holds:*

$$\mathsf{Extract}(\mathsf{Specialise}(\mathsf{Encode}(P, R))) \cong \mathsf{Specialise}(P)$$

Since Datalog programs do not contain any function symbols, the following is an immediate corollary of the previous theorem.

Corollary 4.4 *Let P be a Datalog program, and R be the set of its internally called predicate symbols. Then the following holds:*

$$\mathsf{Extract}(\mathsf{Specialise}(\mathsf{Encode}(P, R))) \cong \mathsf{Specialise}(P)$$

Figure 4 presents the relationships between the HiLog encoding of all internally called predicate symbols of a Prolog program and the specialisation algorithm. Given a Prolog program P, program P' is its residual program generated by specialisation. Program H is obtained by encoding in HiLog the set of internally called predicate symbols of P, and program $H' = \mathsf{Specialise}(H)$. Finally, program H'' is obtained by removing from H' all $\texttt{apply}/N$ clauses, for all $N \geq 2$. Theorem 4.3 states that programs P' and H'' are identical up to predicate renaming. We finally note that the requirement that only used predicate symbols are encoded, does not seriously restrict the class of programs for which the theorem holds. Instead of requiring all encoded predicates to be used, the same result could be achieved by simply adding open calls (calls having variables as arguments) for all unused predicate symbols of a program.

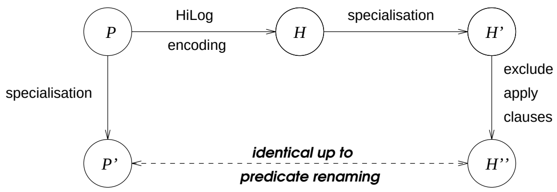

Figure 4: Relationship between HiLog encoding and specialisation.

5 Performance Results

As already stated, the viability of HiLog as a logic programming language depends on its efficient implementation. In particular, the performance of a HiLog implementation must be able to compete with that of modern Prolog systems both for programs that use, and for programs that do not use any higher-order features. The benchmarks of this section show that this is indeed the case.

Table 1 presents normalised times for executing in XSB (version 1.3.1) a set of six standard Prolog benchmarks[2] written by D.H.D. Warren. The normalisation is with respect to the execution times of the original Prolog programs. The two HiLog rows show times for these benchmarks when all internally called predicate

[2] Program `qsort->` is similar to `qsort` but for the `partition/4` predicate which is written using an if-then-else construct that in many Prolog systems avoids choice point creation. All benchmarks were run on a SPARCstation 2 with 64MB of main memory running SunOS 4.1.3. All programs are available by contacting one of the authors.

symbols are considered as HiLog symbols. The compilation scheme that was used is shown in parentheses. The effect of specialisation on the performance of the original Prolog programs was either quite small (about 1-3% for programs `nreverse`, `qsort`, `qsort->` and `deriv`), or non-existent (for the other two); for this reason we do not present times for Prolog compiled without specialisation. We note that for all programs standard Prolog indexing was used.

Having stated theorem 4.3, the first two rows of Table 1 do not provide new information. However, the last two do show the performance penalty of compiling HiLog programs using standard Prolog compilation techniques, or equivalently, the speed-ups obtained by using algorithm Specialise.

Benchmark	`nreverse`	`qsort`	`qsort->`	`deriv`	`query`	`serialise`	*mean*
Prolog (specialisation)	1	1	1	1	1	1	1
HiLog (specialisation)	1	1	1	1	1	1	1
HiLog (standard)	3.4	1.66	2.71	2.07	3.9	1.55	2.4

Table 1: Normalised times for executing standard Prolog benchmarks using XSB.

In summary, using XSB, the specialised HiLog programs execute 2 to 4 times faster than those compiled using no specialisation (the last column gives the geometric mean of the speed-up ratio). To verify that these results generally hold, and do not depend on possible idiosyncrasies of XSB, the programs were source-transformed[3] and run under Quintus Prolog (release 3.0), and SICStus Prolog (version 2.1 #9). Similar speed-ups were observed (Table 2). We note however, that

Benchmark	`nreverse`	`qsort`	`qsort->`	`deriv`	`query`	`serialise`	*mean*
Quintus	4.66	1.41	2.35	1.81	5.04	1.6	2.52
SICStus	3.52	1.8	2.18	1.78	2.48	1.5	2.12

Table 2: Speed-ups obtained by using Quintus and SICStus Prolog.

in general the speed-up obtained by using the specialisation algorithm is a function of the number of predicates of a particular arity. As mentioned in section 2, all predicates of arity N get encoded using the same `apply`$/(N+1)$ predicate. By simply adding to the programs more predicates of used arities, arbitrarily bigger speed-ups than those reported can be achieved. An advantage of specialisation is that the performance of the residual programs does not depend on the number of predicates of a particular arity.

The next set of benchmarks compares the efficiency of programs that use higher-order features. More specifically, we compare, under XSB, the performance of different versions of the generic `maplist` and `closure` higher-order predicates, both operating on an extensional database (EDB) predicate forming a chain of 50 elements. The generic HiLog predicates are the ones given in example 2.1, and are optimised as in example 3.2. In each benchmark two Prolog predicates were used; a generic in which the EDB predicate is a parameter, and a non-generic in which the

[3] The specialisation algorithm implemented in XSB can be used as a source-to-source transformation that dumps the generated residual program in a file. This functionality is available through a compiler option.

EDB predicate is hard-coded. The recursive clauses of the Prolog `maplist` predicates are shown below. Given these clauses, the base clauses of these predicates, and the similarly-looking Prolog predicates for `closure` should be easily deducible.

Generic Prolog (=../2): `map(P,[H1|T1],[H2|T2]) :- C =.. [P,H1,H2], C, map(P,T1,T2).`
Non-Generic Prolog: `map_of_p([H1|T1],[H2|T2]) :- p(H1,H2), map_of_p(T1,T2).`

Table 3 shows two sets of times for these benchmarks normalised to the specialised HiLog, and the non-generic Prolog case, respectively. In XSB, HiLog using specialisation clearly outperforms the set of generic predicates; HiLog programs run between 7 and 8 times faster than generic Prolog programs. Furthermore, compared to the specialised generic HiLog predicates, the non-generic Prolog predicates execute only 10-13% faster. This small overhead is due to the extra argument of the generic HiLog predicates (see predicates `apply_map/3` and `apply_closure/3` of example 3.2).

	Norm. to Generic HiLog		*Norm. to Non-Gen. Prolog*	
Benchmark	`maplist`	`closure`	`maplist`	`closure`
Generic Prolog (using =../2)	7.37	8.11	9.87	11.1
Generic HiLog (standard)	7.29	8.02	8.87	10
Generic HiLog (specialisation)	1	1	1.1	1.13
Non-Generic Prolog	.91	.89	1	1

Table 3: Normalised `maplist` and `closure` benchmarks using XSB.

Finally, to verify the performance gain of using HiLog for generic logic programming across different Prolog implementations, we compared the performance of generic Prolog, generic HiLog, and non-generic Prolog programs under Quintus and SICStus. For Quintus, a generic Prolog program using the library predicate `call/N` (implemented in Prolog) was also tested.

Generic Prolog (`call/N`): `map(P,[H1|T1],[H2|T2]) :- call(P,H1,H2), map(P,T1,T2).`

The results, shown in Table 4, generally resemble those obtained using XSB; but were somewhat unexpected for the expense of running generic Prolog predicates under Quintus.

	Norm. to Generic HiLog			
Benchmark	`maplist`		`closure`	
	Quintus	SICStus	Quintus	SICStus
Gen. Prolog (=../2)	26.5	7.3	26.3	5.2
Gen. Prolog (`call/N`)	40.9	-	40.3	-
Gen. HiLog (spec)	1	1	1	1
Non-Generic Prolog	.88	.77	.89	.94

Table 4: Normalised benchmarks of generic predicates using Quintus and SICStus.

6 Epilogue

HiLog allows the incorporation of certain higher-order constructs in a declarative way within logic programming, while retaining a first-order semantics. In this paper we addressed the issue of efficient execution of HiLog on WAM-based Prolog implementations. We identified the reasons for the inefficiency of naively encoded HiLog programs in predicate calculus, and presented a complete solution to the problem of HiLog implementation. We have shown theoretically that the specialised encoding allows all HiLog programs that do not use any higher-order features to execute at the same speed as Prolog programs. Furthermore, we presented experimental results indicating that generic HiLog executes much faster than generic Prolog, and with only a minimal overhead compared to non-generic Prolog.

The implementation of HiLog specialisation is available through anonymous ftp and can be used in any WAM-based Prolog system as a source-to-source transformation preprocessing step. This, and the fact that an efficient implementation of HiLog requires no changes to the underlying Prolog abstract machine, should allow for "portable" implementations of HiLog by simply changing the input and output predicates of Prolog (i.e. `read/1`, `write/1`, etc.) to accept and display terms that are expressed using the flexible higher-order syntax of HiLog.

The above reasons allow us to claim that HiLog provides a familiar, extremely simple, and efficient logical framework for the incorporation of higher-order features in logic programming languages. In light of the results in this paper, and the given interest of the research community in HiLog, Prolog programmers would rightfully demand furure logic programming systems extended to support HiLog functionality. It is reasonable to expect that HiLog may become an interesting successor of Prolog.

Acknowledgements

This research was supported in part by NSF Grants CCR-910259 and CCR-9404921.

References

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

[2] W. Chen and D. S. Warren. An intensional logic of (multi-arity) set abstractions. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 97–110, Paris, 1991. The MIT Press.

[3] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.

[4] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Extracting determinacy in logic programs. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 424–438, Budapest, Hungary, 1993. The MIT Press.

[5] S. K. Debray. A simple code improvement scheme for Prolog. *Journal of Logic Programming*, 13(1):57–88, May 1992.

[6] M. A. Derr, S. Morishita, and G. Phipps. Design and implementation of the Glue-Nail database system. In *Proceedings of the ACM SIGMOD*, pages 147–156, Washington, D.C., May 1993.

[7] J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-programming in Logic*, pages 229–244, Leuven, Belgium, April 1990.

[8] T. Hickey and S. Mudambi. Global compilation of Prolog. *Journal of Logic Programming*, 7(3):193–230, November 1989.

[9] T. J. Hickey and D. A. Smith. Toward the Partial Evaluation of CLP Languages. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 43–51, Yale University, New Haven, U.S.A., June 1991.

[10] J. W. Lloyd and J. C. Sheperdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3 & 4):217–242, October/November 1991.

[11] Y. Lou and Z. M. Ozsoyoglu. LLO: An object-oriented deductive language with methods and method inheritance. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 198–207, 1991.

[12] I. S. Mumick and K. A. Ross. Noodle: A language for declarative querying in an object-oriented database. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Proceedings of the Conference on Deductive and Object-Oriented Databases*, pages 360–378, Phoenix, Arizona, December 1993. LNCS-760, Springer-Verlag.

[13] G. Nadathur and D. Miller. An overview of λ Prolog. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 810–827, Seatle, 1988. MIT Press.

[14] R. Paterson-Jones and P. T. v. Wood. Extending the WAM to implement HiLog. Unpublished Manuscript (also see Poster Abstract in Proceedings of ILPS-93, page 654), 1993.

[15] G. D. Plotkin. A note on inductive generalisation. *Machine Intelligence*, 5:153–163, 1970.

[16] K. A. Ross. Relations with relation names as arguments: Algebra and calculus. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 346–353, San Diego, California, 1992.

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

[18] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Second International Logic Programming Conference*, pages 127–138, Uppsala, 1984.

[19] P. Van Roy. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming*, 19/20:385–441, May/July 1994.

[20] P. Van Roy and A. M. Despain. The benefits of global dataflow analysis for an optimizing Prolog compiler. In S. Debray and M. Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 501–515, Austin, 1990. The MIT Press.

[21] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., October 1983.

[22] W. Winsborough. Multiple specialization using minimal-function graph semantics. *Journal of Logic Programming*, 13(2&3):259–290, July 1992.

[23] E. Yardeni, T. Frühwirth, and E. Shapiro. Polymorphically typed logic programs. In F. Pfenning, editor, *Types in Logic Programming*, chapter 2, pages 63–90. The MIT Press, 1992.