Curried Least General Generalization: a framework for higher order concept learning

Srinivas Padmanabhuni and Randy Goebel
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1
E-mail: {srinivas, goebel}@cs.ualberta.ca
WWW: http://www.cs.ualberta.ca/~{srinivas, goebel}
Voice: +1-403-492-2683
Fax: +1-403-492-1071

Koichi Furukawa
Graduate School of Media and Governance
Keio University
5322 Endo, Fujisawa, Kanagawa 252, Japan
E-mail: furukawa@sfc.keio.ac.jp
WWW: http://www.sfc.keio.ac.jp/~furukawa
Voice: +81-466-47-5111 Ext. 3235 or 3231
Fax: +81-466-47-5350

Abstract. Continued progress with research in inductive logic programming relies on further extensions of their underlying logics. The standard tactics for extending expressivity include a generalization to higher order logics, which immediately forces attention to the computational complexity of higher order reasoning.
A major thread of inductive logic programming research has focussed on the identification of preferred hypothesis sets, initiated by Plotkin's work on least general generalizations (LGGs). Within higher order frameworks, a relevant extension of LGG is Furukawa's hyper least general generalization (HLGG) [FIG97].
We present a relevant higher order extension of Furukawa's HLGG based on currying, which we call Curried Least General Generalization (CLGG). The idea is that the formal difficulties with the reasoning complexity of a higher order language can be controlled by forming new hypothetical terms restricted to those obtainable by Currying. This technique subsumes the inductive generalization power of HLGG, provides a basis for a significant extension of first order ILP, and is theoretically justified within a well understood formal foundation.

1 Motivation and Introduction

First-order logic has been the major focus of research in the field of machine learning, as developed within *inductive logic programming* (ILP) [Mug92, Rae93].

The typical ILP framework uses first-order Horn clauses as the underlying representation language for inductive generalization. The computational difficulty of dealing with general first-order clauses is well acknowledged, which justifies the ILP concentration on the Horn clause restriction, and the development of several induction algorithms [Pop70, Plo70, Plo71a, Plo71b] for clausal representation. This restriction, which makes the clausal language computationally attractive, also makes induction algorithms efficient enough to be practical for real application.

The standard ILP implementation strategy is based on procedural engines for carrying out the inverse of deduction on first order clausal syntax. An important part of the historical development of logic programming are the relatively efficient algorithms for restricted deduction using resolution, which provide the basis for languages like Prolog, and which have been widely used in a variety of artificial intelligence applications over the past two decades. Unlike deduction, induction has multiple possible outcomes, so ILP introduces additional complexity, due to the multiple outcomes possible with the inversion of deduction algorithms. The consequence is the desire to distinguish the "best" of the outcomes, thus the motivation for adopting Plotkin's ideas and their derivatives.

Despite the common first order clausal restriction, there are many situations where the expressivity of higher order logics provides for direct representation of knowledge that is otherwise difficult to express, so the development of reasoning systems that manipulate fragments of higher order languages continues. Some (relatively) efficient deduction algorithms for higher-order logic [NM90] have been successfully developed. This has led to the implementation of higher-order languages like λ-prolog [NM90] and HOL [GM93], which are based on deduction mechanisms in certain *restricted* forms of higher-order logic. These languages have been successfully used for formal reasoning in many different areas, including hardware design and verification, reasoning about security, proofs about real-time systems, semantics of hardware description languages, compiler verification, program correctness, modeling concurrency, and program refinement.

Further, there are certain aspects of inductive and deductive reasoning with first order logic, which provide the motivation for considering higher order extensions to the first order logic based algorithms for ILP.

For efficiency considerations in first order deduction systems, suitable restrictions are required that enable the design of computationally feasible reasoning algorithms. These restrictions are typically cast as higher-order meta-information—and used in even relatively pedestrian language extensions. Consider the case of the language PROLOG, based on deduction of first-order clauses. In Prolog, there are a variety of meta-predicates which are quite often used to guide the "pure deduction" component of the computation. Some of these predicates are **not**, ! (cut), ;(or), setof, and bagof. So higher-order predicates are a part of the meta-information necessary for implementing deduction systems in first-order logic.

In the induction of first-order clauses in ILP, one use of higher-order meta-information is as a bias to control the complexity of the induction process in

generalizing from ground examples to first-order formulae. Some of the examples of such higher-order meta information include information about the mode of arguments of clauses, as well as second-order schema to guide the predicate invention phenomenon in the induction process. These are typical examples of higher-order meta information used in a variety of ILP systems like CIA[RB92] and RDT [KW92].

The above factors provide a strong motivation for research on identifying subsets of higher-order logic which can be efficiently exploited for induction. Keeping the computational costs in mind, one might reason that investigation of higher-order logics is not a worthwhile project to undertake. But the above argument *vis-a-vis* tractable subsets of higher-order logic is a promising research strategy. Moreover development of induction algorithms for higher-order logics is potentially fruitful in generating meta-information for both inductive and deductive reasoning with first-order logic programs. Some recent studies by Muggleton [FM92] [MJ94] suggests potentially important directions of research in the inductive generalization of higher-order clauses. He suggests that the use of higher-order logic in induction might help in improving both efficiency and expressivity in certain induction domains. Feng and Muggleton [FM92] develop an algorithm for generalization in a restricted form of λ-calculus and demonstrate its use in a class of program transformations. But the main problem in [FM92] is the use of a λ-calculus-based abstraction formalism which complicates the development of algorithms for generalization. Complexity issues for such algorithms are not clearly presented in that system.

Here we present a higher-order extension to the LGG (least general generalization) algorithm of Plotkin [Plo70], the ground-breaking work in machine learning which forms the basis of a majority of ILP algorithms [Mug92, Rae93]. We exploit the technique of *currying* [Sch24, Cur30], commonly used in applicative (functional) programming, for inductive logic programming and develop a framework for *Curried least general generalization* (CLGG).

Currying provides a uniform basis for exportability of ideas from first-order logic based induction to higher-order logic induction. Though currying doesn't provide the full power of λ-calculus, we show how many important classes of useful higher-order structures can be obtained by using first-order algorithms on curried forms of first-order expressions. We then discuss the different issues concerned with controlling the complexity of such induction process in a variety of cases, and discuss feasible higher-order generalizations. An interesting outcome is the extendibility of the various forms of bias restrictions used in ILP to higher order induction, which is a key factor in development of a computationally attractive higher-order learning system.

A system based on higher-order induction can be used to capture some potentially useful kinds of higher-order meta information for ILP. In addition there are diverse applications of such a system which can be studied such as classes of program transformations.

We also extend the idea of relative least general generalization (RLGG) [Plo70] to our subset of higher-order logic obtained by using currying on first-

order terms and clauses. We present generalizations of the various algorithms in Plotkin's RLGG framework to work for this restricted subset of higher-order logic. We discuss the meaningful higher-order generalizations achievable by these extended algorithms, which are otherwise not achievable in ILP. The issues involved in implementing any such extended algorithm are examined, in particular the exportability of efficiency ideas in ILP, like syntactic bias, to guide higher-order induction. In addition, some important applications of such higher-order induction system are examined. This includes a new interpretation of HLGG [FIG97] by CLGG.

In summary, we discuss a plethora of possible directions in which this work can be extended.

2 Currying as a foundation for CLGG

The most natural language for describing higher-order expressions mathematically is the λ-calculus, whose language consists of structures called λ-terms. A λ-term is either:

1. A variable , e.g.x, is a λ-term;
2. An application MN , where both M and N are λ-terms;
3. An abstraction $\lambda x.M$, where M is a λ-term.

In the form MN, we say that M is applied to N. Similarly we say that $\lambda x.M$ abstracts x in M.

The variable occurrences in a $\lambda-$term are of two types: free or bound. An occurrence of a variable x in a term P is bound iff it is a part of P with the form $\lambda x.M$; otherwise it is free.

For any λ-term M,N and any variable x, M [N/x] is defined to be the result of substituting N for every free occurrence of x in M, and changing bound variables to avoid clashes.

The essential mechanism of manipulating λ-calculus terms is the syntactic process called β-reduction. The β-reduction rule is defined as follows:

(λx.E) M $\longleftrightarrow_\beta$ E [M/x],

where the necessary restrictions for x not being free, apply.

So the language of λ-calculus is very convenient for representing higher-order terms. Implicit in the above representation of function applications is the understanding that we will use only function applications of the form MN, (M applied to N), where M and N are both λ-terms. But in many functional programming languages, function applications are multi-argument in nature, i.e. a function is applied to a multitude of arguments.

The phenomenon of *Currying* [Sch24, Cur30], is used to convert multiple argument functions to a form representable in λ-calculus. *Currying* refers to the process of converting a multi-argument function application of the form M N ... P (where M,N,... P are all λ-terms) to another expression of the form (((...(MN)...)P) which is representable in λ-calculus.

In the framework of inductive logic programming, the underlying language is the language of first-order terms. The domain of first order terms (*T(F,X)*) constructed from the set of functions F, and the set of variables X consists of terms defined recursively as follows:

- A variable is a term
- If f is an *n-ary* function symbol, and $t_1, \ldots, t_n$ are terms, then $f(t_1, t_2, \ldots, t_n)$is a term.

A first order term can be converted by *currying* to a λ-term by use of a single binary function called *apply*, to first-order terms pairwise. A first-order variable x is converted to x, and a first-order term of the form $f(t_1)$ is converted to $apply(f, conv(t_1))$, where $conv(t_1)$ denotes the converted form of t_1. Extending this to $f(t_1, t_2, \ldots, t_n)$, we get the form $apply(apply(...apply(f, conv(t_1)), conv(t_2))$ $..., conv(t_n)$, where $conv(t_i)$ represents the converted form of t_i.

The currying process thus enables us to represent any first-order term as a λ-term.

3 Curried least general generalization

3.1 Language

The advantage of applying *currying* to first-order terms as shown in the previous section is that it has only one function, namely *apply*, and any multi-argument or constant function reduces to a constant in the term. Therefore a binary function constant *plus* will be treated the same as a unary function constant like *successor*, or a zero-ary function constant like 0. This removes any difference between functions of different arity.

Extending the idea, we remove the difference between predicates of different arities, and treat predicates as multi-argument functions returning only true or false. This enables us to use predicates as constants in our language.

In our term structure , a term belonging to $T(F, X, P)$, the set of curried terms, is defined by the following inductive rules:

1. Any first order or higher-order variable x, (a multi-argument function or predicate) is a term.
2. An application apply($t1$,$t2$), where $t1, t2 \in T(F, X, P)$, is a term.

This is the same as the original definition of first-order term except that only one binary function symbol (apply) is used, and the differences between higher-order and first-order terms evaporates. Any term in this superset of first order logic shall be equivalently referred to as a *curried first order term* or simply *curried term*. A *curried clause* is a set of curried terms with appropriate signs.

The set of terms in this language forms a superset of the set of first order terms, and includes additional higher-order terms not representable in first order syntax. We do not concentrate on the full λ-calculus for our language because of complexity considerations and show that even this conservative higher order

extension of first order logic terms is capable of providing useful higher-order expressions which are significantly more expressive than first order logic.

Now we present a model for inductive generalization for this language subset of full λ-calculus by extending ideas from Plotkin's RLGG framework to this superset of first order logic.

3.2 θ-subsumption under the new representation

In ILP, the model of induction is based on the relation between pairs of clauses termed as θ-subsumption. We extend the notion of θ-subsumption for curried first order terms, defined as follows:

Definition 1 (Curried θ-subsumption) *A curried clause P is said to θ-subsume another curried clause Q, iff there exists a substitution θ such that $P\theta \subseteq Q$.*

This definition induces a generality relation between any two curried clauses.

For example, the clause containing only the term $apply(f, X)$ θ-subsumes the clause $\{apply(f, 1), apply(g, 2)\}$. This relation induces a lattice on the clauses in our language, similar to the ILP case.

3.3 The LGG algorithm for clauses under the new representation

Here we show that when we apply the LGG algorithm for first order clauses to the clauses in our representation scheme, we get a higher order concept. We call the output of this algorithm a curried least general generalization (CLGG). The LGG algorithm, when applied to the curried representation, will extract the higher-order predicates from the set of input first order predicates.

The steps in our algorithm for obtaining the CLGG are as follows:

(1) Convert any function from the form $f(t_1 \ldots t_n)$
to $app(app(\ldots app(f, t_1), t_2) \ldots, t_n)$.
(2) Then apply the LGG algorithm which runs as follows:

1. The LGG of terms $f(s_1, s_2, \ldots, s_n)$ and $f(t_1, t_2, \ldots, t_n)$ is $f(LGG(s_1, t_1), \ldots, LGG(s_n, t_n))$. The LGG of terms $f(s_1, \ldots, s_n)$ and $g(t_1, \ldots, t_n)$, where $f \neq g$ is the variable v where v represents this pair of terms throughout.
2. The LGG of two terms $p(s_1, s_2, .., s_n)$ and $p(t_1, t_2, \ldots, t_n)$ is $\mathrm{p}(LGG(s_1, t_1), \ldots, LGG(s_n, t_n))$,
3. LGG is undefined when the sign or predicate symbols are unequal.
4. The LGG of two clauses C_1 and C_2 is $\{l : l_1 \in C_1$ and $l_2 \in C_2$ and l_1 has the same sign and predicate symbol as l_2 and $l = lgg(l_1, l_2)\}$.

Consider, for example, the two clauses $\{apply(f, 1)\}$ and $\{apply(g, 1).\}$. The CLGG of these two clauses is $\{apply(X, 1)\}$.

3.4 CRLGG: Curried relative least general generalization of clauses

The advantage of ILP systems over other learning systems is in their ease of using background knowledge in guiding the induction process. In Plotkin's learning framework, the LGG definition is generalised to include generalizations in the presence of background knowledge. This generalization in the presence of background knowledge is called relative least general generalization (RLGG).

Let P be a set of clauses in our language, i.e., P is a set of curried first order clauses. Let C and D be two curried first order clauses. The curried relative least general generalization $CRLGG_P(C, D)$ of C and D relative to P, is the least general clause within the θ-subsumption lattice for which $P \wedge CRLGG_P(C, D) \rightarrow C \wedge D$.

Similar to the RLGG of ILP, the CRLGG of a pair of clauses in our system can also be pretty large, thus requiring suitable restrictions for achieving feasible higher-order generalizations.

4 Some meaningful higher-order generalizations achievable by CLGG

By removing the distinction between the unary and n-ary functions, we are able to represent higher order terms. Here we show that this representation in CLGG is able to capture some elementary higher order properties which cannot be captured in first order logic, without recourse to λ-calculus.

Example 1 Transitivity. Consider the first order predicates brighter-than and lighter-than. We know that if lighter-than(X,U) and lighter-than(U,Z) are both true, then lighter-than(X,Z) is also true for any X,U and Z. Similar property holds true for brighter-than. If we were to capture this generic higher-order property of **transitivity** of predicates, it is impossible to capture this notion in one first order clause. But in our curried syntax, such a property can be captured by a clause $\{apply(apply(P, X), Z) \leftarrow apply(apply(P, X),$
$Y), apply(apply(P, Y), Z).\}$. We can obtain instantiations of the clause for the predicate constants lighter-than and brighter-than respectively.

Example 2 Sortedness. Consider a list of integers. For convenience we shall represent a list in the prolog syntax as opposed to the curried constructor based notation for lists. Say we have a list $[X|Y]$, we know that the predicate ascending-order-sorted(L), where L is a list, can be represented by the set of clauses, $\{ascending - order - sorted([X]), ascending - order - sorted([X|[Y|Z]]) \leftarrow X <= Y, ascending{-}order{-}sorted([Y|Z])\}$. Similar set of clauses for descending-order can be shown. But if we can capture the underlying relation in either case which is the generic property of **sortedness**, we can represent it conveniently in the curried syntax: $\{sorted([X]),$
$sorted([X|[Y|Z]]) \leftarrow X <= Y, sorted([Y|Z])\}$. This notion of the higher order property of sortedness cannot be captured in first order logic.

Example 3 Inverse & Identity . Consider the generic mathematical operation of inverse. If we were to define the property of inverse and identity for each mathematical operation we would have to define individual inverse and identity predicates for each mathematical operation. Consider addition(Plus) where we assume 0 to be the additive identity. The inverse predicate would be defined by plus(inv-plus(x),x)=0. And analogously for multiplication we would have mult(inv-mult(x),x)=1. We can generalize the two operations of inverse and identity for any algebraic operation, and represent these generic versions of inverse and identity in the curried syntax. The generic higher-order predicates of **inverse** and **identity** can be defined by the equality relation $apply(apply(P, apply(apply(inverse, P), x)), x) = apply(identity, P)$. Here we can substitute add or multiply for P, and get the unary minus and reciprocal operations for the inverse respectively and 0, 1 for identities, respectively.

Example 4 Program transformations. Many examples representing similarities in structure between different logic programs can also be be captured in our curried syntax. This justifies the use of our generalization algorithms in program transformations. For example, a clause like $apply(P, apply(f, X)) \leftarrow apply(f, apply(P, X))$ represents a generic structure for programs of the type $addtwo(s(x)) \leftarrow s(addtwo(x))$, where addtwo represents addition by two and s represents the successor function. So program transformations is one high potential area where our generalization model has promising applications.

5 Issues in the use of CLGG algorithm to build a feasible higher order learning system

The advantage of using CLGG is in the fact that the algorithm is a direct application of the first order LGG algorithm. Hence this enables us to export ideas used in reducing the complexity of the generalization process in first order LGG-based systems like GOLEM [MF90], to our higher order CLGG system.

The extension of the features used to control the explosion of choices in GOLEM needs to be studied with reference to the modified curried syntax. In this context the first question that needs to be examined is the capability of GOLEM to generalize in the new modified syntax. In the following subsection, we show some examples, executed in GOLEM [MF90], which show the feasibility of using restrictions analogous to GOLEM for curried terms. This provides a basis for an implementable higher order generalization system based on LGG.

5.1 Experiments with GOLEM

In GOLEM, the underlying bias restrictions to the language of clauses are both semantic and syntactic. Here we show the achievability of generalization of certain classes of curried higher order terms and also curried first order terms using GOLEM. We present two sample runs from GOLEM for curried representation

each showing generalization capabilities of first order and higher order curried terms respectively.

Example 1

```
% Background Knowledge
apply(milk,a1).
apply(milk,a2).
apply(milk,a3).
apply(milk,a4).
apply(aquatic,a5).
apply(aquatic,a6).
apply(aquatic,a7).
apply(aquatic,a8).
apply(apply(class,a1),mammal).
apply(apply(class,a2),mammal).
apply(apply(class,a3),mammal).
apply(apply(class,a4),mammal).
apply(apply(class,a5),fish).
apply(apply(class,a6),fish).
apply(apply(class,a7),fish).
apply(apply(class,a8),fish).

%  Positive Examples

apply(apply(class,a1),mammal).
apply(apply(class,a2),mammal).
apply(apply(class,a3),mammal).
apply(apply(class,a4),mammal).

apply(apply(class,a5),fish).
apply(apply(class,a6),fish).
apply(apply(class,a7),fish).
apply(apply(class,a8),fish).
% Negative Examples

apply(apply(class,a5),mammal).
apply(apply(class,a6),mammal).
apply(apply(class,a7),mammal).
apply(apply(class,a8),mammal).
apply(apply(class,a1),fish).
apply(apply(class,a2),fish).
apply(apply(class,a3),fish).
apply(apply(class,a4),fish).
apply(milk,a5).
```

```
% GOLEM output

apply(apply(class,A),mammal) :- apply(milk,A).
apply(apply(class,A),fish) :- apply(aquatic,A).
```

This example shows how the terms in the first order component of curried terms are still generalizable in GOLEM under the curried representation.

Example 2

```
% Background Knowledge

!- randseed.

apply(apply(mem,0),[0]).
apply(apply(mem,1),[1]).
apply(apply(mem,2),[2]).
apply(apply(mem,3),[3]).
apply(apply(mem,4),[4]).
apply(apply(mem,0),[0,0]).
apply(apply(mem,1),[0,1]).
apply(apply(mem,0),[1,0]).
apply(apply(mem,0),[2,0]).
apply(apply(mem,1),[1,1]).
apply(apply(mem,1),[2,1]).
apply(apply(mem,2),[2,2]).
apply(apply(mem,2),[3,2]).
apply(apply(mem,3),[2,3]).
apply(apply(mem,3),[4,2,3]).
apply(nat,0).
apply(nat,1).
apply(nat,2).
apply(nat,3).
apply(nat,4).
apply(apply(memb,0),[0]).
apply(apply(memb,1),[1]).
apply(apply(memb,2),[2]).
apply(apply(memb,3),[3]).
apply(apply(memb,4),[4]).
apply(apply(memb,0),[0,0]).
apply(apply(memb,1),[0,1]).
apply(apply(memb,0),[1,0]).
apply(apply(memb,0),[2,0]).
apply(apply(memb,1),[1,1]).
apply(apply(memb,1),[2,1]).
```

```
apply(apply(memb,2),[2,2]).
apply(apply(memb,2),[3,2]).
apply(apply(memb,3),[2,3]).
apply(apply(memb,3),[4,2,3]).

%Positive Examples

apply(apply(mem,0),[0]).
apply(apply(mem,1),[1]).
apply(apply(mem,2),[2]).
apply(apply(mem,3),[3]).
apply(apply(mem,4),[4]).
apply(apply(mem,0),[0,0]).
apply(apply(mem,1),[0,1]).
apply(apply(mem,0),[1,0]).
apply(apply(mem,0),[2,0]).
apply(apply(mem,1),[1,1]).
apply(apply(mem,1),[2,1]).
apply(apply(mem,2),[2,2]).
apply(apply(mem,2),[3,2]).
apply(apply(mem,3),[2,3]).
apply(apply(mem,3),[4,2,3]).
apply(apply(memb,0),[0]).
apply(apply(memb,1),[1]).
apply(apply(memb,2),[2]).
apply(apply(memb,3),[3]).
apply(apply(memb,4),[4]).
apply(apply(memb,0),[0,0]).
apply(apply(memb,1),[0,1]).
apply(apply(memb,0),[1,0]).
apply(apply(memb,0),[2,0]).
apply(apply(memb,1),[1,1]).
apply(apply(memb,1),[2,1]).
apply(apply(memb,2),[2,2]).
apply(apply(memb,2),[3,2]).
apply(apply(memb,3),[2,3]).
apply(apply(memb,3),[4,2,3]).

% Negative examples

apply(apply(mem,0),[1,2]).
apply(apply(mem,3),[]).
apply(apply(mem,0),[1]).
apply(apply(memb,0),[1,2]).
apply(apply(memb,3),[]).
```

```
apply(apply(memb,0),[1]).
apply(nat,[]).

% GOLEM output

apply(apply(A,B),[B|C]).
apply(apply(A,B),[C,D|E]) :- apply(apply(A,B),[D|E]).
```

This example illustrates the higher order capabilities of the LGG algorithm under our curried syntax. Both predicates *mem* and *memb* have the same structure within the output program representable by the curried outcome returned above. Therefore we can conclude that currying added to GOLEM has higher order capabilities not captured in the original GOLEM.

Additional observations The two examples in the previous section, show the applicability of semantic and syntactic bias restrictions used by GOLEM even in the curried representation. But GOLEM has certain other kinds of meta information too, e.g. modes of predicates, which is provided by the user to restrain the size of the RLGG obtained, and to control the explosion of possible alternatives. Such type of meta information needs to be incorporated in computation of CRLGG in order to gain similar efficiency. Some first order examples in GOLEM were not generalizable in a controlled manner in the curried form, due to the lack of the appropriate meta-information for curried terms. But a suitable change in the appropriate component of GOLEM (for dealing with mode information for curried representation of the terms) should help us achieve the full power of GOLEM for our curried representation.

5.2 Efficiency Issues

In the previous subsection it was shown that currying provides the capability of generalizing certain kinds of higher order information in GOLEM. It was shown that simple first order terms generalizable under GOLEM are also generalizable under the curried syntax. This leads to the conclusion that the bias restrictions on the clauses in GOLEM, which make the task of generalizing with RLGG achievable, are also applicable to the curried first order terms. In this context the observations made in the previous subsection stress the need for a change in the part of the procedural engine of GOLEM which deals with the mode information of arguments of predicates. A systematic study of the restricted subset of curried first order terms needs to be done as to what the bias restrictions imposed in GOLEM mean in the curried context. More precisely, the bias restrictions like generative clauses, determinacy and functional dependencies need to be studied in the curried context.

6 Applications of CLGG

As seen above, there are a variety of different higher order predicates that can be characterized by curried first order terms. So CLGG can be exploited in a variety of application domains where higher-order information is used. We briefly describe two such classes of applications.

6.1 Program transformations

Some restrictions of the CLGG can be used to generate transformations equivalent to certain program transformations. As shown above, CLGG can be used to capture similar programs into one common structure. This enables us to reason about equivalence of different programs based on templates of programs written in curried form which are obtained by generalization. In the following, we consider a special case of CLGG, which again resembles a class of program transformation algorithms.

Hyper Least General Generalization: a special case of CLGG If we restrict our curried generalization to generalization of two predicates at a time, along with generalization of variables but not involving any function generalization, we get a restricted form of generalization. We can call this new form of generalization after Furukawa's term hyper least general generalization (HLGG) [FIG97]. HLGG is defined for two literals like

$$p(g(a), a)$$
$$q(g(b), b)$$

The generalization of two literals that have *different* predicate symbols causes the invention of a new predicate. Let the two clauses to be generalized by HLGG [FIG97] be $C1$ and $C2$, and the two literals chosen from $C1$ and $C2$ be $p(T1)$ and $q(T2)$, respectively. During the HLGG process on $C1$ and $C2$, assume that $p(T1)$ and $q(T2)$ are generalized by HLGG and the result $gen\_p\_q(T)$ is obtained. This new predicate $gen\_p\_q(T)$ is defined by

$$\begin{aligned} gen\_p\_q(T) &:- p(T1). \\ gen\_p\_q(T) &:- q(T2). \end{aligned} \tag{1}$$

where T, $T1$ and $T2$ are terms and $T = LGG(T1, T2)$. The new predicate $gen\_p\_q(T)$ represents $p \vee q$, or a superconcept of p and q.

The generalized procedure induced via generalization in HLGG is equivalent to an application of the program transformation **folding** operation [BD77] to the original clauses, using the newly invented predicate. That is, the folding of clauses $C1$ and $C2$ by definition (1) yields the same effect as applying HLGG to $C1$ and $C2$.

This folding operation can also be considered as a kind of relative generalization. That is, performing LGG relative to the definitions of newly invented predicates will produce the HLGG C of $C1$ and $C2$. This follows because

$$BK \cup \{C\} \models C1 \wedge C2$$

where BK denotes background knowledge. This suggests an efficient implementation of clause-HLGG by first introducing new predicates using literal-HLGG and then performing RLGG.

HLGG can be formulated as a special case of CLGG, involving a two step process. In the first step, we compute the CLGG of the two terms which are generalizable by HLGG. Let the two terms involved in the HLGG computation be $p(T1)$ and $q(T2)$. The CLGG of the two terms is of the form app(X,T), where T is the curried translation of LGG of T1 and T2, and X is a new variable not found in either T1 or T2. We then replace the X in $app(X,T)$ by a predicate constant gen_p_q. On translation of this curried expression into normal form and addition of the two clauses as in the HLGG definition above, we have the required expression for HLGG.

The equivalence of the HLGG, shown above to be a restricted application of CLGG, and the program transformation technique of folding, suggests that CLGG can be used for more complex program transformations.

6.2 Bias Generator for ILP systems

As indicated in the introduction, higher order templates are commonly used in specifying bias for induction of logic programs. In particular second order clauses form the basis for bias to control the induction in the ILP systems CIA [RB92] and RDT [KW92]. For example, if the bias is the clause $\{P(X) \leftarrow Q(X), R(X)\}$ then the available vocabulary for the induction is the set of instances of the above clause.

Given ground positive examples with suitable background examples together with negative examples, the generalized algorithm based on currying can generate curried higher order clauses like the above clause used as bias in RDT [KW92]. So CLGG provides a tool for automating the process of generating biases in ILP. The biases generated from curried generalization of ground examples can be used to form the appropriate bias needed to guide the induction process in ILP systems which use higher-order schema as the syntactic bias. This idea provides a useful tool for experimenting with suitable biases based upon ground example inputs.

7 Conclusions and Scope for future work

This paper introduces a novel concept in first order induction algorithms endowing them with the capability of generalizing higher-order clauses which are otherwise not representable in first order logic. The important difference from

an earlier work [FM92] is the non-reliance on λ-calculus as the higher order representation language. In [FM92], λ-calculus is used as the underlying paradigm and generalization algorithms involved are complex due to this representation.

In our presentation, we have exported the notion of *currying* from functional programming to induction of first order logic expressions. This resulted in a superset of first order logic in which some key higher-order functions can be represented without taking recourse to λ-calculus. The generalization algorithms applied to first order logic in ILP have been used to generate higher-order clauses without recourse to λ-calculus. Extensions of first order generalization algorithms in ILP to curried first order logic have been presented. The GOLEM examples show the feasibility of inducing higher-order clauses within this framework. Additionally, at least two important application areas of CLGG based generalization, namely program transformations and bias generation for ILP, have been outlined.

Further studies is underway on the following extensions of the framework presented:

1. A theoretical characterization of semantic and syntactic restrictions used in GOLEM, when applied to curried terms. This also includes the study of different classes of restrictions of the curried first order logic and generation of meaningful clauses corresponding to such restrictions.
2. Use of CLGG as a program transformation tool.
3. Use of the CLGG as a tool for automating the process of selection of biases for certain first order induction algorithms like RDT.
4. Incorporation of mode based semantic information into the curried generalization algorithm in extended GOLEM.

Acknowledgements

This work has been supported by the Natural Sciences and Engineering Research Council of Canada, and by the Canadian Federal Networks of Centres of Excellence Institute for Robotics and Intelligent Systems.

References

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programts. *Journal of the ACM*, 24(1):44–67, 1977.

[Cur30] Haskell B. Curry. Grundlagender kombinatorischen logik. *Am. J. Math.*, 52:509–536, 1930.

[FIG97] K. Furukawa, M. Imai, and R. Goebel. Hyper least general generalization and its application to higher-order concept learning. Reserach Memo IEI-RM 97001, SFC Research Institute, Keio University, 5322 Endo, Fujisawa-shi, Kanagawa 252, Japan, 1997.

[FM92] C. Feng and S. Muggleton. Towards inductive generalisation in higher order logic. In D. Sleeman and P. Edwards, editors, *Proceedings of the Ninth International Workshop on Machine Learning*, San Mateo, California, 1992. Morgan Kaufman.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[KW92] J. Kietz and S. Wrobel. *Controlling the Complexity of Learning in Logic through Syntactic and Task-Oriented Models.* Inductive Logic Programming. Academic Press, 1992.

[MF90] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, Tokyo, Japan, 1990.

[MJ94] Stephen Muggleton and C.David Page Jr. Beyond first-order learning : Inductive learning with higher-order logic. Technical Report PRG-TR-13-94, Oxford University, UK, 1994.

[Mug92] S. Muggleton, editor. *Inductive logic programming.* Academic Press, New York, 1992.

[NM90] G. Nadathur and D. Miller. Higher-order horn clauses. *Journal of the ACM*, 37(4):777–814, 1990.

[Plo70] G.D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 6, pages 153–163. Edinburgh University Press, Edinburgh, 1970.

[Plo71a] G.D. Plotkin. Automatic methods of inductive inference. Ph.d. dissertation, University of Edinburgh, Edinburgh, Scotland, 1971.

[Plo71b] G.D. Plotkin. A further note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 6, pages 101–124. Edinburgh University Press, Edinburgh, 1971.

[Pop70] R.J. Popplestone. An experiment in automatic deduction. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages ???—??? Edinburgh University Press, Edinburgh, 1970.

[Rae93] Luc De Raedt. A brief introduction to inductive logic programming. In *Proceedings of the 1993 International Symposium on Logic Programming*, pages 45–51, Vancouver, Canada, October 26-29 1993.

[RB92] L. De Raedt and M. Bruynooghe. Interactive theory revision: an inductive logic programming approach. *Machine Learning*, 8(2), 1992.

[Sch24] M. Schonfinkel. Uber die baustine der matematischen logik. *Math. Annalen*, 92:305–316, 1924.

This article was processed using the LaTeX macro package with LLNCS style