Towards Inductive Generalisation in Higher Order Logic

Cao Feng, Stephen Muggleton
The Turing Institute, George House, 36 North Hanover Street,
Glasgow G1 2AD, UK

Abstract

In many cases, higher order (Horn) clauses are more suited to express certain concepts when relations between predicates exist. However, to date there has been no appropriate higher order formalism within which efficient inductive generalisation can be carried out. This paper describes inductive generalisation in M_λ – a higher order formalism which not only retains the expressiveness of λ-calculus but also provides for effective and efficient inductive generalisation. The main strength of R_λ is twofold: it is a higher formalism extension of the (clausal) first order logic and it can be mechanised in a way similar to the first order case in Horn clause form. For a class of restricted M_λ, their least general generalisation (LGG) is unique, and so is their most general unification. Inductive generalisation in M_λ is implemented in the algorithm HOLGG. This algorithm has been applied to some interesting induction problems in the induction of higher order rule templates and automatic program transformation.

1 Introduction

Generalisation forms the basis of most inductive learning systems. In first order logic, generalisation has been well-understood [Plotkin1970, Plotkin1971, Reynolds1970], and many algorithms have been devised based on these principles [Muggleton and Feng1990]. Using a first order inductive tool [Muggleton and Feng1990], we can obtain the clauses:

$$\forall X \forall Y \forall Z.ancestor(X,Y) \leftarrow ancestor(X,Z), ancestor(Z,Y). \quad (1)$$

$$\forall X \forall Y \forall Z.less\_than(X,Y) \leftarrow less\_than(X,Z), less\_than(Z,Y). \quad (2)$$

which are the generalisation respectively of the facts $less\_than(1,3)$, $less\_than(2,4)$, ... and the facts $ancestor(john, steve)$, $ancestor(steve, mike)$, ...

The clauses in (1) and (2) are very similar in the sense that (1) can be obtained from (2) by exchanging the predicate symbol *ancestor* with *less_than* in all its appearances, and vice versa. If we allow for a special variable P, it is not difficult to see that both clauses in (1) and (2) are "substitution instances" of the clause:

$$\forall X \forall Y \forall Z.P(X,Y) \leftarrow P(X,Z), P(Z,Y). \quad (3)$$

with P being substituted by *ancestor* for (1) and *less_than* for (2). As P is a predicate variable, this clause needs a higher order language that goes beyond the first order predicate logic used in present machine learning research and applications.

In general, a term E is more general than another term F whenever there is a substitution θ for which $E\theta = F$. This is normally called term θ-subsumption. Obviously formula (3) is more general than both (1) and (2). Because the clause in (3) contains the higher order variable "P", this generality cannot be understood under its meaning in first order logic. It has to be made precise by formal semantic and syntactic definitions in higher order logic. Thus we need to consider the problem of how P, X, Y and Z are interpreted in (1), (2) and (3). This has been partially tackled by logic programming in higher order Horn clauses [Nadathur and Miller1990], which is an extension of (first order) logic programming methods to deal with higher-order terms.

Obviously, the higher order clause is a more powerful representation both in terms of expressiveness and efficiency. Conversely, if we are given the higher order

clause such as the clause in (3) we hope that we can more quickly find the rules (1) and (2) with respect to some given facts (perhaps in higher order).

2 Motivation

In order to introduce a higher order language we need to address several issues pertinent in inductive learning. The main concerns are:

- **Expressiveness.** Meta-level information about inductive problems can, in many cases, only be expressed adequately with higher order terms. The induction of program transformation rules in [Huet and Lang1978] is a case in point.
- **Induction.** Inasmuch machine learning is concerned, we need to consider how (3) can be obtained, especially from the given formulae such as (1) and (2). This gives rise to the need of the induction of (3). Similar to the case in first-order logic [Plotkin1970, Plotkin1971, Muggleton and Feng1990], this may be solved by generating the least general generalisation (LGG) of terms in this language.
- **Efficiency.** To achieve practical efficiency, deduction and induction in this language must be easy to compute. In specific, corresponding to first order logic the lower bound of generalisations and unifications of terms must be unique as both are unique for the language of first order terms.

Following Robinson, unification has become the basis of resolution-based logic deduction. Huet [Huet1975] found a semi-deterministic unification algorithm for higher-order terms. Inspired by the success of the logic programming language Prolog, λProlog [Nadathur and Miller1990] has been developed as a full higher-order extension of Prolog. For example, the formula (3) is a λProlog clause. λProlog has been successfully applied in writing program transformers, theorem provers and a number of other areas [Paulson1986, Dietzen and Pfenning1989].

Similar to most general unification (MGU), the least general generalisation (LGG) of terms [Plotkin1970, Muggleton and Feng1990] plays an important role in the emerging field of *inductive logic programming*, which is evolved from logic programming and conventional machine learning [Muggleton1990]. However, as the field matures the need for various higher-order notions has arisen in order to guide the induction of Horn clauses from facts (ground atoms). These include rule templates [Raedt and Bruyonooghe1990, Kietz and Wrobelto appear, Harao1990], predicate determinations
and modes [Muggleton and Feng1990], variable types [Lavrac and Dzeroski1991, Quinlan1990] and predicate commutativity [Lavrac and Dzeroski1991].

At present, higher order terms already have many applications in machine learning: these include inductive learning [Muggleton and Feng1990], analogical reasoning [Harao1990], constructive induction [Raedt and Bruyonooghe1990] and model-driven induction [Kietz and Wrobelto appear]. It is argued that higher-order terms provide elegant expressions for many problems. However, the higher order formalisms used in research to date are unsatisfactory and lack a sound semantics. In particular they deliberately avoid using λ-calculus. They therefore lose the expressive power of higher-order logic.

In a higher order logic, a term E is more *general* than another term F, or F is more specific than E, if and only if there is a substitution θ such that $E\theta$ is λ-convertible to F. This is denoted by $E \geq_\theta F$ or $F \leq_\theta E$. The relation $\geq_\theta$ is transitive, nonsymmetric and reflexive and thus defines a partial ordering over terms. $E =_\theta F$, or E is θ-equal to F if and only if $E \geq_\theta F$ and $F \geq_\theta E$. When $E \geq_\theta F$ but $E \neq_\theta F$, we say $E >_\theta F$ or $F <_\theta E$.

A term E is a *common generalisation* of a set of terms $\mathcal{T}$ if and only if E is more general than each of the terms in $\mathcal{T}$. A term E is a *least general generalisation* of a finite set of terms $\mathcal{T}$ if and only if: 1) E is a common generalisation of $\mathcal{T}$; and 2) for any $F <_\theta E$, F is not a common generalisation of $\mathcal{T}$.

It is shown in [Feng and Muggleton1991] that in general there are multiple or an infinite number of solutions for the LGG of two higher-order terms that are expressed in full $\alpha\beta\eta$ λ-calculus (see Appendix A). In order to compute LGGs efficiently, it is desirable to have a restricted λ calculus such that MGU and LGG are unique in the condition of not sacrificing effectiveness. It is, indeed, only the formulae similar to (3) in Section 1 that we are most interested in. To achieve this we will have to place some additional conditions on λ-calculus and hopefully we can find a manageable subclass of the general λ-calculus. One such restriction is L_λ [Miller1990]. L_λ [Miller1990, Pfenning1991] is a restricted subset of λProlog which has unique MGU and LGG. But it is ill-equipped to express many problems. In particular it cannot represent recursions in its terms which we shall see in Section 6.

3 M_λ: a restricted higher order language

Readers unfamiliar with simple (typed) λ-calculus may go to the Appendix A for the relevant information on λ-calculus. Throughout this paper, the terminology is similar to that used in logic programming, especially Prolog. Variables are denoted by upper case letters U to Z; formulae and terms by other upper case letters; constants by lower case letters; types by Greek letters α, β and γ; and substitutions by Greek letters θ, δ and etc., when confusion does not arise. The constant "$\leftarrow$" is often used as an infix operator for convenience of understanding. In a few occasions we will use calligraphical upper-case letters to denote sets. Suitable super- or sub-scripts may also be used.

A L_λ [Miller1990] term contains only free variables that are applied to bound variables. For example, the formula

$$\lambda X.\lambda Y.\lambda Z.(P(X,Y) \leftarrow P(X,Z), P(Z,Y)). \quad (4)$$

is a L_λ term. The free variable P only has arguments that are bound variables in the formula. P is said to be applied to X and Y and λX is called an abstraction from $\lambda Y.\lambda Z.(P(X,Y) \leftarrow P(X,Z), P(Z,Y))$. A variable in an abstraction is called a bound variable, otherwise it is a free variable. We will also say a variable is free to a term (or subterm) if it is not bound by that term (or subterm). When there are many successive abstraction variables in a formula such as in (4) we may write "λXYZ".

The term in (4) is closely related to (3) as the quantifier"$\forall X.F$" is an abbreviation for "$\Pi(\lambda X.F)$" and Π is a constant expressing universal quantification (and similarly "$\exists X.F$" is for "$\Sigma(\lambda X.F)$").

Though MGU and LGG is unique in L_λ, it is too constrained to express simple terms that contain recursion. Practical examples of the restrictions of L_λ can be seen in Section 6. But now let us look at a simple modification of the above term:

$$\lambda XYZ.(P(s(X),Y) \leftarrow P(X,Z), P(Z,s(Y))). \quad (5)$$

Because it contains a constant s in the arguments of P, (5) is not a L_λ term which is important in many logic programming problems (see Section 6). We shall introduce some extra-logical extensions to λProlog. This will be called M_λ. In this calculus, a subterm such as $s(X)$, called an object term, is explicitly allowed.

An *object term* consists of externally bound variables and constants of distinct types from any bound variables without abstraction. A variable X in a term F is *externally bound* if F is a subterm of some term E and X is bound by an abstraction in E outside of F. The use of object terms gives rise to the necessity of the other extensions in M_λ.

M_λ is a restricted typed λ-calculus. In M_λ: 1) It is allowed to perform α, β, δ_0 and η conversions on terms. The δ_0 conversion rule is described in Appendix B; 2) Any free variables in M_λ terms are only applied to arguments that are object terms; 3) It contains at least constants χ and ψ such that for any object term of M_λ, E, $\chi(E) = E_2$ and $\psi(E) = E_1$ if $E = (E_1 E_2)$, and $\psi(E) = \chi(E) = E$ otherwise. (Both are undefined for non-object terms).

Clearly M_λ is an extension of the L_λ language [Miller1990], in which the δ_0 rule is not permitted and the arguments of free variable functions can only be externally bound variables. χ and ψ are analytical selectors in an analytical syntax as defined by McCarthy. Fortunately many programming languages, such as Lisp and Prolog, have these functions. In M_λ these are further restricted to the application to object terms, which do not contain abstractions as defined in this paper. Also note these extensions are introduce constants into the formalism and therefore are extra-logical.

4 M_λ normal and nonredundant terms

A term is M_λ *normal* if and only if it is in $\alpha\beta\delta_0\eta$ normal form and contains no irreducible δ_0 expressions. We will be interested in LGGs and MGIs that are M_λ normal. The following definition is adopted from that in Section 2. E is the M_λ *normal LGG* of a set of M_λ normal terms $\mathcal{T}$, if and only if: 1) E is M_λ normal and is also a common generalisation of $\mathcal{T}$; and 2) F is not a common generalisation of $\mathcal{T}$ for any M_λ normal term $F <_\theta E$.

For M_λ normal term $S = p(U, U(a))$, the substitution containing δ_0 conversions such as $\theta = \{U/\lambda X.\mathit{if}\ X = a\ \mathit{then}\ E_1\ \mathit{else}\ E_2\}$ is not applicable [Curry *et al.*1958]. If applied it results in $S\theta = p((\lambda X.\mathit{if}\ X = a\ \mathit{then}\ E_1\ \mathit{else}\ E_2), E_1)$, which cannot be reduced further because X in $X = a$ is a free variable to X. Thus this term is not a M_λ normal term.

Let E be a M_λ term and E' be a θ-equal term to E. A subterm F of E will have some trace F', so to speak, in E'. F' is called the *residual* of F. A subterm F of E is *redundant* if $E'' = E' - \{F'\}$ and $E'' =_\theta E$, i.e the residual of F can be removed and still maintain θ-equality.

Clearly, any redundancy can only happen in terms

with free variable functors. If E is a subterm of F with free variable functor and no other subterm of F has the same functor, then any term containing only constants and repetitive bound variables in E is redundant: Because E is the only subterm that has the free variable as its functor, we are free to devise various substitutions to decompose E into its subterms and then remove all constants and repetitive bound variables from it. The operation to remove redundant subterms is called a reduction (not to be confused with α, β, δ_0 and η reductions of λ calculus). After simple reductions, constants will appear only in subterms that appear in multiple places of a terms. [Feng and Muggleton1991] describes an algorithm that can reduce a M_λ to its nonredundant form. In the rest of the paper we refer terms to M_λ normal, nonredundant terms and their LGGs to M_λ normal, nonredundant LGGs.

We need also to consider unification, which is the dual of LGG. E is the *common instance* of a set of M_λ terms $\mathcal{T}$ if E is more specific than each element of $\mathcal{T}$. E is the most general instance (MGI) of a set of M_λ terms $\mathcal{T}$ if and only if: 1) E is the common instance of $\mathcal{T}$; and 2) F is not the common instance of $\mathcal{T}$ for any $F >_\theta E$. The substitution θ for which $E_i\theta = E, \forall E_i \in \mathcal{T}$ is called the most general unifier (MGU). M_λ normal MGU is similarly defined except that E, F and $E_i \in \mathcal{T}$ are all M_λ normal terms. It is proved in [Feng and Muggleton1991], in normal and nonredundant form the MGU and LGG of M_λ terms are unique. The unification algorithm is also given in [Feng and Muggleton1991].

5 Implementation

LGG and unification in M_λ is implemented in Prolog, and is called HOLGG. The LGG algorithm has two parses. The first parse is ELGG which collects all the multiple appearances of subterms into a set of triples $\Gamma = \{\langle \tilde{X}, S_1, T_1\rangle\}$. Then Γ is sorted to obtain $\Gamma = \bigcup_{i=1}^{m} \Gamma_i$ for which $\Gamma_i = \{\langle \tilde{X}, S_j, T_j\rangle\}$ and each S_{j1} and S_{j2} in Γ_i have the same functor and so do T_{j1} and T_{j2}. In the second parse, CLGG takes S, T and Γ_i ($i = 1, ..., m$) as input and produces F. CLGG also calls the algorithm VLGG.

MLGG: anti-unification (LGG) algorithm for M_λ normal terms.
INPUT: two M_λ normal terms S and T of the same type;
OUTPUT: a M_λ normal term $F = MLGG(S, T)$.
1. $\Gamma = ELGG(S, T, \emptyset)$;
2. Sort Γ such that $\Gamma = \bigcup_{i=1}^{n} \Gamma_i$ for which $\Gamma_i = \{\langle \tilde{X}, S_i(S_{j1}, ..., S_{jn}), T_i(T_{j1}, ..., s_{jk})\rangle\}$ for $i = 1, ..., m$;
3. $F = CLGG(S, T, \Gamma, \emptyset)$.

ELGG
INPUT: S, T and the binding variables $\tilde{X}$;
OUTPUT: $\Gamma = \{\langle F_p, S_p, T_p\rangle\}$ where F_p, S_p have different functors.
1. If $S = \lambda\tilde{Y}.S_1$ and $T = \lambda\tilde{Y}.T_1$, then $\Gamma = ELGG(S_1, T_1, \tilde{X}\tilde{Y})$;
2. If $S = S_0(S_1, ..., S_n)$, $T = T_0(T_1, ..., T_n)$ and $S_0 = T_0 = C$, where C is a constant or $C \in \tilde{X}$, then $\Gamma = \bigcup_{i=1}^{n} ELGG(S_i, T_i, \tilde{X})$;
3. If $S_0 \neq T_0$, then $\Gamma = \{\langle \tilde{X}, S, T\rangle\}$.

CLGG
INPUT: S, T, the sorted Γ and the binding variables $\tilde{X}$;
OUTPUT: F is the M_λ normal LGG of S and T.
1. If $S = \lambda\tilde{Y}.S_1$ and $T = \lambda\tilde{Y}.T_1$, $F = \lambda\tilde{Y}.CLGG(S_1, T_1, \Gamma, \tilde{X}\tilde{Y})$;
2. If $S = S_0(S_1, ..., S_n)$, $T = T_0(T_1, ..., T_n)$ and $S_0 = T_0$, then $F = C(F_1, ..., F_n)$ where $F_i = CLGG(S_i, T_i, \Gamma, \tilde{X})$;
3. If $S = S_0(S_1, ..., S_n)$, $T = T_0(T_1, ..., T_n)$ and $S_0 \neq T_0$, and $\langle \tilde{X}, S, T\rangle$ is in Γ_i and there exist F_i ($i = 1, ..., l$),
   3.1. If S_0 and T_0 are constants or free variables, $F = V_{S_0,T_0}(F_1, F_2, ..., F_l)$;
   3.2. If $S_0 \in \tilde{X}$ and T_0 is a constant or a free variable, $F = V_{S_0,T_0}(S_0, F_1, F_2, ..., F_l)$;
   3.3. If S_0 is a constant or a free variable and $T_0 \in \tilde{X}$, then $F = V_{S_0,T_0}(T_0, F_1, F_2, ..., F_l)$;
   3.4. If $S_0 \in \tilde{X}$ and $T_0 \in \tilde{X}$, then $F = V_{S_0,T_0}(S_0, T_0, F_1, F_2, ..., F_l)$; where $F_S = VLGG(S_{i1}, VLGG(..., VLGG(S_{i,j-1}, S_{ij}, \tilde{X}), ...), \tilde{X})$ and $F_T = VLGG(T_{i1}, VLGG(..., VLGG(T_{i,j-1}, T_{ij}, \tilde{X}), ...), \tilde{X})$ so $S = (...((\lambda\tilde{X}.F_S)F_1)...F_l)$ and $T = (...((\lambda\tilde{X}.F_T)F_1)...F_l)$;
4. If $S = S_0(S_1, ..., S_n)$, $T = T_0(T_1, ..., T_n)$ and $S_0 \neq T_0$,
   4.1. If S_0 and T_0 are constants or free variables, $F = V_{S_0,T_0}(X_1, X_2, ..., X_l)$;
   4.2. If $S_0 \in \tilde{X}$ and T_0 is a constant or free variable, $F = V_{S_0,T_0}(S_0, X_1, X_2, ..., X_l)$;
   4.3. If S_0 is a constant or a free variable and $T_0 \in \tilde{X}$, then $F = V_{S_0,T_0}(T_0, X_1, X_2, ..., X_l)$;
   4.4. If $S_0 \in \tilde{X}$ and $T_0 \in \tilde{X}$, $F = V_{S_0,T_0}(S_0, T_0, X_1, X_2, ..., X_l)$; where $X_i \in \tilde{X}$

$(i = 1, ..., l)$ are bound variables in S and T.

VLGG: Variablisation of terms in M_λ.
INPUT: two terms S and T and binding variables $\tilde{X}$;
OUTPUT: $F = VLGG(S, T, \tilde{X})$.
1. If $S = \lambda\tilde{Y}.S_1$ and $T = \lambda\tilde{Y}.T_1$, then $F = \lambda\tilde{Y}.VLGG(S_1, T_1, \tilde{X}\tilde{Y})$;
2. If $S = T = C$ and C is a constant or $C \in \tilde{X}$, then $F = C$;
3. If $S = C(S_1, ..., S_n)$ and $T = C(T_1, ..., T_n)$ and C is a constant or $C \in \tilde{X}$, then $F = C(VLGG(S_1, T_1, \tilde{X}), ..., VLGG(S_n, T_n, \tilde{X}))$;
4. If S, T contain variables in $\tilde{X}$, then fail otherwise $F = V_{S,T}$ for a variable named by S and T.

6 Applications

LGG has played an important role in inductive logic programming in first-order logic. The following examples show the applications of LGG to acquire higher-order clause templates from given first-order clauses.

Example 1. Given the first-order facts:

less_than(0,2), less_than(1,3), ...
less_than(0,3), less_than(1,4), ...
less_than(0,1), less_than(1,2), ...

we are able to obtain the following, using an algorithm such as Golem [Muggleton and Feng1990]:

$$\forall XYZ.less\_than(X, Y) \leftarrow less\_than(X, Z), less\_than(Z, Y),$$

and similarly

$$\forall XYZ.ancestor(X, Y) \leftarrow ancestor(X, Z), ancestor(Z, Y),$$

In both clauses X, Y and Z are universally quantified. Note $\forall X.F$ is the abbreviation of $\Pi(\lambda X.F)$ and Π expresses universal quantification. Their higher-order LGG is

$$\forall XYZ.P(X, Y) \leftarrow P(X, Z), P(Y, Y),$$

where P is a free variable and "$\leftarrow$" is an infix constant. P may then become universally quantified as this generalisation is accepted.

Though L_λ can still be used to express the higher-order term in Example 1, the clause in the following example, which contains recursion and represents a majority of problems that we are interested in, cannot be expressed within L_λ. This is because L_λ forbids the existence of constants in the arguments of free variables.

Example 2. Given the first-order clauses with X, Y, Z and W universally quantified,

$$\forall XYZW.reverse(cons(X, Y), Z) \leftarrow reverse(Y, W), append(W, cons(X, nil), Z).$$
$$\forall XYZW.insert\_sort(cons(X, Y), Z) \leftarrow insert\_sort(Y, W), insert(X, W, Z).$$

we can obtain

$$\forall XYZW.P(cons(X, Y), Z) \leftarrow P(Y, W), Q(W, X, Z).$$

where "*reverse*" is a version often referred to as "naive reverse", P and Q are free variables, and "*cons*" is a list processing function.

One may observe that the first-order facts seem to be the objects in the induction of first-order clauses. These clauses then become objects that characterise the properties of *higher-order objects* — in this particular case it is second order predicate constants such as "*reverse*" and "*insert*". At this "order", we are mainly concerned with the properties of the predicates, the first order objects will be universally quantified, and they are "taken for granted" when studying objects that may apply on them. After this, the higher-order objects may become universally quantified.

If we extend this scenario further, we can imagine that through progressive quantification clauses of successive orders can be induced that characterise objects of higher-order objects.

Another application area is the discovery of program transformation rules. Huet and Lang [Huet and Lang1978] discussed methods for program transformation of recursive computations into iterative ones. A set of second order clause templates for transformation were suggested and they are applied through second order unification to produce more efficient programs based on the Darlington and Burstall [Burstall and Darlington1977] method. As they remarked, the opposite problem with regarding to the discovery of such templates is a difficult task. Few templates are known and no automatic methods exist for performing such discovery.

Example 3. The higher-order logic clause in Example 1, though interesting, is computationally inefficient. To satisfy the first predicate $P(X, Y)$, it needs to non-deterministically satisfy $P(X, Z)$ and then $P(Z, Y)$. More efficient programs for *ancestor* and *less_than* are

respectively:

$$\forall XYZ.less\_than(X,Y) \leftarrow successor(X,Z), less\_than(Z,X).$$
$$\forall XYZ.ancestor(X,Y) \leftarrow parent(X,Z), ancestor(Z,X).$$

where $successor(X,Z)$ expresses that Z is the successor of X in Peano's formalism, and $parent(X,Z)$ states that Z is the parent of X. Both are computational more efficient. Thus possible program transformations are:

$$(\forall XYZ.less\_than(X,Y) \leftarrow successor(X,Z), less\_than(Z,X))$$
$$\Longleftarrow$$
$$(\forall XYZ.less\_than(X,Y) \leftarrow less\_than(X,Z), less\_than(X,Y)),$$

$$(\forall XYZ.ancestor(X,Y) \leftarrow parent(X,Z), ancestor(Z,X))$$
$$\Longleftarrow$$
$$(\forall XYZ.ancestor(X,Y) \leftarrow ancestor(X,Z), ancestor(X,Y)).$$

The LGG produces a program transformation template, though the conditions for the transformation are omitted.

$$(\forall XYZ.P(X,Y) \leftarrow Q(X,Z), P(Z,X)))$$
$$\Longleftarrow$$
$$(\forall XYZ.P(X,Y) \leftarrow P(X,Z), P(X,Y)).$$

In fact, such a template is applicable when Q is a special case of P (i.e. P by one).

Example 4. The recursive list reverse program is described in 3. A more efficient iterative (tail recursive) version of it is:

$$\forall XYZWU.reverse1(cons(X,Y),Z,W) \leftarrow append([X],Z,U), reverse1(Y,U,W))$$

where "$reverse1$" contains an accumulator Z. When "$reverse1$" starts with "$reverse1(List, nil, ReversedList)$" and is terminated by "$reverse1(nil, ReversedList, ReversedList)$", it yields the reversed list. Thus we have a transformation

$$(\forall Z.reverse1(nill,Z,Z)\ \&$$
$$\forall XYZWU.reverse1(cons(X,Y),Z,W) \leftarrow append([X],Z,U), reverse1(Y,U,W))$$
$$\Longleftarrow$$
$$(reverse(nil,nil)\ \&$$
$$\forall XYZW.reverse(cons(X,Y),Z) \leftarrow reverse(Y,W), append(W,cons(X,nil),Z)).$$

We also know another transformation which concerns with the addition of the elements in a list:

$$(\forall Z.sumlist1(nil,Z,Z)\ \&$$
$$\forall XYZWU.sumlist1(cons(X,Y),Z,W) \leftarrow add(X,Z,U), sumlist1(Y,U,W))$$
$$\Longleftarrow$$
$$(sumlist(nil,0)\ \&$$
$$\forall XYZW.sumlist(cons(X,Y),Z) \leftarrow sumlist(Y,W), add(W,X,Z)).$$

where "*plus*" is a function that returns the addition of two numbers. When started with "$sumlist1(List, 0, SumOfList)$" and terminated by "$sumlist1(nil, SumOfList, SumOfList)$", the iterative computation also returns the sum of the elements in the list. The LGG of the two is:

$$(\forall Z.P1(nil,Z,Z)\ \&$$
$$\forall XYZWU.P1(cons(X,Y),Z,W) \leftarrow Q(Z,X,U), P1(Y,U,W))$$
$$\Longleftarrow$$
$$P(nil,V)\ \&$$
$$\forall XYZW.P(cons(X,Y),Z) \leftarrow P(Y,W), Q(W,X,Z)).$$

with free variables P, Q, $P1$ and V.

This is an alternative expression of McCarthy's transformation [McCarthy1960]. For the sake of convenience we have omitted the conditions for this transformation to apply. It is in fact that, among others, V must be the lower bound element of the appropriate type and Q be a transitive and communicative function. This problem can be addressed by relative least general generalisation (RLGG) that will be discussed briefly in Section 7. However, its detail is beyond the scope of this paper.

The other potential application areas are analogical reasoning and the automatic acquisition of grammar rules from example sentences and the generalisation of proofs. However we will not discuss them in this paper.

7 Conclusion and future research directions

Recently, inductive logic programming [Muggleton1990] has witnessed a growing trend in utilising higher-order (or meta-level) logical notions in existing ILP framework. This is motivated, in part, by the need to develop more effective and efficient ILP methods. These notions are often adopted as declarative biases in many forms including functional constraints on the predicates in clauses and

templates for the clauses being induced. However current methods lack a coherent framework for accommodating these notions. In this paper, we found a class of higher order terms that are sufficiently expressive and still have unique MGI and LGG. Their generalisations also proved to be computationally efficient. This can be demonstrated by the various applications in Section 6. This lays the foundation for studying the relative least general generalisation of such terms.

Our future research work is concerned with expanding the current (first order) ILP framework, in which higher-order inductive inference may be described as the discovery of a hypothesis H from examples and background knowledge such that:

$$M \wedge B \wedge H \vdash E^+$$
$$M \wedge B \wedge H \not\vdash E^- \qquad (6)$$

if $M \wedge B \not\vdash E^+$. M in relation (6) represents a set of higher-order (λ Prolog) clauses. The hypotheses in H now can be either first-order or higher-order clauses. B is the background knowledge, E^+ and E^- are respectively the set of positive and negative examples. Corresponding to existing ILP theory, it is necessary to develop methods for generalisation in higher-order logic. In doing so we hope to achieve two aims: a) to develop more efficient methods for inducing first-order clauses, and b) to induce higher-order clauses such that they can aid induction of both classes of logic programs.

From the experience of ILP in first-order logic, in the next step we need to study the relative least general generalisation (RLGG) for higher-order logic programs in the presence of background knowledge. Similar to the first-order case [Muggleton and Feng1990], we may have to deal with a restricted logical model of the background knowledge. We also need to investigate the computability of such a model of the background knowledge. If this is successful the results will have implications in ILP and to the discovery of automatic program transformation techniques.

Acknowledgements. The authors are grateful to the ILP group at the Turing Institute. We are also thankful to Dale Miller and Frank Pfenning for providing general information in higher-order logic programming.

A Syntax of λ-calculus terms

A term in (simply typed) λ-calculus can be one of the following:

1. **Atom.** A variable or constant (of type α) is a term (of type α);
2. **Application.** An application of E (of type $\beta \to \alpha$) to F (of type β) is a term $(E\ F)$ (of type α);
3. **Abstraction.** The abstraction of a term F (of type α) on a variable X (of type β) is the term $(\lambda X.F)$ (of type $\beta \to \alpha$) that binds X in the scope F.

The type of a term F is denoted by $\tau(F)$. One may verify that (1), (2) and (3) in Section 1 are terms, and $\tau(P) = (\alpha, \alpha \to \beta)$ and $\tau(\leftarrow) = (\beta, \beta, \beta \to \gamma)$, assuming that $\tau(X) = \tau(Y) = \tau(Z) = \alpha$. Types are not essential in some results. They are polymorphic when used.

We denote $(...((F\ E_1)\ E_2)\ ...E_n)$ by the expression $F(E_1, E_2, ..., E_n)$ if F is an atom, and its type $(\alpha_1 \to (\alpha_2 \to ...(\alpha_n \to \beta)...))$ by $(\alpha_1, \alpha_2, ..., \alpha_n \to \beta)$, where $\tau(E_i) = \alpha_i$. F is called the functor, E_i $(i = 1, 2, ..., n)$ the arguments and n the arity of F. We also abbreviate $(\lambda X_1.(\lambda X_2...(\lambda X_n.F)...))$ to $(\lambda X_1 X_2...X_n.F)$ and $X_1 X_2...X_n$ to $\tilde{X}$.

The *order* of an atom is the depth of the nesting of parentheses in its type + 1. The order of a term is the highest order of its atoms. The order of X, Y and Z in (3) is 1, and the order of P is 2. The order of (3) is therefore 2.

A.1 λ-conversions

Let $F\{X/E\}$ be the operation of substitution that replaces each occurrence of X in F by E. X is free in F if it does not occur in the scope of an abstraction in F that binds X. E is free for X in F if E does not appear in the scope of an abstraction in F that binds X. These two conditions are used to avoid possible name clashes in the following definition of λ conversions.

A *substitution* is a set of ordered pairs $\theta = \{X_i/E_i \mid i = 1, ..., n\}$, where X_i are distinct variables and each E_i is a term of the same type as X_i. The application of θ to a term F is denoted by $F\theta$ and it is the term $(...((\lambda\tilde{X}.F)E_1)E_2)...)E_n)$. Intuitively for each $X_i (i = 1, ..., n)$ $F\theta$ is the results of replacing X at each place in F by the subterm E_i. The composition of substitutions, denoted by $\theta \cdot \delta$, is the same as defined in first-order logic (sometimes we may omit the dot).

The *conversion rules* of the general λ-calculus is:

1. α**-rule.** $\lambda X.F$ to $\lambda Y.(F\{X/Y\})$ if Y is free for X in F, and vice versa;
2. β**-rule.** $(\lambda X.F)E$ to $F\{X/E\}$ if E is free for X in F, and vice versa;
3. η**-rule.** $\lambda X.FX$ to F if X is not free in F,

and vice versa.

The ***convertibility***[1] of two terms is an equivalence relation. The rules can be carried out in both left-to-right and right-to-left directions. λ-Convertible terms are considered to be equal to each other. An application of a conversion rule is called a α (or β and etc.) ***reduction*** when applied in the left-to-right direction and an ***expansion*** in the opposite direction. A λ term that cannot be reduced by rules of any kind is said to be in ***normal form.***

Church and Rosser proved that the normal form of a term is unique. For any convertible λ terms E and F, there is a term S in normal form such that E and F can be reduced to S. They also proved the normal form of a term can be obtained by attacking the leftmost reduction on one-by-one basis until no reduction is possible. We should bear in mind at this point, apart from α, β and η rules, there are other conversion rules that also maintain Church and Rosser properties such as the δ and δ_0 rules that will be introduced in Appendix B.

B Extension to δ_0 conversion

We will first consider a conversion rule of δ_0, which is a variant of the δ conversion rule studied by Church in an extension to $\alpha\beta\eta$ λ-calculus. The δ_0 conversion rule is defined as follows. If E and F are in $\beta\delta_0\eta$ normal form without variables free in E and F:

1. $\delta_0 EF = \lambda XY.X$ if E is α-convertible to F;
2. $\delta_0 EF = \lambda XY.Y$ otherwise.

$\lambda XY.X$ is known as "true" and $\lambda XY.Y$ as "false" in the standard λ-calculus (In Church's system, truth is expressed by $\lambda XY.XY$ — the combinatory number 1 – and falsity by $\lambda XY.X(XY)$ — the number 2. This is an arbitrary choice).

δ_0 rule is similar to a weaker form of the equality theory introduced in modern deductive logic. Similarly to $\alpha\beta\eta$ λ-calculus, the application of the δ_0 rule from left-to-right is called a δ_0 reduction. It is shown that δ_0 reductions maintain the Church-Rosser properties of λ-calculus when the reduction expressions are carried over from the original (perhaps not in $\beta\delta_0\eta$ normal form) terms. We shall restrict ourselves to such a λ-calculus.

The term $\lambda XY.X$ selects the first argument of X and Y, whereas $\lambda XY.Y$ selects the second. Using the δ_0 rule we can construct a term: $F = (\lambda X.((\delta_0 XN)E_1)E_2)$. When F is applied to a term M, we obtain $FM = (((\delta_0 MN)E_1)E_2)$ which corresponds to "if M is α convertible to N, then E_1 else E_2", where $\delta_0 MN$ is the δ_0 reduction expression and E_1 and E_2 are the terms applied upon by this reduction. M, N, E_1 and E_2 are in $\beta\delta_0\eta$ normal form. M and N contain no variables free in them. We shall denote F by "$\lambda X.if\ X = N\ then\ E_1\ else\ E_2$". Clearly, $(\lambda X.E_1)$ is convertible to $(\lambda X.if\ X = N\ then\ E_1\ else\ E_2)$ if E_1 is α-convertible to E_2.

[1] Note often an α conversion is used prior to a β conversion to change the names of bound variables.

References

[Burstall and Darlington1977] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1), 1977.

[Curry *et al.*1958] H. Curry, R. Feys, and W. Craig. *Combinatory Logic (Volume I).* North Holland, Amsterdam, Holland, 1958.

[Dietzen and Pfenning1989] S. Dietzen and F. Pfenning. Higher-order and modal logic as a framework for explanation-based generalization. In B. Spatz, editor, *Proceedings of the sixth international workshop on machine learning. Cornell University, Ithaca, New York*, pages 447 – 449. San Mateo, CA: Morgan Kaufmann, June 1989.

[Feng and Muggleton1991] Cao Feng and Stephen Muggleton. Least general generalisation in higher order logic. TIRM, The Turing Institute, George House, 36 NOrth Hanover St, Glasgow, UK, 1991.

[Harao1990] Masateru Harao. Analogical reasoning based on higher order unification. In *First International Conference on Algorithmic Learning Theory*, pages 151–163, Tokyo, Japan, 1990. Japanese Society for Artificial Intellligence.

[Huet and Lang1978] G. Huet and B. Lang. Proving and applying program transformations expressed with second order patterns. *Acta Informatica*, 11:31–55, 1978.

[Huet1975] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Kietz and Wrobelto appear] J. Kietz and S. Wrobel. Controlling the complexity of learning in logic through syntact ic and task orieted models. In S. Muggleton, editor, *Proceedings of the First International Workshop on Inductive Logic Programming*, Viana de Castelo, Portugal, to appear. Academic Press.

[Lavrac and Dzeroski1991] N. Lavrac and S. Dzeroski. Learning nonrecursive definitions of relations with

linus. In Y. Kodratoff, editor, *EWSL '91: machine learning: proceedings of the European working session on learning*, pages 265–281, Porto, Portugal., 1991. Berlin: Springer-Verlag.

[McCarthy1960] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine (part 1). *Communications of the Association of Computational Machinery*, 3:184–204, 1960.

[Miller1990] D. Miller. A logic programming language with λ-abstraction, function variables and simple unification. In P. Schroeder-Heister, editor, *Extensions of logic programming*, pages 237–258. Sprinter-Verlag LNCS, 1990.

[Muggleton and Feng1990] S. Muggleton and C. Feng. Efficient induction of logic programs. In *First International Conference on Algorithmic Learning Theory*, pages 369–381, Tokyo, Japan, 1990. Japanese Society for Artificial Intellligence.

[Muggleton1990] S. Muggleton. Inductive logic programming. In *First International Conference on Algorithmic Learning Theory*, volume Also in New Generation Computing, 1991, Vol 8, pages 42–61, Tokyo, Japan, 1990. Japanese Society for Artificial Intellligence.

[Nadathur and Miller1990] G. Nadathur and D. Miller. Higher-order horn clauses. *Journal of the Association for Computing Machinery*, 37(4):777–814, 1990.

[Paulson1986] L. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3(3):237–258, 1986.

[Pfenning1991] F. Pfenning. Unification and anti-unification in the Calculus of Constructions. In G. Kahn, editor, *Sixth Annual Symposium on Logic in Computer Science*, pages 74 – 85. IEEE, July 1991.

[Plotkin1970] G.D. Plotkin. A note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163. Elsevier North-Holland, New York, 1970.

[Plotkin1971] G.D. Plotkin. A further note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 6*, pages 101–124. Elsevier North-Holland, New York, 1971.

[Quinlan1990] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[Raedt and Bruyonooghe1990] L. De Raedt and M. Bruyonooghe. Constructive induction by analogy: a new method to learn how to learn? In K. Morik, editor, *EWSL'89: proceedings of the fourth european working session on learning*, Montpellier, France, 1990. London: Pitman.

[Reynolds1970] J.C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 135–151. Elsevier North-Holland, New York, 1970.