A Strong Complete Schema for Inductive Functional Logic Programming*

J. Hernández-Orallo M.J. Ramírez-Quintana

DSIC, UPV, Camino de Vera s/n, 46020 Valencia, Spain.
{jorallo,mramirez}@dsic.upv.es

Abstract. A new IFLP schema is presented as a general framework for the induction of functional logic programs (FLP). Since narrowing (which is the most usual operational semantics of FLP) performs a unification (mgu) followed by a replacement, we introduce two main operators in our IFLP schema: a generalisation and an inverse replacement or intra-replacement, which results in a generic inversion of the transitive property of equality. We prove that this schema is *strong* complete in the way that, given some evidence, it is possible to induce any program which could have generated that evidence. We outline some possible restrictions in order to improve the tractability of the schema. We also show that inverse narrowing is just a special case of our IFLP schema. Finally, a straightforward extension of the IFLP schema to function invention is illustrated.
Keywords: Functional Logic Programming, Inductive Logic Programming, Function Invention, Induction of Auxiliary Functions, Narrowing, Inverse Narrowing.

1 Introduction

Inductive logic programming (ILP) [9] is the branch of machine learning that studies concept learning in a logical framework. Namely, ILP deals with the induction of logic programs (i.e. finite sets of Horn clauses) from examples and background knowledge.

The use of logic programming for learning is mainly based on the idea that logic programs are a single representation for examples, background knowledge and hypotheses. However, logic languages like Prolog (the most representative language of this paradigm) lack some programming facilities such as evaluable and nested functions, types, higher order programming and lazy evaluation. Although these features are well supported by functional languages, they lack the computing power provided by logical variables and unification. Hence, the interest in the integration of both families of languages has grown over the last few years.

Integrated languages fully exploit the facilities of logic programming in a general sense: functions, predicates and equality. One relevant approach [4, 6]

* This work has been partially supported by CICYT under grant TIC 98-0445-C03-C1.

to integration is functional logic programming where the programs are logic programs which are augmented with Horn equational theories. A lot of work has been invested in the development of the semantics of integrated languages. Therefore, it has been shown that the main semantic properties of logic programs also hold for functional logic programs (least model, fixpoint semantics) [1][1]. Operational semantics is defined in terms of semantic unification or $\mathcal{E}$-unification [15] (i.e., general unification wrt an equational theory $\mathcal{E}$). Narrowing [5, 14] is a sound and complete $\mathcal{E}$-unification method for theories which satisfy some requirements (such as confluence and termination properties or the absence of extra variables in the condition of the equations). Narrowing can be seen as a combination of resolution from logic programming and term reduction from functional programming. Hence, it is widely accepted that narrowing is the key to describing operational semantics of functional logic languages.

In [3] we have presented a framework for the induction of functional logic programs (IFLP) from (positive and negative) examples. The evidence is composed of equations, and their rhs's are normalised wrt the background knowledge and the theory to be induced. In logic programming, the induction can be made top-down (starting from the most general program and refining it by specialisation) or bottom-up (starting from positive data as a program and generalising it). In the case of functional logic programs, we cannot follow a top-down direction because the examples are equations, and the most general program $X = Y$ would not make the program terminating nor confluent. As a consequence, the kernel of our method was an inverse narrowing mechanism (similar to the inverse resolution operator of ILP) which selects pairs of equations to obtain an equation which is usually more general than the original ones. The starting set of equations is a generalisation of the positive examples which is made by replacing terms by variables at some occurrences. In fact, the algorithm combines inverse narrowing and generalisation in each step. The method is effective, but it is too specific for those cases where auxiliary terms are involved.

Let us show this with an example.

Example 1. Consider the following evidence

$$E^+ = \left\{ \begin{array}{l} e_1^+ : f(a) = r(g(b,b)) \\ e_2^+ : h(a,b) = r(a) \end{array} \right\}$$

and suppose that sufficient negative examples are provided to justify the program

$$P = \left\{ \begin{array}{l} r_1 : h(Y,a) = g(Y,b) \\ r_2 : f(X) = r(h(b,X)) \\ r_3 : h(a,b) = r(a) \end{array} \right\}$$

However, P could never be induced by inverse narrowing. This is because the example e_1^+ directly relates the function symbols f, r and g (we are not considering other constant symbols in the equations), whereas the equations r_1 and r_2 from P define the function f in terms of r and g but through the function h. This last function can be thought of as an auxiliary function in the definition

[1] In this paper we do not address any questions related to declarative semantics.

of f. The generalisation step in the inverse narrowing approach does not take this possibility into account since there is no positive evidence that links the symbols f and h nor the symbols h and g.

In this paper, we define a new framework, the IFLP schema, as a general and strong complete framework for solving the IFLP problem. By strong completeness we refer to the capability of inducing *all* possible programs such that the positive examples hold wrt them but the negative examples do not. The term 'strong' is due to the fact that, in this context, weak completeness makes no sense since it is always possible to find a program that covers all the positive examples and none of the negative ones: the positive examples themselves. Other completeness results could be stated in terms of some extra conditions that the program should follow (e.g. Progol). The idea is to generalise the way in which the narrowing relation is inverted to induce theories which use auxiliary functions. The inductive method proposed is closely related to the transitive property of equality. More exactly, we define a new operator that reverses the direction in which transitivity is applied. Then, we prove that the schema is complete in the sense mentioned above. We also show that the IFLP schema is rather general to have inverse narrowing as one of its instances. Finally, we deal with the function invention problem which can be easily formalised in our schema. In this context, we can consider an invented function as an auxiliary function of a new signature that extends the hypothesis language with new functions.

The work is organised as follows. In Section 2, we recall the main concepts of functional logic programming and we formalise the narrowing semantics we focus on. Section 3 reviews the inverse narrowing approach and analyses the way in that theories are induced. This motivates the introduction of new operators to overcome the limitations of inverse narrowing. The IFLP schema is defined in Section 4. The strong completeness of the schema is discussed in Section 5. Section 6 shows that inverse narrowing is an instance of our schema. In Section 7, the setting is easily changed to include function invention. Finally, Section 8 concludes the paper and discusses future work.

2 Preliminaries

We briefly review some basic concepts about equations, Term Rewriting Systems and $\mathcal{E}$-unification. For any concept which is not explicitly defined, the reader may refer to [2, 8, 15].

Let Σ be a set of *function symbols* (or functors) together with their arity[2] and let $\mathcal{X}$ be a countably infinite set of *variables*. Then $\mathcal{T}(\Sigma, \mathcal{X})$ denotes the set of *terms* built from Σ and $\mathcal{X}$. The set of variables occurring in a term t is denoted $Var(t)$. This notation naturally extends to other syntactic objects (like clause, literal, ...). A term t is a *ground term* if $Var(t) = \emptyset$. A *substitution* is defined as a mapping from the set of variables $\mathcal{X}$ into the set of terms $\mathcal{T}(\Sigma, \mathcal{X})$. An *occurrence* u in a term t is represented by a sequence of natural numbers. $O(t)$ and $\bar{O}(t)$

[2] We assume that Σ contains at least one constant.

denote the *set of occurrences* and *non-variable occurrences* of t respectively. $t_{|u}$ denotes the *subterm* of t at the occurrence u and $t[t']_u$ denotes the *replacement* of the subterm of t at the occurrence u by the term t'. An equation is an expression of the form $l = r$ where l and r are terms. l is called the left hand side (lhs) of the equation and r is the right hand side (rhs). An equational theory $\mathcal{E}$ (which we call *program*) is a finite set of equational clauses of the form $l = r \Leftarrow e_1, \ldots, e_n$. with $n \geq 0$ where e_i is an equation, $1 \leq i \leq n$. The theory (and the clauses) are called *conditional* if $n > 0$ and *unconditional* if $n = 0$. An equational theory can also be viewed as a (Conditional) Term Rewriting System (CTRS) since the equation in the head is implicitly oriented from left to right and the literals e_i in the body are ordinary non-oriented equations. Given a (C)TRS $\mathcal{R}$, $t \rightarrow_{\mathcal{R}} s$ is a rewrite step if there exists an ocurrence u of t, a rule $l = r \in \mathcal{R}$ and a substitution θ with $t_{|u} = \theta(l)$ and $s = t[\theta(r)]_u$. A term t is said to be in *normal form* wrt $\mathcal{R}$ if there is no term t' with $t \rightarrow_{\mathcal{R}} t'$. We say that an equation $t = s$ is normalized wrt $\mathcal{R}$ if t and s are in normal form. $\mathcal{R}$ is said to be *canonical* if the binary one-step rewriting relation $\rightarrow_{\mathcal{R}}$ is terminating (there is no infinite chain $s_1 \rightarrow_{\mathcal{R}} s_2 \rightarrow_{\mathcal{R}} s_3 \rightarrow_{\mathcal{R}} \ldots$) and confluent ($\forall\, s_1, s_2, s_3 \in \mathcal{T}(\Sigma, \mathcal{X})$ such that $s_1 \rightarrow^*_{\mathcal{R}} s_2$ and $s_1 \rightarrow^*_{\mathcal{R}} s_3, \exists\, s \in \mathcal{T}(\Sigma, \mathcal{X})$ such that $s_2 \rightarrow^*_{\mathcal{R}} s$ and $s_3 \rightarrow^*_{\mathcal{R}} s$). An $\mathcal{E}$-unification algorithm defines a procedure for solving an equation $t = s$ within the theory $\mathcal{E}$. Narrowing is a sound and complete method for solving equations wrt canonical programs. Given a program P, a term t *narrows* into a term t' (in symbols $t \overset{u,l=r,\theta}{\hookrightarrow}_P t'$[3]) iff $u \in \bar{O}(t)$, $l = r$ is a new variant of a rule from P, $\theta = mgu(t_{|u}, l)$ and $t' = \theta(t[r]_u)$. We write $t \hookrightarrow^n_P t'$ if t narrows into t' in n narrowing steps.

3 The Inverse Narrowing Approach

In this section, we briefly outline the inverse narrowing approach we have presented in [3]. The algorithm was composed of two operators: Consistent Restricted Generalisation and Inverse Narrowing.

Since we had to ensure posterior satisfiability, the inverse narrowing method began generating all possible restricted generalisations from each positive example which was consistent with both positive and negative examples. It was computed by the Consistent Restricted Generalisation operator.

Definition 1. Consistent Restricted Generalisation CRG
An equation $e = \{l_1 = r_1\}$ is a consistent restricted generalisation (CRG) wrt E^+ and E^- and an existing theory $T = B \cup P$ if and only if e is a restricted generalisation for some equation of E^+[4] (always oriented left to right) and there does not exist: (1) a narrowing chain using e and T that yields some equation

[3] Or simply $t \overset{l=r,\theta}{\hookrightarrow}_P t'$ or $t \overset{\theta}{\hookrightarrow}_P t'$ if the occurrence or the rule is clear from the context. Also, the subscript P will usually be dropped when clear from the context.

[4] An equation $t = s$ is a restricted generalisation of an equation $r = m$ if $\exists\sigma : \sigma(t) = r \wedge \sigma(s) = m$ and $\forall x(x \in Var(s) \Rightarrow x \in Var(t))$.

of E^-, and (2) a narrowing chain using e and T that yields a different normal form for some lhs different from the rhs which appeared in the equations of E^+.

Secondly, the inverse narrowing operator was defined as an operator that generates an equation from two equations.

Definition 2. Inverse Narrowing
Given a functional logic program P, we say that a term t conversely narrows into a term t', and we write $t \overset{u,l=r,\theta}{\hookleftarrow}_P t'$, iff $u \in O(t)$, $l = r$ is a new variant of a rule from P, $\theta = mgu(t_{|u}, r)$ and $t' = \theta(t[l]_u)$. The relation $\hookleftarrow_P$ is called the inverse narrowing relation.

Now, we will concentrate our attention on how the inverse narrowing approach induces equations. Suppose that $s = t$ and $l = r$ are the equations selected by the algorithm, such that $t_{|u}$ unifies with r with $\theta = mgu(t_{|u}, r)$. Then, $s = \theta(t[l]_u)$ and $l = r$ are the two equations induced in an inverse narrowing approach step. It is easy to see the relationship between this algorithm and the transitive property of equality. In what follows, for the sake of legibility, we consider x, y and z to be subterms at the occurrence ϵ. The next rationale is still valid for any other occurrence in a term.

The transitive property is expressed as:

$$x \to y \wedge y \to z \Rightarrow x \to z \ (1)$$

whereas an inverse narrowing approach step can be also represented as:

$$x \to z \wedge y \to z \Rightarrow x \to y \wedge y \to z \ (2)$$

where $x \to y$ in (2) is the equation computed by inverse narrowing from $x \to z$ and $y \to z$. However, (2) is not a real inversion of transitivity because it begins from two equations (one of the premises and the result) of the formula (1) and it generates its other premise. To have a constructive inversion of the transitivity of equality, the behaviour of the algorithm should be as follows:

$$x \to z \Rightarrow x \to y \wedge y \to z \ (3)$$

where $x \to y$ and $y \to z$ are the result of this constructive inverse narrowing. Notice that the term y in the above formula (3) is new. The following schema not only extends the setting to cope with this inverse transitive, but also to cope with inverse replacement. This is the mechanism which will allow us to introduce auxiliary functions in the inductive process.

4 IFLP Schema

Let us denote the set of function symbols of arity ≥ 0 which appear in a program P as Σ_P. In the same way, Σ_{E^+}, or simply Σ^+, denotes the set of function symbols of arity ≥ 0 which appear in the positive evidence E^+.

As we have stated, narrowing is based on a mgu, which is a specialisation, followed by a replacement. It is logical then to base the induction of functional logic programs on an inversion of these deductive operators. Consequently, we introduce two operators: an inverse specialisation, namely a generalisation, and an inverse replacement.

Definition 3. Unrestricted Generalisation (UG)
An equation $e' = \{l' = r'\}$ is an unrestricted generalisation (UG) of an equation $e = \{l = r\}$ if and only if there exists a substitution θ such that $\theta(l') = l$ and $\theta(r') = r$.

Definition 4. Single Intra-Replacement (SIR)
Given an equation $s = t$, choose any occurrence ω of t and any function symbol $F \in \Sigma^+$ to construct a new term q in the following way:

$$q = t[\phi]_\omega$$

where $\phi = F(X_{k,1}, X_{k,2}, \ldots, X_{k,n})$, $n \geq 0$ is the arity of F and $X_{k,i}$ are different fresh variables. The subscript k is used to distinguish these variables from other variables in previous or subsequent uses of this operator.

As output, the SIR operator produces a first equation D_k as: $s = q$, and a second equation E_k as: $q_{|\omega} = t_{|\omega}$

The first result from this definition is that D_k and E_k make true that $s \hookrightarrow^2 t$, i.e. s can be narrowed into t in two narrowing steps, because $s \overset{\epsilon, s=q, \emptyset}{\hookrightarrow} q$ and $q \overset{\omega, q_{|\omega}=t_{|\omega}, \emptyset}{\hookrightarrow} t$. Following the definition, and taking into account both D_k and E_k, the operator SIR can only generalise. However, if the occurrence ω is a variable X, the second equation is of the form $t = X$. If we remove this equation, it can be said that SIR specialises. Despite this seemingly contradictory behaviour, the operator must be used interactively in order to specialise a variable into a term which has more than one function symbol.

Example 2. Suppose an original equation $f(g(a)) = b$ and $\Sigma^+ = \{f, g, h, a, b\}$ with their corresponding arities. By choosing the occurence $\omega = \epsilon$ and $F = h$, we generate the following two equations:

a first equation D_k as: $f(g(a)) = h(X_{k,1}, X_{k,2})$
and a second equation E_k as: $h(X_{k,1}, X_{k,2}) = b$

We can apply the same operator to D_k at occurrence $\omega' = 2$ and $F = a$. This gives

a third equation D_{k+1} as: $f(g(a)) = h(X_{k,1}, a)$
and a fourth equation E_{k+1} as: $a = X_{k,2}$

It is easy to show that the original equation is covered by the program which can be constructed from $D_k, E_k, D_{k+1}, E_{k+1}$. However, it would be interesting to be able to specialise the lhs of E_k's and to allow more then one new symbol on the lhs.

Both things can be obtained by using the following simple operator:

Definition 5. Syntactic Folding (SF)
Given two equations $E_1 = \{l_1 = r_1\}$ and $E_2 = \{l_2 = r_2\}$ with r_1 being a variable such that there exists an occurrence ω such that $r_1 \equiv (l_2)_{|\omega}$, a new folded equation

can be constructed as $l_2[l_1]_\omega = r_2$. *The same applies if such an occurrence is in* r_2.

In the previous example E_{k+1} and E_k could be folded into $h(X_{k,1}, a) = b$ by using the occurrence ω=2.

Example 3. Consider Example 1 again. If the first equation from the evidence is selected, i.e. $f(a) = r(g(b, b))$, and the SIR operator is applied at occurrence $\omega = 1$ and with function symbol h, the following two equations are generated:

a first equation D_k is: $f(a) = r(h(X_{k,1}, X_{k,2}))$

and a second equation E_k as: $h(X_{k,1}, X_{k,2}) = g(b, b)$

We can apply the same operator to D_k at occurrence $\omega' = 1.1$ and $F = b$. This produces:

a third equation D_{k+1} as: $f(a) = r(h(b, X_{k,2})$

and a fourth equation E_{k+1} as: $b = X_{k,1}$

If SIR is applied again to D_{k+1} but now at occurrence $\omega' = 1.2$ and $F = a$, this gives:

a fifth equation D_{k+2} as: $f(a) = r(h(b, a)$

and a sixth equation E_{k+2} as: $a = X_{k,2}$

Equation D_{k+2} can be generalised into $f(X_{k,2}) = r(h(b, X_{k,2})$ which is one rule of program P. By using the SF operator, E_k and E_{k+2} can be folded into $h(X_{k,1}, a) = g(b, b)$ and then folded again by using E_{k+1} into $h(b, a) = g(b, b)$ which can then be generalised into $h(X_{k,1}, a) = g(X_{k,1}, b)$, which is another rule of the program.

These three operators are able to construct virtually any term as the following lemma and theorem show:

Lemma 1. *Select any term* r *constructable from* $\mathcal{T}(\Sigma)$. *Given any equation* $s = t$ *and any occurrence* ω *of* t *there exists a finite combination of the SIR and SF operators that generates these two equations:*

a first equation D *as:* $s = q$

and a second equation E *as:* $q_{|\omega} = t_{|\omega}$

where $q = t[r]_\omega$.

Proof. Let us prove this lemma by mathematical induction. Consider d equal to the depth of the tree which can be drawn from r, e.g. $f(g(a, h(a, a)))$ has depth 4.

For $d = 1$, the lemma is obvious because it is only necessary to apply the SIR operator at occurrence ω with the term $\phi = r$.

Let us suppose the hypothesis that the lemma is true for k. Then, we have to show that it is true for $k+1$. Consider that $r_{|u} = g(a_1, a_2, \ldots, a_n)$ where a_i are function symbols of arity 0, i.e. constants, and $u = x_1.x_2.\ldots.x_k$ and there is no other occurrence at level $k+1$ but the a_i. By hypothesis we have been able to construct two equations for depth k:

a first equation D_k as: $s = q$

and a second equation E_k as: $q_{|\omega} = t_{|\omega}$

where $q = t[r']_\omega$, with r' being $r[a]_{x_1.x_2.\ldots.x_k}$ where this a does not appear again in r'. Since this a appears once, it is obvious that this step could have been avoided and we could have a variable X instead of a term a as well.

Let us apply the SIR operator to the first equation at occurrence $\omega' = x_1.x_2.\ldots.x_k$ with $\phi = g(X_{k,1}, X_{k,2}, \ldots, X_{k,n})$, $n \geq 0$ is the arity of F and $X_{k,i}$ are different fresh

variables. This generates two equations:
a first equation D_{k+1} as: $s = q'$
and a second equation E_{k+1} as: $q'_{|\omega'} = q_{|\omega'}$
where $q' = q[\phi]_{\omega'}$. We can apply the SIR operator n times with function symbol a to D_{k+1} at all its n positions giving respectively:
a first equation D_i as: $s = q'[a_i]_i$
and a second equation E_i as: $a = X_{k,i}$
These E_i can be used jointly with D_{k+1} by operator SF to construct a new equation A, $s = q[g(a_1, a_2, \ldots, a_n)]_{\omega'}$, which is equal to $s = t[r]_\omega$, and D_i can be used jointly with D_{k+1} by operator SF for a second equation, $q[g(a_1, a_2, \ldots, a_n)]_{\omega'} = q_{|\omega'}$. Finally, since the rhs of this last equation is X, we can apply a SF operator to this last equation and E_k giving an equation B, $r = t_{|\omega}$. Both A and B are precisely the equations D and E of the lemma.

Since this holds for $k+1$ if it holds for k, we can affirm that it holds for all k. □

Theorem 1. *Select any term r' which is constructable from $\mathcal{T}(\Sigma, \mathcal{X})$. Given any equation $s = t$ and any occurrence ω of t, there exists a finite combination of the SIR, the SF and the UG operators that generates these two equations:*
a first equation D'_k as: $s = q'$, and a second equation E'_k as: $q'_{|\omega} = t_{|\omega}$, where $q' = t[r']_\omega$.

Proof. Given the equation $s = t$ and any term r', consider a new term r such that any variable in r' is substituted by a function symbol of arity 0. Obviously, this r is ground, and, by lemma 1 it can be constructed by a finite combination of the operators SIR and SF, resulting in a first equation D_k as $s = q$, and a second equation E_k as $q_{|\omega} = t_{|\omega}$, where $q = t[r]_\omega$.
Take D_k and use a UG to obtain a new equation $s = t[r']_\omega$ which is equal to $s = q'$. In the same way all the E_k can be generalised to obtain a new equation $r' = t_{|\omega}$. □

5 Strong Completeness of the IFLP Schema

Theorem 1 is essential to be able to show that any possible intermediate term that may be used in a derivation can be induced by using the operators of the IFLP Schema. This leads to the following strong completeness result:

Theorem 2. Strong Completeness
Given a finite program P, and a finite evidence E generated from P, such that every rule of P is necessary for at least one equation of the positive evidence (i.e. if removed some positive example is not covered), and $\Sigma_P = \Sigma^+$, i.e., all function symbols of the program appear in the positive evidence, then the program can be induced by a finite combination of the operators presented in the IFLP schema, that is to say, Unrestricted Generalitation (UG), Single Intra-Replacement (SIR) and Syntactic Folding (SF).

Proof. Select any rule $r \equiv \{s = t\}$ from P. Since it is necessary, it is used in at least one derivation of one example, say $a_0 = a_n$. We express this derivation as:

$$a_0 \overset{u_1, l_1 = r_1, \theta_1}{\hookrightarrow} a_1 \overset{u_2, l_2 = r_2, \theta_2}{\hookrightarrow} a_2 \hookrightarrow \ldots \overset{u_n, l_n = r_n, \theta_n}{\hookrightarrow} a_n$$

If $n = 1$, i.e. the derivation $a_0 \overset{u_1, l_1 = r_1, \theta_1}{\hookrightarrow} a_1$ then we have that under the IFLP schema we can generate a first equation D_k as: $a_0 = a_0$, and a second equation E_k as: $(a_0)_{|\omega} = (a_1)_{|\omega}$, such that $\omega = u_1$, and what has to be introduced is $q = (a_0)_{|\omega}$. This can be done as was shown in Theorem 1. The last equation E_k can be generalised in order to match $l_1 = r_1$.

Let us assume the hypothesis that we have been able to generate all the $l_i = r_i$ upto $n - 1$. Then, for n we have :

$$a_0 \overset{u_1, l_1 = r_1, \theta_1}{\hookrightarrow} a_1 \overset{u_2, l_2 = r_2, \theta_2}{\hookrightarrow} a_2 \hookrightarrow \ldots \overset{u_n, l_n = r_n, \theta_n}{\hookrightarrow} a_n$$

Since it has been generated to a_{n-1} we only have to show that it is possible to generate the equation that allows for narrowing from a_{n-1} to a_n, i.e. $a_{n-1} \overset{u_n, l_n = r_n, \theta_n}{\hookrightarrow} a_n$. However, this step is no different from the step we proved for $n = 1$, so we can find this $l_n = r_n$ and the hypothesis is true for all n. Thus, the theorem is proven. □

Strong Completeness is not usual in the inductive literature (except [12]), because, without additional information (e.g. modes) it entails intractability. However, the previous theorem discovers a set of operators which are sufficient to induce *any possible* program. Further work is centred on finding restrictions which preserve completeness or bring the schema to tractability. Among the latter there are at least two ways possible. A first option is to fix a selection criterion (e.g. compression) ensuring completeness wrt this criterion by using an ordered search space and mode declarations (e.g. [11]). A second one is to study uncomplete but still powerful instances of the schema and provide efficient algorithms for them. The first option is in progress by the authors through the use of genetic programming as in [17]. The second option was precisely undertaken in [3] and the next section discusses its relation to the preceding schema.

6 Inverse Narrowing as an Instance of the IFLP Schema

In this section, we show that Inverse Narrowing is just an instance of our generic IFLP Schema. This relationship allows a more detailed study of our previous algorithm, its limitations and its extensions to cover more difficult cases without falling into intractability.

First of all, it is evident that, according to the definition given in Section 3, CRG is just a restriction of the UG. Secondly, Inverse narrowing was defined as an operator that generates an equation from two equations. On the contrary, SIR generates two equations from one equation. This operator, iterated and combined with the other operators of the IFLP Schema is a generalisation of inverse narrowing as the following corollary shows:

Corollary 1. *For any equation $s = t$ such that we can make an inverse narrowing step: $t \overset{u, l = r, \theta}{\hookleftarrow}_P t'$ to obtain a pair of equations $s = t'$ and $l = r$, then these two equations can be obtained in the IFLP schema.*

Proof. Just apply the operators of the IFLP schema to obtain a first equation D_k as: $s = q$, and a second equation E_k as: $q_{|\omega} = t_{|\omega}$, such that $\omega = u$, where $q = t[l]_\omega$ and

$t_{|\omega} = r$.
The only difference is that $t' = \theta(q)$, i.e., a substitution is applied, but this difference vanishes if we select $q = \theta(t[l]_\omega)$ and then we generalise the second equation E_k to make it match $l = r$. □

7 Extending Inverse Narrowing

As we have stated before, the IFLP Schema should suggest different ways to generalise inverse narrowing to cope with more complex cases. In this work, it has been shown that if a function symbol did not appear in some convenient conditions, it could not be induced by inverse narrowing. In this way, we can extend inverse narrowing to allow fresh variables on the rhs's and where the secondary equation can be obtained either from a set of generalised equations from the evidence or by the introduction of a new term function symbol $F(X'_1, X'_2, \ldots, X'_m)$ into an equation of the form $F(X'_1, X'_2, \ldots, X'_m) = Y$, which obviously can be used in any occurrence of the other equation since Y unifies with anything.

7.1 Function Invention

The invention of predicates is an open area of research in ILP [13, 16, 10, 7]. In the case of unconditional functional logic programs it is expected that function invention would be even more necessary than First Order Horn Logic [16].

In our strong completeness theorem, we assumed that $\Sigma_P = \Sigma^+$, i.e., all function symbols of the program appear in the positive evidence.

One of the reasons for the introduction of this general schema is that in the case where the relation $\Sigma_P \supset \Sigma^+$ is strict, we can extend Σ^+ with new and fresh function symbols of different arities, thus making the invention of new functions possible. The set of *inventable* functions is denoted by Σ^i, and the SIR operator can then construct terms by using function symbols from $\Sigma^+ \cup \Sigma^i$.

Under the extension of the signature, it is clear that the IFLP Schema is able to invent functions. The procedure resembles the approach presented in [10], where maximal utilisation predicates are introduced and then refined. In our case, they are refined by the possible introduction of function symbols in different occurrences at different stages.

On the contrary, in order to extend our previous inverse narrowing approach [3] with function invention, we are forced to act in the reverse way due to the nature of this procedure. Our extended inverse narrowing is able to do inventions of this kind but adding equations of the form $F(a, a, \ldots, a) = Y$ where F is a new function symbol from Σ^i and a is a constant which appears in Σ^+. These equations can be used as secondary equations in an inverse narrowing step. The use of inverse narrowing on the lhs is also required (this was already done when learning from background knowledge in [3]). Therefore, the approach becomes too general for practical purposes, as the following example illustrates.

Example 4. Let us consider the example of inducing the product function from scratch, which requires the invention of a function for addition. To do this, make $\Sigma^i = \{+\}$ where $+$ has arity 2. Given the following evidence:

(E_1^+) $ss0 \times ss0 = ssss0$
(E_2^+) $sss0 \times ss0 = ssssss0$
(E_3^+) $sss0 \times s0 = sss0$
(E_4^+) $0 \times sss0 = 0$
(E_5^+) $ss0 \times 0 = 0$
(E_6^+) $ssss0 \times 0 = 0$
(E_7^+) $s0 \times ss0 = ss0$

(E_1^-) $ss0 \times sss0 = sssssssss0$
(E_2^-) $sss0 \times sss0 = ssssssss0$
(E_3^-) $ss0 \times sss0 = ssss0$
(E_4^-) $sss0 \uparrow ss0 = ssss0$
(E_5^-) $0 \times s0 = s0$
(E_6^-) $s0 \times 0 = ss0$

We can proceed as follows. The equation $\{0 \times X = 0\}$ is just a generalisation of E_4^+. From $\Sigma^i = \{+\}$, we introduce the equation $+(0,0) = Y$, which can be expressed in infix notation as $E_1 \equiv 0+0 = Y$. From Σ^+ we introduce the equations $E_2 \equiv s(0) = Y'$ and $E_3 \equiv 0 = Y''$. We make inverse narrowing at occurrence ϵ of the rhs of E_1 with E_3 and we have $E_4 \equiv 0+0 = 0$ that can be generalised into $X+0 = X$. By repeatedly using inverse narrowing on different occurrences we can obtain the following equation $X + s(Y) = s(X+Y)$. Although, in this case, this involves only three steps, in general it would be necessary to use heuristics or mode declarations. Even with all this, some systems (e.g. [11]) are helped by some examples of the addition in the evidence. The equation $sX + Y = X \times Y$ can be obtained as was shown in [3] since the equations of addition are already generated.

At the end of the process, the following program can be constructed: $P = \{0 \times X = 0, sX \times Y = X \times Y + Y, X + 0 = X, X + s(Y) = s(X+Y)\}$.

8 Conclusions and Future Work

The IFLP Schema is shown to be a general and strong complete framework for the induction of functional logic programs. Theoretically, this allows the induction of functional logic programs with auxiliary functions and, if the signature is conveniently extended, it can be used to invent functions. Moreover, although intricate combinations of the operators which have been presented may be needed in order to obtain the rules of the intended program, function symbols are introduced one by one. This makes extending our previous algorithm with the new operators possible, since it is based on genetic programming techniques.

This theoretical work is a necessary stage in a more long-term project to explore the advantages of extending the representational language of ILP to functional logic programs. This is also subject to a convenient extension of our schema to conditional theories, which will make it possible to use and compare the same examples and background knowledge as in ILP problems. A less theoretical ongoing work is centred on the development and implementation of a more powerful but still tractable algorithm than the one presented in [3].

References

1. P.G. Bosco, E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. A complete semantic characterization of K-leaf, a logic language with partial functions. In

Proceedings of the IEEE Symposium on Logic Programming, pages 318–327. IEEE Computer Society Press, N.W., Washington, 1987.
2. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19-20:583–628, 1994.
3. J. Hernández and M.J. Ramírez. Inverse Narrowing for the Induction of Functional Logic Programs. In *Proc. Joint Conference on Declarative Programming, APPIA-GULP-PRODE'98*, pages 379–393, 1998.
4. S. Hölldobler. Equational Logic Programming. In *Proc. Second IEEE Symp. on Logic In Computer Science*, pages 335–346. IEEE Computer Society Press, 1987.
5. H. Hussmann. Unification in conditional-equational theories. Technical report, Fakultät für Mathematik und Informatik, Universität Passau, 1986.
6. J. Jaffar, J.-L. Lassez, and M.J. Maher. A logic programming language scheme. In D. de Groot and G. Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 441–468. Prentice Hall, Englewood Cliffs, NJ, 1986.
7. K. Khan, S. Muggleton, and R. Parson. Repeat learning using predicate invention. In C.D. Page, editor, *Proc. of the 8th International Workshop on Inductive Logic Programming, ILP'98*, volume 1446 of *Lecture Notes in Artificial Intelligence*, pages 165–174. Springer-Verlag, Berlin, 1998.
8. J.W. Klop. Term Rewriting Systems. *Handbook of Logic in Computer Science*, I:1–112, 1992.
9. S. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(4):295–318, 1991.
10. S. Muggleton. Predicate invention and utilisation. *Journal of Experimental and Theoretical Artificial Intelligence*, 6(1):127–130, 1994.
11. S. Muggleton. Inverse entailment and progol. *New Generation Computing Journal*, 13:245–286, 1995.
12. S. Muggleton. Comnpleting inverse entailment. In C.D. Page, editor, *Proc. of the 8th International Workshop on Inductive Logic Programming, ILP'98*, volume 1446 of *Lecture Notes in Artificial Intelligence*, pages 245–249. Springer-Verlag, Berlin, 1998.
13. S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In S. Muggleton, editor, *Inductive Logic Programming*, pages 261–280. Academic Press, 1992.
14. U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. Second IEEE Int'l Symp. on Logic Programming*, pages 138–151. IEEE, 1985.
15. J.H. Siekmann. Universal unification. In *7th Int'l Conf. on Automated Deduction*, volume 170 of *Lecture Notes in Computer Science*, pages 1–42. Springer-Verlag, Berlin, 1984.
16. I. Stahl. The Appropriateness of Predicate Invention as Bias Shift Operation in ILP. *Machine Learning*, 20:95–117, 1995.
17. A. Varsek. *Genetic Inductive Logic Programming.* PhD thesis, University of Ljubljana, Slovenia, 1993.