FLIP:User's Manual (v0.7)

C. Ferri-Ramrez, J. Hernndez-Orallo, M.J. Ramrez-Quintana, *†

April 25, 2000

Abstract

We present the FLIP system, an application that implements a framework for the Induction of Functional Logic Programs (IFLP) from facts. IFLP can be seen as an extension to the now consolidated field of Inductive Logic Programming (ILP) to induce equational theories. Some results and benchmarks obtained during the experimentation of the induction system are described.

1 Introduction

This report describes the FLIP system, an implementation of the IFLP framework presented in [5]. IFLP can be defined as the functional (or equational) extension of ILP. Since the beginning of last decade, Inductive Logic Programming (ILP) has been a very important area of research as an appropriate framework for the inductive inference of first-order clausal theories from facts. As a machine learning paradigm, the general aim of ILP is to develop tools, theories and techniques to induce hypotheses from examples and background knowledge. ILP inherits the representational formalism, the semantical orientation and, the well-established techniques of logic programming.

Over the last decade, it has been theoretically and experimentally demonstrated that *functional logic languages have more expressive power in comparison to functional languages and a better operational behavior in comparison to logic languages* [3]. One relevant approach [6] to integration is the functional logic programming where the programs are logic programs which are augmented with Horn equational theories.

In [4] we presented a general framework for the induction of functional logic programs (IFLP) from examples, generalizing the scope of the ILP where we only consider the unconditional case. For simplicity, the (positive and negative) examples are expressed as pairs of ground terms or ground equations where the right term is in normal form. Positive examples represent terms that will have to

*DSIC, UPV, Camino de Vera s/n, 46022 Valencia, Spain. E-mail: {cferri,jorallo,mramirez}@dsic.upv.es.

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

be proven equal using the induced program, whereas negative examples consist of terms that do not have to be proven equal. The approach is based on the idea of the inverse resolution of ILP. Starting from the generalization of positive examples to include variables as arguments of functions, we defined an inverse narrowing mechanism which selects pairs of equations to obtain an equation which is more general than the original ones from the generalized examples. The selection criteria is based on the number of positive examples that can be derived from the equation. Then, if the generated equation covers more positive examples or subsumes other equations, then it is added to the previously selected equations (probably replacing some of them). This process is repeated until a program (or set of equations) is valid according to some selection criteria.

2 The IFLP framework

IFLP can be defined as the functional (or equational) extension of ILP[1]. The goal is the inference of a theory (a functional logic program P) from evidence (a set of positive and optionally negative equations E).

The algorithm starts by generating a set of initial programs. This set is built by the generalization of the rules of the positive examples e_i^+. This generalization process is limited by a number of restrictions that eliminates many rules that will be inconsistent with the examples or useless for the induction process. We name this limited generalization as Consistent Restricted Generalization or CRG. A formal description of the CRG can be found in [4].

From this initial set of simple programs, the algorithm selects a pair of them. The choice is done matching one to one each initial program and selecting the pair that covers more positive examples with the minimum length of the rules.

On the selected pair of rules a step of inverse narrowing is applied. The mechanism allows to generate new programs starting by two different rules. A detailed description of the method is illustrated in [4]. The new programs produced by the inverse narrowing application are added to the initial set of programs. The main loop of the algorithm starts with this set of programs.

This loop consists of the selection of the pair of rules where apply the inverse narrowing method is to be applied for generating news programs. The loop finishes when a program that covers all the positive examples (without covering any negative example) is found, or when the loop has reached a fixed maximum number of steps.

Hence, the algorithm exits when a program is induced such that accepts all positive examples and that fails with every negative example. Otherwise it returns the best program that has been generated when the loop has reached the limit number of steps.

We refer to [1] to a complete description of the IFLP framework.

[1]It is obvious that any problem expressed in the ILP framework can also be expressed in the IFLP framework, because all the positive facts e_i^+ of an ILP problem can be converted into equations of the form $e_i^+ = true$ and all the negative facts e_j^- can be expressed as $e_j^- = false$.

2.1 Learning from a background knowledge

We use two different methods for the induction of programs using background theory. Both of them permit the introduction of function symbols from background knowledge into the program that is being induced. Briefly the two methods are:

2.1.1 Inverse narrowing with the Background theory

The first of these methods consists of the inverse narrowing operator with a rule of the background theory. The rules from the background theory that are introduced in the induction process are the base cases of the background theory because these have simple expressions in the right hand side. In this way it is possible to obtain new equations with the background function in their right hand side.

2.1.2 Including the background theory in the examples

The second method uses a very different approach. It mainly consists in expressing the right hand side of some examples with some functions of the background theory. For instance, $prod(X, 0) = 0$, is completely equivalent to $prod(X, 0) = sum(0, 0)$, but in the last one, we have introduced a function of the background theory in the induction process.

3 The FLIP system

The FLIP system is a project built in C, with more than 5000 lines of code that implements the Inductive Functional Logic Programming framework. The system includes a menu user interface, a simple parser, a narrowing solver, an inverse narrowing method, and a CRG generator.

The system uses the design and implementation of functional logic programming defined by Alpuente, Julian and Moreno in [7].

In [1] there is a complete description of the implementation of FLIP.

The FLIP system works with two sets of facts: the positives examples and the negatives ones, respectively and optionally background knowledge. Starting from these examples the system applies the IFLP algorithm until a solution program is discovered or it has overpassed a limit number of steps of the main loop of the induction process.

3.1 Program options

The files with the examples and theory are passed to FLIP by means of arguments and options when it is called. If you execute FLIP with the option $-?$, it prints out:

```
[cesar@curret]$ flip -?

######################### {\sf FLIP} System (v0.7) #########################
```

```
usage: ./flip [-t] [-b background_file] [-u] [-e] [-n] [-f] [-p depth]
[-s steps] positive_examples_file [negative_examples_file]
    -t          Switches off the equation termination filter.
    -b file     Uses the specified file as the background knowledge.
    -u          Uses the union operator in the first step.
    -e          Introduces the Background functions in the examples.
    -n          Inverse narrowing with full program clauses combination.
    -f          Stops immediately when a program is found that covers all the examples.
    -p depth    Depth limit of the narrowing solver(default 19).
    -s steps    Maximum number of steps of the main induction loop(default 10).
```

Here we comment the purpose of each option:

- **-t** *Termination filter*: FLIP provides a simple termination filter that permit to reject some non-terminating equations. There are cases where it is interesting to generate programs with non-terminating equations, for instance $minus$ or max, see [2] for details.
- **-b** *Background Knowledge*: The user specifies to FLIP a file with a background theory that will be used in the induction process.
- **-u** *Union operator*: FLIP can use, if the user wants, the union operator in the first step of the process. As we argued in [1] the use of the union operator is not suitable in some cases, because it generates long programs with patches. Consequently, this option should be restricted.
- **-e** *Background Introduction* : The system, by default, introduces the background functions with inverse narrowing, nevertheless the user can indicate to insert these functions in the examples, see 2.1.
- **-n** *Inverse narrowing* : The inverse narrowing operator is applied by default between the two best equations of the combined programs. With this option the user indicates FLIP to apply inverse narrowing between every clauses of the pair of programs (full program clauses combination).
- **-f** *Stop immediately* : We have found problems with some non-terminating equations, because the filter we have implemented is quite naive (the termination problem is far from being simple). When this option is enabled, FLIP immediately exits whenever it finds a solution program. In this way we avoid the possibility of finding a non-terminating equation before the step is completed.
- **-p** *Narrowing depth* : The system uses a narrowing solver that perfoms a depth search. Thus, to eliminate possible infinite loops in non-terminating programs a depth limit is fixed, by default it is 19.
- **-s** *main loop limit*: This option sets a maximum number of steps of the main loop of the induction process, by default the limit is 10.

4 Examples of using FLIP

In this section we show how FLIP solves a simple problem, concretely the sum of two naturals:

The target program is:

$$\begin{aligned} sum(X,0) &= X \\ sum(s(X),Y) &= s(sum(X,Y)) \end{aligned}$$

where $0, s(0), s(s(0))...$ represent the infinite set of natural numbers. Consider we generate the following file *sum.pex* with positives examples of this problem:

$$\begin{aligned}
sum(0,0) &= 0, \\
sum(s(0),s(0)) &= s(s(0)), \\
sum(0,s(0)) &= s(0), \\
sum(s(s(0)),0) &= s(s(0)), \\
sum(s(0),0) &= s(0), \\
sum(s(0),s(s(0))) &= s(s(s(0))), \\
sum(s(s(0)),s(s(0))) &= s(s(s(s(0)))), \\
sum(s(0),0) &= s(0), \\
sum(s(s(s(0))),s(0)) &= s(s(s(s(0)))), \\
sum(s(s(s(0))),s(s(0))) &= s(s(s(s(s(0)))))
\end{aligned}$$

And the following file *sum.nex* with negative ones:

$$\begin{aligned}
sum(s(0),0) &= 0, \\
sum(0,0) &= s(0), \\
sum(s(0),s(0)) &= s(0), \\
sum(s(0),0) &= s(s(0)), \\
sum(0,s(0)) &= s(s(0))
\end{aligned}$$

Note that the syntax of the files are very simple, just equations separated by commas, without any other type of information: types, mode... So the next step is just executing FLIP:

```
[cesar@curret]$ flip sum.pex sum.nex
```

The program prints in the standard output some information of inducing process. First it shows a line that notifies the user the generation of CRG equations. When this phase is finished, every equation generated is shown together with its coverage:

```
######################### FLIP System (v0.7) ##########################
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
@@@@@@@@@@@ CRG process @@@@@@@@@@@
Initial CRG Equations
0 sum(0,0) = 0
1 sum(0,X0) = X0
2 sum(X0,0) = X0
.
.
55 sum(s(s(X0))),s(s(X0))) = s(s(s(s(X0)))))
56 sum(s(s(X0)),s(X0)) = s(s(s(s(0)))))
57 sum(s(s(X0)),s(X0)) = s(s(s(X0))))

coverage of program 0: 1 positives, map 1
coverage of program 1: 2 positives, map 5
coverage of program 2: 4 positives, map 153
```

When all the programs have been shown, FLIP chooses the best two, according to the IFLP criterion, and performs inverse narrowing between them:

```
1 Best program: number 2, 4 positives, map 153
2 Best program: number 7, 3 positives, map 262
 %%%%%%% Inverse Narrowing %%%%%%%%

 Inverse Narrowing between:
sum(X0,0) = X0
sum(X0,s(0)) = s(X0)

 Resulting equation:
sum(X0,s(0)) = sum(s(X0),0)

 Analyzed equation:
sum(X0,s(0)) = sum(s(X0),0)
```

The new equations provided by the inverse narrowing process are generalized by the CRG operator:

```
@@@@@@@@@@@@ CRG process @@@@@@@@@@@
 CRG Equations generated:
0 sum(X0,s(0)) = sum(s(X0),0)
1 sum(X0,s(X1)) = sum(s(X0),0)
2 sum(X0,s(X1)) = sum(s(X0),X1)
3 sum(X0,s(0)) = s(sum(X0,0))
4 sum(X0,s(X1)) = s(sum(X0,0))
5 sum(X0,s(X1)) = s(sum(X0,X1))
```

Every new equation forms a program, whose coverage is shown. At the end of the step it is checked whether the best program satisfies the requerimets of the target program, if it covers them, the process has ended anf FLIP prints the program. First the original one is shown, and then, a filtered version without redundant equations:

```
Best program:
0 sum(X0,s(X1)) = sum(s(X0),X1)
1 sum(X0,0) = X0

 &&&&&&&&&&&&&&&& Solution program &&&&&&&&&&&&&
0 sum(X0,s(X1)) = sum(s(X0),X1)
1 sum(X0,0) = X0
 Number of examples covered: 10, optimality 13.459432

 &&&&&&&&&&&&&&&& Filtered solution program &&&&&&&&&&&&&
0 sum(X0,s(X1)) = sum(s(X0),X1)
1 sum(X0,0) = X0
 Number of examples covered: 10, optimality 13.459432
```

5 Results

We have tested our system with several examples of simple programs. The algorithm usually finds the solution in a few steps of the loop, but it depends mainly on the number of rules of the solution program, and on the quality of the initial examples.

Two sets of limited and very significant examples permits the system to generate more general solutions and more quickly. So, the initial selection of the examples that could be supplied to the system is very important.

Some relevant functional logic programs induced can be seen in Table 1. Non-recursive functional logic programs which were derived are shown in Table 2. Finally, more complex problems can be generated with the use of background knowledge, as illustrated in Table 3. More extensive and updated information can be found in [2].

Induced program	Description	Steps
sum(s(X),Y) = s(sum(X,Y)) sum(0,Y)=Y	Sum of two natural numbers	1
length(p(X,Y)) = s(length(X)) length(λ)=0	Length of a list	2
consec(p(X,Y)) = consec(X) consec(p(p(X,Y),Y)) = true	Returns true if the list has two consecutive elements	1
drop(0,X) = X drop(s(X),p(Y,Z)) = drop(X,Y)	Drops the N last elements of a list	1
app(p(X,Y),Z) = p(app(X,Z),Y) app(λ,X) = X	Appends two lists	1
member(p(X,Y),Z) = member(X,Z) member(p(X,Y),Y) = true	Returns true if Z is in a list	1
last(p(X,Y)) = last(X) last(p(λ,X)) = X	Last element of a list	1
geq(s(X),s(Y)) = geq(X,Y) geq(X,0) = true	Equal or greater function	1
sum(s(X),Y) = s(sum(X,Y)) sum(0,Y)=Y prod(s(X0),X1) = sum(prod(X0,X1),X1) prod(0,X0) = 0	Addition and Multiplication at the same time	3
mod3(0) = 0 mod3(s(0)) = s(mod3(0)) mod3(s(s(0))) = s(s(mod3(0))) mod3(s(s(s(X0)))) = mod3(X0)	The mod 3 operation	3
even(s(s(X))) = even(X) even(0) = true	Returns true if a natural number is even	1

Table 1: Some recursive programs induced without background knowledge

References

[1] C. Ferri, J. Hernández, and M.J. Ramírez. The flip system: From theory to implementation. *submitted to Machine Learning*, 2000.

[2] C. Ferri, J. Hernández, and M.J. Ramírez. The FLIP system homepage. `http://www.dsic.upv.es/~jorallo/flip/`, 2000.

[3] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19-20:583–628, 1994.

[4] 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.

[5] J. Hernández and M.J. Ramírez. A Strong Complete Schema for Inductive Functional Logic Programming. In *Proc. of the Ninth International Workshop on Inductive Logic Programming, ILP'99*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 116–127, 1999.

[6] S. Hölldobler. Equational Logic Programming. In *Proc. Second IEEE Symp. on Logic In Computer Science*, pages 335–346. IEEE Computer Society Press, 1987.

[7] P. Julian and G. Moreno. Nuevas Semanticas para Programas Logicos Funcionales. Technical report, Technical report, Department of Information Systems and Computation, Valencia University of Technology, 1987.

Induced program	Description
cup(y,up,X1,X2,n,y,X3)=true, cup(y,up,X1,X2,side,y,X3)=true	The Cup learning task where the arguments represent: BottomIsFlat, Concavity(up,not-up,no), Expensive, HandleOnTop,Fragile, Handle (n, top, side), Light, MadeOf(ceramic,paper,styrofoam)
enjoysport(sunny,X,Y,Z,W,V)=yes enjoysport(cloudy,X,Y,Z,W,V)=yes	The EnjoySport learning task where the arguments represent: Sky, AirTemp, Humidity, Wind, Water, Forecast
playtennis(overcast,X1,X2,X3)=yes playtennis(sunny,X1,normal,X3)=yes playtennis(rain,X1,X2,weak)=yes	The playtennis learning task where the arguments represent: Outlook, Temperature, Humidity, Wind
top(push(X,Y))=Y	Top element of a stack
customer(p(p(X0,X1),X2))=customer(p(X0,X1)) customer(p(X0,children))=true customer(p(X0,woman))=true customer(p(X0,teacher))=true	Returns true if the atributes define a good customer

Table 2: Some non-recursive programs induced without inverse narrowing

Induced program	Description	B.K.	Steps
rev(p(X0,X1)) = app(rev(X0),p(λ,X1)) rev(λ) = λ	Reversal of a list	append	2
suml(p(X0,X1)) = sum(suml(X0),X1) suml(p(λ,X0)) = X0	Sum of a list of natural numbers	sum	1
maxl(p(X0,X1)) = max(X1,maxl(X0)) maxl(p(λ,X0)) = X0	Max of a list of natural numbers	max	1
prod(s(X0),X1) = sum(prod(X0,X1),X1) prod(0,X0) = 0	Product of two natural numbers	sum	1
fact(s(X0)) = prod(fact(X0),s(X0)) fact(0) = s(0)	Factorial function	sum, prod	4
sort(p(X0,X1)) = inssort(X1,sort(X0)) sort(λ) = λ	Inefficient sort of a list	inssort, gt, if	1

Table 3: Some programs induced with background knowledge