

Constructive Inductive Logic Programming

Stéphane Lapointe

Defence Research Establishment
Valcartier

2459 Blvd. Pie XI north
POB 8800, Courcellette (Québec)
Canada, G0A 1R0
lapointe@drev.dnd.ca

Charles Ling

Department of Computer Science
University of Western Ontario

London (Ontario)
Canada, N6A 5B7
ling@csd.uwo.ca

Stan Matwin

Ottawa Machine Learning Group
Department of Computer Science

University of Ottawa
Ottawa (Ontario)
Canada, K1N 6N5
stan@csi.uottawa.ca

Abstract*

Constructive learning, the capacity to extend a fixed initial representation by adding new non-observational terms, is an active research topic in machine learning. We show how the very notion of constructiveness has evolved, and define with some precision the sort of learning which should be considered constructive in the context of Inductive Logic Programming. We begin by investigating the differing extent to which two classes of new terms, useful or necessary predicates, enrich a representation. We then characterize two situations, partial utilization and zero utilization of a new necessary predicate, in which learning methods based on inverse resolution lack a sufficiently representative set of instances to allow a concept to be learned. In the absence of a more representative set we conclude that efforts must move from inverse resolution towards a more powerful technique, that is, inverse implication. The notion of *sub-unification* allows us to invert implication with a small number of examples. These ideas are expressed in a Constructive Inductive Logic Programming algorithm CILP, based on inverting implication. CILP learns some natural classes of Horn clauses with partially- and zero-utilized terms not learnable by methods based on inverse resolution.

1 Introduction

There are three good reasons for the recent interest in Inductive Logic Programming (ILP) (deRaedt [1992], Muggleton [1991a] and Furukawa [Muggleton and Furukawa, 1992]). First, ILP offers a good solution to a useful task—learning intensional descriptions of relations from examples and relevant background knowledge. This task in itself, however, has been addressed in the learning community before the advent of ILP [Shapiro, 1981; Hoff et al., 1982].

Second, ILP has greater representational power than learners based on propositional attribute-value representations, particularly when a domain theory is being learned from a set of examples. However in such cases the initial language used by the learner may be inadequate to define the finite set of Horn clauses representing the theory. It is then essential to invent new predicates which extend the language in order to learn the theory. Certain new ILP systems [Quinlan, 1990; Muggleton and Feng, 1990] use a fixed language; they cannot invent new predicates and may fail to learn the theory expected from a set of examples. In general, however, ILP offers potential for predicate invention, allowing learnability of theories which cannot be expressed in the fixed initial language.

Third, ILP is based on solid formal foundations. Its logical framework supports the posing and addressing of interesting questions, i.e. the issue of the completeness of different generalization transformations with respect to induction [Rouveirol, 1991], or the problem of determining the relative power of different ILP methods [Ling, 1992; Lapointe and Matwin, 1992]. This paper investigates such a question: a precise characteristic of the notion of constructive learning in the ILP context.

One must go all the way back to 1983 to find a definition of constructive learning in the literature: "Constructive generalization rules generate inductive assertions that use descriptors not present in the original observational statements" [Michalski, 1983]. This idea, implemented in the author's AQ system with so-called L-rules, draws on background knowledge to provide descriptor predicates not present in the examples. CIGOL [Muggleton and Buntine, 1988] used the same approach called *absorption* in the inverse resolution context. Where does the required background knowledge come from? We take the view that a constructive learner's concept descriptions should use terms which are neither in the observational statements, nor in the background knowledge, rather than simply transfer parts of the background knowledge to the concept description.

This direction was first taken by Muggleton [1987] with the introduction in DUCS of *intra-construction* (a special case of the 'W'-operator). Constructiveness is achieved here by converting one set of rules into another composed of more compact rules in which re-occurring conjunctions of literals are replaced by a single predicate denoting the concept represented by the conjunction. This invented predicate is the new, non-observational term.

*This work is supported by grants from the Natural Sciences and Engineering Research Council of Canada.

Meaningful invented predicates make domain theory more compact yet more comprehensible. These constructs are not necessary, however, because they can be replaced by the conjunction of literals that define them. Only recursive predicates are necessary, as they cannot be eliminated in this manner.

The ability to learn recursively-defined predicates is essential for truly constructive ILP. For instance, we want to be able to learn the definition of multiply from the following set of examples¹:

```
multiply(0,A,0)
multiply(1,B,B)
multiply(3,4,12)
```

knowing only the definition of successor. Any finite definition of multiply involves learning the invented predicate plus from these examples. None of the systems mentioned earlier can invent the recursive definition of plus from the above set of examples because it does not constitute a *representative set*² of instances of plus. For propositional logic, *characteristic sample*, introduced by Muggleton [1991b], is equivalent to basic representative set. We show below that on many occasions representative example sets will not be available, and that this situation is in fact typical when necessary recursively defined predicates must be invented. Consequently new approaches to ILP are needed if true constructiveness—the ability to learn definitions of new necessary predicates from examples—is to be achieved.

We introduce the concept of a *generating term* based on *sub-unification* that allows us to move from generalization based on inverse resolution (IR) to generalization based on inverting implication (II) [Lapointe and Matwin, 1992; Muggleton, 1992]. We show how an II system achieves constructiveness, and describe the applicability and limitations of our view of Constructive Inductive Logic Programming (CILP). Finally, we illustrate the discussion with examples showing how the learning task is achieved.

2 Constructiveness in ILP

2.1 Useful and Necessary New Terms

A criterion of learnability divides new terms into the two categories of useful or necessary. We call a new term *useful* if its invention does not affect the learnability of the theory. Useful new terms can be eliminated from a theory without affecting its learnability. A new term is (*functionally*) *necessary* if a theory is not learnable without it according to the learnability criterion. The learnability criterion can be either finite learning/axiomatization or pac-learnability [Valiant, 1984]. This paper uses a finite learning criterion, that is, the theory is definable by a finite number of clauses. A new predicate's *definition* or *hypotheses* is the set of definite clauses which have it in their head. A necessary new predicate has at least one of these clauses directly defined

recursively; that is, the body (antecedent) of the clause contains the same predicate which appears in its head (consequent). Necessary new predicates must meet this condition because new predicates not defined in this way can be eliminated without affecting the finite learnability result.

2.2 Partial Utilization Problem

Lack of sufficient data is a common problem in learning hypotheses of theoretical terms [Ling, 1992]. Often the operator 'W' creates insufficient instances of a new term for IR to construct its desired definition. Following Buntine [1989] we call such new predicates *partially-utilized*. In general the fewer instances of a new predicate, the more difficult it is to infer hypotheses of it. In the worst case it may not be possible for IR to infer the definitions of a new predicate at all.

For a given set of hypotheses defining the new predicate newp, the *least model* of newp is the set of all instances of it that can be derived from the definition of newp. True instantiation of hypothesized clauses of an observable predicate containing newp in their body produces a set of newp instances which must also be true. This set of instances is defined as the *projected set* of newp. Consider the example of learning multiplication through invention of a new term:

```
mul(X,s(Y),Z) <-
    mul(X,Y,W),newp1(X,W,Z).
newp1(X,0,X).
newp1(X,s(W),s(Z)) <- newp1(X,W,Z).
```

The least model of newp1 is the set of all relations of plus, i.e. $\{(x,y,x+y) \mid x,y \text{ are integers}\}$. The projected set of newp1, i.e. $\{(x,xy,x+xy) \mid x,y \text{ are integers}\}$, is a proper subset of the least model; $\text{plus}(s^2,s^5,s^7)$ does not appear in the projected set³.

A new predicate newp is *fully utilized* in a theory if its projected set equals its least model. If new predicates created by 'W' are fully utilized, all examples of the new terms are available to IR, which can then infer the definition of the new term easily. This is not often the case.

A new predicate newp is *partially utilized* for any finite set of hypotheses of newp when all projected sets of newp are proper subsets of the least model of newp. Ling [1991] established that a *basic representative set* provides the minimum set of examples for successful inference using inverse resolution. A basic representative set (BRS) of a clause is a set of ground atoms which appear in some true instantiation of the clause. For example, the two sets $\{\text{plus}(s^1,s^2,s^3), \text{plus}(s^1,s^1,s^2)\}$ and $\{\text{plus}(s^3,s^1,s^4), \text{plus}(s^3,s^0,s^3)\}$ are BRS for the clause $\text{plus}(X,s(W),s(Z)) \leftarrow \text{plus}(X,W,Z)$. If a BRS is included in the projected set of newp created by the 'W'-operator, the 'V'-operator can construct the new term's clause if it is applied to the examples in the BRS. However these examples are not distinguishable from others in the projected set, so an excessive number of faulty clauses

¹Throughout the paper, names starting with capital letters denote variables, while other names denote constants.
²We define in the next section what we mean by a representative set of examples for learning recursively defined predicates.

³In sequel, s^e , where e is an expression, denotes an integer constant equal to the value of e . $s^e(X)$ denotes an integer equal to the value of $e+X$.

may be inferred. Inverse implication, discussed in the next section, alleviates this problem.

Hypotheses for a partially utilized new predicate cannot be constructed using IR if all "important" data are missing. More specifically, if no basic representative set of any possible target hypothesis of a new predicate is included in its projected set, no hypotheses can be inferred for the new predicate. In this case the new predicate is *zero utilized*. Consider the example of learning the exponential function $f(X) = 2^X$, $X \geq 1$ defined as:

$f(s(0), s^2)$.
 $f(s(X), Z) \Leftarrow f(X, W), \text{double}(W, Z)$.
 $\text{double}(s^2, s^4)$.
 $\text{double}(s(X), s(s(Z))) \Leftarrow \text{double}(X, Z)$.

The new term *double* is zero-utilized here because the projected set of *double* contains only $\text{double}(s^2, s^4)$, $\text{double}(s^4, s^8)$, $\text{double}(s^8, s^{16})$, $\text{double}(s^{16}, s^{32})$ and so on. Clearly this definition of *double* is not constructable using inverse resolution because some instances of any instantiation of its definition do not appear in its projected set. Another example of zero utilization is the logic program for *reverse* defined by a new predicate *rev3* of arity three; again, some sequence instances cannot be projected. Zero utilization represents a *fundamental limitation* of inventing theoretical terms using IR.

3 Basic Concepts

3.1 Inverse Resolution vs Inverse Implication

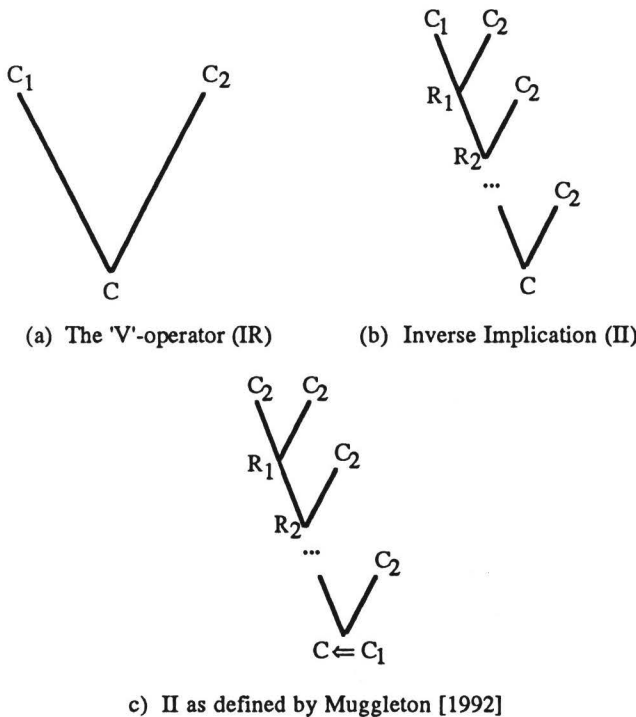


Figure 1: Inverse Resolution (IR) vs Inverse Implication (II).

The 'V'-operator in IR infers a clause in one resolution step. Consider the two clauses C and C_1 in Figure 1(a). 'V' infers C_2 such that C_1 and C_2 can be resolved to C .

Inverse Implication (II) is an extension of the 'V' operator. Rather than inferring a clause used in a single resolution step, II infers a clause used in a *chain* of resolutions (in Figure 1(b), C_2 given C and C_1). There is a close relation between II in Figure 1(b) and Muggleton's definition [1992] of the process which is illustrated in Figure 1(c): because $C_1, C_2 \vdash C$, thus $C_2 \vdash C \Leftarrow C_1$.

Assume the clause of the new predicate *newp* is in the form:

$\text{newp}(\dots) \Leftarrow \text{newp}(\dots)$.

To infer the right clause for the new predicate *newp*, two instances must be available from the 'V'-operator which lie one resolution step from each other. A new term may be partially utilized or zero utilized, in which case it yields instances more than one resolution step apart; 'V' only works with instances one step apart, the instances forming a BRS. This poses a problem for IR methods that use 'V' and require a BRS. II on the other hand can infer a clause from two examples many resolution steps apart. It allows many partially utilized and some zero utilized new predicates to be inferred and is clearly a more powerful learning technique.

3.2 Terminology

The method of inferring C_2 in II has been detailed by Lapointe [1992]. Intuitively, it consists of first finding the differences between C_1 and C by a process called *sub-unification*. These differences are called the *embedding terms*. The smallest *generating terms* are then identified which will reproduce the embedding term when applied repeatedly. Clause C_2 is then inferred with these generating terms representing the arguments in the clause head, and its corresponding variables representing the arguments of the clause body. Use of the smallest generating term possible inverts the greatest number of resolution steps and produces the most general C_2 .

Thus if $C_1 = p(s(a), b)$ and $C = p(s^5(a), t^2(b))$, C_1 is sub-unifiable with C using the embedding terms $\langle s^4(V_1), t^2(V_2) \rangle$. Sub-unified terms are $s(a)$ and b . Their embedding terms have as generating terms $\langle \{s^4(V_1)/1, s^2(V_1)/2, s(V_1)/4\}, \{t^2(V_2)/1, t(V_2)/2\} \rangle^4$. The clause C_2 inferred with generating terms of the same maximum depth for all the arguments is:

(1) $p(s^2(V_1), t(V_2)) \Leftarrow p(V_1, V_2)$.

with a maximum depth of 2. In inverse resolution with the 'V'-operator only the following solution of depth 1 is produced:

(2) $p(s^4(V_1), t^2(V_2)) \Leftarrow p(V_1, V_2)$.

The more general solution (1) is preferable.

In sequel, input sub-unification means sub-unification applied to input arguments only.

⁴ g/n denotes a generating term g of depth n , $\{\dots\}$ denotes the set of all generating terms of a given argument, and $\langle \dots \rangle$ denotes sets of generating terms for several arguments.

4 Construction of New Predicates

4.1 Target Logic Programs

The method presented here is restricted to the subclass of all logic programs which are recursive and which satisfy one of the two following syntactic formats, in which all predicates are assumed to have mode declarations:

- (i) $p(\dots).$
 $p(\dots) \Leftarrow p(\dots).$
- (ii) $q(\dots).$
 $q(\dots) \Leftarrow q(\dots), \text{newp}(\dots).$
where newp is a new predicate defined as
 $\text{newp}(\dots) \Leftarrow \text{newp}(\dots). \text{newp}(\dots).$

Regularities in the expected syntax of the target definition form the basis of the CILP algorithm. CILP first tries to induce a definition in format (i), then in format (ii), and then requests more examples from the user. It repeats this process until successful.

There are some constraints on predicate arguments. The n^{th} argument of predicate p in the body of (i) above is either a variable with a unique occurrence in the generating term representing the n^{th} argument of the predicate p in the head of (i), or is identical to that term (if sub-unification is identical to unification). The same constraint applies to input arguments in the definition of q in (ii). We can observe that output arguments of q can be modified by newp in (ii) and consequently cannot be used to construct the generating terms. Every argument of newp is a variable appearing in another predicate in the clause. The output arguments of newp are the output arguments of q in the head; the output arguments of q in the body are the input arguments of newp . For instance, the logic program

$$q(f(a, Y), b, Y).$$
$$q(f(X, Y), b, Z) \Leftarrow q(X, b, W), \text{newp1}(X, W, Z).$$

can be learned by our method (the last argument of both q and newp1 is an output argument). When the new predicate newp1 satisfies (i), we refer to newp1 as learnable at level 1. Were newp1 is to be defined according to (ii) as $\text{newp1}(\dots) \Leftarrow \text{newp1}(\dots), \text{newp2}(\dots)$, learning newp2 from instances of q is at level 2. Learnability at levels higher than 1 is an open problem whose difficulty increases with the level, as the utilization problem becomes more and more severe with each next level.

Although CILP is limited to a constrained class of logic programs, in practice its two recursive formats describe a very useful class of logic programs: the subclass of the primitive recursive functions which includes all the common algebraic functions on the integer.

4.2 Arguments of the New Predicate

When a new predicate is invented, its arity and its bindings to other variables in the clause must be determined. CILP begins by defining all the variables occurring in the clause as the arguments of the new predicate:

$$q(f(X, Y), b, Z) \Leftarrow q(X, b, W), \text{newp}(X, Y, W, Z).$$

4.2.1 Types of Argument

A new predicate's argument can be categorized into three types:

- *Necessary arguments*:. Arguments essential to the concept being invented; it would not make sense without them. All output arguments of predicates in the clause are necessary. In the example above, the desired output Z is produced by newp , and consequently is its necessary argument. The recursive computation would be meaningless if the output W of the recursing predicate q is not used in newp , so W is considered necessary as well.
- *Useless arguments*. Arguments which do not affect the learnability of a new concept. Useless arguments neither hinder nor help the induction of the new concept, so removing them changes nothing.
- *Harmful arguments*:. Arguments preventing the new concept from being learned. Harmful arguments must be removed if a new concept is to be learned successfully.

Arguments introduced by the inputs to other predicates (X and Y above) can be of any type — their type is unknown. When the definition of a new predicate is being induced, arguments with unknown type must be eliminated if they hinder the learning process.

4.2.2 Removing Arguments

CILP incorporates a technique to identify harmful arguments so they can be removed. Induction is undertaken with all arguments in place. First the deepest generating terms of depth n for all necessary arguments are obtained. Other arguments of unknown type are then inspected. They are kept if and only if they also have a generating term of this depth. An example may help clarify this. Suppose the second and third arguments of some predicate are necessary and both have generating terms of depth 1 and 3 for a given sub-unification, while the first argument with unknown type has a single generating term of depth 1. The first argument would be removed in order to consider the generating terms of depth 3. Removal is reversible; the first argument can be brought back should CILP backtrack and consider the generating terms of depth 1.

4.3 Projected Set of the New Predicate

Once a clause incorporating the new predicate has been constructed, as many instances as possible of the predicate are collected for the induction of its definition. One example is unified with the head and another with the recursing literal in the body. Instances collected in this manner form the new predicate's projected set. Assume the examples $p(s(0), s(0))$, $p(0, 0)$ and $p(s^3(0), s^6(0))$ are given and the induced clause is:

$$p(s(X), Z) \Leftarrow p(X, W), \text{newp}(X, W, Z).$$

The first example unifies with the head and the second with the recursing literal in the body yielding the instance $\text{newp}(0, 0, s(0))$. More instances are desirable and to get them the third example must be used. However no example has $s^2(0)$ as its first argument and there is no way of

building such an example. The only solution is to query an oracle for the required argument⁵. The query in the example would be: "What is $p(s^2(0))$?". The desired answer is $s^3(0)$, from which the instance $\text{newp}(s^2(0), s^3(0), s^6(0))$ can be obtained. The oracle is not required to answer all queries but it should answer as many as possible. The existence of an oracle does not solve the utilization problem; its responses still may not provide a basic representative set (zero utilization). Moreover, example pairs required for collecting instances of the new predicate need not be provided in the training set. In our approach, the missing elements of a pair are identified and then the oracle is queried for their values.

4.4 Base Case

A base case is a unit clause used by a recursive logic program in the last step of a proof by refutation. The induced definition of a predicate must therefore include an appropriate base case in addition to the recursive clause. Finding the base case is a matter of keeping any unresolved fact in the proof of each example after it is reduced by the recursive clause as many times as possible. The base case is the least general generalization (LGG) of the unresolved facts from all examples. Consider this clause and the three examples of it that follow:

C: $p(s(X), Y, s^2(Z)) \Leftarrow p(X, Y, Z)$.
 E_1 $p(0, s(0), s^2(0))$
 E_2 : $p(s(0), 0, s^3(0))$
 E_3 : $p(s^2(0), s(0), s^6(0))$

E_1 cannot be reduced and is kept as-is. E_2 can be reduced once to yield the unresolved fact $p(0, 0, s(0))$. Reducing E_3 twice produces $p(0, s(0), s(s(0)))$. The base case is the LGG of these three facts, that is, $p(0, Y, s(Y))$.

4.5 The CILP Algorithm

We have embodied the Constructive Inductive Logic Programming approach detailed in this paper in an algorithm is called CILP. This algorithm employs the five procedures `PURE_RECURSION`, `LEFT_RECURSION`, `BASE_CASE`, `TOO_SPECIFIC` and `TOO_GENERAL`. We present pseudocode for the algorithm and procedures first and then discuss it.

CILP

- C1.** Let S be the set of positive examples and N be the set of negative examples.
- C2.** $P \leftarrow \text{Pure_Recursion}(S, N)$.
- C3.** If $P \neq \text{Fail}$ then
- C4.** Return(P).
- C5.** $P \leftarrow \text{Left_Recursion}(S, N)$.
- C6.** If $P \neq \text{Fail}$ then
- C7.** Return(P).
- C8.** $S' \leftarrow \text{Read more examples.}$
- C9.** $S \leftarrow S \cup S'$.
- C10.** Goto(2).

⁵Obviously queries about new theoretical terms cannot be asked.

PURE_RECURSION(S, N)

- P1.** Choose a new pair of examples $\langle E_1, E_2 \rangle$ from S . If no more pairs then RETURN(*Fail*).
- P2.** Find all sub-unifications of E_1 in E_2 .
- P3.** For every sub-unification, find all associated generating terms.
- P4.** Find the deepest generating terms $\langle g(X), \dots \rangle$. Arguments of the unknown type are removed if they do not have generating terms of the same depth than necessary arguments. If no more generating terms are available then GOTO(1).
- P5.** Build the clause C of the form $p(g(X), \dots) \Leftarrow p(X, \dots)$.
- P6.** $B \leftarrow$ the target predicate with LGG of unified subterms as arguments.
- P7.** $S' \leftarrow S - \{E_1, E_2\}$.
- P8.** $B \leftarrow \text{BASE_CASE}(B, S', C)$.
- P9.** $P \leftarrow C \cup B$.
- P10.** If `TOO_GENERAL`(P, N) = *Yes* then
- P11.** Remove the current generating terms.
- P12.** GOTO(4) (try other generating terms).
- P13.** RETURN(P) (A satisfactory program has been induced).

BASE_CASE(B, S', P)

- B1.** Reduce all examples of S' using P . Call this new set R .
- B2.** $B \leftarrow \text{LGG}(R \cup \{B\})$.
- B3.** RETURN(B).

TOO_GENERAL(P, N)

- T1.** For each $n \in N$
- T2.** If P can prove n then RETURN(*Yes*).
- T3.** RETURN(*No*).

LEFT_RECURSION(S, N)

- L1.** Choose a new pair of examples $\langle E_1, E_2 \rangle$ from S . If no more pairs then RETURN(*Fail*).
- L2.** Find all input sub-unifications of E_1 in E_2 .
- L3.** For every input sub-unification, find all associated generating terms.
- L4.** Find the deepest generating terms $\langle g(X), \dots \rangle$. If no more generating terms are available then GOTO(1).
- L5.** Build the clause C of the form $p(g(X), \dots) \Leftarrow p(X, \dots), \text{newp}(\dots)$ that introduces the new predicate newp having all variables of the clause as arguments. The generating term applies to input arguments only.
- L6.** Collect the projected set of newp using the oracle if needed. Call this set PRO .
- L7.** $Q \leftarrow \text{PURE_RECURSION}(\text{PRO}, \emptyset)$.
- L8.** If $Q = \text{Fail}$ then
- L9.** Remove the current generating terms.
- L10.** GOTO(4) (try other generating terms).
- L11.** $B \leftarrow \text{BASE_CASE}(\text{Nil}, S, C \cup Q)$.
- L12.** $P \leftarrow C \cup B$.
- L13.** If `TOO_GENERAL`($P \cup Q, N$) = *Yes* then

- L14.** Remove the current generating terms.
- L15.** GOTO (4) (try other generating terms).
- L16.** RETURN (P U Q) (A satisfactory program has been induced).

Some points require explanation.

- RETURN always exits the procedure.
- The body of an 'if' statement is denoted by indentation.
- The PURE_RECURSION procedure tries to induce a recursive definition of the target predicate without inventing a new predicate. At step P10, the consistency test is performed. If some negative examples are covered then the current generating terms are removed and other generating terms are tried.
- The LEFT_RECURSION procedure tries to induce a definition that introduces a new predicate. Once the instances of the new predicate are collected then the PURE_RECURSION procedure is called on these instances. At step L8, it tests if the PURE_RECURSION procedure could induce a definition for the collected instances. If it could not then the current generating terms are removed and other generating terms are tried.

4.6 Examples

4.6.1 Exponential Functions

This simple example illustrates a case where the new predicate is zero-utilized. Consider the following exponential function:

$$f(X) = 2^X \quad \text{for every } X \geq 1$$

It can be defined recursively as follows:

$$f(1) = 2$$

$$f(X+1) = f(X) * 2$$

To induce a logic program that represents this function a new predicate that doubles natural numbers must be invented. The set of positive examples S is $\{f(s^1, s^2), f(s^3, s^8)\}$ and the set of negative example N is $\{f(s^2, s^5)\}$.

Because of the lack of space, we show here only some of the relevant intermediate steps of the algorithm (see [Lapointe et al., 1993] for more details).

The procedure PURE_RECURSION fails since the negative example is covered. At step L5, the following clause is obtained:

$$f(s(V_1), Z) \Leftarrow f(V_1, W), \text{newp}(V_1, W, Z).$$

The second and the third arguments of newp (i.e. W, Z) are necessary and the type of the first is unknown. At step L6, the oracle is queried because the output of $f(s^2, ?)$ is unknown. A new example $f(s^2, s^4)$ is provided and the projected set PRO is obtained:

$$\{ \text{newp}(s^1, s^2, s^4), \text{newp}(s^2, s^4, s^8) \}$$

The Procedure PURE_RECURSION applied to the set PRO produces: $\text{newp}(s(V_2), s^2(V_3)) \Leftarrow \text{newp}(V_2, V_3)$.

The first argument of unknown type was considered harmful because it does not have a generating term of depth 2 as the second and the third arguments. It has been removed.

At steps P6-P8, the base case for the new predicate is obtained, that is, $\text{newp}(s^2, s^4)$. At step L11, the base case for f is obtained, $f(s^1, s^2)$.

A satisfactory program is then induced:

$$f(s^1, s^2).$$

$$f(s(V_1), Z) \Leftarrow f(V_1, W), \text{newp}(W, Z).$$

$$\text{newp}(s^2, s^4).$$

$$\text{newp}(s(V_2), s^2(V_3)) \Leftarrow \text{newp}(V_2, V_3).$$

This definition of newp (doubling integer) is zero utilized (as discussed in section 2.2). Similar results would be obtained for any exponential functions of the form:

$$f(X) = N^X, N \geq 2, X \geq 1.$$

4.6.2 Other Examples

A recursive logic program representing polynomials of degree 1 (general form: $p(X) = aX + b$) does not need a new predicate to represent polynomials of degree 1 as the following illustrates:

$$p(0, s^b).$$

$$p(s(X), s^a(Z)) \Leftarrow p(X, Z).$$

On the other hand, polynomials of degree 2 (general form: $p(X) = aX^2 + bX + c, a > 0$) need an auxiliary predicate that performs a kind of addition. CILP, while inducing a logic program that evaluates any polynomial of degree 2, learns this necessary new predicate. Given the following positive examples:

$$p(0, s^c)$$

$$p(s^2, s^{4a+2b+c})$$

$$p(s^4, s^{16a+4b+c})$$

and the following negative example $p(0, s^{c+1})$ where a, b and c are constants, induction will succeed in learning a definition of both p and the necessary new predicate. During the process the oracle will provide two other examples of p , that is $p(s(0), s^{a+b+c})$ and $p(s^3, s^{9a+3b+c})$. The projected set of newp will be

$$\text{newp}(0, s^c, s^{a+b+c})$$

$$\text{newp}(s(0), s^{a+b+c}, s^{4a+2b+c})$$

$$\text{newp}(s^2, s^{4a+2b+c}, s^{9a+3b+c})$$

$$\text{newp}(s^3, s^{9a+3b+c}, s^{16a+4b+c})^6$$

The induced program will be

$$p(0, s^c).$$

$$p(s(V_1), Z) \Leftarrow p(V_1, W), \text{newp}(V_1, W, Z).$$

$$\text{newp}(0, s^c(A), s^{a+b+c}(A)).$$

$$\text{newp}(s(V_1), s^{a+b}(V_2), s^{3a+b}(V_3)) \Leftarrow$$

$$\text{newp}(V_1, V_2, V_3).$$

where $\text{newp}(V_1, W, Z)$ represents a special version of addition used in the definition of any polynomial of degree 2, i.e. $2aV_1 + W + (a+b) = Z$.

Two other classical examples that involve the invention of necessary predicates are also learnable by our method. The usual definition of multiply which uses a predicate to perform addition can be gracefully learned; the new predicate plus is invented and a harmful argument is removed in the process. In the case of the other example, reverse, a

⁶Even though the four instances of newp form basic representative sets, they cannot be obtained by the 'W' operator if the two additional examples of p are not given together with the three initial examples. For CILP, three initial examples suffice, and the other two are obtained from the oracle during the induction process.

predicate to add an element to the end of a list is invented. This new predicate contains a useless argument.

5 Conclusion and Future Work

This paper presents a relational learning method called CILP that performs truly constructive induction by inventing necessary theoretical terms. Our characterization of the training sets required by other recent ILP programs shows how our method differs from these systems, and emphasizes the importance of its ability to learn from a sparse training set. Because our approach is based on inverting implication rather than inverse resolution it can infer clauses from two examples that are far apart in a chain of resolution steps. Many new, partially-utilized and some zero-utilized predicates can now be inferred in the CILP framework. The class of definitions learnable by CILP includes a number of useful non-trivial logic programs. We have shown how our approach learns first level new terms. Moving to higher levels would mean, e.g., learning multiply and plus from the examples of exponential. Learning at an arbitrary level is a difficult challenge; resolving it would allow one to learn any primitive recursive function at very low levels of utilization. In CHAMP [Kijisirikul et al., 1992], new predicates can be learned at several levels of learnability, but only for selected examples in which the provided h-easy model and/or BRS can be projected to new predicates. SIERES [Wirth and O'Rorke, 1991] does predicate invention but also requires a BRS for the necessary predicate to be learned.

Although our CILP algorithm has not been fully implemented, the procedures PURE_RECURSION and LEFT_RECURSION are implemented in the LOPSTER system (see [Lapointe, 1992; Lapointe and Matwin, 1992] for experimental results).

Future work includes generalizing our approach to situations where examples involve different constants and do not necessarily belong to intersecting chains of recursion. Another interesting problem is the inclusion of intensional background knowledge in the form of intensional rules.

References

- Buntine, W. (1989). "Constructive induction in definite clause logic", unpublished memo, 1989.
- deRaedt, L. (1992). European Workshop on Inductive Logic Programming, Vienna, ECAI, 1992.
- Hoff, B., R. S. Michalski, et al. (1982). "INDUCE2 - A Program for Learning Structural Descriptions From Examples.", Technical Report 82-5, UIUC, 1982.
- Kijisirikul, B., M. Numao, et al. (1992). "Discrimination-based Constructive Induction of Logic Programs.", Procs. of AAAI-92, San Jose, pp. 44-49, 1992.
- Lapointe, S. (1992). "Induction of Recursive Logic Programs.", Master thesis, University of Ottawa, 1992.
- Lapointe, S., C. Ling and S. Matwin (1993). "Constructive Inductive Logic Programming.", Third Inductive Logic Programming Workshop, Bled, Slovenia, 1993.
- Lapointe, S. and S. Matwin (1992). "Sub-unification: a tool for efficient induction of recursive programs.", Procs. of Ninth International Conference on Machine Learning, Aberdeen, pp. 273-280, Morgan Kaufmann, 1992.
- Ling, X. C. (1991). "Inductive learning from good examples.", Procs. of Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91), Sydney, pp. 751-756, 1991.
- Ling, X. C. (1992). "Inventing Necessary Theoretical Terms.", Technical Report TR 302, Dept. of Computer Science, University of Western Ontario, 1992.
- Michalski, R. S. (1983). "A Theory and Methodology of Inductive Learning." in Machine Learning, R. S. Michalski, J. Carbonell and T. Mitchell, ed., pp. 84-134, 1983.
- Muggleton, S. (1987). "Duce, an oracle based approach to constructive induction.", Procs. of Tenth International Joint Conference on Artificial Intelligence (IJCAI-87), Milan, pp. 287-292, 1987.
- Muggleton, S. (1991a). Inductive Logic Programming ILP-91, Viana de Castelo, 1991.
- Muggleton, S. (1991b). "Inverting the Resolution Principle.", *Machine Intelligence 12*, 1991.
- Muggleton, S. (1992). "Inverting Implication.", Procs. of European Workshop on Inductive Logic Programming, Vienna, 1992.
- Muggleton, S. and W. Buntine (1988). "Machine invention of first-order predicates by inverting resolution.", Procs. of Fifth International Conference on Machine Learning, pp. 339-351, 1988.
- Muggleton, S. and C. Feng (1990). "Efficient induction of logic programs.", Procs. of First International Conference on Algorithmic Learning Theory, Tokyo, OMSHA, 1990.
- Muggleton, S. and K. Furukawa (1992). International Workshop on Inductive Logic Programming, Tokyo, ICOT, 1992.
- Quinlan, J. R. (1990). "Learning logical definitions from relations.", *Machine Learning 5*: pp. 239-266, 1990.
- Rouveirol, C. (1991). "Semantic Model for Induction of First Order Theories.", Procs. of Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91), Sydney, pp. 685-690, 1991.
- Shapiro, E. (1981). "Inductive Inference of Theories from Facts.", Technical Report TR 192, Dept. of Computer Science, Yale U., 1981.
- Valiant, L. G. (1984). "A theory of the learnable.", *Comm. ACM 27*(11): pp. 1134-1142, 1984.
- Wirth, R. and P. O'Rorke (1991). "Constraints on predicate invention." Procs. of Eighth International Workshop on Machine Learning, Evanston, pp. 457-461, Morgan Kaufmann, 1991.