Chapter 2

Regular Papers

Inverting Implication with Small Training Sets

David W. Aha[1], Stephane Lapointe[2], Charles X. Ling[3] and Stan Matwin[4]

[1] Naval Center for Applied Research in Artificial Intelligence, Code 5514, Naval Research Laboratory, Washington, DC 20375 USA
[2] DREV, P.O. Box 8800, Defence Research Establishment Valcartier, Courcelette, Quebec, G0A 1R0 Canada
[3] Department of Computer Science, University of Western Ontario, London, Canada
[4] Department of Computer Science, University of Ottawa, Ontario Canada

Abstract. We present an algorithm for inducing recursive clauses using inverse implication (rather than inverse resolution) as the underlying generalization method. Our approach applies to a class of logic programs similar to the class of primitive recursive functions. Induction is performed using a small number of positive examples that need not be along the same resolution path. Our algorithm, implemented in a system named CRUSTACEAN, locates matched lists of generating terms that determine the pattern of decomposition exhibited in the (target) recursive clause. Our theoretical analysis defines the class of logic programs for which our approach is complete, described in terms characteristic of other ILP approaches. Our current implementation is considerably faster than previously reported. We present evidence demonstrating that, given randomly selected inputs, increasing the number of positive examples increases accuracy and reduces the number of outputs. We relate our approach to similar recent work on inducing recursive clauses.

1 Introduction

Inducing recursive relational Horn clauses, or logic programs, is important for several reasons. First, recursion is the basic technique in logic programming, where iteration is achieved by recursion. Second, important classes of functions (e.g., primitive recursive functions) are defined recursively. Third, as we have pointed out in an earlier paper (Lapointe, Ling, & Matwin, 1993), constructive learning in the inductive logic programming (ILP) context requires the ability to learn recursive relations.

The shortcomings of existing ILP systems for inducing recursive clauses have been discussed in the recent literature, and are by now well understood (Lapointe & Matwin, 1992; Muggleton, 1992; Idestam-Almquist, 1993). It is generally believed that the θ-subsumption mechanism is the main source of difficulty in inducing recursive clauses in popular ILP systems (e.g., FOIL (Quinlan, 1991) and GOLEM (Muggleton & Feng, 1990)). Several authors have initiated work on ILP systems that are not founded on θ-subsumption. These systems are based on the idea that generalization, necessary for inducing general clauses from specific examples, should be based on *inverse implication* rather than *inverse resolution*.

We have proposed an approach and an implementation that shows how a specific inverse implication technique can induce a class of important recursive functions given very few examples.

In this paper, we extend our earlier work by proving that our method is complete (i.e., with respect to implication) for a specific class of logic programs that is structurally similar to the class of primitive recursive functions. Syntactically, the class of programs that our CRUSTACEAN system can learn has the form

$$\begin{aligned} &p(A_1, \ldots, A_n). \\ &p(A_1, \ldots, A_n) \leftarrow p(B_1, \ldots, B_n), \end{aligned}$$

where A_i and B_i are possibly complex logical terms (e.g., list constructors). This is a good indication of the power, in terms of expressiveness, of the class we have considered since it is well known that the class of primitive recursive functions can be used to represent almost any algebraic function. The class of purely logical relations contains such practically useful logic programs as appending and splitting lists, deleting a given element from a list, reversing lists, and arithmetic functions such as addition and factorial. In addition, our approach requires only a very small number of training examples. Methods based on θ-subsumption typically (e.g., in a classroom environment) require tens of training instances to induce some of the functions mentioned above.

This paper describes CRUSTACEAN, a system that generalizes the inverse implication approach first applied in LOPSTER (Lapointe & Matwin, 1992). CRUSTACEAN relaxes one of the limiting assumptions of LOPSTER, which required that training instances belong to the same chain of recursive calls. By removing this assumption, much less user's knowledge of the recursive relation is required than was required by LOPSTER. In LOPSTER, the user had to know either the base clause of the recursion or the recursive clause (i.e., in order to supply two examples that belong to the same chain of recursive calls). In CRUSTACEAN, the input examples are completely independent. This direction of research is motivated by our goal to develop efficient, practically usable ILP systems that assume no additional knowledge other than the strong hypothesis bias (i.e., the class of logic programs the system can induce). The current version of CRUSTACEAN has been significantly optimized compared to the version first reported in (Aha, Ling, Matwin, & Lapointe, 1993). We describe some of these optimizations in Sect. 3. Section 4 of this paper describes a theoretical characterization of CRUSTACEAN's abilities, Sect. 5 summarizes empirical results, and Sect. 6 relates our contributions to other work on inverse implication. We begin by describing some of the basic concepts underlying our approach in Sect. 2.

2 Basic Concepts

There are three notions, underlying our approach. These notions were first introduced, in a slightly different form, in LOPSTER (Lapointe & Matwin, 1992).

They are useful for building tools that can then be used to hypothesize the recursive concept definition from the structure of terms of $n \geq 2$ examples of that concept. These notions are (1) subterms, (2) embedding terms, and (3) generating terms. We will define them here informally, using an example as an illustration.

Given two terms t_1 and t_2, t_1 is a *subterm* of t_2 iff there exists a third term t_e, called an *embedding term* of t_1, such that substituting t_1 for a variable in t_e (i.e., embedding t_1 in t_e) we obtain a term identical to t_2 up to an additional substitution necessary to produce the specific constants (in general, subterms) in t_1 and t_2. For example, suppose that the two terms are $t_1 = []$ and $t_2 = [c, a] = [c, a|[]]$. Then t_1 is a subterm of t_2, where $t_e = [c, a|\text{Subterm}]$, and substituting $[]$ for the variable *Subterm* in t_e yields t_2 (i.e., the additional substitution in t_1 is vacuous here).

If t_1 is a subterm of t_2 realized by the embedding term t_e, then the *generating term* of t_e (of *depth n*) is a term t^V, such that V occurs exactly once in t^V, and that substituting t^V for V $n-1$ times in t^V produces t_e up to a substitution. Intuitively speaking, the generating term decomposes the embedding term into repeated applications of n identical substitutions. Again, let us consider the above example. The embedding term $[c, a|\text{Subterm}]$ has two generating terms. The first is $t^V = [c, a|V]$. It has a depth of 1 because we can obtain t_e from t^V directly (i.e., there is no need to substitute t^V for V). There is also a generating term at depth 2 for t_e, namely $t^V = [X|V]$. If we substitute t^V for V once, then we obtain $[X|[Y|V]] = [X, Y|V]$.

We can informally describe the mechanics of CRUSTACEAN as follows. It assumes that the target theory consists of one purely recursive clause R and one base clause B. If CRUSTACEAN is given positive examples P_1 and P_2, then P_1 can be proven by resolving B_1 and R repeatedly where B_1 is a specialization of B. Therefore, arguments of B_1 are subterms of P_1. Similarly, arguments in B_2 are subterms of P_2. Therefore, to infer the recursive clause R and the base clause B, CRUSTACEAN computes, for all the training examples, all the subterms. The base clause is then induced from the *least general generalization* (lgg) (Muggleton & Feng, 1990) of these subterms. The resulting embedding terms must contain the same generating term since the same recursive clause R is used in the proofs of the positive examples (i.e., except for those positive examples that are instantiations of the base clause). Therefore, the next step is to find the matching generating terms that form the basis for constructing the recursive clause. The next section describes CRUSTACEAN and an example of its processing in more detail.

3 Description of CRUSTACEAN

CRUSTACEAN is a generalization of Lapointe and Matwin's (1992) LOPSTER system.[5] Both systems are based on inverse implication rather than θ-

[5] An earlier version of CRUSTACEAN was described in (Aha, Ling, Matwin, & La-

subsumption. Their main difference is that CRUSTACEAN does not require that the positive instances be on the same inverse resolution path. Effectively, this means that the user does *not* need to know the definition of the target relation a priori.

Table 1. CRUSTACEAN induces a set of recursive relations and filters them according to constraints provided by its inputs

```
 Given: $P$: a set of positive examples
        $N$: a set of negative examples

CRUSTACEAN($P$,$N$)
  1. Relations ← Induce_relations($P$,$N$)
  2. Output: Filter_relations(Relations,$P$,$N$)

Induce_relations($P$,$N$)
  1. $S$ ← Subterms($P$)
  2. $G$ ← Generating_terms($S$)
  3. $C$ ← Combinations($S$,$G$)
  4. $\forall c \in C$:
    IF Legal_match($c$) THEN
       4.1 $b$ ← Base_clause($c$)
       4.2 IF Inconsistent($b$,$N$) THEN
           4.2.1 $r$ ← Recursive_clause($c$)
           4.2.2 Relations ← Relations $\bigcup \{\langle b, r \rangle\}$
  5. Output: Relations

Filter_relations(Relations,$P$,$N$)
  1. Discard all redundant relations
  2. Discard all infinitely recursive relations
  3. Discard all relations that resolve with any $n \in N$
  4. Discard all relations that cannot resolve with all $p \in P$
  5. Output: Remaining relations
```

Table 1 summarizes CRUSTACEAN's control algorithm and I/O behavior. CRUSTACEAN inputs both a set of positive and a set of negative examples of a predicate and outputs singly recursive relations LP=(BC,RC) with base clause BC and recursive clause RC such that

pointe, 1993). The current version removes the requirement to constrain the set of constants that may appear in the induced relations, attempts to induce relations from all matching sets of generating lists rather than only those with highest summed depth, requires that their summed depth be at least two, and incorporates several efficiency improvements.

1. LP resolves with each positive example and no negative example and
2. at least one of the positive examples can resolve with RC.

Each argument BC_j ($1 \leq j \leq$ arity(LP)) of BC is the lgg of S_{ij} ($1 \leq i \leq n$), where S_{ij} is a subterm of P_{ij} (i.e., the j^{th} argument of positive example P_i) obtained by recursively applying a structural decomposition operator D_j N_i times. Each argument RC_j in the head of RC is the lgg of the subterms generated by applying D_j to P_{ij}, where each P_i contributes N_i subterms. Each argument C_j of RC's (single) condition comes from applying D_j once to RC_j.

CRUSTACEAN's behavior is illustrated using the last_of example (Table 2) throughout the remainder of this section. In our illustration, pairs of subterms and embedding terms are represented using square bracket notation for lists. "Subterm" is used in the embedding terms to refer to the location of the subterm within the corresponding entire term. For example, the pairing [a,[Subterm]] pairs a subterm "a" with an embedding term "[Subterm]" for the term "[a]". We represent lists as dotted pairs (e.g., "[a,b]" is our shorthand for "pair(a,pair(b,[]))").

CRUSTACEAN calls Induce_relations to induce relations for the given predicate and then calls Filter_relations to discard those that do not pass all of its filters.

The positive examples are last_of(a,[c,a]) and last_of(b,[x,y,b]). Function Induce_relations begins by computing, for each positive example's arguments, its set of subterms and their embedding terms (Step 1). Step 2 then computes their corresponding generating terms. The first example's subterms and embedding terms are shown below on the left. The corresponding generating terms and depths are displayed to their right.

```
[ [a,Subterm] ],                [ [empty,0] ],
[ [[c,a],Subterm],              [ [empty,0],
  [c,     [Subterm,a]],           [pair(1),1],
  [[a],   [c|Subterm]],           [pair(2),1],
  [a,     [c,Subterm]],           [pair(2)/pair(1),1],
  [[],    [c,a|Subterm]] ]        [pair(2),2 & pair(2)/pair(2),1] ]
```

There is one list for each argument of last_of. Since the first argument is a constant, it has only one subterm (i.e., itself). There are five subterms for the second argument (i.e., [c,a]). For example, the third subterm of the second argument is [a]. Its embedding term is [c|Subterm] since the substitution of [a] for *Subterm* in this expression yields the complete second argument [c,a]. Its only generating term is pair(2) at depth 1, meaning that the subterm is accessed by decomposing the argument once using *pair* and selecting its second argument. When we write *none* for a generating term, we mean that it is not defined for that subterm (i.e., no decomposition is required to access the subterm from the complete argument because they are identical). Two generating terms exist for the last subterm of the second argument. One of these yields the subterm via two recursive decompositions of pair(2) while the other does so via a combination of two such decompositions at depth 1 (i.e., the former decomposes the second argument once per recursive call while the latter decomposes it twice per recursive call). The same information for the second example is shown below.

```
[ [b,Subterm] ]                     [ [none,0] ]
[ [[x,y,b],Subterm],                [ [none,0],
  [x,      [Subterm,y,b]],            [pair(1),1],
  [[y,b],  [x|Subterm]],              [pair(2),1],
  [y,      [x,Subterm,b]],            [pair(2)/pair(1),1],
  [[b],    [x,y|Subterm]],            [pair(2),2 & pair(2)/pair(2),1],
  [b,      [x,y,[Subterm]]],          [pair(2)/pair(2)/pair(1),1],
  [[],     [x,y,b|Subterm]] ]         [pair(2),3 &
                                       pair(2)/pair(2)/pair(2),1] ]
```

Induce_relation's third step computes, for each example separately, the cross products of its subterms and its generating terms. Some combinations are discarded since they cannot contribute to the induction of recursive clauses. For example, combinations at different non-zero generating depths are discarded since they require a different number of decompositions per argument. This yields the six combinations of subterms (left) and generating terms (right) below for the first example.

```
[[a,Subterm], [[c,a],Subterm]]        [[none,none],0]
[[a,Subterm], [c,    [Subterm,a]]]    [[none,pair(1)],1]
[[a,Subterm], [[a],  [c|Subterm]]]    [[none,pair(2)],1]
[[a,Subterm], [a,    [c,Subterm]]]    [[none,pair(2)/pair(1)],1]
[[a,Subterm], [[],   [c,a|Subterm]]]  [[none,pair(2)],2]
[[a,Subterm], [[],   [c,a|Subterm]]]  [[none,pair(2)/pair(2)],1]
```

Although the last two combinations have identical combinations of subterms, they are paired with different generating terms and depths. The nine combinations for the second example's subterms (left) and generating terms (right) are shown below.

```
[[b,Subterm],[[x,y,b],Subterm]]       [[none,none],0]
[[b,Subterm],[x,     [Subterm,y,b]]]  [[none,pair(1)],1]
[[b,Subterm],[[y],   [x|Subterm]]]    [[none,pair(2)],1]
[[b,Subterm],[[y,b], [x,Subterm,b]]]  [[none,pair(2)/pair(1)],1]
[[b,Subterm],[[b],   [x,y|Subterm]]]  [[none,pair(2)],2]
[[b,Subterm],[[b],   [x,y|Subterm]]]  [[none,pair(2)/pair(2)],1]
[[b,Subterm],[b,     [x,y,[Subterm]]]] [[none,
                                         pair(2)/pair(2)/pair(1)],1]
[[b,Subterm],[[],    [x,y,b|Subterm]]] [[none,pair(2)],3]
[[b,Subterm],[[],    [x,y,b|Subterm]]] [[none,
                                         pair(2)/pair(2)/pair(2)],1]
```

The cross product of the combinations found for these two positive examples thus yields $6 \times 9 = 54$ total combinations, which is output by the call to Combinations in Step 3. Step 4 in Induce_relations tests these 54 combinations. For each combination, it first attempts to match their generating lists. For example, the first attempted match involves the following lists.

```
[[a,Subterm],[[c,a],Subterm]]        [[none,none],0]
[[b,Subterm],[[x,y,b],Subterm]]      [[none,none],0]
```

This match is immediately rejected because the summed depths of these lists of generating terms is zero, whereas the summed depths must be at least two to ensure, with high probability, that the induced recursive clause is sufficiently general.[6] If the summed depth is zero, then both examples' combinations of subterms are instantiations of the base clause. For such combinations, the recursive clause cannot be induced (i.e., because the pattern of decompositions performed during recursion cannot be determined). If the summed depths was only one, then one combination of subterms would require exactly one recursive call. Later, we will show how CRUSTACEAN induces the arguments for the head of the recursive clause by computing the lgg's of the examples' arguments prior to decomposition. In this case, the lgg would be computed over only one decomposition per each argument, which would cause them to be overly specific. Thus, CRUSTACEAN requires that the summed depths of a match be at least two.

One of the remaining 53 matches being considered attempts to match the second combinations for each instance:

```
[[a,Subterm],[c,[Subterm,a]]]        [[none,pair(1)],1]
[[b,Subterm],[x,[Subterm,y,b]]]      [[none,pair(1)],1]
```

Here the summed depth is two, which is sufficient. Lists of generating terms match only if the generating terms match for each argument. The *none* generating term matches any term. All other generating terms must match exactly, both in the function used to decompose the term and its selected argument (i.e., the second arguments in both of these lists decompose their arguments using the "pair" function's *first* argument). In this case, the lists match. Step 4.1 then computes the matching lists' base clause from their subterms, which are shown below for this example.

```
 First example: (a,c)
Second example: (b,x)
```

A potential base clause is induced from these subterms by computing the least general generalization (Muggleton & Feng, 1990) of each argument. This match thus yields last_of(A,B) as a potential base clause. Step 4.2 discards potential base clauses that resolve with any of the negative examples. For example, this clause is discarded since it resolves with the only negative example (i.e., last_of([x,y],[x])). However, the match between the following subterms and generating terms yields last_of(A,[A]), which does not resolve with any negative example.

```
[[a,Subterm],[[a],[c|Subterm]]]          [[none,pair(2)],1]
[[b,Subterm],[[b],[x,y|Subterm]]]        [[none,pair(2)],2]
```

[6] The induced recursive clause may still contain irrelevant constants under some circumstances.

In this case, Induce_relations calls Recursive_clause on this combination to induce a recursive clause. The arguments in the head of the recursive clause are taken from the least general generalization over the arguments from the iterative decompositions of each example. Each example contributes the number of iterations equal to the depth of its list of generating terms. The first iteration is the example itself, while the i^{th} iteration is generated by applying the decomposition specified by the list of matched generating terms $i-1$ times to the example. The iterative decompositions for this match, one from the first example and two from the second example, are shown below.

```
(a,[c,a])
(b,[x,y,b])
(b,[y,b])
```

This last iteration is generated by applying the decomposition [none,pair(2)] to (b,[x,y,b]). The lgg of these iterations yields the arguments (A,[B,C|D]). Thus, the head of the induced recursive clause is last_of(A,[B,C|D]). The recursive call of the induced recursive clause is constructed by applying the decomposition specified by the matched generating terms. This yields last_of(A,[C|D]), and the complete induced relation is shown below.

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

Although the recursive clause shown here differs from the more typical

```
last_of(A,[B|C]) :- last_of(A,C),
```

the induced clause is more accurate; there is no need to invoke the recursive clause when only one element remains in the list and it does not resolve with the base clause.

Note that, if the summed depths of the matched lists of generating terms was one, then only one iteration would be used to induce the recursive clause, which causes it to be overly-specific. Thus, as mentioned earlier, Legal_match (Step 4) requires that the summed depths of the list of generating terms in a match must be at least two.

Only one of the other matched combinations yields a base clause that does not unify with any negative example. That combination is shown below.

```
[[a,Subterm],[[],[c,a|Subterm]]]        [[none,pair(2)],2]
[[b,Subterm],[[],[x,y,b|Subterm]]]      [[none,pair(2)],3]
```

Their lists of generating terms match, their iterations are

```
(a,[c,a])
(a,[a])
(b,[x,y,b])
(b,[y,b])
(b,[b]),
```

and the induced base and recursive clauses are shown below.

```
last_of(A,[]).
last_of(A,[B|C]) :- last_of(A,C).
```

Thus, Induce_relations outputs only two induced relations for last_of.

Filter_relations is then called with these two relations. They differ, so both pass the redundancy filter. They also pass a simple test that detects infinitely recursive relations. They are then tested to see if they resolve with any negative example. The second relation resolves with negative example last_of([x,y],[x]), indicating that it is too general. Thus, it is discarded, leaving only the correct target relation (see also Table 2), which does not resolve with the negative example and does resolve with all of the positive examples.

4 Theoretical Analysis

This section describes a theoretical analysis of CRUSTACEAN. We provide a characterization of the structure of logic programs that it can induce. Furthermore, we show that our algorithm is complete for the family of logic programs that we have defined. That is, it induces all *least general* logic programs of the family that are consistent with the training examples. The hypothesis language of CRUSTACEAN is restricted to a specific class or family of singly-recursive logic programs. The following definition captures this targeted family.

Definition 1. (Definition of SR) SR is defined as the family of singly-recursive logic programs LP=(BC, RC) of the following syntactic form:

$$BC = p(a_1, a_2, ..., a_n).$$
$$RC = p(b_1, b_2, ..., b_n) \leftarrow p(c_1, c_2, ..., c_n).$$

where p is a predicate symbol, a_i, b_i, and c_i are terms, and, for every i $(1 \leq i \leq n)$ either

1. $c_i \neq b_i$, c_i is not a ground term (i.e., it contains at least one variable), and c_i is a subterm of b_i, or
2. $c_i = b_i$.

Furthermore, the first condition above must be satisfied for at least one i $(1 \leq i \leq n)$.

Any logic program LP $\in$ SR consists of a base clause BC and a singly-recursive clause RC. The last part of this definition ensures that at least one argument is decomposed (i.e., simplified) by applications of the recursive clause. This avoids meaningless logic programs that have an infinitely recursive clause such as $p(X) \leftarrow p(X)$. The family SR includes many interesting and non-trivial recursive logic programs.

Example 1.

$$\begin{aligned}\mathrm{BC}_1 &= \mathrm{last\_of}(A, [A])\\ \mathrm{RC}_1 &= \mathrm{last\_of}(A, [B, C|D]) \leftarrow \mathrm{last\_of}(A, [C|D])\end{aligned}$$

$\mathrm{LP}_1 = (\mathrm{BC}_1, \mathrm{RC}_1)$ belongs to the family SR; the first arguments of the recursive clause satisfy the second condition of Definition 1 and the second arguments satisfy the first condition.

Muggleton's (1992) definition of n^{th} powers and n^{th} roots of a clause will be useful for describing the output of our algorithm. First, we need to define the function L, which contains only linear derivations of Robinson's (1965) function R.

Definition 2. (Resolution closure) Let T be a set of clauses. The function L is recursively defined as

$$\begin{aligned}&L^1(T) = T\\ &L^n(T) = \{C | C_1 \in L^{n-1}(T), C_2 \in T, C \text{ is the resolvent of } C_1 \text{ and } C_2\}\end{aligned}$$

Definition 3. (n^{th} powers and roots of a clause) A clause D is an n^{th} power of a clause C iff D is a clause in $L^n(\{C\})$ except for the renaming of variables. Similarly, C is an n^{th} root of D if and only if D is an n^{th} power of C.

For instance, $p(f(f(X))) \leftarrow p(X)$ is a second power (i.e., the square) of $p(f(X)) \leftarrow p(X)$.

Proposition 4. *(Completeness of* CRUSTACEAN *for SR) Given a set of positive examples P and a set of negative examples N,* CRUSTACEAN *will output every logic program LP=(BC,RC) in SR with the following properties.*

1. *LP is complete and consistent with respect to P and N, and there exists a $p \in P$ such that BC θ-subsumes a $n^{th} \geq 1$ power of $\{RC, p\}$.*
2. *LP is a lgg under implication of P, or recursive clause RC is the least general n^{th} root of RC' for a given $n \geq 2$, where LP'=(BC,RC') is a lgg under implication of P.*

We can only provide a sketch for Proposition 4's proof at this time for a simple extension of CRUSTACEAN.[7] Let $\mathrm{GT}=\{\mathrm{GT}_1, \ldots, \mathrm{GT}_n\}$ be an arbitrary list of matched generating terms constructed from examples P with arity n. Let the depth of each $p_i \in P$ be D_i. Each GT_j defines how to decompose each p_{ij} into an arbitrary subterm s_{ij} $(1 \leq j \leq n)$. Let BC and RC be the base and recursive clauses constructed from this match. Each argument BC_j is the lgg of subterms s_{ij} for all i. For all $p_i \in P$, RC resolves with each argument p_{ij} to yield

[7] The extension is a filter in Filter_relations that removes all outputs θ-subsumed by other outputs. We plan to implement this in the near future.

s_{ij}, requiring $D_i \geq 0$ resolution steps. Therefore, LP=(BC,RC) is complete with respect to P. After construction, a filter ensures that LP cannot resolve with any $n \in N$. Thus, it is also consistent with N. Since at least one $D_i \geq 1$, then BC θ-subsumes the resolvent of RC_{D_i} and p_i, where RC_{D_i} is the D_i^{th} power of RC.

By construction, LP implies every clause in P. Our hypothesized simple extension of CRUSTACEAN adds a filter for removing outputs that θ-subsume other outputs (e.g., LP=(BC,RC), where another output LP'=(BC',RC') exists such that BC θ-subsumes BC' and RC θ-subsumes RC'). Since BC is constructed from the lgg of subterms of P, and RC from the lgg of generating term applications on P, then LP can be an lgg under implication of P. However, GT_j can contain $S \geq 2$ repeating sequences of decomposition operators. CRUSTACEAN outputs every LP=(BC,RC_s), where s is a divisor of S. Fewer repeats correspond to simpler and often more general recursive definitions. Since RC_S is a S/s power of RC_s under substitution θ_s, then each RC_s is a least general S/s root of RC_S, where LP=(BC,RC_S) is a lgg under implication of P. This completes our sketch.

CRUSTACEAN's output consists of all the least general logic programs of SR under implication for all depths of recursive clauses (i.e., the depth of a recursive clause corresponds to the number of times it is used to prove all the positive examples). Since the θ-subsumption notion of generality is only incomplete for self-recursive clauses (i.e., for expressing generality relations between different powers or roots of a clause) (Muggleton, 1992), no output is more general under θ-subsumption than any other output, although one might be more general than another under implication (e.g., as when the first recursive clause is a root of the second).

Example 2. Let $P = \{p(f(f(0,a),b)),\ p(f(f(f(f(0,a),b),a),b))\}$ and let $N = \{p(s(0))\}$. Consider the following logic programs belonging to SR:

$$\begin{aligned}
\mathrm{LP}_1 : \mathrm{BC}_1 &= p(0)\\
\mathrm{RC}_1 &= p(f(V,X)) \leftarrow p(V)\\
\mathrm{LP}_2 : \mathrm{BC}_2 &= p(0)\\
\mathrm{RC}_2 &= p(f(f(V,a),b)) \leftarrow p(V)\\
\mathrm{LP}_3 : \mathrm{BC}_3 &= p(X)\\
\mathrm{RC}_3 &= p(f(V,X)) \leftarrow p(V)\\
\mathrm{LP}_4 : \mathrm{BC}_4 &= p(0)\\
\mathrm{RC}_4 &= p(f(f(V,X),b)) \leftarrow p(V)
\end{aligned}$$

LP_2 is output because it is a lgg of P under implication. RC_1 is the least general second root of RC_2 (up to a substitution) and BC_1=BC_2. Therefore LP_1 is also output. LP_3 is not output since overly general BC_3 resolves with the negative example. The other outputs are complete and consistent with respect to P and N. However, RC_4 is not a lgg since it is more general than RC_2 and has the same term structure. Furthermore, is not a least general n^{th} root of any other RC' for a given LP'=(BC_4,RC'). Therefore, LP_4 is not output.

5 Empirical Evaluation

The following two subsections summarize our evaluations of CRUSTACEAN. Section 5.1 evaluates its performance when the inputs are manually selected, thus approximating optimal conditions. Section 5.2 instead evaluates its performance when the inputs are randomly selected, which tests CRUSTACEAN's robustness.

5.1 Evaluation with Manually Selected Inputs

We conducted ten experiments with CRUSTACEAN to investigate its ability to induce relations when given positive examples not on the same inverse resolution chain.[8] The relations used in the experiments were previously used to test LOPSTER (Lapointe & Matwin, 1992). However, the positive examples used here were modified so as to not be on the same inverse resolution path.[9] The positive examples, negative examples, and target relation for each experiment are summarized in Table 2. CRUSTACEAN output the correct target relation for each of the experiments; no other relations were output. Average run-times were collected from a set of 25 runs on a SUN SPARCstation 10 and are reported in Table 3. CRUSTACEAN is implemented in Quintus Prolog.

5.2 Evaluation with Randomly Selected Inputs

Although this evaluation demonstrated that CRUSTACEAN performs well with manually selected inputs, it provided no insight on how CRUSTACEAN performs under less optimal conditions. Therefore, we evaluated its performance when the inputs were not manually selected. We hypothesized that as more positive examples were input, CRUSTACEAN's accuracy would increase and its number of outputs would decrease.

In these experiments, terms whose types are recursive data structures (e.g., list, successor) were generated randomly from a uniform distribution on structure depth in the range $[0, 4]$. All constant terms and list subterms were randomly generated from a uniform distribution on the 26 lower case letters. Some arguments were generated so that they satisfied relation-dependent constraints when positive examples were requested. For example, when generating a positive example for the *member* relation, the first argument was selected randomly from the list's elements.

We examined CRUSTACEAN's performance when it was given two and three randomly selected positive instances. We varied the number of negative

[8] These were also used to test an earlier version of CRUSTACEAN reported by Aha, Ling, Matwin, and Lapointe (1993). However, this more advanced version does not constrain what constants are allowed to appear in the induced clauses. It also reduces the cpu time required to solve these ten problems by approximately 24%.

[9] The experiment with factorial contained positive examples that were on the same inverse resolution chain, but this was unavoidable due to its definition.

Table 2. Sets of positive (+) and negative (−) examples given as input in the 25 experiments and the relations induced by CRUSTACEAN

+ append([d],[],[d])	+ member(3,[1,2,3,4])
+ append([a,b],[c],[a,b,c])	+ member(j,[k,j])
- append([d],[e],[d,e,f])	- member(3,[1,2,5])
- append([a],[],[])	- member(a,[c,d])
append([],A,A).	member(A,[A|B]).
append([A|B],C,[A|D])	member(A,[B,C|D])
:- append(B,C,D).	:- member(A,[C|D]).
+ delete(c,[a,b,c,d],[a,b,d])	+ noneIsZero([s(s(0))])
+ delete(1,[2,1],[2])	+ noneIsZero([s(0),s(s(s(0)))])
- delete(d,[d,2],[d])	- noneIsZero([s(0),0])
delete(A,[A|B],B).	noneIsZero([]).
delete(A,[B,C|D],[B|E])	noneIsZero([s(A)|B])
:- delete(A,[C|D],E).	:- noneIsZero(B).
+ extractNth(s(s(0)),[y,z],z)	+ plus(s(0),s(0),s(s(0)))
+ extractNth(s(s(s(0))),[a,b,c,d],c)	+ plus(s(s(0)),0,s(s(0)))
- extractNth(s(s(0)),[x,y],x)	- plus(0,s(0),s(s(0)))
extractNth(s(0),[A|B],A).	- plus(s(0),0,0)
extractNth(s(s(A)),[B,C|D],E)	- plus(0,0,s(0))
:- extractNth(s(A),[C|D],E).	plus(0,A,A).
	plus(s(A),B,s(C))
	:- plus(A,B,C).
+ factorial(s(s(s(0))),	+ reverse([1,2],
s(s(s(0)))* (s(s(0))*s(0)))	append(append([],[2]),[1])
+ factorial(s(s(0)),s(s(0))*s(0))	+ reverse([x],append([],[x]))
- factorial(s(s(s(s(0)))),	- reverse([x,y],append([],[x]))
*(s(s(s(s(0)))),s(s(s(0))))	reverse([],[]).
fact(s(0),s(0)).	reverse([A|B],append(C,[A])) :-
fact(s(s(A)),s(s(A))*B)	:- reverse(B,C).
:- fact(s(A),B).	
+ last_of(a,[c,a])	+ split([x,y],[x],[y])
+ last_of(b,[x,y,b])	+ split([1,2,3,4],[1,3],[2,4])
- last_of([x,y],[x])	- split([x,y],[x,y],A)
last_of(A,[A]).	- split([a],[],[a])
last_of(A,[B,C|D])	- split([],[],[a])
:- last_of(A,[C|D]).	- split([a],[],[])
	split([],[],[]).
	split([A,B|C],[A|D],[B|E]) :-
	:- split(C,D,E).

Table 3. Average runtimes (25 runs) in cpu seconds and number of combinations of subterms processed for the ten relations (see Table 2)

Name	Cpu Time	#Combinations
append	0.7	981
delete	0.7	1152
extractNth	0.5	765
factorial	0.5	912
last_of	0.1	54
member	0.1	78
noneIsZero	0.1	50
plus	0.1	156
reverse	0.4	720
split	12.3	10948

instances from 0 through 25. Test sets included 100 randomly selected instances Positive and negative instances were generated with equal probability. Selections of all terms and instances were done with replacement. The accuracy of an output relation on a test set is defined as the percentage of correct classifications it provides among those instances. The accuracy of a set of output relations is their average accuracy. The accuracies displayed in Table 4 for each relation are averages across all runs. Runs were repeated ten times for each relation, where each run used a different randomly selected training set for each relation.

Table 4 summarizes the results for each relation when ten negative instances were input. Accuracies increase as more positive instances are input. However, they are not 100% for two reasons. First, the randomly generated positive examples can share structure or constants that yield overly-specific outputs. Second, the randomly generated negative examples do not always resolve with the overly general outputs. Both of these conditions can prevent CRUSTACEAN from outputting the (correct) target relation. Nevertheless, our experiments provide evidence that accuracy increases with the number of positive input examples. Although not shown here, accuracy also increases with the number of negative input examples.

For all but two relations, the number of outputs decreased sharply as the number of positive examples increased because the only lists of generating terms that can match for $n+1$ positive examples must be a subset of those that match for the first n examples.[10] Increasing numbers of negative examples also reduced the number of outputs, in this case because they filtered overly-general outputs.

[10] Except when adding additional positive examples satisfies CRUSTACEAN's constraint that the summed depths of a match be at least two. This occurred for *delete* and *split* here.

Table 4. CRUSTACEAN's average accuracies and number of outputs (10 runs) given two or three randomly selected positive examples with ten randomly selected negative examples

Name	2 Positive Inputs		3 Positive Inputs	
	Accuracy	#Outputs	Accuracy	#Outputs
append	63.0%	22.5	73.8%	15.0
delete	61.7%	20.4	71.3%	20.4
extractNth	60.0%	12.0	78.0%	5.0
factorial	78.5%	8.7	88.1%	3.5
last_of	74.4%	2.7	88.4%	1.4
member	65.2%	2.2	76.2%	1.8
noneIsZero	72.9%	7.0	79.3%	6.5
plus	63.5%	9.8	85.5%	5.8
reverse	80.5%	22.0	85.5%	8.9
split	78.3%	15.6	85.7%	22.3

6 Relationship with Previous Work

Since recursive relations are prevalent in logic programs written in PROLOG, recursive clause generalization under implication received extensive studies in the last few years. Lapointe and Matwin (1992) first proposed and implemented an inverse implicator called LOPSTER. When given two positive examples C and D of the same predicate, LOPSTER infers a recursive clause R such that R can be resolved with C and their resolvents *repeatedly* to prove D. That is, $C, R \models D$. This is equivalent to $R \models D \leftarrow C$.

Muggleton (1992) described a theoretically complete (though inefficient) algorithm that essentially computes generalization R from the recursive clause $D \leftarrow C$. However, both systems require the two examples given (C and D) to be on the same resolution chain resolved with R. This means that some a priori knowledge on the target theories must be known. This restriction is removed in our CRUSTACEAN system. It inputs a set of examples that are not necessarily on the same resolution chain. It infers a recursive clause and a base clause such that resolving positive examples with the recursive clause (eventually) yields instances of the base clause.

Idestam-Almquist (1993) proposed *recursive anti-unification* as a method for computing the minimal general generalization under implication (MinGGI) of a set of recursive clauses. There are two major differences between his algorithm and LOPSTER, which is related to CRUSTACEAN. First, LOPSTER generalizes from two examples C and D, which can be regarded as one recursive clause $D \leftarrow C$, while MinGGI generalizes from a set of recursive clauses. The MinGGI of one recursive clause is simply the clause itself. Second, Idestam-Almquist's algorithm induces minimally general recursive clauses. LOPSTER instead induces the *least* general recursive clauses that maximize the number of recursive calls in the resolution proof (i.e., by selecting matched lists of generating terms with maximal depth). This allows it to produce the simplest recursive clauses. This clause is more general than other recursive clauses under self-resolution. For instance, given the two examples

$$p(s(s(s(s(0)))))$$
$$p(s(s(s(s(s(s(0))))))),$$

LOPSTER yields $p(s(X)) \leftarrow p(x)$. On the other hand, given the clauses

$$p(s(s(s(s(0))))) \leftarrow p(0)$$
$$p(s(s(s(s(s(s(0))))))) \leftarrow p(0),$$

MinGGI would output $p(s(s(X))) \leftarrow p(X)$.

Another recent ILP system that is closely related to CRUSTACEAN is Cohen's (1993) FORCE2. Its class of learnable theories is similar to CRUSTACEAN's: one "closed" linearly recursive, *ij*-determinate clause and one base clause. The techniques used in FORCE2 are also closely related in essence to the ones in CRUSTACEAN in several aspects, except that FORCE2 can be regarded as a model-driven system, while CRUSTACEAN is data-driven. Instead of finding recursive literals (with generating terms) from the structural difference of subterms via sub-unification between given examples as in our systems, FORCE2 enumerates all possible recursive literals. Although FORCE2 has been proven to be a PAC-learning algorithm due to its restrictive representation, the enumeration algorithm would inevitably generate many recursive literals (and clauses) that would not be induced by CRUSTACEAN.

Another difference is that FORCE2 is provided with an oracle that answers the query if a positive example is a specialization of the (unknown) base clause. This oracle is needed because, after FORCE2 hypothesizes the recursive clause, it then hypothesizes other instances of the target predicate to be true by "force-simulating" (i.e., decomposing) given positive examples using the recursive clause repeatedly, until the oracle says the hypothesized instance is a specialization of the base clause. All such instances are collected and generalized to form the base clause via lgg.

CRUSTACEAN does not interact with an oracle. The only constraint it has is that the positive examples can be reduced to specializations of a common but unknown base clause. A large number of subterms of positive examples are explored to find such specializations, as well as the recursive literals for the recursive clause. Therefore, CRUSTACEAN can potentially produce overly-specific or overly-general base clauses, which result in unintended recursive clauses. Most of these unintended clauses can be removed by checking their consistency with the positive and negative examples, and through a few other constraints and heuristics. If our system were given the same oracle as in FORCE2, then its efficiency would be improved greatly.

Another primary difference between CRUSTACEAN and the algorithms proposed by Muggleton, Idestam-Almquist and Cohen is that its class of learnable theories is quite restricted. The recursive clause must be purely rather than linearly recursive, and their are certain restrictions on the terms. We feel that these restrictions, and other heuristics, are justified by CRUSTACEAN's efficiency (i.e., it performed well on the benchmark ILP clauses in Sect. 5). This again reflects our goal of designing efficient and practical ILP systems that assume little or no knowledge on the target theories except the class of such theories.

7 Conclusion

We have described CRUSTACEAN, a system that induces recursive relations from examples using a form of inverse implication. We have detailed the algorithm, as well as an example that illustrates its working. CRUSTACEAN produces purely recursive logic programs, syntactically similar to primitive recursive functions. We have also characterized the class of functions for which CRUSTACEAN is complete; CRUSTACEAN will produce the least general logic programs and their corresponding n^{th} roots (Muggleton, 1992) with respect to the the class of considered logic programs.

We believe that our approach is a practical method for inducing recursive clauses. The class of functions for which it is applicable is larger than it may seem at first glance since, unlike some other systems (e.g., IRES (Rouveirol, Puget 1990)), the literals are unflattened. When selecting the input examples, the user does not need to know the base clause, which is required by LOPSTER, or the recursive clause, which is required by practically all approaches based on θ-subsumption. Only a small number of examples are required. Finally, empirical results indicate that many practical and non-trivial recursive relations (e.g., list reversal, usually taught only in the second year of the Computer Science curriculum) can be learned in under one cpu second.

Future work will includes proving the conjecture expressed in Proposition 4, implementing the extension mentioned in its sketch, and further improving CRUSTACEAN's efficiency. We will also plan to extend CRUSTACEAN so that it can induce a larger class of logical relations (i.e., left-recursive relations) of the form

$$\begin{array}{l} p(\ldots). \\ p(\ldots) \leftarrow p(\ldots), q(\ldots). \end{array}$$

Acknowledgements

Thanks to our reviewers for their useful comments. The authors' email addresses are aha@aic.nrl.navy.mil, lapointe@jupiter.drev.dnd.ca, ling@csd.uwo.ca, and stan@csi.uottawa.ca respectively.

References

Aha, D. W., Ling, C. X., Matwin, S., & Lapointe, S. (1993). Learning singly-recursive relations from small datasets. In *Proceedings of the IJCAI-93 Workshop on Inductive Logic Programming.* Chambery, France: Unpublished.

Cohen, W. (1993). Pac-learning a restricted class of recursive logic programs. In *Proceedings of the Third International Workshop on Inductive Logic Programming* (pp. 73–86). Bled, Slovenia: J. Stefan Institute.

De Raedt, L. (1992). *Interactive theory revision: An inductive logic programming approach.* London: Academic Press.

Idestam-Almquist, P. (1993). Recursive anti-unification. In *Proceedings of the Third International Workshop on Inductive Logic Programming* (pp. 241–254). Bled, Slovenia: J. Stefan Institute.

Lapointe, S., Ling, X. C., & Matwin, S. (1993). Constructive inductive logic programming. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence.* Chambery, France: Morgan Kaufmann.

Lapointe, S., & Matwin, S. (1992). Sub-unification: A tool for efficient induction of recursive programs. In *Proceedings of the Ninth International Conference on Machine Learning* (pp. 273–281). Aberdeen, Scotland: Morgan Kaufmann.

Ling, X. C. (1991). Inductive learning from good examples. In *Proceedings of the Twelvth International Conference on Artificial Intelligence* (pp. 751–756). Sydney, Australia: Morgan Kaufmann.

Muggleton, S. (1992). Inverting implication. In *Proceedings of the First European Workshop on Inductive Logic Programming.* Vienna, Austria: Unpublished.

Muggleton, S., & Buntine, W. (1988). Machine invention of first order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning* (pp. 339–352). Ann Arbor, MI: Morgan Kaufmann.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. *Proceedings of the First International Workshop on Algorithmic Learning Theory* (pp. 368–381). Tokyo, Japan: Japanese Society for Artificial Intelligence.

Quinlan, J. R. (1991). Determinate literals in inductive logic programming. In *Proceedings of the Twelvth International Joint Conference on Artificial Intelligence* (pp. 746–750). Sydney, Australia: Morgan Kaufmann.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery,* **12**, 23–41.

Rouveirol, C., & Puget, J. F. (1990). Beyond inversion of resolution. In *Proceedings of the Seventh International Conference on Machine Learning* (pp. 122–130). Austin, TX: Morgan Kaufmann.