Prolog, Refinements and RLGG's

Claude Sammut

School of Computer Science and Engineering
University of New South Wales
Sydney 2052 Australia
claude@cse.unsw.edu.au
http://www.cse.unsw.edu.au/∼claude

Abstract. Cohen's [1] refinement rules provide a flexible mechanism for introducing intentional background knowledge in an ILP system. Whereas Cohen used a limited second order theorem prover to implement the rule interpreter, we extend the method to use a full Prolog interpreter. This makes the introduction of more complex background knowledge possible. Although refinement rules have been used to generate literals for a general-to-specific search, we show how they can also be used as filters to reduce the number of literals in an RLGG algorithm. Each literal constructed by the LGG is tested against the refinement rules and only admitted if a refinement rule has been satisfied.

1 Introduction

Most current ILP systems use some form of declarative bias to restrict the search space of the learner. Cohen [1] introduced *refinement rules* as a flexible way of specifying the form of literals that may be introduced into the head and body of the clause being constructed. His refinement rules are written in a second order language. This allows the rules to be expressed very succinctly. A disadvantage of this scheme is that the theorem prover is quite limited in its capabilities. In this paper we show how the language of the refinement rules can be based on a full Prolog interpreter with some simple second order extensions. We also show how this system can be used to achieve the same effects as lazy modes in Srinivasan's work with Progol [10].

In his FLIPPER system, Cohen [1] used refinement rules to generate literals for a general-to-specific search similar to FOIL's [6]. We show how the same kind of rules can be used to produce a saturated clause [8] [7] that can be used either in a general-to-specific search or in a specific-to-general search as used by Plotkin [5] or by Muggleton [2] in GOLEM. The crucial observation here is that the refinement rules can be used to filter literals that are created by Relative Least General Generalisation [5].

Learning based on the RLGG has fallen out of favour, mainly because the Least General Generalisation of two clauses often creates a very large number of redundant literals that must be avoided or removed by a variety of heuristics. However, the RLGG remains a useful technique, especially when a specific-to-general search is most suitable for a particular problem. It is well suited to

learning tasks in which we are interested in obtaining a *characteristic* rather than a *discriminant* concept description. For example, in computer vision we may wish to learn to describe objects that appear in an image. In this case, it may not be useful to learn to simply distinguish between, say, a cup and a table because this does not capture the properties of the cup itself. For this reason, we are interested in obtaining generalisations of the training set that are as specific as possible.

In the following section, we review the basic ideas behind refinement rules. We then show how the refinement rule interpreter can be implemented as an extension to Prolog. Sections 4 and 5 demonstrates how refinement rules can be used in conjunction with an RLGG algorithm to limit the number of literals generated.

2 Refinement Rules and Generalisation

In this section, we briefly summarise some of the key ideas introduced by Cohen. Like FOIL, FLIPPER is a covering algorithm that creates a new clause by starting with a head and an empty body. New literals are introduced into the body of the clause to specialise it. The efficacy of the literal is tested, as in FOIL, by information gain. Where FLIPPER differs from FOIL is the way in which it constructs new literals. Literals are generated by refinement rules. Two types of refinement rule may be defined. A *head rule* has the form:

$$\langle A, Pre, Post \rangle$$

where A is a positive literal, Pre is a conjunction of literals and $Post$ is a set of positive literals. A body rule has the form:

$$\langle \leftarrow B, Pre, Post \rangle$$

where B is a positive literal and Pre and $Post$ are as above.

There must only be one head rule stating that A should be used to create the head of the clause being learned, provided that the condition Pre is satisfied. After A has been constructed, the literals in $Post$, are asserted into a temporary database. There may be any number of body rules to generate literals for the body of the clause under construction. The preconditions are tested using a function-free second order theorem prover.

Cohen gives an example of refinement rules that he used for the king-rook-king illegal problem:

$$DB_0 = \{rel(adjacent), rel(equal), rel(less\_than)\}$$

$$illegal(A, B, C, D, E, F) \leftarrow$$
where $true$ asserting $\{row(A), col(B), ..., row(E), col(F)\}$

$$\leftarrow R(X, Y)$$
where $rel(R), CommonType(X), CommonType(Y)$ asserting $\emptyset$

DB_0 is an initial database that contains information that the refinement operation can use. In this case, the database starts with a list of the relations that refinement is allowed to use in building a clause. Expressions of the form,

$$\text{where } \langle preconditions \rangle \text{ asserting } \langle postconditions \rangle$$

inform the system about the conditions that must be true before the rule can be applied and the conditions which must apply afterward. The post conditions are added to DB. In this example, the first refinement rule indicates that the head of the clause should be $illegal(A, B, C, D, E, F)$ and the arguments are rows or columns. Rules of the form, $\leftarrow BodyLiteral$, tell the system the form of literals that may be added to the body of the clause being learned. In this example, the rule indicates that literals like $adjacent(X, Y)$, $equal(X, Y)$ and $less\_than(X, Y)$ are legal as long as X and Y are of the same type.

3 Using Prolog to Interpret Refinement Rules

Refinement rules have been embedded in the *iProlog* interpreter [9] so that Prolog itself is used to evaluate the rules rather than using a special purpose theorem prover. In *iProlog*, the above example is written as:

```
    rel(adjacent).          type(row).
    rel(equal).             type(col).
    rel(less_than).

illegal(A, B, C, D, E, F)
    where true asserting row(A), col(B), ..., row(E), col(F).

(:- R(X, Y))
    where rel(R), type(CommonType), CommonType(X), CommonType(Y).
```

Note that *iProlog* has been extended to allow the predicate symbol to be a variable. So that minimal changes can be made to Prolog we insist that the predicate variable must be bound before execution. Thus, we must add the literal $type(CommonType)$, to first bind $CommonType$. Thus, Cohen's language is more powerful in the sense that his theorem prover will search for an appropriate predicate for $CommonType$, however, in *iProlog*, we now have the full power of Prolog to specify intentional background knowledge. We will illustrate this with an example from image processing [11].

The task is to learn to recognise various blood vessels from x-ray angiograms of the cerebral vasculature. After preprocessing, the image is reduced to a skeleton of line segments, each of which represents a segment of a blood vessel. For example, the following predicates describe an internal carotid artery:

```
    internal_carotid_artery(mb1, 1).
    segment(1, mb1, n, 40, 130, [2]).
```

```
segment(2, mb1, w, 40, 144, [3]).
segment(3, mb1, nw, 35, 135, [4, 5]).
segment(4, mb1, n, 40, 50, [6, 7]).
segment(6, mb1, ne, 20, 170, [8, 9]).
segment(5, mb1b1, e, 10, 100, []).
segment(7, mb1b2, w, 5, 125, []).
segment(8, mb1b3, e, 18, 90, []).
segment(9, mb1b4, n, 15, 100, []).
```

The blood vessel $mb1$, which is an Internal Carotid Artery, starts with segment number 1. Each segment is described by an atom with the following arguments: the segment's segment number, the identifier of the blood vessel to which the segment belongs, the direction of the segment (north, north-east, east, south-east, *etc*), the diameter of the segment, the grey-level intensity of the segment and finally, a list of segments that branch from the end of this segment. A segment's start and end points are where there is a branch or a bend in the blood vessel.

In the following example, we will use the refinement rules, not to generate literals for a general-to-specific search, but to create a saturated clause. The head rule is:

```
internal_carotid_artery(VesselName, StartingSegment)
    where
         true
    asserting
         vessel_name(VesselName),
         seg_list(VesselName, [StartingSegment]).
```

Refinement rules are invoked in a forward chaining manner. The head rule matches the predicate $internal\_carotid\_artery(mb1, 1)$. Since there are no pre-conditions, the head of the new clause is created and the predicates

$$vessel\_name(VesselName) \text{ and } seg\_list(mb1, [1])$$

are asserted into the database. This enables the following body rule to construct literals for the segments of the blood vessel:

```
(:- segment(SegId, VesselName, Dirn, Diam, Inten, SegList))
    where
         seg_list(VesselName, S),
         member(SegId, S)
    asserting
         seg_list(VesselName, SegList),
         diameter(VesselName, SegId, Diam),
         intensity(VesselName, SegId, Inten).
```

For each match of the template and for each solution to the preconditions, a new segment literal is created and the corresponding post-conditions are asserted. We

use the *seg_list* predicate to ensure that all segment literals that are generated are "linked", meaning that there is a path from the starting segment to every other segment that appears in the clause. We use the *diameter* and *intensity* predicates to specify the type and origin of the value found in the segment literal.

After execution of this rule, the clause is:

```
internal_carotid_artery(mb1, 1) :-
        segment(1, mb1, n, 40, 130, [2]),
        segment(2, mb1, w, 40, 144, [3]),
        segment(3, mb1, nw, 35, 135, [4, 5]),
        segment(4, mb1, n, 40, 50, [6, 7]),
        segment(6, mb1, ne, 20, 170, [8, 9]).
```

Intentional background knowledge is required if we wish to include in the concept description which segment has, say, the maximum diameter. This is a little tricky because the refinement rule must scan all of the segments to find the maximum value. First, we require a predicate that can find the maximum value of a measurement. We use the second order extension to make this predicate generic for different types.

```
max(M, V, S, X) :-
        findall(M(V, S, N), M(V, S, N), L),
        max(L, M(V, S, X)).

max([X], X) :- !.
max([M(V, SA, A)|B], M(V, S, X)) :-
        max(B, M(V, SY, Y)),
        (A > Y -> S = SA, X = A; S = SY, X = Y).
```

This can be read as: the maximum value of measurement M in blood vessel, V, occurs in segment, S, and has value, N. The predicate can be invoked from the body rule:

```
(:- max(M, V, S, N))
    where
        measurement(M),
        vessel_name(V),
        max(M, V, S, N).

measurement(diameter).
measurement(intensity).
```

A corresponding rule can be defined for the minimum value. After execution of these rules, the following literals are added to the clause:

```
    max(diameter, mb1, 1, 40)
    max(diameter, mb1, 2, 40)
```

```
max(diameter, mb1, 4, 40)
max(intensity, mb1, 6, 170)
min(diameter, mb1, 6, 20)
min(intensity, mb1, 4, 50)
```

Using Prolog to scan all examples gives us the same capability as Srinivasan's lazy evaluation [10] to accumulate values to pass to a predicate that performs some kind of "global" data analysis. In the case of Srinivasan and Camacho, this was a regression algorithm. *iProlog* includes a number of other data analysis tools (see [9]).

We now describe in some detail how refinement rules are evaluated.

3.1 Evaluating Refinement Rules

A head rule has the form:

Literal where *precondition* asserting *postcondition*.

where *precondition* is any legal Prolog expression as would appear in the body of a clause. *Postcondition* is a conjunction of literals that will be asserted into Prolog's data base.

Training examples are given as ground unit clauses in Prolog's data base. When the refinement rules are invoked by a call to the *refine* built-in predicate, the head rule is invoked. A new clause is created such that *Literal* is the head of the clause, provided that the preconditions are satisfied. The preconditions are tested by invoking the Prolog interpreter to try to find a proof in exactly the same way that it would execute a Prolog program. Often the precondition for a head rule is simply true. Once the clause has been created, each literal in the postcondition is asserted, temporarily, into Prolog's data base. All postconditions are retracted after the refinement process has completed execution.

A body rule has the form:

(:- *Template*) where *precondition* asserting *postcondition*.

Note that the parentheses are needed simply so that the Prolog parser does not confuse this with a syntactically incorrect clause.

Once the head rule has completed execution, the refinement rule interpreter begins cycling through each body rule in text order. The following actions are performed:

1. The preconditions and the template are conjoined to form a single conjunction of literals, C.
2. C is tested by running it as a query to the Prolog interpreter.
3. If C is satisfied, the postconditions are temporarily asserted into Prolog's data base.
4. The literal resulting from satisfying $Template$ is added to the body of the clause.

5. If there is more than one solution to the query C, backtracking finds all solutions and therefore generates more than one literal to add to the clause.

Note that all literals added to the body of the new clause must be calls to defined Prolog predicates.

Like a forward chaining rule interpreter, the refinement rule interpreter repeats its cycle until no more new literals can be found. The result of this process is a saturated clause based on one of the training examples. This may be used as the "bottom" clause in a Progol-style search [3]. If the refinement process is applied to all of the positive training examples, the resulting set of clause can be used to form LGG's as with Plotkin [5] or in GOLEM [2].

The biggest problem with LGG's is that many redundant literals are usually created. In the next section we will see how refinement rules can be used to *filter* the LGG, eliminating unwanted literals.

4 Constrained RLGG's

We follow an approach similar in spirit to the constrained atoms of Page and Frisch [4]. Since refinement rules impose restrictions on the form of literals that can be generated through saturation, it is reasonable to apply the same restrictions to literals constructed by generalisation. Thus, we modify Plotkin's LGG algorithm to filter literals so that whenever an LGG of two literals is found, it is tested against the refinement rules. If no refinement rule is satisfied, the new literal is discarded. As a result, the RLGG's do not grow to impractical sizes.

Assuming the new literal is L, the refinement test proceeds as follows:

1. Search the list of refinement rules $\langle X, Pre, Post\rangle$ (for the head literal) and $\langle \leftarrow X, Pre, Post\rangle$ (for the body literals) to find a rule where X unifies with L. If no match is found, the test fails.
2. If a matching refinement rule has been found for L, invoke the Prolog interpreter to test the preconditions, Pre. If the proof fails, the refinement check fails.
3. If the preconditions are met, temporarily assert L into Prolog's data base. This is necessary so that subsequent tests of other literals can refer to L.
4. Assert the postconditions.
5. Return with a success.

One further variation of the LGG algorithm is required. When inverse substitutions are created and variables are inserted into the terms, each variable is temporarily bound to a unique dummy constant. This is necessary so that when L is asserted into the data base, it has constant values consistent with the variable bindings of other literals in the clause, rather than unbound variables that match anything.

To illustrate how refinement rules can reduce the number of literals in an RLGG, let us continue with the medical imaging example. Suppose we have the following two training instances:

```
internal_carotid_artery(mb1, mb1_1).
segment(mb1_1, mb1, n, 40,  130, [mb1_2]).
segment(mb1_2, mb1, w, 40,  144, [mb1_3]).
segment(mb1_3, mb1, nw, 35, 135, [mb1_4, mb1_5]).
segment(mb1_4, mb1, n, 40,  50, [mb1_6, mb1_7]).
segment(mb1_6, mb1, ne, 20, 170, [mb1_8, mb1_9]).
segment(mb1_5, mb1b1, e,  10,  100, []).
segment(mb1_7, mb1b2, w,   5,  125, []).
segment(mb1_8, mb1b3, e,  18,   90, []).
segment(mb1_9, mb1b4, n,  15,  100, []).

internal_carotid_artery(pk2, pk2_1).
segment(pk2_1, pk2, n, 35, 70, [pk2_2]).
segment(pk2_2, pk2, ne, 40, 100, [pk2_3]).
segment(pk2_3, pk2, w, 40, 120, [pk2_4]).
segment(pk2_4, pk2, n, 50, 40, [pk2_5, pk2_6]).
segment(pk2_5, pk2, nw, 45, 50, [pk2_7, pk2_8]).
segment(pk2_7, pk2, ne, 20, 120, [pk2_9, pk2_10]).
segment(pk2_6, pk2b1, e,  10,  150, []).
segment(pk2_8, pk2b2, w,   5,  175, []).
segment(pk2_9, pk2b3, e,  16,  130, []).
segment(pk2_10, pk2b4, n,  14,  140, []).
```

Using the refinement rules described in section 3, we obtain the following two saturated clauses. Note that at this point we have not yet created any variables but retain all constants.

```
internal_carotid_artery(mb1, mb1_1) :-
        segment(mb1_1, mb1, n, 40, 130, [mb1_2]),
        segment(mb1_2, mb1, w, 40, 144, [mb1_3]),
        segment(mb1_3, mb1, nw, 35, 135, [mb1_4, mb1_5]),
        segment(mb1_4, mb1, n, 40, 50, [mb1_6, mb1_7]),
        segment(mb1_5, mb1b1, e, 10, 100, []),
        segment(mb1_6, mb1, ne, 20, 170, [mb1_8, mb1_9]),
        segment(mb1_7, mb1b2, w, 5, 125, []),
        segment(mb1_8, mb1b3, e, 18, 90, []),
        segment(mb1_9, mb1b4, n, 15, 100, []),
        max(diameter, mb1, mb1_4, 40),
        max(intensity, mb1, mb1_6, 170),
        min(diameter, mb1, mb1_6, 20),
        min(intensity, mb1, mb1_4, 50).

internal_carotid_artery(pk2, pk2_1) :-
        segment(pk2_1, pk2, n, 35, 70, [pk2_2]),
        segment(pk2_2, pk2, ne, 40, 100, [pk2_3]),
        segment(pk2_3, pk2, w, 40, 120, [pk2_4]),
        segment(pk2_4, pk2, n, 50, 40, [pk2_5, pk2_6]),
```

```
        segment(pk2_5, pk2, nw, 45, 50, [pk2_7, pk2_8]),
        segment(pk2_6, pk2b1, e, 10, 150, []),
        segment(pk2_7, pk2, ne, 20, 120, [pk2_9, pk2_10]),
        segment(pk2_8, pk2b2, w, 5, 175, []),
        segment(pk2_9, pk2b3, e, 16, 130, []),
        segment(pk2_10, pk2b4, n, 14, 140, []),
        max(diameter, pk2, pk2_4, 50),
        max(intensity, pk2, pk2_7, 120),
        min(diameter, pk2, pk2_7, 20),
        min(intensity, pk2, pk2_4, 40).
```

Since there are 9 segment literals in the first clause and 10 in the second, a simple LGG of these clauses would result in 90 segment literals in the new clause. However, using the same refinement rules as filters on the LGG, we obtain:

```
internal_carotid_artery(_0, _1) :-
        segment(_1, _0, n, _2, _3, [_4]),
        segment(_4, _0, _5, 40, _6, [_7]),
        segment(_7, _0, _8, _9, _10, [_11 | _12]),
        segment(_11, _0, n, _13, _14, [_15, _16]),
        segment(_15, _0, _17, _18, _19, [_20, _21]),
        segment(_16, _22, _23, _24, _25, []),
        segment(_20, _26, _27, _28, _29, _30),
        segment(_21, _31, _32, _33, _34, []),
        max(diameter, _0, _11, _13),
        min(intensity, _0, _11, _14).
```

which is a considerable improvement on the unconstrained LGG.

5 A Small Trial

It is unlikely that we will ever have a large number of x-ray images as training examples since it is very time consuming for radiologists to provide fully labelled images. Therefore the mode of operation in this application is more one of "programming by example" than data mining. A small trial has been conducted on a sample of 10 x-ray images of the internal carotid artery and 11 negative examples obtained from images of different blood vessels or images from different views.

Apart from the predicates already described, the background knowledge includes the concepts of left and right turns (i.e. changes in direction from one segment to the next); left and right branches (i.e. different blood vessels emanating from the example vessel) and counting predicates providing the number of turns and branches.

The refinement rules were applied to all 10 positive examples, obtaining 10 saturated clauses. The constrained LGG of these clauses produced the following clause:

```
internal_carotid_artery(_0, _1) :-
        segment(_1, _0, n, _2, _3, [_4 | _5]),
        left_turns(_0, _6),
        right_turns(_0, _7),
        left_branches(_0, 2),
        right_branches(_0, 2).
```

This describes a blood vessel that has an initial segment heading North. It has some left and right turns and exactly two left branches and two right branches. While the size of the training set is extremely small, this trial demonstrates that refinement rules can be effective in constraining the LGG to produce a concise and understandable concept description.

Acknowledgments

Much of the motivation for this work came from the medical imaging application described here. Thanks to Tatjana Zrimec for providing the x-ray images, the preprocessing software and the domain knowledge needed for this problem. Thanks also to Mike Bain for many helpful discussions on refinement rules.

References

1. Cohen, W.: Learning to classify English text with ILP methods. In L. de Raedt (Eds.), Advances in Inductive Logic programming. IOS Press (1996) 124–143
2. Muggleton, S., Feng, C.: Efficient induction of logic programs. In First Conference on Algorithmic Learning Theory. Omsha, Tokyo (1990)
3. Muggleton, S.: Inverse Entailment and Progol. New Generation Computing **13** (1995) 245–286.
4. Page, C. D., Frisch, A. M.: Generalization and Learnability: A study of constrained atoms. In S. Muggleton (Eds.): Inductive Logic Programming. Academic Press (1992) 29–61
5. Plotkin, G. D.: A further note on inductive generalization. In B. Meltzer and D. Michie (Eds.): Machine Intelligence 6. Elsevier, New York (1971)
6. Quinlan, J. R.: Learning Logical Definitions from Relations. Machine Learning **5** (1990) 239–266
7. Rouveirol, C., Puget, J.-F.: Beyond Inversion of Resolution. In Proceedings of the Seventh International Conference on Machine Learning. Morgan Kaufmann (1990)
8. Sammut, C. A., Banerji, R. B.: Learning Concepts by Asking Questions. In R. S. Michalski Carbonell, J.G. and Mitchell, T.M. (Eds.): Machine Learning: An Artificial Intelligence Approach, Vol 2. Morgan Kaufmann, Los Altos, California (1986) 167–192
9. Sammut, C.: Using background knowledge to build multistrategy learners. Machine Learning **27** (1997) 241–257
10. Srinivasan, A., Camacho, R.: Experiments in numerical reasoning with Inductive Logic Programming. Journal of Logic Programming (in press)
11. Zrimec, T., Sammut, C.A.: A Medical Image Understanding System. Engineering applications of Artificial Intelligence **10**(1) (1997) 31–39.