# Term Rewriting Systems

November 13, 2018

In this tutorial, you will be asked to implement a Term Rewriting System (TRS). The input of a TRS is a *term* called the *input term* and a set of *rewrite rules*.

Terms are something you already know – a *constant* is a term (e.g. `harry`, `william`, `[]`). Moreover any other *function symbol*, whose arguments are terms, is also a term (e.g. `f(a)`, `f(f(f(X)))`, `'.'(a,[])`, `[a,b]`, `[a,[b,[[[c]]],d,[e],f]]`). In Prolog, you have used terms as arguments to predicates.

A rewrite rule has the form $l \to r$, where $l$ and $r$ are terms. Using a procedure called *matching*, TRS finds a place inside the input term, which can be unified with $l$ and replaces it with $r$. For example a rewite rule `f(X)` $\to$ `g(X)` applied to the input term `f(f(a))` can yield `g(f(a))`, but also `f(g(X))`.

For more background on TRS, please consult Wikipedia article on TRS.

## 1 Match & replace

Your first task is to implement the `match` predicate. Given a set of rewrite rules and an input term, it applies at least one of the rules to at least one of the applicable locations in the input term. Represent a rewrite rule as a term `rule(`$l$`,`$r$`)`.

**Example.**

```
?- match([rule(f(X),g(X))], f(f(a)), Out).
Out = g(f(a)) ;
Out = f(g(a)) ;
...
```

Note that the two answers above is the **minimum** you should get on this input. If you get more correct answers, such as `Out = g(g(a))`, that's fine. There won't be any effect on the assignment's evaluation. However, I would suggest not to include that one, because you might loose termination (see below).

You may even receive the answer `Out = f(f(a))`, which means "no rule applied". That's also fine, no effect on evaluation. In fact, I would suggest to include that one, because such implementations are simpler (implies more readable and debuggable).

**Hint #1.** Using unification only, you can find the left-hand-side $l$ of a rule as the top-most symbol in the input term. In other words, getting the first answer in the example above (`Out = g(f(a))`) is easy. How do you reach the "internal" `f(a)`?

You are supposed to use the `=..` predicate (see documentation), which decomposes a term into a list of arguments. And lists are something you already know how to work with. For example:

```
?- f(f(a),g(h)) =.. L.
L = [f, f(a), g(h)].
```

Can you see the arguments being placed in the list `L`?

Note that the predicate works both ways:

```
?- T =.. [f, f(a), g(h)].
T = f(f(a), g(h)).
```

**Hint #2.** You may encounter a strange situation, when a variable `X` in the rule `[rule(f(X),g(X))]` is also instantiated:

```
?- match([rule(f(X),g(X))], f(f(a)), Out).
Out = g(f(a)) ,
X = f(a) ;
Out = f(g(a)) ,
X = a ;
...
```

So far, it is not a huge issue, but this behavior will be problematic in the next task. You may want to use `copy_term` (see documentation) to create a "copy" of a rewrite rule just before it's applied.

```
?- copy_term(rule(f(X),g(X)), R).
R = rule(f(_G302), g(_G302)).
```

Note that `R` is the same rewite rule as the original one (even with variable bindings), merely no variables between the original and the copy are shared. You can even split the copy into the left-hand-side and the right-hand-side:

```
?- copy_term(rule(f(X),g(X)), rule(L,R)).
L = f(_G2695),
R = g(_G2695).
```

Now, feel free to unify `L` with a subterm of the input term and replace it with `R`! Job done.

**Termination.** Not all sets of rewrite rules terminate. Consider rules `[rule(f(X),g(X)), rule(g(X),f(X))]`, which essentialy encode that `f` and `g` are completely interchangeable symbols. Your implementation must terminate on such inputs.

I suggest to ensure termination by applying **one** rewrite rule in **one** location in the input term. An example output of such an implementation, which can be done within 12 lines of code, is:

```
?- match([rule(f(X),g(X)),rule(g(X),f(X))], f(g(a)),Out).
Out = g(g(a)) ;
Out = f(f(a)) ;
Out = f(g(a)) ;   % no rule applied here
false.
```

**Evaluation.**

- Implementation, which meets the basic requirements gets 3 points.
- Moreover, a terminating implementation get 1 additional point.
- Implementation, which does not bind variables in the set of rules, gets 1 additional point.

# 2 Fixpoint

Fixpoint is a term, rewritten to such a degree that no rewrite rule can be applied to it. For example, given a single rewrite rule `rule(f(X),g(X))`, the input term `f(f(f(a)))` can be rewritten to `g(f(f(a)))`, then to `g(g(f(a)))` and finally to `g(g(g(a)))`. The last term can not be rewritten and hence it is a fixpoint.

In the second task, using the `match` procedure, find *any* term, which is a fixpoint from a given input term. Your predicate fixpoint should work as follows:

```
?- fixpoint([rule(f(X),g(X))], f(f(f(a))), Out).
Out = g(g(g(a))).
```

**Determinism.** In the example above, there is only 1 fixpoint. However, that might not be the case always. If there are more fixpoints, you can return any of them. For example, the answer to

```
?- fixpoint([ rule(f(X),g(X)),
              rule(f(X),h(X)) ],
    f(f(a)), Out ).
```

can be either `g(g(a))`, `g(h(a))`, `h(g(a))` or `h(h(a))`. Any of these answers is correct.

The only requirement is that the predicate always returns exactly 1 answer. Even if no rule applies, it must return the input term.

**Termination.** Feel free to assume that the given set of rules never leads to an infinite chain of rewrites. For example, assume that rules `[rule(f(X),g(X)), rule(g(X),f(X))]` will never be given to `fixpoint`.

**Testing.** If not sure whether your implementation is correct, try emulating the `flatten` predicate:

```
?- fixpoint([ rule( [[A|B]|C], [A|[B|C]]) ,
              rule(    [[]|X],     X    ) ],
            [[a],b,[[[c],d]]], Out).
Out = [a, b, c, d].
```

Another application is to find a conjunctive-normal-form of a propositional formula:

```
?- fixpoint([
        rule( not(not(X)),     X                    ),
        rule( not(or(X,Y)),    and(not(X),not(Y))   ),
        rule( not(and(X,Y)),   or(not(X),not(Y))    ),
        rule( or(and(X,Y),Z),  and(or(X,Z),or(Y,Z)) ),
        rule( or(X,and(Y,Z)),  and(or(X,Y),or(X,Z)) )],
    not(or(not(a), and(not(b),c))), Out).
Out = and(a, or(b, not(c))).
```

**Evaluation.**

- Implementation, which meets the basic requirements gets 3 points.
- Moreover, a deterministic predicate gets 2 additional points.

# 3 Search

Finally, implement the `search` predicate that can derive all terms, that can be obtained from the input term by a *sequence of applications* of rewrite rules. For example:

```
?- search([ rule(f(X),g(X)),
            rule(g(X),f(X))],
          f(g(a)), Out).
Out = f(g(a)) ; % optional
Out = g(g(a)) ;
Out = f(f(a)) ;
Out = g(f(a)) ;
false.
```

Please note that these two rewrite rules are not terminating (`f(X)` $\rightarrow$ `g(X)` $\rightarrow$ `f(X)` $\rightarrow \ldots$). You can handle this e.g. by using a closed-list.

**Notes.** The first answer in the example above (when no rule has been applied) is optional. If the input term is or is not among the answers, the evaluation of your implementation will not be affected.

The same applies if any answer is given more than once. Just make sure that your implementation produces all 3 distinct answers in the example above (`g(g(a))`, `f(f(a))`, `g(f(a))`) *at least once*.

**Transitive closure.** The set of all terms which can be rewritten from the input term is called a *transitive closure* of a TRS. Consider rewriting `f(a)` using two rules `rule(f(X),g(X))` and `rule(g(X),f(X))`. The

transitive closure has 4 terms: `f(g(a))`, `g(g(a))`, `f(f(a))`, `g(f(a))`). In this case, the transitive closure is finite.

However, that is not always the case. Consider the trivial example of rewriting `f(a)` using the `rule(f(X), f(f(X)))`. The transitive closure is infinite: `f(a)`, `f(f(a))`, `f(f(f(a)))`, ....

**Termination.** Make sure that your implementation terminates even if the transitive closure is infinite. For example, the following query must terminate:

```
?- search( [ rule(f(X), g(f(X))),
             rule(g(X), f(g(X)))],
           f(g(a)),
           f(g(f(g(f(g(a))))))
         ), !.
true.
```

This probably means that you are going to implement bredth-first-search. Nevertheless, feel free to use a search strategy of your choice. Iterative-deepening depth-first-search is another option.

**Evaluation.**

- Implementation which is terminating on instances with a finite transitive closure will get 3 points.
- Implementation which is terminating also on instances with an infinite transitive closure will get 5 points.