IMS 2015
12th International Mathematica Symposium

Extending *Mathematica* with anti-patterns

Mircea Marin
West University of Timisoara, Timisoara 300223 Romania

Abstract

Pattern matching is a widely spread mechanism to search for relevant data by specifying a pattern which specifies the features we are interested in. We usually specify the features of the objects we want to match, but there are many situations when we want to of exclude objects with certain characteristics.
Antipatterns are an extension of the notion of pattern that allows to express negative information in a via a complement symbol, and offer a compact and expressive representation for sets of terms. This concept was proposed by Claude Kirchner and his coworkers in 2007, and it turns to be useful in programming languages (especially rule-based programming) and for symbolic computation.
We developed a small package, called `Antipatterns``, that enables the direct use of anti-patterns in *Mathematica* programs. We describe the capabilities of our package and illustrate its usage on a simple example.

Introduction

Patterns are a formalism which is widely used in all areas of computer science to describe objects which are of interest to us. Negation is specific to human thinking, and most of the time the characterization of the objects we are looking for is a combination of both positive and negative conditions. For example, we would like to search for cars which have certain positive features (e.g., color, model type) but which lack some undesirable features (e.g., engine type, year of fabrication). Although negation is a characteristic of human thinking, the standard notion of pattern used in languages with rule-based programming capabilities, including *Mathematica*, is designed to describe only positive conditions. This limitation has been recognized by many computer scientists, who proposed several interesting extensions of the notion of pattern with capabilities to express negative information.

Anti-patterns **[3.4]** are a natural extension of the notion of pattern with a unary complement operator denoted by `Not`. The anti-patterns produced by applying the complement operator to a pattern describe negative features, that is, features that should not be satisfied by some part of the term.

To illustrate the expressive power of this formalism, consider the pattern `f[x_,x_]`. This pattern matches all terms of the form `f[t,t]`. The anti-pattern `Not`[f[x_,x_]] matches all terms *except* those matched by the pattern `f[x_,x_]`.

As noticed in **[2]**, the expressive power of anti-patterns becomes much more interesting when anti-patterns are nested inside each other. For example, if `f[x_,y_]` is used to describe cars with brand `x` and color `y`, and `watch[x_,y_,z_]` is used to describe watches with brand `x`, case made of material `y`, and bracelet made of material `z`, then the anti-pattern `Not[car[Not[Fiat],y_]` describes all objects that are either not cars (e.g., all watches), or all cars whose brand is not Fiat. In general, the use of nested negations allows to express complex conjunctions and disjunctions of conditions that must be fulfilled by the terms matched by an anti-pattern.

Anti-patterns become more useful when used in the specification of rule transformations, or rewrite rules, in a rule-based programming language. We propose the notion of *anti-rule* as a natural generalization of the notion of rewrite rule, which can be used in rule-based programming. In contrast to rewrite rules, anti-rules can have an anti-pattern to their left-hand side. As a result, a rule-based programming language with anti-rules allows to reason and work with both positive and negative characterizations of objects.

Our main contribution of is the specification of an algorithm which translates anti-rules into conditional rewrite rules which perform the same rewriting steps, and an implementation of this algorithm in a package, called `Antipatterns``, that enables rule-based programming with anti-rules in *Mathematica*, by translating them into conditional rewrite rules. The motivation for this work was the lack of anti-pattern capabilities in *Mathematica*.

The rest of this paper is structured as follows. In the first section we introduce the syntax and semantics of antipatterns. Next, we introduce the notion of anti-rule for the purpose to bring the expressive power of anti-patterns to rule-based programming. Next, indicate our main contribution: an algorithm to translate anti-rules into conditional rewrite rules. Our translation scheme opens the possibility to extend with anti-rules the capabilities of rules-based programming languages with support for conditional term rewriting. The third section describes our implementation of this translation algorithm as an add-on package of *Mathematica;* it provides a translator of anti-rules into corresponding *Mathematica* conditional rules.

Syntax and interpretation of anti-patterns

One of the most remarkable strengths of *Mathematica*'s core language is its powerful and succint—yet highly readable—symbolic pattern language. *Mathematica*'s patterns are a natural generalization of regular word expressions to describe classes of symbolic structures, with immediate use for defining individual functions and transformation rules.

We assume the reader is familiar with the syntax and semantics of patterns as described in the User Guide for the core language of *Mathematica*. From now on we wil use the notation ⟦`ptn`⟧ to denote the set of expressions matched by a *Mathematica* pattern *ptn*.

Syntax

Anti-patterns ar a simple—yet highly powerful—extension of patterns: at the syntactic level, the only visible difference is the usage of a distinguished unary operator `Not`, whose argument must be a *Mathematica* pattern or anti-pattern. Their syntax can be defined recursively as follows:

$$aptn ::= ptn \;\big|\; \texttt{Not}[aptn] \;\big|\; aptn[aptn\,']_p$$

where `ptn` denotes an arbitrary *Mathematica* pattern, and $aptn[aptn\,']_p$ denotes the result of replacing the subexpression of `aptn` at position *p* with the anti-pattern `aptn'`. A *variable* pattern is of the form `x_` where *x* is a *Mathematica* symbol.

For example, the following expressions are anti-patterns:

```
watch[Not[Rolex],bracelet[x_],case[Not[x]]]
f[x_,Not[g[y_,x,Not[h[y]]]]]
Not[Not[f[x_,x_]]]
```

The *complement depth* of a subexpression *e* in an antipattern `aptn`, denoted by $\texttt{cdepth}_{aptn}(e)$, is the number of occurrences of `Not` along the path from the root position of `aptn` to the position where *e* occurs in `aptn`. For example, the complement depths of `x_`, `x`, and `h[y]` in `f[x_,Not[g[y_,x,Not[h[y]]]]]` are 0, 1, 2 and respectively.

We also require that anti-patterns should not contain occurrences of the same variable pattern at positions separated by an occurrence of the complement operator `Not`. For example, `f[x_,Not[g[y_,x,Not[h[y]]]` is a valid anti-pattern, but `f[x_,Not[g[y_,x,Not[h[y_]]]` is not because the variable pattern `y_` occurs at positions {2,1,1} and {2,1,3,1,1}, and `Not[...]` occurs at position {2,1,3} between them.

We also introduce the following auxiliary notion: given an anti-pattern `aptn`, we define the set `FVar(aptn)` of its *free variables* to be the set of pattern variables with occurrences which are not below an occurrence of `Not[...]`. For example `FVar(f[x_,Not[g[y_,x,Not[h[y]]]]])={x}` because `x` is the only pattern variable that is not below Not in this anti-pattern. Although `y`, occurs in this antipattern, it is not free but local to the complement subpattern `Not[g[y_,x,Not[h[y]]]]`.

The notion of anti-pattern proposed by us is a natural generalization of the notion of anti-patterns for first-order terms proposed by Claude Kirchner in [8].

Semantics

First, we define the semantics of *syntactic* anti-patterns, that is, anti-patterns where all function symbols (except `Not`) have not special attributes (such as being associative, commutative, etc.) In this case, the semantics of an anti-pattern is defined recursively as follows:

⟦`aptn`⟧`::=`⟦`ptn`⟧ if `aptn`≡`ptn` and `ptn` is a normal *Mathematica* pattern.

$[\![aptn[\texttt{Not}[aptn\,']]_p]\!] ::= [\![aptn[\texttt{z\_}]_p]\!] - [\![aptn[aptn\,']_p]\!]$ where

- *p* is a position without occurrences of `Not` above it in `aptn`,
- `z_` is a fresh variable pattern

This semantics captures the intuitive meaning of anti-patterns. For example:

- `Not[f[_]]` matches any term whose root symbol is not `f`, or any term with root symbol `f` and with more than 1 argument.
- `Not[g[x_, x_]]` matches any term, except those of the form `g[t,t]` where *t* is some term.
- `watch[color[Not[red]],bracelet[Not[leather]]]` matches any term of the form `watch[color[c],bracelet[m]]` where `c`≠`red` and `m`≠`leather`.
- `f[Not[g[Not[h[x_]],Not[u[x_]]]]]` matches any term `f[t]` with *t* different from `g[u,v]`, where either *u* is not of the form `h[s]` or *v* is not of the form `h[t]`. For example, this anti-pattern matches the terms `f[a]`, `f[g[h[a],f[b]]]`, but does not match the term `f[g[f[a],f[a]]]`.
- `f[x_,Not[g[Not[h[y]],y_,x]]]` matches any term of the form `f[s,t]` where either *t* is not of the form `g[u,v,s]`, or is of the form `g[h[v],v,s]`. For example, this anti-pattern matches `f[a,f[c,b]]`, `f[a,g[h[b],c,b]]`, and `f[a,g[h[c],c,a]]`.

This last example illustrates the powerful interplay between nested patterns and pattern variables whose values are referred to impose further matching constraints in subsequent complement subpatterns:

- the value of the pattern variable `x_` is referred in the complement subpattern `Not[g[Not[h[y]],y_,x]]`
- the value of the pattern variable `y_` is referred in the complement subpattern `Not[h[y]]`

Matching with anti-patterns becomes more interesting when the terms contain function symbols with reqular equational theories associated. Given an equational theory *E*, we write $s =_E t$ to indicate that *s* and *t* are equal in *E*. The equational theory *E* is called *regular* if $s =_E t$ implies that *s* and *t* contain the same same variables. From now on, we assume implicitly that *E* is a regular equational theory.

First, we recall the standard notion of matching in equational theories: we say that a pattern `ptn` matches a term *t* in *E*, and write

$ptn \ll_E t$, if there exists a matching substitution σ such that the instance of ptn with σ, which we denote by $ptn/.\sigma$, is equal with t in E. Formally, this means that $ptn\ /.\ \sigma =_E t$ for some substitution σ. This notion is generalized to anti-patterns as follows (Definition 13 from [3]): given an anti-pattern $aptn$ with function symbols associated with the regular equational theory E, we say that $aptn$ matches a term t in E, and write $aptn \ll_E t$, if we are in one of the following cases:

Case 1. FVar($aptn$)=$\emptyset$ and either (a) $aptn$ is a pattern and $aptn\ /.\ \sigma =_E t$ or (b) $aptn = aptn[\text{Not}[aptn\ ']]_p$ and $aptn[s]_p =_E t$ for some term s, but $aptn[aptn\ ']_p$ does not match t in E

Case 2. Otherwise, FVar($aptn$) $\neq \emptyset$, σ is a grounding substitution for FVar($aptn$), and aptn/.σ matches t according to Case 1.(b).

To understand the intuition behind **Case 2** of this definition, let's consider the anti-pattern f[Not[a],x_] where f is an associative function symbol (that is, f[s, f[t, u]] = f[f[s, t], u] for all terms s, t, and u.) Also, let's consider the term f[a,f[b,c]]. The anti-pattern $aptn$=f[Not[a],x_] is intended to match with matching substitution {x->t} any term which is equal modulo associativity of f to a term of the form f[s,t] where $s \neq_E$ a. Since f[a, f[b, c]] $=_A$ f[f[a, b], c], the substitution $\sigma = \{$x $\to$ c$\}$ is a matcher of $aptn$ with f[a,f[b,c]] because $aptn\ /.\ \sigma$ = f[Not[a], c] matches with f[f[a,b],c]. Note that the alternative definition

Case 2'. If FVar($aptn$) $\neq \emptyset$ $aptn = aptn[\text{Not}[aptn\ ']]_p$, then $(aptn.\ /\ \sigma)[s]_p =_E t$ for some term t, but but $aptn[aptn\ ']_p$ does not match t in E

gives the counterintuitive answer that f[Not[a],x_] does not match f[a,f[b,c]].

Programming with anti-rules

The most straightwforward to program with anti-patterns is to use them as left-hand sides of transformation rules. Formally, an anti-rule is a transformation rule of the form

$$aptn \to rhs$$

where $aptn$ is an anti-pattern and rhs is a term that specifies the symbolic structure of the transformation from the bindings of the variables that occur free in $aptn$.

For example, the application of the anti-rule

$$f[x\_,\ z : \text{Not}[g[\text{Not}[h[y]],\ y\_,\ x]]] \to \{x,\ z\}$$

to the term f[a,g[h[c],c,a]] should succeed by computing the matcher $\sigma = \{$x $\to$ a, z $\to$ g[h[c], c, a]$\}$ and the corresponding result {a,g[h[c],c,a]}.

Anti-rules are a relatively new concept to functional and rule-based programming. Rewriting with anti-rules is supported in very few languages, whereas rewriting with conditional rewrite rules is widely used in several programming languages. Therefore, it is desirable to try to translate anti-rules into conditional rewrite rules which perform the same rewrite steps, and to formalize this translation as an algorithm. The purpose of the following subsection is to indicate such a translation algorithm.

Translation of anti-rules into conditional rewrite rules

First, we describe an algorithm to compute matching substitutions between anti-patterns and terms, by means of solving (systems of) *anti-matching equations*, that is, equations of the form $at \ll s$ where at is an anti-pattern and s is some term. In order to solve an anti-matching equation, we define an operation dec($at \ll s$) which computes a system of simpler anti-matching equations by decomposing the anti-matching equation at $\ll$ s. The decomposition is defined as follows:

1. If at is a term t, the newly produced system is

$$\text{dec}(t \ll s) := (t \ll s)$$

2. Otherwise, the set P of positions in at for subterms with root symbol Not is not empty, and we can compute the non-empty set $P_{\min}$ of minimal positions of P w.r.t. prefix order. If $P_{\min} = \{p_1, \ldots, p_k\}$ then

$$\text{dec}(at \ll s) := \left(t \ll s,\ at\,|_{p_1} \ll z_1,\ \ldots,\ at\,|_{p_k} \ll z_k\right)$$

where $z_1, \ldots, z_k$ are fresh variables and $t = at[z_1]_{p_1} \ldots [z_k]_{p_k}$.

In both cases, the system of anti-matching equations produced from an anti-matching equation is of the form

$$(t \ll s,\ \text{Not}[at_1] \ll z_1,\ \ldots,\ \text{Not}[at_k] \ll z_k)$$

We are ready now to define the notion of matcher of an anti-matching equation.

Definition. Let $at \ll s$ be an anti-matching equation, and

$$\text{dec}(at \ll s) = (t \ll s,\ \text{Not}[at_1] \ll z_1,\ \ldots,\ \text{Not}[at_k] \ll z_k)$$

A *matcher* of $at \ll s$ is a substitution θ for the free variabes in at, which the following two conditions hold:

1. The matching equation $t \ll s$ has a matcher σ.
2. $x\,/.\,\theta = x\,/.\,\sigma$ for every free variable x of at, and none of the the anti-matching equations $at_i\ /.\ \sigma \ll z_i\ /.\ \sigma$ has a matcher.

This is, obviously, a recursive definition which can be used to test if $at \ll s$ has a matcher, and to compute it if it exists.

Next, we define the translation of an anti-rule

$$aptn \to rhs$$

Let

$$\text{dec}(\text{aptn} \to \text{rhs}) = (t \ll s, \text{Not}[at_1] \ll z_1, \ldots, \text{Not}[at_k] \ll z_k) \quad (1)$$

This implies that the application of this anti-rule to a whole term u can replace u with $rhs/.\theta$ if and only if

1. $u = t\theta$, *and*
2. For every $1 \le i \le k$, the anti-matching equation $at_i\theta \ll z_i\,\theta$ has no solution.

To check these conditions, it is desirable to have defined the boolean functions `match` and `antiMatch`, such that

- `match[s, t] == True` if and only if the term s matches t.
- `antiMatch[at,s]==True` if and only if the anti-matching equation $at \ll s$ has a matcher.

If `match` and `antiMatch` are defined in this way, then the translation of the anti-rule $aptn \to rhs$ into a conditional rewrite rule with the same rewriting behavior is straightforward: it is

```
t :→ r /; And[antiMatch[atk, zk] == False, …, antiMatch[atk, zk] == False]
```

```
t :→ r /; And[antiMatch[at1, z1] == False, …, antiMatch[atk, zk] == False]
```

where at_i and z_i for $1 \le i \le k$ are determined by equation (1).

Decision algorithms for matching are well-known in the literature, and some of them are known to run in linear time. *Mathematica* has a built-in method `MatchQ[s, t]` which returns `True` if the matching equation $t \ll s$ has a matcher, and `False` otherwise. Therefore, we assume that the function `match` is already available, and focus on finding a suitable definition for the function `antiMatch[at,s]`.

We define `antiMatch` for an anti-pattern $aptn$ indirectly, by constructing a ssytem of conditional rewrite rules R_{aptn} such that

R_{aptn} rewrites the term `antiMatch[aptn, t]` to `true` if and only if the anti-matching equation $aptn \ll t$ has a solution.

R_{aptn} is defined recursively as follows. If

$$\text{dec}(\text{aptn} \to \text{True}) = (t \ll \text{True}, \text{Not}[at_1] \ll z_1, \ldots, \text{Not}[at_k] \ll z_k) \quad (2)$$

then $R_{aptn} = \{\rho_{aptn}\} \bigcup \bigcup_{k=1}^{p} R_{at_i} \bigcup \overline{R_{aptn}}$ where ρ_{aptn} is the conditional rewrite rule

```
antiMatch[t, x_] :→ True /; And[
    match[t, x] == True,
    antiMatch[atk, zk] == False,
    …,
    antiMatch[atk, zk] == False]
```

with `x_` a fresh pattern variable; and $\overline{R_{aptn}}$ consists of the following $p+1$ transformation rules:

```
antiMatch[aptn, x_] :→ False /; match[t, x] == True

antiMatch[aptn, x_] :→ False /; antiMatch[at1, z1]==True

…

antiMatch[aptn, x_] :→ False /; antiMatch[atp, zp]==True
```

The definition of `antiMatch` becomes much simpler if the conditions of rewrite rules are allowed to contain irreducibility tests. In this case, the definition of R_{aptn} is reduced to $R_{aptn}=\{\rho_{aptn}\} \bigcup \bigcup_{k=1}^{p} R_{at_i} \bigcup \overline{R_{aptn}}$ where ρ_{aptn} is the conditional rewrite rule

```
antiMatch[t, x_] :→ True /; And[
    match[t, x] == True,
    antiMatch[atk, zk] ⇥,
    …,
    antiMatch[atk, zk] ⇥]
```

where s⇥ indicates that s is an irreducible term. With this capability, the translation of the anti-rule $aptn \to rhs$ gets simpler too: It is the conditional rewrite rule

```
t :→ r /; And[
    antiMatch[atk, zk] ⇥,
    …,
antiMatch[at_k, z_k] ⇥]
```

Implementation

We implemented an add-on package called `Antipatterns`` that enables the usage of anti-patterns in *Mathematica*, via two methods:

1. `antiMatch[`*`aptn,term`*`]` which yields `True` if *`apatn`* is an anti-pattern that matches term, and `False` otherwise.
2. `AntiRule[`*`aptn`* `→` *`rhs`*`]` which converts the anti-rule *`aptn→rhs`* into a corresponding transformation rule of *Mathematica*.

The implementation is based on the algoritm described in the previous subsection. I illustrate how to use the capabilities of this package to translate the anti-rule

```
f[x_,r1[y_,Not[r2[x, y]]],Not[g[x,z_,h[Not[r3[z,t_]]]]]] → r2[x,y]
```

After loading the add-on package `Antipatterns`` with the command

```
Get["Antipatterns.m"]
```

we can proceed as follows:

```
(* Assign to r the result of translating this anti-rule *)
r = AntiRule[f[x_, r1[y_, Not[r2[x, y]]], Not[g[x, z_, h[Not[r3[z, t_]]]]]] →
   r2[x, y]]
```

```
f[x_, r1[y_, x$1988_], x$1989_] /; ! antiMatch[r2[x, y], x$1988] &&
   ! antiMatch[g[x, z_, h[! r3[z, t_]]], x$1989] :→ r2[x, y]
```

The value assigned to `r` indicates the conditional rewrite rule of *Mathematica* that simulates the rewrite behaviour of the anti-rule given as input to `AntiRule`. Our implementation of `antiMatch[`*`aptn,t`*`]` does not analyze the structure of aptn at definition time; instead, it relies on a method called `AntiMatchQAux`, which is private to the implementation of our package, and is in charge to implement the proper behavior of `antiMatch` by analyzing the structure of its first argument at runtime.

```
antiMatch[p_ /; FreeQ[p, Not], t_] := MatchQ[t, p];
antiMatch[p_, t_] := AntiMatchQAux[t, EliminateAntipatterns[p]];
```

It is interesting to note that our implementation of anti-rules works well with anti-patterns that contain symbols with properties given by regular equational theories, such as associativiy, commutativity, and idempotence. For example, if we declare

```
Attributes[f] = {Flat, OneIdentity};
```

and define the anti-rule

```
r = AntiRule[f[u : Not[a], v_] → {u, v}];
```

then

```
Replace[f[a, f[b, c]], r]
```

```
{f[a, b], c}
```

yields the expected result because `f[a, f[b, c]]` $=_A$ `f[f[a, b], c]` and $\sigma = \{u \to f[a, b], v \to c\}$ is a matcher of the anti-pattern `f[`*`u`*` : Not[a], v_]` with `f[f[a, b], c]`.

Conclusion

Anti-patterns are a recently proposed formalism intended to extend the notion of pattern with the possibility to describe negative conditions as well, in a concise and easy-to-understand way. They have been implemented in the Tom pattern matching language **[1]**, which is particularly well suited for programming various transformations on tree structures and XML based documents. Based on the sucess of Tom and a few other functional programming languages that enable programming with anti-patterns, we believe that it makes sense to extend *Mathematica* with capabilities ot program with them.

The main contribution of this paper is an algorithm to translate anti-rules into conditional rewrite rules of *Mathematica*, that do simulate the rewrtiting behavir of anti-rules. The algorithm was implemented into an add-on package called `Antipatterns``, which empowers the *Mathematica* programmers with the possibility to program with anti-patterns.

References

A. [1] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on Java. In Proceedings of the 18th Conference on Rewriting Techniques and Applications, volume 4533, pages 36–47. Springer, 2007.

B. [2] H. Cirstea, C. Kirchner, R. Kopetz, and P.-E. Moreau. Anti-patterns for rule-based languages. Journal of Symbolic Computation, 45(5):523-550, 2010. Symbolic Computation in Software Science.

C. [3] C. Kirchner, R. Kopetz, and P.-E. Moreau. Anti-pattern matching. In Proceedings of the 16th European Symposium on Programming, volume 4421, pages 110–124. Springer, 2007.

D. [4] C. Kirchner, R. Kopetz, and P.-E. Moreau. Anti-pattern matching modulo. In C. Martın-Vide, editor, Proceedings of the 2nd International Conference on Language and Automata Theory and Applications, vol- ume 5196, pages 275–286, Tarragona, Spain, 2008. Springer.