Rules and Patterns in Maxima

M. Talon revised and enriched by R. Fateman

February 17, 2022

1 Introduction.

We attempt to overcome the abstruse and compact chapter of the Maxima manual devoted to the subject of rules and patterns, and provide a more tutorial-like introduction to the subject. We also try to provide a detailed introduction to the workings of the Lisp language program (source `matcom.lisp`) underlying the Maxima pattern matching system.

These notes are based on the manual and comments by R. Fateman and R. Dodier in the Maxima mailing list and on drafts, as well as experimentation. This is not however, intended to be exhaustive. But, contrary to the manual which presents the various functions in alphabetical order, we try here to build a logical progression, starting from the matching functions and then moving to the replacement functions.

To summarize, in advance: The idea is that by identifying patterns in expressions we can guide a computation to transform those expressions to satisfy some computational need. This is by contrast with a more typical procedure-oriented specification, based on applying functions to expressions. A pattern-directed manipulation of expressions is the principal mode for user-written procedure definitions in Mathematica. Some syntactic sugar in the language tends to hide this from the casual user.

We use extended examples of Maxima programming by declaring patterns and replacements later in this presentation as a method for implementing manipulation of quaternions.

We remind the reader that the methods of pattern-matching and replacement are of necessity transformed into an intermediate conventional language, so they may be interpreted or executed in a conventional sequential manner. That intermediate language is Common Lisp.

These notes are supplemented by a first section describing in general terms what is pattern matching and what are some useful applications, as well as the problems this involves. Some parts of this perspective may need to be somewhat acquainted to the maxima rules and patterns mechanisms. Finally we have added an appendix showing how one can extend the maxima pattern matching tools to tackle problems that are not easily solved using the built in tools, for example in the case of non commutative multiplication, using maxima functions to build new rules.

2 A perspective on pattern matching

Since we do not have a computer algebra system that knows all of mathematics, and it is unlikely that we will ever encounter one, it becomes important for us to be able to add the specialized knowledge that we need to an existing system.

There are several well-known ways of adding information. Here are some of them. Setting values, such as $\pi = 3.14159$ which is only approximately right. Defining functions like absolute value.

```
myabs(x) := if x<0 then -x else x.
```

But that works only if x is an explicit number, so this is probably insufficient for a symbolic system. Maybe

```
myabs(x) := if  numberp(x) then (if x>0 then x else -x)
                           else abs(x) /*symbolic result*/
```

This too does not include all the knowledge we might encode. We know that $f(z)^2$ is positive if $f(z)$ is real, even if we don't specifically know the value of z or the explicit definition of f – just that it maps whatever z might be to real numbers. We might try something like this:

```
myabs(x) := if  numberp(x) then (if x>0 then x else -x)
                           else
            if ``it looks like an even power of an expression
 that can only take on real values'' then x
                           else abs(x) /*symbolic result*/
```

We haven't said how to compute that test, but even so, this too is incomplete. For instance, for real z we know that $\cos z + 1$ is non-negative.

We can continue to elaborate on this function, but there are two issues to notice.

- First, the program is getting more complicated and we might (or others) introduce bugs more cases.
- It might get rather slow, as more and more computation is needed to determine the program flow. For instance, there is a program, `sqfr` which can tell us if polynomial p can be represented as q^2 for some polynomial q. It is a non-trivial computation though, and you might not wish to use it if it is likely to be unsuccessful.
- The description of what the program actually does may become unwieldy, and even an attentive user might not be able to figure out what the program does in any particular case. In fact, the documentation might diverge from the actual program, as it is "improved".

There are several other ways of organizing the code, but one that seems to be particularly attractive is the use of patterns.

Let us start by defining testing programs (called predicates: a program that returns true or false) like this:

```
nonnegnum(p) := is (numberp(p) and p>=0)
```

and associate this predicate what we call a pattern variable, `nnp` this way:

```
matchdeclare(nnp,nonnegnum)
```

Then we can add a rule to our system via `tellsimp` to teach it how to simplify `myabs`. We will explain `tellsimp` and other programs in more detail later, but for the moment, here is how we say that if the simplifier encounters `myabs(43)` where 43 is `nonnegnum`, then `myabs(43)` should simplify to 43.

```
tellsimp( myabs(nnp), nnp)$
```

Therefore `myabs(43)` returns 43. `myabs(-43)`returns `myabs(-43)` because there is no tellsimp rule for that case.

The next part could fix this:

```
negnum(p) := is (numberp(p) and p<0)$
matchdeclare(negp, negnum)$
tellsimp (myabs(negp), -negp)$
```

now `myabs(-43)` returns 43.

At this point, `myabs(z^2)` will not trigger any rules and so will be "simplified" to `myabs(z^2)` by just leaving it alone.

Define another predicate:

```
squared(p):=  /* return true if p can be expressed as an
even power of a real-valued expression, or a product of even powers of
real-valued expressions otherwise false.*/

matchdeclare (sp2,squared)$

tellsimp(myabs(sp2), sp2)$
```

Even though this is beginning to look complicated, we can still try to identify individual "facts" with particular tellsimp rules. If we are careful, the rules are mutually independent – they don't overlap. This can be tricky, but it is a way of tamping down bugs.

It is also possible to define match programs that just return a list of bindings, using `defmatch`, or rule programs via `defrule` that are independent of the simplifier, and are independently applied.

An example which is of mathematical interest is the following:

This `mysabs` example, even as it trends toward complexity, does not illustrate a particular hazard that is often encountered: where a pattern expression is too general and the user can expect one set of matches, but the matching program in the computer identifies a different way. This is especially of concern for patterns involving addition and multiplication, where the user's expectations may be at odds with the implementation in the computer system. For instance, assume pattern variables `a,b` are declared "true" and can match anything. We can imagine that a pattern `u*v` might match an expression `3*z` as `[u=3,v=z]` but also `[u=z, v=3]`. We might be surprised to see it match `[u=1, v=3*z]`, but this is mathematically consistent. More surprisingly, it can even match `[u=45, v=3*z/45]`. How could that last one be proper? Consider the pattern `f(u,u*v)` matching the expression `f(45,3*z)`. Indeed,

```
matchdeclare([u,v],true)$
defmatch(m1, f(u,u*v))$
m1(f(45,3*z));
```

returns `[u=45,v=3*z/45]`

Sometimes it is clear that the pattern can match a given expression in only one way. In other cases there are multiple matches possible,and this is a problem. The Maxima program will return the first and only one it finds, and that might be different from the user's expectation.

One way out of this is for the matching program to generate another assignment of variables still consistent with predicates, if the previous assignment fails, etc. This is called "backtracking"and can be expensive in some cases. It can be useful and sufficiently efficient in small expressions.

One computer algebra system, Mathematica, uses matching with backtracking. Detailed knowledge of the matching program (whether in Maxima or Mathematica) can help avoid wrong turns. This may require an understanding of the ordering of terms in sum or product expression, or other details. A better way is to define a *collection of patterns* so carefully that the result is that only one pattern, the uniquely right one, matches any given expression.

A natural question to ask is whether pattern-replacement rules are "better" than conventional programs. This has a formal mathematical answer, as follows: In principle it is possible to perform any computation that can be done in a conventional sequential language using a collection of pattern-replacement rules. Or vice versa. [1] The choice of technique may be a matter of personal taste, convenience, efficiency of implementation and simplicity or readability of notation.

The expected use of pattern matching, to augment the simplifier, or to substitute for writing conventional programs that define some new mathematical function, can be straightforward if the pattern variables are used to essentially check the properties of individual arguments to a function. It is more difficult to use if the patterns involve segmenting terms in a sum or product, because these operators have the algebraic properties of commutativity and associativity. There is an alternative which deliberately recognizes this, and more explicitly partitions such expressions[2]

[1]A formalization of a primitive notion of pattern-replacement rules, Semi-Thue systems, is provably "Turing complete". That is, it is equivalent in computational power to a conventional programming language in that each can be used to simulate the other, or to compute anything that is conventionally computable. Another formalism related to the notion of pattern matching is logical "unification". This is a process of solving symbolic equations, but where symbols can occur on both sides.

[2]Partitioning of Algebraic Subexpressions in Computer Algebra Systems: Or, Don't use matching so much.

Here's a brief introduction to that, using the partitioning program as part of the predicate for the match variable `pp`. (You may skip this section on first reading.)

```
load(partition),
(matchdeclare (pp, partition_expression("+",constantp,0,"+",g,'ANSpp)),
tellsimp(foo(pp),ANSpp),
declare([a,b,c],constant), /* put "constant" property on symbols a,b,c
*/
/* try it out */
[foo(a+b+c+x+y+3),foo(w),foo(b), foo(a*b+x),foo(a*b)]);

-->
[g(c+b+a+3,y+x),foo(w),foo(b),g(a*b,x),foo(a*b)]

notes:
foo(a+b+c+x+y+3) shows that it works.
foo(w) is unchanged because w is not a sum.
foo(b) is unchanged because b is not a sum, even though b is constant
foo(a*b+x) shows that Maxima knows a*b is a constant if a,b, are each
constant
foo(a*b) again is not a sum..
```

That predicate for `pp` uses the program `partition_expression` which partitions an expression E that is a sum, choosing all those pieces that can pass the test `constantp`, and starting with the initial value 0, adds them all together to get a result S. The program then sets the global variable `ANSpp` to `g(S,E-S)` and returns true. Why is this global variable idea in there? Pattern matches must return true or false, indicating a match or non-match. If the replacement is returned, that's ambiguous – what if the desired replacement is literally false? We can't return that (i.e. we can't say "the pattern matched and we return the result false".) So we say "the pattern succeeded and the result is stored in some variable."

Just to review: Here the intent is to separate components of a sum E into two parts: those items that are constant, S, versus those that are not. Each of the two parts will be a sum, initially zero. The result will be g(S,E-S), and `ANSpp` will be set to g(S,E-S). If foo is applied to something that is not a sum, it remains unchanged.

If you are continuing to read here, you are probably in agreement that rules and pattern-replacement methods can be useful in computer algebra systems, and want to become more acquainted with them.

3 The matcher.

The first step is to formalize the notion of a match. We have a pattern P which looks like a regular algebraic expression, and an expression E. The pattern P=f(a,b) and the expression E=f(a,b) match, because each component of P and E correspond to the exact same symbols.

Including a neat integration example.
https://people.eecs.berkeley.edu/ fateman/papers/partition-new2.pdf

We next define "match variables", which are wildcards to match and select appropriate parts of the target expressions. The `matchdeclare` function declares such wildcards, each of which is associated with a condition or predicate P.

If we allow the match variable v to match "anything" then the pattern P=f(v,b) and the expression E=f(a,b) match, with the binding [v=a]. The command to make v match any subexpression is matchdeclare(v, true).

example

```
matchdeclare(v,true)$
defmatch(m0, h(v))$
m0(h(a)) ;   returns [v=a]
```

a safer way of writing and using patterns is to make the names local, since the above interaction leaves v assigned the value a. While this is useful, it can be confusing if such assignments are surprises.

```
block( [v],  m0(h(a))
```

Often we want the matching process to require that a variable v can be bound to an expression E only a function q(E) evaluates to true. We have already used the predicate q = true which allows any expression to match. Well, almost.

```
defmatch(m6,h(v,v))$
m6(h(1,1));   returns [v=1]
m6(h(1,2));   returns false
```

That is, the variable v is used twice in the pattern, and it must match the same expression each time. (Note that "same" is a tricky concept for a CAS. Is $\sin^2 x + \cos^2 x$ the same as 1? For the pattern matcher, that's too much work.

Let's try some more interesting predicates for match variables: We set variable `ann` and `bnn` to match arbitrary atoms except numbers (mnemonic: nn= non-number), variable `cna` (not atom) to match non-atoms, and variable `numonly` to match a number. The syntax is

```
matchdeclare(ann,pred1,bnn,pred1,cna,pred2,numonly,numberp);
```

Here `pred1` is a logical function taking one variable which will be applied to `ann` before allowing it to match in a pattern.

```
pred1(x):=atom(x) and not numberp(x);
```

Similarly for `bnn` etc.

For `cna`, we define

```
pred2(x):= not atom(x);
```

For `numonly` we can take simply the built-in predicate `numberp`.

Alternatively, we can group `ann` and `bnn` together within [], and we can avoid the clutter of defining extra names and use lambda functions in the `matchdeclare`:

```
matchdeclare([ann,bnn], lambda([r],atom(r) and not numberp(r)),
              cna,lambda([r], not atom(r)),
              numonly,numberp);
```

An example of more mathematical interest:

```
%i1) apred(r):= if (r#0 and freeof(x,r)) then true else false;
(%o1)          apred(r) := if (r # 0) and freeof(x, r) then true
(%i2) pred(r):= if freeof(x,r) then true else false;
(%o2)                pred(r) := if freeof(x, r) then true
(%i3) matchdeclare(a,apred, b,pred, c,pred);
(%o3)                                done
(%i4) defmatch(m,a*x^2+b*x+c);
(%o4)                                  m
(%i5) m(3*x^2+1);
(%o5)                        [c = 1, a = 3, b = 0]
(%i6) subst(%,[(-b+sqrt(b^2-4*a*c))/(2*a),(-b-sqrt(b^2-4*a*c))/(2*a)]);
                                 %i         %i
(%o6)                        [-------, - -------]
                              sqrt(3)    sqrt(3)
(%i7)
```

Advanced remark: there is a more general form for predicates, which allow them to apply to several wildcards at once. For example, as in the manual:

```
 gt (i, j) := integerp(j) and i < j;$
matchdeclare (i, integerp, j, gt(i))$
```

In this situation, where the predicate for `j` is `gt(i)`, the system appends `j` at the end of the arguments of `gt`, and thus runs `gt(i,j)` so `j` matches only when `j>i`. It is necessary that j be bound before i in the sequence of matching.

Internally the effect of `matchdeclare` is to put information on the property list of the wildcard variables, where that information will be used when "compiling" patterns using those variables. This is the way we can peek under the covers in Lisp to see the property list of a symbol.

```
(%i1) matchdeclare(a,pred1)
(%i2) :lisp (symbol-plist '$a)
(MPROPS (NIL MATCHDECLARE ($PRED1)))
```

A pattern is a Maxima expression generally involving user-named functions like f,g,h, and built-in kernels (such as sin, cos, exp, etc.) all of which are constants in the pattern. Also some of the symbols in the pattern are the declared match variables. Technically, the aim of a pattern matcher is to determine if a given expression can be "unified" with the pattern: that is, it is of the same form as the pattern, with wildcards substituted by actual values, with other variables and constants matched by themselves. [3]

To step through this matching in Maxima let us use the `defmatch` function that will be described in more detail later on, and we've already used it in passing. In its most simple incarnation reads:

```
(%i4) defmatch(m1,h(ann,bnn,numonly) ); /* define a match program m1*/
(%o4)                          m1
```

One can also use the function `defrule` which can be used to produce essentially the same effect. defrule says to bind those names and put them into a rule right-hand side: pattern –> replacement.

```
(%i5) defrule(r1,h(ann,bnn,numonly),['numonly = numonly,'bnn = bnn,'ann = ann])$
```

m1(h(q,r,34)); r1(h(q,r,34));

everyone is happy here, with the result [numonly=34,bnn=r,ann=q]

If we have the pattern ann*bnn*numonly, we don't have so much luck.

```
 defmatch(m1,ann*bnn*numonly)
```

Let us try to match $2xy$ with these rules. Perhaps you expect to collect x in `ann` y in `bnn` (or the converse) and 2 in `numonly`. Instead one gets:

```
(%i6) m1(2*x*y);
(%o6)                                    false
(%i7) r1(2*x*y);
(%o7)                                    false
```

To understand this, let us relax the predicate on variable `ann`. Note it is sufficient to run `matchdeclare` and redefine the rule:

```
(%i8) matchdeclare(ann,atom,bnn,pred1,cna,pred2,numonly,numberp)$

(%i9)  defmatch(m1,ann*bnn*numonly)$
```

[3]It is possible to write a whole language based on a theory of unification in first-order logic,such is the language Prolog.

```
(%i10) m1(2*x*y);
(%o10)                    [numonly = 2, bnn = x y, ann = 1]
```

We can now explain various facets of the matching mechanism. The variable bnn was matched first, and it greedily collected all the non-numeric atoms, those that matched its predicate, and collected them as a product. That left the expression as 2, which can match numonly, leaving the expression as 1. That can match ann now, since 1 is an atom! So we are done! It may not be what you expected. First `defmatch` creates a matching program `m1`, and puts on the Maxima property list (`MPROP`) of the Lisp symbol `$m1` a `$RULE` property which is a Maxima list (or `MLIST`) containing two values: the "left-hand side" to be matched and the replacement result. Here the replacement is an empty Maxima list "[]" which in Lisp looks like `((MLIST) )`.

The command `disprule` can be used for displaying the same information but using the Maxima language. Note that defrule or defmatch data have in their definition access to the appropriate match variables and thus their property list, which contain the corresponding predicates. This information is accessed when `defrule` or `defmatch`is used.

A subsequent change in the match variables will not affect `m1` or `r1` unless there is a new call to `defrule` ! That is, we would have to re-run the commands.

In the case of `%o10` the matcher has correctly associated `d` to the number 2. The matcher has shuffled terms around using product commutativity to satisfy the predicates, it is not a syntactic or string matcher like the one in grep, awk, etc. but it uses knowledge of algebraic properties to do matching. It is more puzzling that `a=1` and `b=x*y`.

Recall that `b` matches symbols that are not numbers. The addition and multiplication operators are commutative and nary. In these situations the matcher is **greedy** and for a given match variable, it attempts to gather all the symbols or subexpressions appearing in the addition or the multiplication that satisfy the associated predicate. For example in `%o14` above, `b` matches the three symbols `x,y,z`. Now what about `a`? The matcher is able to "invent" that `x*y*z = 1*x*y*z` and then tries to associate `a` with 1. In the first `matchdeclare a` is forbidden to be a number. But this matcher has **no backtracking capability**, so having set `b=xy` will not come back to set a=x, b=y, hence the matcher fails. This explains the results in %o6 and %o7. When we relax the condition on a, the matcher can set a=1 and then everything works OK.

Remark: let us insist that in the case of (commutative) addition and multiplication, the matching is done in a possibly non intuitive way. Let us illustrate this by the following example:

```
(%i1) matchdeclare(aa,atom,bb,all);
(%o1)                              done
(%i2) defmatch(rr,aa+bb);
(%o2)                               rr
(%i3) rr(x+y+x*y);
(%o3)                   [bb = x y + y + x, aa = 0]
(%i4) matchdeclare(aa,all,bb,atom);
(%o4)                              done
(%i5) defmatch(rr,aa+bb);
(%o5)                               rr
(%i6) rr(x+y+x*y);
```

```
(%o6)                          [bb = y + x, aa = x y]
```

The result in %o3 is similar to what we have seen above. The matcher attributes greedily everything to bb and satisfies aa with 0. The second example with inverted order is more interesting. First we see that the matcher works in inverse lexicographic order, it first tries to satisfy bb and then aa in both cases. Note it is the same for products. Second we see that the matcher collects all the atoms in the expression and outputs the *sum* of these atoms in bb. This may be quite unintuitive since bb has the predicate "atom" but at the end is *not* atomic, but the sum of all atoms. A term which satisfies more than one predicate is taken by the first predicate which it satisfies. Each predicate is tested against all operands of the sum or product before the next predicate is evaluated. The order in which the matching is done is inverse lexicographic in the pattern variables. Note that the order of terms in the match pattern is not necessarily the order in which you typed the terms. The expression is possibly rearranged by simplification. If this causes problems, the recommended fix is to define predicates that completely partition the expression uniquely.

One can also use the following trick to convert a sum or a product to an expression whose pieces can be manipulated separately (this would also allow to manipulate non commutative multiplication):

```
(%i1) matchdeclare(p, ?mplusp);
(%o1)                                  done
(%i2) defrule(rp,p, foo(first(p),foo(rest(p))));
(%o2)                 rp : p -> foo(first(p), foo(rest(p)))
(%i3) rp(a+b+c);
(%o3)                            foo(c, foo(b + a))
(%i4) apply1(a+b+c,rp);
(%o4)                         foo(c, foo(foo(b, foo(a))))
```

Here one uses the predicate mplusp at the lisp level to avoid being caught in the machinery for sums and products.

When we have both atoms and non atoms such as sin(x) or sin(y) or numbers, the matcher is able to completely partition expressions as below.

```
(%i20) matchdeclare(a,pred1,b,pred1,c,pred2,d,numberp)$
(%i21) defmatch(r2,b*c*d)$
(%i22) r2(2*x*y*sin(u)*cos(w));
(%o22)                 [d = 2, c = sin(u) cos(w), b = x y]
```

So the matcher has attributed the product of all atoms (not numbers) to b, the product of all non-atoms to c and the number to d. In general it will partition the expression in three components corresponding to mutually exclusive predicates, and this partitioning is unique, independent of order of factors.

Thus to separate pieces that are free of x and pieces that are free of y, where x and y can be specified by the call to the matching program, consider this:

```
(%i1) matchdeclare(nox,freeof(x), noy,freeof(y))$
```

```
(%i2) defmatch(separate, nox+noy,x,y)$ /*notice extra arguments! */
(%i3) separate(sin(u)+4*v+cos(v)+5*u,u,v);
(%o4) [x=u,y=v,noy=sin(u)+5*u,nox=cos(v)+4*v]
```

If the command was `answer:separate(x+y+4,x,y)` we could not object to where 4 was matched, since it is free of x and free of y. As it happens, `noy` collects its terms first, so the result is `[x=x,y=y,noy=x+4,nox=y]`. One neat way of picking out, from that expression, the value for (say) `noy` is `subst(answer,noy)` which gives `x+4`. Another way is to observe that there is a side effect of calling the match program `separate`. The values of `nox` and `noy` are set, and so it is safer to have a local binding:

```
(%i5) what_is_free_of_v(W):=block([nox,noy], separate(W,u,v), noy)$
(%i6) what_is_free_of_v(sin(u)+4*v+cos(v)+5*u);
(%o6)      sin(u)+5*u
```

The match variables in a sum or product are tested in reverse alphabet order, so in `a+b+c`, the `c` is the first variable and its predicate get first crack at the expression.

When the formula to be matched is not a sum or product, there is no problem from commutativity or functions with variable number of arguments (nary functions), and patterns work in a straightforward way, with matching essentially traversing the internal structure of expressions, matching locations to patterns. For example

```
(%i16) matchdeclare([a1,a2,a3],true)$       /* a1,a2,a3 match anything */
(%i17) defmatch(pat3, h(g(a1,3),a2,r(a3)))$ /* made-up functions h,g,r */
(%i18) pat3(h(g(43,3),w+4,r(y+7)));
(%o18)  [a3=y+7,a2=w+4,a1=43]
```

Returning to sums and products, there are some possibly unexpected matches related to the representation of expressions in the program: observe the terms with a negative sign in a sum or quotients in a product

```
(%i24) r2(x*y/z);  /* recall, defmatch(r2,b*c*d), d is a number, c non-atom  */
                                                 1
(%o24)                         [d = 1, c = -, b = x y]
                                                 z
```

First the matcher has inserted a factor 1 to match `d`, but note that $1/z$ appears in the non-atoms. This is because $1/z$ is indeed not an atom, `op(1/z)="/"`. Analogously for a subtraction, where `-1*y` is not an atom:

```
(%i25) defmatch(r2,b+c)$
(%i26) r2(x-y);
(%o26)                            [c = - y, b = x]
```

Finally let us look at how the matcher works with expressions containing sub-expressions:

```
(%o27) matchdeclare(a,pred1,b,pred2,c,numberp,d,pred1)$
(%i29) defmatch(r3,c+a*b+3*d);
defmatch: 3 d
         will be matched uniquely since sub-parts would otherwise be ambigious.
defmatch: b a
         will be matched uniquely since sub-parts would otherwise be ambigious.
(%i31) r3(5+x*sin(y)+3*z);
(%o31)                 [d = z, c = 5, a = x, b = sin(y)]
```

This pattern worked as we might have hoped, but this is fortuitous: because the pattern program first looked for `3*d`, where `d` is a symbolic atom. If it had first found the subexpression `x*sin(y)`, it would have divided it by 3 and tried `pred1` on the expression `x*sin(y)/3` which would fail. That's the meaning of the message. That is, the pattern matcher is going to try only once (uniquely) to match a sub-part, and if that fails, it won't try again on another sub-part. There is insufficient guidance (there are no anchors) in the pattern. This is undoubtedly a subtle point, and attempts to clarify the warning message have apparently not succeeded!

3.1 If one is interested in the inner workings.

Let us see how the matcher works in more details on a simple case. The following works only on Maxima systems that do not automatically compile code into machine language. One of those is Clisp, and that was used for this example.

```
(%i1) matchdeclare(a,atom);
(%o1)                                done
(%i2) defmatch(r,h*a);
(%o2)                                  r
(%i3) r(h*x);
(%o3)                               [a = x]
(%i4)
```

Here we point out that the pattern should contain undeclared objects which are matched only by themselves. These are anchors, and the anchor here is the object `h`. In fact, the process doesn't really "match" `h`, it divides by it, and sees if what is left matched `a`. We can see the function generated by `defmatch` by doing:

```
(%i4) :lisp(symbol-function '$r)
#<FUNCTION LAMBDA (TR-GENSYM1) (DECLARE (SPECIAL TR-GENSYM1))
  (CATCH 'MATCH
   (PROG NIL (DECLARE (SPECIAL))
    (SETQ TR-GENSYM1 (MEVAL '((MQUOTIENT) TR-GENSYM1 $H)))
    (COND ((DEFINITELY-SO '(($ATOM) TR-GENSYM1)) (MSETQ $A TR-GENSYM1))
     ((MATCHERR)))
    (RETURN (RETLIST $A))))>
(%i4) ?trace(?matcherr);
;; Tracing function MATCHERR.
(%o4)                             (matcherr)
(%i5) :lisp(trace $r)
```

```
;; Tracing function $R.
($R)
(%i5) r(h*x);
1. Trace: ($R '((MTIMES SIMP) $H $X))
1. Trace: $R ==> ((MLIST SIMP) ((MEQUAL SIMP) $A $X))
(%o5)                              [a = x]
(%i6) r(h*sin(x));
1. Trace: ($R '((MTIMES SIMP) $H ((%SIN SIMP) $X)))
2. Trace: (MATCHERR)
1. Trace: $R ==> NIL
(%o6)                               false
```

We see that TR-GENSYM1, a generated name for the argument is bound to '((MTIMES SIMP) $H $X) that is `h*x`. The first thing done is to divide by h to get x. Then the first clause of the COND tests the predicate associated with `a`, namely atom() on x. If x is an atom, the program sets a=x,and returns to the caller with the list [a=x]. If that clause fails the second consists of a call to (matcherr) throws the 'MATCH tag and has result nil. The 'MATCH is caught by the CATCH, and returning nil. The idea is that any time the program finds the match must fail, it does an immediate "throw" to exit the procedure. The general pattern matching program composes the code to follow along the matching process. The pieces of code are emitted by various functions (getdec, compileatom, compileplus, compiletimes) as bits of lisp data (which, after being inserted into the framework with catch and throw can be interpreted as lisp functions (and compiled to machine code). One can see these various bits of code emitted by tracing said functions. For example:

```
(%i33) :lisp(trace compiletimes)
...
(%i35) defmatch(r,a*b);
  0: (COMPILETIMES TR-GENSYM16 ((MTIMES SIMP) $A $B))
  0: COMPILETIMES returned
       ((COND
         ((PART* TR-GENSYM16 '($B $A)
                '((LAMBDA (#:G494)
                    (DECLARE (SPECIAL #:G494))
                    (COND
                     ((DEFINITELY-SO '(($PRED2) #:G494)) (MSETQ $B #:G494))
                     ((MATCHERR))))
                  (LAMBDA (#:G495)
                    (DECLARE (SPECIAL #:G495))
                    (COND
                     ((DEFINITELY-SO '(($PRED1) #:G495)) (MSETQ $A #:G495))
                     ((MATCHERR)))))))
         (T (MATCHERR))))
(%o35)
                              r
```

where the part:

```
        (COND
                          ((DEFINITELY-SO '(($PRED2) #:G494)) (MSETQ $B #:G494))
                          ((MATCHERR)))
```

is in fact issued by getdec as seen by tracing it. We see that the symbol-function of `$r` is built piece by piece by several functions.

The very peculiar behavior of the matcher in the presence of sums or products is programmed in `PART+` for sums and `PART*` for products which can be found in `matrun.lisp`. In particular the following at the end of `PART+`:

```
(setq saved 0)
     (mapc
      #'(lambda (z)
  (cond ((null (setq val (catch 'match (mcall (car preds) z)))) nil)
(t (setq saved (add2* saved val))
   (setq e (delete z e :count 1 :test #'equal)))))
      e)
```

is what collects the sum of the parts satisfying the considered predicate in `saved` and deletes them from the expression `e`. The predicates are arranged in a list, and only when the above returns is the list truncated to get to the next predicate. One can see in the above `(%i35) defmatch(r,a*b)` that `PART*` is given as arguments, the expression, the list of pattern variables in reverse lexicographic order, and the list of predicates.

4 The functions defining rules, and transformation rules.

We have seen a particular form of the defmatch function. Its general form is the following:

```
defmatch(name, pattern,x_1,...,x_n)
```

that is one may specify extra "pattern arguments" besides the wildcards declared by matchdeclare. However their use is not as flexible as that of the wildcards as illustrated below.

```
(%i1) pred1(x):=atom(x) and not numberp(x);
(%o1)              pred1(x) := atom(x) and (not numberp(x))
(%i2) pred2(x):= not atom(x);
(%o2)                         pred2(x) := not atom(x)
(%i3) matchdeclare(a,pred1,b,pred2);
(%o3)                                   done
(%i4) defmatch(r3,a+x+b*u,x);
(%o4)                                    r3
(%i5) r3(y+z+sin(t)*u,z);
(%o5)                       [a = y, b = sin(t), x = z]
```

Here we have two wildcards and two extra variables `x` and `u` in the pattern. The variable `x` is declared in defmatch and the variable `u` is undeclared, and will match only itself. The pattern argument x is given value z, but in fact z has to be given as argument to r3, and matches itself, so there is no new information. Indeed:

```
(%i6) r3(y+z+sin(t)*u,x);
(%o6)                                false
```

So x doesn't play a role of wildcard. On the other hand these pattern arguments allow to dissect for example products (this doesn't work with wildcards, as we have seen):

```
(%i7) defmatch(r4,x*y,x,y);
(%o7)                                 r4
(%i8) r4(u*v,u,v);
(%o8)                           [x = u, y = v]
```

Presumably canonical ordering of the product x*y and order of the pattern variables are involved in that specific identification. Note that pattern variables can be kernels: With the wildcard a being an atom:

```
(%i9) defmatch(r4,a*y,y);
(%o9)                                 r4
(%i10) r4(u*v,u);
(%o10)                          [a = v, y = u]
(%i11) r4(u*sin(v),sin(v));
(%o11)                        [a = u, y = sin(v)]
```

Anyways what can we do with the result of applying a rule defined by defmatch? One can substitute the sequence of equalities in any expression of the pattern variables or even use them directly. Beware there is a difference:

```
(%i12) a^2+y^2;
                                    2    2
(%o12)                             y  + u
(%i13) subst(%o11,a^2+y^2);
                                   2       2
(%o13)                          sin (v) + u
```

In %i12 the a=u has been applied but not the y=sin(v). Only the wildcards have been effectively substituted. Using subst gives a complete result.

The defrule function allows to specify a replacement directly. Its syntax is:

```
defrule(name,pattern,replacement)
```

Note that no extra pattern variables as in defmatch are allowed, but one is free to use undeclared variables that will be matched by themselves. The replacement is any expression depending on these variables and wild cards. It may be some mathematical function or some more textual function as we have shown at the beginning

```
defrule(r2,a*b*d,['a = a,'b = b,'d = d]);
```

The replacement pattern is here chosen so that this rule has the same effect than `defmatch(r2,a*b*d)`. Replacement involving mathematical functions is for example:

```
(%i1) pred2(x):= not atom(x);
(%o1)                     pred2(x) := not atom(x)
(%i2) matchdeclare(a,atom,b,integerp,c,pred2);
(%o2)                              done
(%i3) defrule(r5,a+b*c+x,exp(a)/b*c/x);
defmatch: c b
         will be matched uniquely since sub-parts would otherwise be ambigious.
                                                    a
                                                  %e  c
(%o3)                     r5 : x + b c + a -> -----
                                                   b x
(%i4) r5(u+3*sin(v)+x);
                                     u
                                   %e  sin(v)
(%o4)                              ----------
                                      3 x
```

Note that `x` is undeclared in the pattern and matched by itself in the expression. It is very important to identify precisely `b` and `c` in `bc` since they are transformed into `c/b` which could easily be erroneously `b/c`. This is achieved by declaring `b integerp` and `c` non atom.

We end this discussion of defrule by an interesting textual replacement function appearing in the manual:

```
(%i1) matchdeclare (aa, atom, bb, lambda ([x], not atom(x)));
(%o1)                              done
(%i2) defrule (r1, aa + bb, ["all atoms" = aa, "all nonatoms" = bb]);
(%o2)         r1 : bb + aa -> [all atoms = aa, all nonatoms = bb]
(%i3) defrule (r2, aa * bb, ["all atoms" = aa, "all nonatoms" = bb]);
(%o3)          r2 : aa bb -> [all atoms = aa, all nonatoms = bb]
(%i4) r1 (8 + a*b + %pi + sin(x) - c + 2^n);
                                                         n
(%o4)     [all atoms = %pi + 8, all nonatoms = sin(x) + 2  - c + a b]
(%i5) r2 (8 * (a + b) * %pi * sin(x) / c * 2^n);
                                                      n + 3
                                             (b + a) 2      sin(x)
(%o5)       [all atoms = %pi, all nonatoms = ---------------------]
                                                      c
```

The results are here a partition of atoms and non atoms in a sum and a product respectively. However while `c` is certainly an atom it appears in the expressions in the form `-c` or `1/c` which both are not atoms. The predicates being logically disjoint aa and bb are indeed partitions of the considered expressions determined without ordering ambiguity. Note also that the simplifier has written $8 * 2^n$ as $2^(n + 3)$.

With this example one can indeed show the greediness problem: let us define instead of two mutually exclusive predicates two overlapping predicates:

```
(%i1) matchdeclare(aa, atom, bb, lambda ([x], not atom(x) or numberp(x)));
(%o1)                               done
(%i2) defrule (r1, aa + bb, ["all atoms" = aa, "all nonatoms" = bb]);
(%o2)         r1 : bb + aa -> [all atoms = aa, all nonatoms = bb]
(%i3) r1 (8 + a*b + %pi + sin(x) - c + 2^n);
                                                  n
(%o3)     [all atoms = %pi, all nonatoms = sin(x) + 2  - c + a b + 8]
```

Note that a number satisfies both predicates for aa and bb, so that 8 is taken greedily by the first variable bb, and thus does not appear in "all atoms".

At this point we may have a lot of rules defined, a lot of wildcards declared by matchdeclare, etc. There are convenience functions to list this stuff (disprule, propvars, printprops).

One may get for example:

```
(%i27) disprule(all);
(%t27)                   rr1 : 3 d + c + a b -> []

(%t28)                   rr2 : x + b u + a -> [x]

(%t29)                      rr3 : x y -> [x, y]

(%t30)                       rr4 : a y -> [y]


                                             a
                                           %e   c
(%t31)                  rr5 : x + b c + a -> -----
                                             b x

(%o31)                [%t27, %t28, %t29, %t30, %t31]

(%i32) propvars (matchdeclare);
(%o32)                         [a, b, c, d]
(%i33) printprops (all, matchdeclare);
(%o33)           [pred1(d), numberp(c), pred2(b), pred1(a)]
```

We see that the rules are labelled by `disprule`, wildcards are listed by `propvars(matchdeclare)` and the associated predicates are listed by `printprops`. In a similar vein `clear_rules()` kills the existing rules.

Once rules have been defined there is a mechanism to apply them automatically and recursively in an expression. One does that using the functions `apply1, apply2, applyb1`.
`apply1(exp,rule_1,...,rule_n)` apply `rule_1` to `exp` repeatedly until it fails. The the same rule on all subexpressions, left to right (recall that an expression is a tree, usually with operators at nodes and leafs are atoms). Then one does the same with `rule_2` etc. The difference with `apply2` is that it applies in order `rule_1,...,rule_n` to each subexpression before passing to

the next one. Finally applyb1 does the same but starting from bottom. The recusrsions are limited by maxapplydepth from top to bottom and maxapplyheight from bottom towards top.

A trivial example: going from dot product to * product.

```
(%i1) matchdeclare(a,true,b,true);
(%o1)                              done
(%i2) defrule(r1,a.b,a*b);
(%o2)                        r1 : a . b -> a b
(%i3) apply1((x+y.(z+t.u)).(v+w.g),r1);
(%o3)                     (g w + v) (y (z + t u) + x)
```

Going the other way round.

```
(%i1)  matchdeclare(a,lambda([e],not atom(e) and (op(e)="*")));
(%o1)                              done
(%i2) defmatch(r,a);
defmatch: evaluation of atomic pattern yields: a
(%o2)                                r
(%i3) defrule(r2,a,subst(".","*",a));
(%o3)                      r6 : a -> subst(., *, a)
(%i5) apply1((g*w + v)*(y*(z + t*u) + x),r2);
(%o5)                 (v + g . w) . (y . (z + t . u) + x)
```

Transforming only the leaf products in the expression tree:

```
(%i7) maxapplyheight:1$

(%i8) applyb1((g*w + v)*(y*(z + t*u) + x),r6);
(%o8)                   (v + g . w) (y (z + t . u) + x)
```

Suppose one wants to get the arguments of the leaf dot products:

```
(%i1) matchdeclare(a,lambda([e],not atom(e) and (op(e)=".")));
(%o1)                              done
(%i2) defrule(r3,a,print(args(a)));
(%o2)                      r3 : a -> print(args(a))
(%i3) maxapplyheight:1$
(%i5) applyb1((v + g . w)*(y*(z + t . u) + x),r3)$
[g, w]
[t, u]
```

Using recursive functions like apply1 may lead to infinite recursion if the replacement patterns still matches:

```
(%i6) matchdeclare(a,atom);
```

```
(%o6)                                   done
(%i7) defrule(r,a,[a]);
(%o7)                               r : a -> [a]
(%i8) apply1(x+y,r);
INFO: Binding stack guard page unprotected
Binding stack guard page temporarily disabled: proceed with caution
(%i9) :lisp(trace $r)
($R)
(%i9) apply1(x+y,r);
  0: ($R ((MPLUS SIMP) $X $Y))
  0: $R returned NIL
  0: ($R $X)
  0: $R returned ((MLIST SIMP) $X) T
  0: ($R ((MLIST SIMP) $X))
  0: $R returned NIL
  0: ($R $X) ..............
```

We see that r applies to x, giving [x], then doesn't apply to [x]. It then descends recursively to x inside [x] and infinite recursion begins. Note that applyb1 doesn't suffer from the same problem:

```
(%i5) applyb1(x+y,r);
  0: ($R $X)
  0: $R returned ((MLIST SIMP) $X) T
  0: ($R $Y)
  0: $R returned ((MLIST SIMP) $Y) T
  0: ($R ((MLIST SIMP) ((MPLUS SIMP) $X $Y)))
  0: $R returned NIL
(%o5)                                  [y + x]
```

Here r applies on x and gives [x] but is then blocked, similar for y, so it goes upper to x+y and returns [x+y].

Remark: In case the replacement pattern is a lambda function one has to apply the lambda function to the matching variable to get the result, such as:

```
(%i1) matchdeclare(a,atom);
(%o1)                                   done
(%i2) defrule(r,a,lambda([e],print("a =", e))(a));
(%o2)                  r : a -> lambda([e], print(a =, e))(a)
(%i3) r(x);
a = x
(%o3)                                    x
```

Note the a inside the lambda is not captured automatically, and stays in the printout a=x. This is a remark of R. Dodier.

5 The let rules.

These rules are simplification rules specifically for use with products, namely products of atoms to positive or negative powers and kernels such as `sin(x), n!, f(x,y)`. They are defined in nisimp.lisp.

Some of the arguments of `prod` may be declared in a `matchdeclare` statement, but as in the case of `defmatch`, other arguments mays be specified in the let rule. Moreover one can specify a predicate which specifies when these let arguments match. The replacement is any rational function, and the matched arguments are substituted in it. A typical let rule takes the form:

```
let(product,replacement,predicate,arg_1,...arg_n);
```

To be more precise positive powers are grouped in a numerator and negatives one in a denominator. Powers of an atom in an expression match only when the power is greater or equal than in the pattern, either in the numerator or denominator.

In fact let rules may be grouped in packages, so there is an extended form:

```
let([product,replacement,predicate,arg_1,...arg_n],package_name);
```

The let rules are applied to an expression by the commands:

```
letsimp(expression); or
letsimp(expression,package_name); or
letsimp(expression,pack_1,pack_2,...);
```

The command `letsimp` simplifies separately the numaerator and the denominator. If one wants that cancellations occur between numerator and denominator, one needs to set

```
letrat:true;
```

Letsimp acts by applying the let rule repeatedly until the expression doesn't change any more. However this needs some comments.

```
(%i1) t(x,y):=true;
(%o1)                          t(x, y) := true
(%i2) let(x*y,x+y,t,x,y);
(%o2)                   x y --> y + x where t(x, y)
(%i3) letsimp(x*y+y*z);
(%o3)                            y z + y + x
```

We note that the `letsimp` has been distributed across the addition. The part `x*y` matches and becomes `x+y` while `y*z` doesn't match and is left as is. Similarly

```
(%i4) letsimp(x*y*z);
(%o4)                              y z + x z
```

so z which is not declared is left in identical way in the replacement. Things are more tricky when adding powers. Looking at nisimp.lisp one sees that the program to be traced is nisletsimp. We get:

```
(%i5) trace(letsimp);
(%o5)                              [letsimp]
(%i6) :lisp(trace nisletsimp)

(NISLETSIMP)
(%i6) letsimp(x^2*y);
                         2
1 Enter letsimp [letsimp(x  y)]
  0: (NISLETSIMP
      ((MRAT SIMP ($X $Y) (#:X468 #:Y469)) (#:Y469 1 (#:X468 2 1)) . 1))
    1: (NISLETSIMP ((MTIMES RATSIMP) ((MEXPT RATSIMP) $X 2) $Y))
      2: (NISLETSIMP ((MPLUS SIMP) ((MEXPT SIMP) $X 2) ((MTIMES SIMP) $X $Y)))
        3: (NISLETSIMP ((MEXPT SIMP) $X 2))
        3: NISLETSIMP returned ((MEXPT SIMP) $X 2)
        3: (NISLETSIMP ((MTIMES SIMP) $X $Y))
          4: (NISLETSIMP ((MPLUS SIMP) $X $Y))
            5: (NISLETSIMP $X)
            5: NISLETSIMP returned $X
            5: (NISLETSIMP $Y)
            5: NISLETSIMP returned $Y
          4: NISLETSIMP returned ((MPLUS) $X $Y)
        3: NISLETSIMP returned ((MPLUS) $X $Y)
      2: NISLETSIMP returned ((MPLUS) ((MEXPT SIMP) $X 2) ((MPLUS) $X $Y))
    1: NISLETSIMP returned ((MPLUS) ((MEXPT SIMP) $X 2) ((MPLUS) $X $Y))
    1: (NISLETSIMP 1)
    1: NISLETSIMP returned 1
  0: NISLETSIMP returned
       ((MQUOTIENT) ((MPLUS) ((MEXPT SIMP) $X 2) ((MPLUS) $X $Y)) 1)
                  2
                 x  + (x + y)
1 Exit  letsimp ------------
                     1
                                          2
(%o6)                            y + x  + x
```

At position 1 `nisletsimp` sees the product of x^2 and y. This product matches the rule thus at position 2 `nisletsimp` sees $x^2 + xy$. Then at position 4 `xy -> x+y` and one gets the result $x^2 + x + y$, the quotient by 1 being due to rational representation. Hence the question: how is it that one application of the rule to $x^2 * y$ gives $x^2 + xy$ and not $x^2 + y$? This is understood if one writes $x^2 * y = x * (x * y) -> x * (x + y)$. Let us try to see if this works for higher powers:

```
(%i8) letsimp(x^3*y);
```

```
                                       3    2
(%o8)                            y + x  + x  + x
```

This is explained by the chain:

$x^3 * y = x^2 * (x * y)$ -> $x^2 * (x + y) = x^3 + x * (x * y)$ -> $x^3 + x^2 + x * y$ -> $x^3 + x^2 + x + y$

Finally let us look at:

```
(%i9) letsimp(x^3*y^2);
                             2          3      2
(%o9)                       y  + 3 y + x  + 2 x  + 3 x
```

This is explained by : $x^3 * y^2$ -> $x^2 * y * (x + y) = x^3 * y + x^2 * y^2$. The first one is $x^3 + x^2 + x + y$, the second $x * y * (x + y)$ -> $x * (x + y) + y * (x + y)$ -> $x^2 + y^2 + 2 * (x + y)$, yielding the above result.

This explains the nature of the recursion occurring in let rules. As is the case with `defmatch` it is frequently more convenient to use variables in patterns defined by matchdeclare. For example in below, variables `m` and `n` are matched to `a1` and `a2` and substituted in the replacement. An example is in the manual:

```
(%i1) matchdeclare ([a, a1, a2], true)$
(%i2) oneless (x, y) := is (x = y-1)$
(%i3) let (a1*a2!, a1!, oneless, a2, a1);
(%o3)         a1 a2! --> a1! where oneless(a2, a1)  /* that is a2=a1-1 */
(%i4) letrat: true$
(%i5) let (a1!/a1, (a1-1)!);
                        a1!
(%o5)                   --- --> (a1 - 1)!
                        a1
(%i6) letsimp (n*m!*(n-1)!/m);
(%o6)                     (m - 1)! n!
```

Finally there are some convenience functions to manipulate let rules.

```
letrules(); display rules in current package, by default default_let_rule_package
letrules(pkg) ; display rules in pkg
let_rule_packages;  is a list of all packages, including default_let_rule_package
remlet(product,package_name); removes rule for product in package_name
remlet();  removes all rules from current package
remlet(all); idem
remlet(all,package_name); removes all rules in package_name, and the name itself.
current_let_rule_package can be assigned to any package name.
```

We see that the let rules allow to simplify differently expressions. However they have to be called explicitely by letsimp(). We are now going to discuss similar simplifying functions which are directly hooked in the maxima simplifier, and are thus called automatically. They are called `letsimp` (called before the simplifier) and `letsimpafter` (called after the simplifier).

6 Tellsimp and tellsimpafter.

Both have the form:

```
tellsimp (pattern, replacement);
tellsimpafter (pattern, replacement);
```

The manual says: `tellsimp` is used when it is important to modify the expression before the simplifier works on it, for instance if the simplifier "knows" something about the expression, but what it returns is not to your liking. If the simplifier "knows" something about the main operator of the expression, but is simply not doing enough for you, you probably want to use `tellsimpafter`.

The pattern is comprised of wild cards declared with defmatch and other variables or kernels and operators which are taken literally as above. For tellsimp the pattern may not be a sum, product, single variable, or number. For tellsimpafter pattern may be any nonatomic expression in which the main operator is not a pattern variable.

The simplification rule is associated to the main operator. If the name of the rule is foorule, one can trace it by `:lisp(trace foorule)` and see it (with `maxima -l clisp`) with

```
:lisp (symbol-function '$foorule).
```

A simple example:

```
(%i1) matchdeclare([a,b,c]:all);
(%o1)                                   done
(%i2) tellsimpafter(a+b*c,a-b*c);
defmatch: c b
         will be matched uniquely since sub-parts would otherwise be ambigious.
(%o2)                             [+rule1, simplus]
```

Note the rule is associated to main operator "+" and named `+rule1`. It is also associated to the simplus generic simplification functions for sums. Tracing it gives:

```
(%i3) :lisp(trace +rule1)
(+RULE1)
(%i3) x+y*z;
  0: (+RULE1 ((MPLUS) $X ((MTIMES SIMP) $Y $Z)) 1 NIL)
    1: (+RULE1 ((MPLUS) $X ((MTIMES SIMP) -1 $Y $Z)) 1 NIL)
    1: +RULE1 returned ((MPLUS SIMP) $X ((MTIMES SIMP) -1 $Y $Z))
  0: +RULE1 returned ((MPLUS SIMP) $X ((MTIMES SIMP) -1 $Y $Z))
(%o3)                                  x - y z
```

The following is trickier:

```
(%i1) matchdeclare([a,b],all);
(%o1)                               done
(%i2) tellsimpafter(a+b,a-b);
(%o2)                         [+rule1, simplus]
(%i3) :lisp(trace +rule1)
(+RULE1)
(%i3) x+y;
  0: (+RULE1 ((MPLUS) $X $Y) 1 NIL)
    1: (+RULE1 ((MPLUS) $X $Y) 1 NIL)
    1: +RULE1 returned ((MPLUS SIMP) $X $Y)
    1: (+RULE1 ((MPLUS) ((MTIMES SIMP) -1 $X) ((MTIMES SIMP) -1 $Y)) 1 T)
    1: +RULE1 returned
         ((MPLUS SIMP) ((MTIMES SIMP) -1 $X) ((MTIMES SIMP) -1 $Y))
    1: (+RULE1
        ((MPLUS) 0 ((MPLUS SIMP) ((MTIMES SIMP) -1 $X) ((MTIMES SIMP) -1 $Y)))
        1 NIL)
    1: +RULE1 returned
         ((MPLUS SIMP) ((MTIMES SIMP) -1 $X) ((MTIMES SIMP) -1 $Y))
  0: +RULE1 returned ((MPLUS SIMP) ((MTIMES SIMP) -1 $X) ((MTIMES SIMP) -1 $Y))
(%o3)                            (- y) - x
```

Here we see that `a` is matched to 0, and `b` to `(x+y)` so the result is probably unexpectedly `-x-y`. We see once more the problems associated with the fact that for "+" and "*" the matcher is greedy and constants like 0 or 1 are added freely in the game. For `tellsimp`, main operator "+" is forbidden, and indeed if one tries it:

```
(%i1) matchdeclare([a,b],all);
(%o1)                               done
(%i2) tellsimp(a+b,a-b);
tellsimp: warning: rule will treat '+' as noncommutative and nonassociative.
(%o2)                         [+rule1, simplus]
(%i3) x+y;
INFO: Binding stack guard page unprotected
Binding stack guard page temporarily disabled: proceed with caution
Maxima encountered a Lisp error:
Binding stack exhausted.
```

Once more interaction between the tellsimprule and the simplifier causes infinite loop. These function being very automatic can cause many unexpected results. But they can also produce nice results.

It is possible to check how tellsimp and tellsimpafter rules are successively applied. Setting

```
announce_rules_firing: true;
```

before defining the simplification rules, will show how they are applied in a simpler way than tracing them.

As an example let us present quaternions in this way. Here `z0` plays the role of unit quaternion, `z1,z2,z3` are usually denoted `i,j,k`. The dot product plays the role of quaternionic product, but we need to introduce scalars as coefficients and impose distributivity with respect to addition and product with a scalar.

```
(%i1) dotdistrib:true$
(%i2) dotscrules:true$
(%i3) tsa(i,j,k,s)::=buildq([i,j,k,s],apply(          /* Utility macro */
        tellsimpafter,[concat(z,i).concat(z,j),s*concat(z,k)]))$
(%i4) for kk in [[1,2,3,1],[2,1,3,-1],[3,1,2,1],[1,3,2,-1],
          [2,3,1,1],[3,2,1,-1]] do ([i,j,k,s]:kk,tsa(i,j,k,s))$
(%i5) for kk in [[0,1],[1,0],[0,2],[2,0],[0,3],[3,0],[0,0]] do
        ([i,j]:kk,tsa(i,j,i+j,1))$
(%i6) for k from 1 thru 3 do tsa(k,k,0,-1)$
(%i7) for i from 0 thru 3 do
      apply(declare,[[concat(a,i),concat(b,i)],scalar])$
(%i8) A: sum(concat(a,i)*concat(z,i),i,0,3)$
(%i9) B: sum(concat(b,i)*concat(z,i),i,0,3)$
(%i10) rat(expand(A.B),z0,z1,z2,z3);
(%o10)/R/ (a0 b3 + a1 b2 - a2 b1 + a3 b0) z3
 + ((- a1 b3) + a0 b2 + a3 b1 + a2 b0) z2
 + (a2 b3 - a3 b2 + a0 b1 + a1 b0) z1
 + ((- a3 b3) - a2 b2 - a1 b1 + a0 b0) z0
(%i11) cA:-a3*z3 - a2*z2 - a1*z1 + a0*z0$             /* Conjugate of A */
(%i12) nA:a3^2  + a2^2  + a1^2  + a0^2$               /* Norm of A = A.cA */
(%i13) rat(expand(A.cA/nA),z0,z1,z2,z3);
(%o13)/R/                            z0               /* The unit quaternion */
```

One can check the proper declaration of scalars and of rules with `propvars(scalar);` and `disprule(all);` In %o10 we see the product of two quaternions A and B. If a0=b0=0 only the vector part appears and we see that the quaternionic product reduces to the vector product. In %o13 we see that every non nul quaternion is invertible, i.e. quaternions form a field.

The gamma algebra of quantum field theory can be treated in a similar way, we refer to

`https://github.com/dprodanov/clifford`

by D. Prodanov. This formalism is also amenable to treat the case of groups defined by generators and relations

`https://en.wikipedia.org/wiki/Presentation_of_a_group`

7 Appendix

The aim of this section is to describe in more details the relation between rules and functions. It is based on private communications of Prof. Fateman. First, a rule defines a function. To see this one has to use a maxima compiled with gcl or clisp since with sbcl the function is immediately compiled. We shall take a very simple example for brevity:

```
(%i1) matchdeclare(aa,atom);
(%o1)                                  done
(%i2) defrule(r,aa+x,x+exp(aa));
                                                     aa
(%o2)                       r : x + aa -> x + %e
(%i3) r(x+y);
                                          y
(%o3)                                   %e  + x
(%i4) :lisp(symbol-function '$r);

(LAMBDA (TR-GENSYM0)
  (DECLARE (SPECIAL TR-GENSYM0))
  (CATCH 'MATCH
    (PROG ($AA)
      (DECLARE (SPECIAL $AA))
      (SETQ TR-GENSYM0 (MEVAL '((MPLUS) TR-GENSYM0 ((MMINUS) $X))))
      (COND
        ((DEFINITELY-SO '(($ATOM) TR-GENSYM0)) (MSETQ $AA TR-GENSYM0))
        ((MATCHERR)))
      (RETURN
        (VALUES (MEVAL '((MPLUS SIMP) ((MEXPT SIMP) $%E $AA) $X)) T)))))
(%i4) :lisp(symbol-plist '$r)

(MPROPS (NIL $RULETYPE $DEFRULE $RULE
             ((MEQUAL) ((MPLUS SIMP) $AA $X)
              ((MPLUS SIMP) ((MEXPT SIMP) $%E $AA) $X))))
(%i4) disprule(r);
                                                     aa
(%t4)                       r : x + aa -> x + %e
```

So we have defined a rule `r` with one pattern variable `aa`, and one anchor `x`, which does its job. The program associated to the rule is attached to `$r` in the symbol-function of `$r`. Moreover a description of the rule is present in the property list of `$r` and displayed by `disprule(r)`. Finally `$r` has no symbol-value.

The way the program works is cleared by tracing it (here with gcl):

```
(%i5) :lisp(trace $r)

($R)
(%i5) r(x+y);

  1> ($R ((MPLUS SIMP) $X $Y))
  <1 ($R ((MPLUS SIMP) $X ((MEXPT SIMP) $%E $Y)) T)
                                          y
(%o5)                                   %e  + x
```

Thus the input TR-GENSYM0 to the function `$r` is `(list (MPLUS SIMP) \$X \$Y)` and the output is composed of two values, first the transformed expression, and second the boolean value T. This is because of the (values * *) at the end of the function. The beginning of the function is the matching engine. In this simple case, it works this way: since the rules tries to match `x+aa`, `x` is subtracted to the input TR-GENSYM0 and what remains must be `aa`. But `aa` must be an atom. In this case, applying to `x+y` it happens that `y` is an atom and thus one sets `aa=y`. Otherwise one runs matcherr. For example `r(x+y+z)` returns false. This is because matcherr just does (throw 'match nil), the tag 'match is catched by the CATCH 'MATCH which gets out of the program outputting just one value nil, showed by the clisp trace:

```
(%i4) r(x+y+z);
1. Trace: ($R '((MPLUS SIMP) $X $Y $Z))
1. Trace: $R ==> NIL
(%o4)                               false
```

In more complicated examples the pattern matcher works in essentially the same way processing recursively to identify the various patterns to match. But there is no backtracking and if some inside matcher enters matcherr the whole match returns false. Example:

```
%i1) matchdeclare(aa,atom,bb,numberp);
(%o1)                               done
(%i2) defrule(r,aa+bb*y,[aa,bb]);
(%o2)                     r : bb y + aa -> [aa, bb]
(%i3) r(x+2*y);
(%o3)                              [x, 2]
(%i4) :lisp(symbol-function '$r)

(LAMBDA (TR-GENSYM0)
  (DECLARE (SPECIAL TR-GENSYM0))
  (CATCH 'MATCH
    (PROG ($AA $BB TR-GENSYM1)
      (DECLARE (SPECIAL $AA $BB TR-GENSYM1))
      (SETQ TR-GENSYM1 (RATDISREP (RATCOEF TR-GENSYM0 (MEVAL '$Y))))
      (COND
        ((DEFINITELY-SO '(($NUMBERP) TR-GENSYM1))
         (MSETQ $BB TR-GENSYM1))
        ((MATCHERR)))
      (SETQ TR-GENSYM0
            (MEVAL '(($RATSIMP)
                     ((MPLUS) TR-GENSYM0 ((MTIMES) -1 TR-GENSYM1 $Y)))))
      (COND
        ((DEFINITELY-SO '(($ATOM) TR-GENSYM0)) (MSETQ $AA TR-GENSYM0))
        ((MATCHERR)))
      (RETURN (VALUES (MEVAL '((MLIST SIMP) $AA $BB)) T)))))
```

Here the pattern variable bb is identified as the coefficient of the anchor y in the rational expression of TR-GENSYM0 and tested to be a number. In this case one subtracts the product of bb and y from the expression to get aa. Then aa is tested to be an atom.

We have described in some details the way the rules work, i.e. they are functions which take as input one argument, an expression, and produce as output, either the transformed expression and T, if there is match or NIL otherwise. In fact producing two values NIL,NIL does no harm in this case and is more symmetric. For managing more sophisticated transformation rules it becomes necessary to write directly the function to be applied as a maxima function and transform it to a maxima rule. To this aim we reproduce here a program communicated by Prof. Fateman, that he calls function2maxrule.lisp:

```
;; generate a rule suitable for use by apply1 etc from a function
;; that returns a usual single value, a non-nil result or false.
;; multiple values: (result,  true)
;; otherwise  (false false)
;;  note that if your functions actually returns successfully with
;; the result false, you can’t use this.

(defun $convertfunction2rule (rulename thefun )
  ;; generate a named function that returns a multiple-value
  ;; of 2 items.  either  (false false)   or (theReplacement with subst,  true)
  (let* ((e (gensym "e"))
 (ans (gensym "a"))
 (body

  (coerce ‘(lambda(,e) ;function of one argument
     (let((,ans (mfuncall (quote ,thefun) ,e)))
       (if (null ,ans)
   (values nil nil) ;; false false for maxima
   (values ,ans
   t))))
  ’function)
   ))
    (setf (symbol-function rulename) body))
  ;; I added stuff to make disprule happy, otherwise useless.
  (setf (getf (symbol-plist rulename) ’mprops)
’(nil $ruletype $defrule $rule ((mequal) "foo" "bar")))
  (meta-add2lnc rulename ’$rules)
  rulename)
```

The basis of this program is that the maxima function “thefun” is applied to the expression `e` using the maxima function call `mfuncall`. A lisp function is created from the maxima function by the same technique used for example in coerce-float-fun (plot.lisp) and assigned to the symbol function of the rule rulename. In this expression `’(lambda ....)` is just a lisp list of symbols, which can be manipulated, for example by interpolating the values of the initial expression `,e` and the transformed one `,ans` and it is the `(coerce ... ’function)` which transforms this list in a lambda function definition. The variables `e` (the expression) and `ans` (the transformed expression) are protected from capture by using gensyms, they take a concrete value when the rule is applied to some concrete expression.

One then has to define the maxima function “thefun” able to transform the expression, but for that we need to identify patterns in the expression using the above mechanisms matchdeclare and defmatch. The maxima function can manipulate these patterns in any way, and use any conditions it likes on them. This is very flexible, but there is always the difficulty of properly selecting the patterns in the expression.

Let us apply this technique to compute the normal ordering of creation and annihilation operators. So take two symbols `a` and `b` with a non commutative multiplication noted ”.” and the commutation relation $b.a = a.b + 1$. Using it one wants to put all the a at the left and all the b at the right, which we call normal order. A very similar problem is to write any element of the enveloping algebra of the algebra of rotations in canonical order, up to lower order terms (this is the Poincaré–Birkhoff–Witt theorem). Here one has 3 elements a, b, c with commutation relations $b.a = a.b + c$ plus circular permutations on (a, b, c), and one wants to write any polynomial in a, b, c with the a first, then the b and finally the c. Sticking to our case, the following does the job:

```
/* To express dot products as x.(y.(z.t)) etc. one makes dotassoc:false
and then expand(expr,0,0)  similarly for the reverse */

dotassoc:false;
dotexptsimp:false;
dotdistrib:true;
dotscrules:true;

/* The expression to be transformed is a sum of terms of the form
x.(y. ...) multiplied by constants. We need to map the transformation rule
around sums and products by constants. */

/* a and b match only themselves */

matchdeclare(cc,true);

defrule(r1,b.a,a.b+1);
defrule(r2,b.(a.cc),a.(b.cc)+cc);

norm_ord(expr):=block([e:expr],
  if (scalarp(e) or e=a or e=b) then return(e)
  else if inpart(e,0)="+" then return(map(norm_ord, e))
  else if inpart(e,0)="*" then return(map(norm_ord, e))
  else if inpart(e,0)="." then return(applyb1(e,r2,r1))
  else false);  /* Should not occur */

load("function2maxrule.lisp");
convertfunction2rule(nord,norm_ord);

/* When using apply1 on this rule the recursion should stop when
applyb1(e,r2,r1) returns false */

/* Normal ordering a^^2.b^^3.a^^2 */

noexp:apply1(a.a.b.b.b.a.a,nord);
dotassoc:true$ dotexptsimp:true$
expand(noexp);
```

Where we load the above lisp program `function2maxrule.lisp`. The ideas in this example have been explained by R. Fateman. Running it gives:

```
(%i10) convertfunction2rule(nord,norm_ord);
```

```
(%o10)                                nord
(%i11) noexp:apply1(a.a.b.b.b.a.a,nord);
(%o11) 2 (a . (a . b) + a . (a . (a . (b . b)))) + 4 (a . (a . b))
            + 4 (a . (a . (a . (b . b)))) + a . (a . (a . (a . (b . (b . b)))))
(%i12) dotassoc:true$

(%i13) dotexptsimp:true$

(%i14) expand(noexp);
                   <4>    <3>       <3>    <2>        <2>
(%o14)            a    . b    + 6 (a    . b   ) + 6 (a    . b)
```