Partitioning of Algebraic Subexpressions in Computer Algebra Systems: Or, Don't use matching so much. Including a neat integration example

Richard Fateman

January 11, 2014

Abstract

A popular technique to direct transformations of algebraic expressions in a computer algebra system such as Macsyma/Maxima or Mathematica is to first write out a pattern or template for some expected class of "input" expressions and use a built-in system routine to match it, identifying pieces and giving them names, then proceeding to compute with these parts. Sometimes this is not such a good approach, and an alternative, presented here, may be more appealing.

1 Introduction

A popular technique to direct transformations of algebraic expressions in a computer algebra system such as Macsyma/Maxima or Mathematica is to first write out a pattern or template for some expected class of "input" expressions and use a built-in system routine to match it, identifying pieces and giving them names. Next one writes a rule-replacement where those extracted parts of the template are placed in a new configuration, presumably "the answer" or something closer to it. This can work especially nicely for applications where there is no completely known algorithm: that is, one can chip away at the problem by considering subcases or rules. If there are enough rules to cover all cases that appear in practice, this can look like an algorithm. If the rules in fact *do* cover all possible cases, it may very well be an algorithm. Unfortunately, especially if there are many rules and there are multiple ways of matching even one rule, this method can be inefficient and sometimes ineffective. In some cases the means for expressing a template for matching are sufficiently esoteric that the user may be stymied by the complexity. For example does the pattern `2*k` match the expression 8, binding `k=4`? Does `k*(k+1)` match the expression 12, binding `k=3`? [1]

In this essay we address one common programmatic situation where the pattern must match an expression headed by an associative (n-ary) "commutative operator with identity" such as "+" or "*" where there may be many possible permutations for matching sub-parts to subsets of terms. Matching such a pattern often is equivalent to a thoroughly studied problem in logic, namely "unification with identity"[2].

The obvious method for such a match may lead to an exponential-cost search using backup. Implementation inefficiency is often compounded by the difficulty that users find in composing entirely correct and minimal sets of non-overlapping rules. For example, a Mathematica pattern match to separate items in a sum free of `x` from others might look like `w1___?FreeQ[#,x]&+w2___`. If the expression in question is of length 12, a simple example takes 0.0156 seconds. It rapidly increases with the length so that for length 24, it takes over 72 seconds. The technique suggested here does the equivalent partitioning much faster: For length 24, 0.00005 seconds. For length 1000, about 0.10 seconds. See Appendix 1 for test programs in Maxima and Mathematica.

[1] Maxima does the first of these but not the second. Mathematica does neither. On the other hand Mathematica will do some kinds of combinatorial searches, finding solutions Maxima does not attempt, and the use of "solve" commands finds many other matches that elude the usual heuristics for matching.

[2] The usual computer-algebra-system version of matching is a less powerful and much easier version: for a start, it is asymmetric with the "expression" free of pattern variables. General unification allows variables on both sides of the match. Nevertheless, one can easily pose exponential-time searches in the restricted domain.

In this paper we demonstrate an alternative to using matching that is more direct in implementation and is versatile. We show how this can be used for writing a particularly simple symbolic integration program: a table-driven 20-line program that does many standard textbook problems. The program does not use matching for combinatorial search, although it does use predicate testing as a fundamental (arguably, matching) tool.

2 A Motivating Example

In writing programs to transform algebraic expressions there is often a requirement to partition sums or products into pieces that match various predicates.

For example, consider an integration program that tries to work up a case for integrating an expression that might be described by a template like $ax^3\cos(x)$, where a and x are parameters in the pattern. This is easy if we know in advance the symbolic value of x, say t. Then $t^3 * \cos(t)$ can be divided out and we can find a which is presumably some expression that is free of t. Perhaps we will check the truth of that statement. Maxima's pattern matcher can accomplish this. It is less useful in finding a match for something like this: $f(u)u'$, that is, where one item, or perhaps a product of items, is a derivative of some other item(s) in the product, and f is not literally cos but some *as-yet-to-be-determined* function.

These are tough problems for "pattern matching" in the sense that the expression may need to be transformed in some way in order to match (say simplifying a sum to a product first by factoring, or by actually performing a "synthetic division" and re-simplifying), or may have multiple distinct matches. The general solution can require exhaustive search. We can easily write such search programs, and even spend time to refine such programs to cut down on the search space if possible, but it is not always possible to avoid exponential-time search.

Another tactic is to try a more explicit solution method which assigns parts to the symbols in the general pattern, but by a more guided method. Explicitly envisioning the task as partitioning of an expression helps develop such methods. *In some cases an exhaustive search of combinations may nevertheless be necessary, but it will be done explicitly and intentionally, not because the pattern-matcher is using a default technique implemented as exhaustive search.*

3 Partitioner

We describe a program that works on sums, products, and potentially other expressions with a different head operator. It is given a predicate `p`, a program that returns true or false given a subexpression of that main expression. The partitioning program separates the given expression into pieces that pass and those that fail the predicate. To make the program useful, we set up 4 other arguments as described below. We then have a fairly general partitioner. We show how to use it in subsequent sections.

Since we will collect those pieces that pass the predicate and those that don't, we need some structure for storing the collections. Let y (for "yes") be the collection that passes the predicate. Initialize y to a parameter `init`. Typically this is 1 for products, 0 for sums. For each term v for which `p(`v`)` is true, we combine v with y by a `combiner`. Typically we find it convenient to combine terms using "*" in the case of products, "+" for sums. Similarly, n (for "no"). Alternatively we can collect terms in sets, or lists, or store them in arrays.

We are almost done with the specification: what do we do with the collected y and n? We could return them as a pair, or try something more devious that would work with other matching facilities. Here's what we programmed. Let the user of the partition program provide 2 more items. The first is an `action` which could just be a dummy function name to apply to the two values y and n. One possibility is for that function to be "[" in which case the partitioner returns a list of [y,n]. The second is a symbolic name `res`, whose value will be bound to the result. This final item is something of a kludge, but makes it easier for certain restricted constructions in pattern matching to get a handle on the result.

Now that we've described the program, we display the Maxima code.

```
/* Our partitioning tool. Note that the expression being partitioned
```

```
is the last argument.
This makes matchdeclare simpler */

partition_expression(operator,pred,init,combiner,action,res,E):=
    block([yes:init,no:init],
   if not atom(E) and inpart(E,0)=operator then
       (map(lambda([r], if apply(pred,[r])=false
              then  no:apply(combiner,[r,no])
              else yes:apply(combiner,[r,yes])),
            inargs(E) ),
       res :: apply(action, [yes,no]), /* result stored as requested */
       true))$

 inargs(z):=substinpart("[",z,0)$
/* inargs is a utility like args() except args(a/b) is [a,1/b]. we want [a,b] */
```

That's it.

EXAMPLE 1: Collect constant terms in a sum

Note that `matchdeclare` is used to associate a predicate with a name. It is not executed in a loop – it is akin to a type declaration. It is not doing any "matching" *per se*. It informs the program that creates patterns for `defmatch`, `tellsimp` and `tellsimpafter` that a pattern matching is successful subject to the additional condition that any assignment to the declared name must pass the associated predicate.

```
(matchdeclare (pp, partition_expression("+",constantp,0,"+",g,'ANSpp)),
tellsimp(foo(pp),ANSpp),
declare([a,b,c],constant),
[foo(a+b+c+x+y+3),foo(w),foo(a), foo(a*b+x),foo(a*b)])
-->
[g(c+b+a+3,y+x),foo(w),foo(a),g(a*b,x),foo(a*b)]
```

Here the intent is to separate components of a sum into two parts: those items that are constant versus those that are not. Each of the two parts will be a sum, initially zero. The result will be `g(yes,no)`, and `ANSpp` will be set to `g(yes,no)`. If `foo` is applied to something that is not a sum, it remains unchanged. Perhaps another rule can be used to transform it.

EXAMPLE 2: Collect factors in a product

This illustrates a task similar to that in example 1, but operating on a product.

```
(matchdeclare(qq, partition_expression("*",constantp,1,"*",g,'ANSqq)),
 tellsimp(bar(qq),ANSqq),
[bar(a*b*3*x*y),bar(w),bar(a+b)])
-->
 [g(3*a*b,x*y),bar(w),bar(b+a)]
```

Here the intent is to separate components of a product into two parts: those items that are constant versus those that are not.

EXAMPLE 3: Separate factors in a product in lists

This illustrates how one can use another mechanism for accumulating factors, namely collecting *lists* of factors.

```
(matchdeclare(ss,
    partition_expression("*",constantp,[],cons,"[",'ANSss)),
tellsimp(bar2(ss),ANSss),
tellsimp(bar3(ss), The_Partitions_Are(ANSss)),
[bar2(3*%pi*xx), bar2(w),bar2(%pi+x+3), bar3(x*3)])
-->
[[[%pi,3],[xx]],bar2(w),bar2(x+%pi+3),The_Partitions_Are([[3],[x]])]
```

EXAMPLE 4: Separate even and odd explicit powers of parameter %v

This example also illustrates the use of Maxima's *sets* {} for collecting data, which might be preferred to lists for some purposes. Note the difference between finding one odd power and finding them all.

```
(matchdeclare(odd,oddp, nov, freeof(%v)),
 defmatch(OneOddPowerp,nov*%v^odd), /*%v is global */
 matchdeclare(tt,
  partition_expression("+",OneOddPowerp,{},adjoin,"[",'ANStt)),
 tellsimp(separatepowers(tt),ANStt),
[block([%v:x],separatepowers(3*x+4*x^5+7*x^10)),
 block([%v:y],separatepowers(3*y+4*y^5+7*x^10)),
 block([%v:a],separatepowers(expand((a+x)^5)))])
-->
[[{3*x,4*x^5},{7*x^10}],
 [{3*y,4*y^5},{7*x^10}],
 [{a^5,10*a^3*x^2,5*a*x^4},{5*a^4*x,10*a^2*x^3,x^5}]]
```

EXAMPLE 5: Implementing a match for f(u)du in an integral

This program looks for a pattern that can be integrated by a simple lookup. It accesses a list of integration formulas keyed on the name of the function `f` which is discovered only after searching through the expression. For certain functions it can compute $\int f(u)du$. In order to do this, the program makes a list of all the factors of the integrand that it can find, and tries some divisions.

We can use `intfudu(expr,x)` directly, or if the expression is not presented as a product, it may be worthwhile to try `intfudu(factor(expr),x)` or perhaps integrate each summand separately.

This program works primarily for functions of a single argument, but we set it up so that it can be generalized to functions of any number of arguments. The usual example of more than one argument is for exponentiation. That is, the "^" operator for u^2 which is encoded as `expt(u,2)`, for $1/u$ which is encoded as `expt(u,-1)` or for 2^u which is `expt(2,u)`. Also it will therefore work for $\exp(u)$ which is `expt(e,u)`.

First, we set up a list of kernel functions f, and a function `intablefun` which returns two items, the integration formula and the derivative of the kernel. This latter item is almost always the same, but for functions of several variables like "^", it matters.

```
 (/* %voi is ``variable of integration'', a global variable */
```

```
  /* shorthand for du/dv used in most table entries */
 dudv(u):=diff(u,%voi),

intablefun(op):= if atom(op) then intable[op] else false,


 intable[otherwise]:=false, /*backstop for undefined kernels */
 intable[log]  : lambda([u],  [-u+u*log(u),dudv(u)]),
 intable[sin]  : lambda([u],  [-cos(u),dudv(u)]),
 intable[cos]  : lambda([u],  [sin(u),dudv(u)]),
 intable[tan]  : lambda([u],  [log(sec(u)),dudv(u)]),
 intable[sec]  : lambda([u],  [log(tan(u)+sec(u)),dudv(u)]),
 intable[csc]  : lambda([u],  [log(tan(u/2)),dudv(u)]),
 intable[cot]  : lambda([u],  [log(sin(u)),dudv(u)]),
 intable[atan] : lambda([u],  [-(log(u^2+1)-2*u*atan(u))/2,dudv(u)]),
 intable[acos] : lambda([u],  [-sqrt(1-u^2)+u*acos(u),dudv(u)]),
 intable[asin] : lambda([u],  [sqrt(1-u^2)+u*asin(u),dudv(u)]),
 intable[sinh] : lambda([u],  [cosh(u),dudv(u)]),
 intable[cosh] : lambda([u],  [sinh(u),dudv(u)]),
 intable[tanh] : lambda([u],  [log(cosh(u)),dudv(u)]),
 intable[sech] : lambda([u],  [atan(sinh(u)),dudv(u)]),
 intable[csch] : lambda([u],  [log(tanh(u/2)),dudv(u)]),
 intable[coth] : lambda([u],  [log(sinh(u)),dudv(u)]),
 intable[asinh]: lambda([u],  [-sqrt(1-u^2)+u*asinh(u),dudv(u)]),
 intable[acosh]: lambda([u],  [-sqrt(u^2-1)+u*acosh(u),dudv(u)]),
 intable[atanh]: lambda([u],  [log(1-u^2)/2+u*atanh(u),dudv(u)]),
 intable[acsch]: lambda([u],  [u*asinh(u)/abs(u)+u*acsch(u),dudv(u)]),
 intable[asech]: lambda([u],  [u*asech(u)-atan(sqrt(1/u^2-1)),dudv(u)]),
 intable[acoth]: lambda([u],  [log(u^2-1)/2+u*acoth(u),dudv(u)]),

 intable["^"]  : lambda([u,v], if freeof(%voi,u) then [u^v/log(u),dudv(v)]
                               else if freeof(%voi,v)
                               then if (v#-1) then [u^(v+1)/(v+1), dudv(u)]
                               else [log(u),dudv(u)]),

/* Rule for integration of 'diff(f(u),u) with respect to u for "unknown" f */

/* Need a fancier rule for integration of 'diff(f(u^2),u) with respect to u.
  Maxima does not have built-in notation for diff with respect to first argument,
  though see pdiff share file */

 intable[nounify(diff)]:lambda([[u]],
  if length(u)=3 and u[2]=%voi then [diff(u[1],%voi,u[3]-1),1]),

/* We can add rules for more operations like this */

 intable[polygamma] :lambda([u,v], if freeof(%voi,u) then [polygamma(u-1,v),diff(v,%voi)]),
 intable[Ci]: lambda([u], [u*Ci(u)-sin(u),dudv(u)]),
 intable[Si]: lambda([u], [u*Si(u)-cos(u),dudv(u)]),
 gradef(Ci(w), cos(w)/w),
 gradef(Si(w), sin(w)/w),
 intable[nounify(bessel_j)] :lambda([u,v], if(u=1) then [-bessel_j(0,v),dudv(v)]),
 intable[nounify(bessel_i)] :lambda([u,v], if(u=1) then [ bessel_i(0,v),dudv(v)]),
```

```
 intable[nounify(bessel_k)] :lambda([u,v], if(u=1) then [-bessel_k(0,v),dudv(v)]),
 intable[polygamma]         :lambda([u,v], if freeof(x,u) then
                               [polygamma(u-1,v),diff(v,x)]),


/* Here are Airy functions.   */
intable[nounify(airy_ai)]:lambda([u],
[-(u*(-3*gamma(1/3)*gamma(5/3)*hgfred([1/3], [2/3, 4/3],
      u^3/9) + 3^(1/3)*u*gamma(2/3)^2*hgfred([2/3],
      [4/3, 5/3], u^3/9)))/(9*3^(2/3)*gamma(2/3)*gamma(4/3)*gamma(5/3)),dudv(u)]),


/* Here are Legendre polynomials P[n](x).  Presumably if n were an explicit
   integer this symbolism would be removed, so we assume it is of symbolic
   order n. */

intable[legendre_p] : lambda([n,u], if freeof(n,u) then
 [(legendre_p(n+1,u)-legendre_p(n-1,u))/(2*n+1), dudv(u)]),

 /* some extra pieces courtesy of Barton Willis */
intable[abs] : lambda([u], [u * abs(u) / 2, dudv(u)]),
intable[signum] : lambda([u], [abs(u), dudv(u)]),
intable[unit_step] : lambda([u], [(u + abs(u))/2, dudv(u)])
)
$

/*etc etc add functions as needed */



(is_sum(x):= is (not(atom(x)) and (inpart(x,0)="+")),

( matchdeclare(ss, partition_expression("*",lambda([u],freeof(%voi,u)),
                                        [],cons,"[",'ANSss))),
 defrule(ddr1,ss,ANSss)), /*this forces the call to partition_expression */

/* Finally, here's the main integration program */

intfudu(E,%voi):= /*integrate E=f(u)*du with respect to %voi*/
 if is_sum(E) then return (map (lambda([r],intfudu(r,%voi)), E)) else
  block([lists,consts,factors, thefuns, therest, thelist, int, df,
         result:false],
   if freeof(%voi,E) then return (E*%voi),
   lists:ddr1(E), /*partition Eression into factors*/
   if lists=false then lists:[[1],[E]],
   factors:second(lists),
   for k in factors do
    if not atom(k) and
      (thefuns:intable[inpart(k,0)])#false and
      (thelist:apply(thefuns,inargs(k)))#false
     then( [int,df]:thelist,
           if freeof(%voi,therest:ratsimp(E/k/df))
               then (result:(therest*int),
```

```
                          return())),
   /*if nothing of form f(u)du worked, then try matching u^1*u'-> u^2/2 */
  if result=false then
      for k in factors do (
          if diff(k,%voi) # 0 and freeof(%voi,therest:ratsimp(E/k/diff(k,%voi)))
               then (result:(therest*k^2/2),
                          return())),
  return(if (result=false) then 'int(E,%voi) else result))
)$
```

This program approximately implements the first stage of Joel Moses' SIN program, and then some. According to Moses [1], SIN'S first stage was able to solve 45 out of the 86 problems presented to James Slagle's earlier SAINT program.

We expect that `intfudu` probably does about 80 percent of freshman calculus problems as given in textbooks, excluding rational function integration.

It was relatively fast to write, but how fast is it to run? Albert Rich has provided a listing of all the 113 explicit test problems in Moses' thesis. Of these, we found that 41 can be done by `intfudu` in a total time of about 0.013 seconds[3]; the Lisp-coded Maxima integration program takes a total of about 0.0069 seconds on the same computer. The answers are almost all identical, the exception being two that differ by a constant. If we rewrite the partitioning program in Lisp[4] the total time taken is about 32 percent faster. Much of the remaining time is taken by calls to programs already compiled. In passing, we note with some relief all of the 113 test problems are (still) successfully integrated by Maxima, whose integration program originated as Moses' routines.

We can further enlarge the scope of this program by adding to `intable`. Making such additional entries *will not slow the program down* as the "kernels" are looked up in a hash table. We could add more complications for functions of several variables, as shown by the example for `Bessel` and `Polygamma`.

4 Additional notes on integration programs

A computer algebra system is expected to have a broader grasp of methods for integration, which might be based on Moses' stage 2, rational function integration, and Risch integration, as well as user-contributed special patterns. Also heuristics such as integration by parts may play a role: partitioning can also be used to simplify this task. We note that `intfudu` may be useful even when there is a more general method for indefinite integration because the answers from those methods may be in a form that is more difficult to comprehend. For example the most straightforward "Risch" methods do trigonometric problems via complex exponentials, and may leave all or some of the results in an unwelcome format. There are opportunities for implementing Risch-like methods to leave results in a preferred trigonometric or inverse trigonometric forms, but detailed discussion of more advanced algorithmic integration methods is beyond the scope of this paper.

Recently another method for integration has been implemented by Albert Rich: a rule-based system called Rubi. `http://www.apmaths.uwo.ca/ãrich/` This system was initially implemented using Mathematica and a collection of disjoint pattern-match and replacement rules; it has been translated entirely to "if-then-else" expressions, and can be run in other systems, e.g. Maxima. An advantage to this approach is that Rich has shown he can produce a higher percentage of correct answers on test suites, and his results are tailored "optimal expressions" with a minimum number of terms or terms of lower complexity. The coverage seems to be mostly of complicated expressions of the usual algebraic, exponential, and trigonometric functions with a sprinkling of special functions.

A suggestion by one referee for this current paper is that there may be more compelling examples for partitioning rather than matching integrands – one involving sums with many terms. We agree that integrands tend to be relatively small, especially textbook examples, and are generally products because integration term-by-term usually works.

[3] on an Intel 3GHz Pentium D

[4] It takes 11 lines of Lisp

How about reducing a large sum by application of trigonometric identities? One could easily collect all expressions E such that $\sin(E)^2$ occurs in a sum, and run a second search for matching $\cos(E)^2$ as well. While this is superficially plausible, we have found that an excellent method is to represent the overall expression Q as a polynomial in sin and cos, and do "long division" by a power k of $r = \sin(E)^2 + cos(E)^2$ for each plausible k and E. Knowing that $Q = r^k * A + B$ tells us that Q is actually equal to $A + B$. This is used in Maxima's `ratsubst` command. Another technique we have used is to replace all trigonometric forms by complex exponentials, rationally simplify using kernels that are simple exponentials, and hope that one can convert back. This too is available, (`exponentialize, ratsimp, demoivre`).

Readers who have extensive pattern-matching transformation suites that partition large sums are invited to suggest applications to the author.

5 Conclusion

If the goal of pattern matching is to partition a sum or a product into separate classes based on a predicate, an explicit partitioning program may be simpler to understand, more efficient, and more general. An application to symbolic indefinite integrate shows how it can be used.

Thanks

Thanks to Barton Willis for running some checks and finding some bugs, and enhancing the program for use in Maxima. Referees for CCA suggested I add one line to `intfudu` so it would integrate explicit sums term-by-term. This is easy but not totally satisfactory for some uses. Previously we returned `false` when `intfudu` failed – a clue to try another method. Now the solution to a sum when some of the terms cannot integrated includes one or more "unevaluated" integrals. These could in principle be handed off to the next technique.

References

[1] Joel Moses, "Symbolic integration: the stormy decade" Comm. ACM, Volume 14 , Issue 8 (August 1971) 548—560

6 Appendix 1 Matching leads to exponential cost

A Mathematica program to generate test sums and then partition them.

```
f1[w1___?(FreeQ[#, x] &) + w2___] := {{w1}, {w2}}
tt[n_] := Sum[A[i, If [OddQ[i], x, y]]  , {i, 1, n}]

test[n_]:=Block[{h=tt[n]}, Timing[f1[h];]]
Do [Print[{i, test[i]}], {i, 1, 24}]
```

Running this test program reveals that longer and longer sequences of expressions like `A[1,x]+A[2,y]+A[3,x]+ ...` are matched and separated into parts that depend on `x` and those that do not. There is a simple expression defining `f1` but executing the match is an exponential function of the length. For i=24, the cost on my computer is over 72 seconds. A similar Maxima test program takes negligible time by using partitioning, as shown below.

```
load("part-in-lisp.lisp")
f1(expr):= block([ans],
        partition_expression(?mplus,freeofx,[],cons ,identity, ans,expr) ,ans)$
tt(n):=sum(a(i, if oddp(i) then x else y),i,1,n);
```

```
h1000:tt(1000)$
f1(h1000)$  /*takes about 0.1 seconds.*/
```

We want to assure the reader that the advantage of partitioning is not simply a trick specific to Maxima or Lisp. A program based on the same partitioning idea can be written using Mathematica as well. Here's the "guts" of such a program to run for this example:

```
PartitionSumByFreeQx[w_] :=
  Block[{yes = {}, no = {}},
  Scan[If [FreeQ[#, x], AppendTo[yes, #], AppendTo[no, #]] &, w];
  Return[{yes,no}]]
```

This program takes about 0.062 seconds in Mathematica 9 on a sum of 1000 terms, about 6 times slower than Maxima, on the same machine.

7 Appendix 2: A Partition Expression program in Lisp

```
(defun $partition_expression  ;; this runs in maxima
    (operator pred init combiner action res e)
  (let ((yes init) (no init))
    (cond ((and (not (atom e))(eq (caar e) operator))
           (map nil
               #'(lambda(r)   ;; maxima value false is lisp nil.
                   (if (mfuncall pred r)
                          (setf yes (mfuncall combiner r yes))
                                ; r fails predicate
                          (setf no (mfuncall combiner r no))))
               (cdr e))
           (mset res (mfuncall action (list '(mlist) yes no))) t)
          (t nil))))
```