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

Richard Fateman

February 24, 2009

Abstract

A popular technique to direct transformations of algebraic expressions in a computer algebra system such as Macsyma/Maxima or Mathematica is to write a out a pattern or template for the expected class of expressions and use a built-in system routine to match it (identifying pieces and giving them names). This is followed by some rule-replacement where those extracted parts of the template are placed in a new configuration, presumably "the answer" or something closer to it. This works especially nicely for applications where there is not a 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, this can constitute 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. One typical problematical situation is when the pattern must match an expression headed by an 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". The corresponding solution may lead to an exponential cost search using backup. Implementation inefficiency is often compounded by the difficulty of composing correct programs through rules. 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.

1 Introduction

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 not such a problem 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 is pretty good at this. But less so in trying something like this: $f(u)u'$, that is, where one item, or perhaps a product of items, is a derivative of some other items 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 a single expression may fail to match, or may have one or more 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 avoid search if possible, but it is not always possible to avoid exponential-time search. Another tactic is to try a more explicit solution method which finds matches, but by a more guided method. Explicitly envisioning the task as partitioning of an expression helps develop such methods. Occasionally, an exhaustive search of combinations may still be necessary, but it will be done explicitly and intentionally, not because the pattern-matcher is using a default technique implemented as exhaustive search.

2 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 we 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 specifcation: 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 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)$ /* utility like args, but avoids / or - */
```

That's it.

EXAMPLE 1: Collect constant terms in a sum

```
(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 terms in a product

Same kind of deal as the previous example, but 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 terms in a product in lists

Same as the previous example but collecting *lists* of terms.

```
(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 also illustrates the use of Maxima's *sets* {} for collecting data, which might be nicer than 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, 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 (if it is a sum) 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, set up a list of kernel functions f, 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 = variable of integration, a global variable wrt these entries */

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),diff(u,%voi)]),
 intable[sin] : lambda([u],  [-cos(u),diff(u,%voi)]),
 intable[cos] : lambda([u],  [sin(u),diff(u,%voi)]),
 intable[tan] : lambda([u],  [log(sec(u)),diff(u,%voi)]),
 intable[sec] : lambda([u],  [log(tan(u)+sec(u)),diff(u,%voi)]),
 intable[csc] : lambda([u],  [log(tan(u/2)),diff(u,%voi)]),
 intable[cot] : lambda([u],  [log(sin(u)),diff(u,%voi)]),
 intable[atan]: lambda([u],  [-(log(u^2+1)-2*u*atan(u))/2,diff(u,%voi)]),
 intable[acos]: lambda([u],  [-sqrt(1-u^2)+u*acos(u),diff(u,%voi)]),
 intable[asin]: lambda([u],  [sqrt(1-u^2)+u*asin(u),diff(u,%voi)]),
 intable[sinh]: lambda([u],  [cosh(u),diff(u,%voi)]),
 intable[cosh]: lambda([u],  [sinh(u),diff(u,%voi)]),
 intable[tanh]: lambda([u],  [log(cosh(u)),diff(u,%voi)]),
 intable[sech]: lambda([u],  [atan(sinh(u)),diff(u,%voi)]),
 intable[csch]: lambda([u],  [log(tanh(u/2)),diff(u,%voi)]),
 intable[coth]: lambda([u],  [log(sinh(u)),diff(u,%voi)]),
 intable[asinh]: lambda([u],  [-sqrt(1-u^2)+u*asinh(u),diff(u,%voi)]),
 intable[acosh]: lambda([u],  [-sqrt(u^2-1)+u*acosh(u),diff(u,%voi)]),
 intable[atanh]: lambda([u],  [log(1-u^2)/2+u*atanh(u),diff(u,%voi)]),
 intable[acsch]:lambda([u], [u*asinh(u)/abs(u)+u*acsch(u),diff(u,%voi)]),
 intable[asech]:lambda([u], [u*asech(u)-atan(sqrt(1/u^2-1)),diff(u,%voi)]),
 intable[acoth]:lambda([u], [log(u^2-1)/2+u*acoth(u),diff(u,%voi)]),

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

/* Rule for integration of 'diff(f(u),u) wrt u for "unknown" f */

/* Need a fancier rule for integration of 'diff(f(u^2),u) wrt u.
  Maxima does not have notation for diff wrt 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),diff(u,%voi)]),
 intable[Si]: lambda([u], [u*Si(u)-cos(u),diff(u,%voi)]),
 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),diff(v,%voi)]),
 intable[nounify(bessel_i)] :lambda([u,v], if(u=1) then [ bessel_i(0,v),diff(v,%voi)]),
 intable[nounify(bessel_k)] :lambda([u,v], if(u=1) then [-bessel_k(0,v),diff(v,%voi)]),

/* 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)),diff(u,%voi)]),


/* 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_(n-1,u))/(2*n+1),
            diff(u,%voi)])
)
$

/*etc etc add functions */

/* Here's the main integration program */
/* %voi is ``variable of integration'', a global variable */

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

 intfudu(exp,%voi):= /*integrate exp=f(u)*du wrt %voi*/
  block([lists,consts,factors, thefuns, therest, thelist, int, df, result:false],
   if freeof(%voi,exp) then return (exp*%voi),
   lists:ddr1(exp), /*partition expression into factors*/
   if lists=false then lists:[[1],[exp]],
   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(exp/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 freeof(%voi,therest:ratsimp(exp/k/diff(k,%voi)))
               then (result:(therest*k^2/2),
                           return()),
  return(result))$
```

This program appears to duplicate the first stage of Joel Moses' SIN program. 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.

It probably does about 80 percent of freshman calculus problems, excluding rational function integration, and is likely to be rather fast.

We can enlarge the scope of this program by adding to `intable`. It is notable that making such additional entries will not slow the program down. We could add more complications for functions of several variables, as shown by the example for Bessel. Here's Polygamma (see below)

```
intable[polygamma] :lambda([u,v], if freeof(x,u) then
[polygamma(u-1,v),diff(v,x)]),
```

Other ways to improve could be based on Moses' stage 2, rational function integration, and Risch integration, as well as user-contributed special patterns, as well as heuristics such as integration by parts, for which partitioning can also be used to simplify the task. We note that there is a more general method for indefinite integration, the answers may be in a form that is difficult to comprehend, so that even if "Risch" can do a problem, a more targeted method may help. In any case, further work on integration goes beyond the scope of this paper.

Thanks

Thanks to Barton Willis for running some checks and finding some bugs. All remaining bugs are, of course, the responsibility of the author.

References

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