A Semantic Matcher for Computer Algebra

Gene Cooperman
GTE Laboratories, Incorporated
40 Sylvan Road
Waltham, MA 02254

1. Introduction

An experimental semantic matcher for computer algebra systems has been developed. Certain new features make it possible to use a rule-based system for tasks which could not previously have been done with traditional rule-based matchers. The new system is also easier and faster to write for many problems than hard-wired code. As an experimental program, little attention has been paid to speed, subject to the requirement of human patience that at least several rules per second be executed.

As a demonstration of its power, a differentiation package has been written for both partial and *total differentiation*. The emphasis was on functionality, rather than convenient user displays. The rules and predicate functions required three pages of code.

Future tests will include an attempt to emulate the higher methods of the PRESS system. [1] A working system would make available to other computer algebra systems the equation-solving abilities of PRESS.

The current implementation runs either as a stand-alone package or as a package in MACSYMA [6]. It contains no parser or display package aside from LISP's own. However, it is hoped that the internal representation is sufficiently simple, so that it can be easily interfaced to other LISP-based systems to take advantage of their parser, display, and built-in algebraic routines.

2. Features of Matcher and Comparison with Some Existing Systems

Features of the matcher include:

1) individually user-declarable attributes for all functions. Current attributes correspond to the abelian group axioms: commutativity, associativity, identity, and inverse. The first three attributes are the same as discussed by McIsaac [7], where commutativity is called symmetry in that article. The matching algorithms for handling attributes here are also similar to [7]. This is similar to SMP's [2] properties on symbols. However, the attributes in this matcher affect only whether a match occurs. They do not cause the arguments to be reordered or expressions altered in the absence of any rules, as does SMP. Michael Genesereth described a matcher with such attributes in an unpublished manuscript [5], but it was not included in MACSYMA.

2) restriction of pattern variables to match expressions according to user-defined predicates. For example, the statement `matchvariable(even, (lambda (x), numberp(x) and is( x mod 2 = 0)))` *might* be used *in conjunction* with the rule `cos(even*var)` $\rightarrow$ `1`, where var matches arbitrary expression. This is similar to MACSYMA's matchdeclare property, or SMP's conditions on generic symbols used in patterns.

3) restriction of function pattern variables to match functions according to user-defined predicates. This is necessary for such rules as `diff(fnc(expr),x)`$\rightarrow$ `partial_diff(fnc(expr))*diff(expr,x)`. "fnc" can be declared a function pattern variable, which matches any symbol in functional position, subject to a user-defined *matchfunction* predicate similar to the *matchvariable* predicate in item 2, above. This feature does not seem to exist in current matchers.

4) the ability to define predicates of more than one argument associated with more than one pattern variable. These are associational predicates. Evaluation of the associational predicate is delayed until all pertinent pattern variables are instantiated. If the matchvariable predicates on the individual variables have been satisfied, and the associational predicate fails, then the last instantiation of a pattern variable of the associational predicate is rejected, and the matcher proceeds as if that pattern variable's matchvariable predicate had failed. For example, `diff(no_has_var,var)` $\rightarrow$ `0`, where an associational predicate waits until both no_has_var and var have been instantiated, and then tests if no_has_var contains var with a custom LISP function. Moses's SCHATCHEN matcher [8] (used internally in MACSYMA) has a similar facility, "loop". By declaring the appropriate functions such as "+" to be commutative, and declaring its identity, 0, the associational predicate will exhibit the same behavior as "loop" facility in testing the associational predicate on all combinations of instantiations of terms which satisfy the individual matchvariable predicates. MACSYMA's user-level pattern matchers do not support this delayed evaluation. The manual on SMP is unclear as to whether this is supported.

5) evaluation of *subexpressions* in the replacement of the pattern within the environment existing at the time of the match, including pattern variable bindings. For example, `a` $\rightarrow$ `a+1` and `a` $\rightarrow$ `match_eval(a+1)` would differ in that 4 would transform to 4+1 under the former rule, and 5 under the latter rule. A second construct, match_eval_splicing acts similarly, but splices a list into a list of arguments. For ex-

ample, if "+" had not been declared associative, one might want a rule: `'+'('+'(rest_of_args), rest_of_args2) →` `'+'(match_eval_splicing(rest_of_args, rest_of_args2))`. The pattern variables rest_of_args are rest pattern variables discussed in item 6. `match_eval` can be especially important to manually control resimplification on a rule by rule basis if full resimplification of expressions after every rule application is not desired. The facility, match_eval, is also provided in Moses's SCHATCHEN matcher [8] under the name eval.

6) "rest" pattern variables able to match several arguments, similarly to the &rest lambda-list keyword in COMMON LISP and SMP's multi-generic symbols for patterns. A rest pattern variable matches a sublist of the list of arguments of a function. By creating a new list to which the rest pattern variable is instantiated, this facility can be thought of as a partial inverse to match_eval_splicing, which splices out one list. For an example, see the discussion of the previous item. A concept similar to rest pattern variables is discussed by McIsaac [7] under the name, "ellipsis".

7) association of flags with every expression and subexpression. This allows information to be cached, allowing more efficient rule-based systems. In Backus-Naur form, the internal form of an expression operated on by the matcher, is:

```
expression ::=
  LISP_atom | ((function {flag}*) {expression}*)
```

The {...}* indicates zero or more of the item in braces. This is similar to MACSYMA's internal form for general representation. An example of its use might be to cache information about an expression. For example, the expression matched to no_has_var in item number 4, above, might have that information cached so that it could be retested later, if the same instantiation was attempted later.

8) grouping of rules into rule-sets which can be separately enabled, disabled, traced, and untraced.

9) utilities for easily defining, obtaining, and removing rules, attributes, and match predicates for variables and functions.

3. Example: Partial and Total Differentiation

The package for partial and total derivatives requires about 30 rules and 80 lines of code in Franz LISP. The code defines the matchvariable, matchfunction, and associational predicates. The package was completed in less than two days. This was facilitated by the availability of tracing of rules for debugging.

The code embodies partial and total differentiation, user-defined derivatives (similar to "gradef" in MACSYMA), and handling of dependencies of variables (similar to "depends" in MACSYMA). It handles both explicit functions such as f(x), and implicit functional dependencies declared by such statements as `depends(y,x)`. The package was used with MACSYMA to take advantage of MACSYMA's parsing, display, and simplification of expressions. Simplification of expressions can also be handled in a rule-based manner, but for reasons of efficiency, one might prefer to carry out low-level simplifications with hard-wired code.

Its use is shown below in a mode in which a single rule at a time is applied by the `match` function. The functions, pd and td, symbolize partial and total differentiation, respectively. The symbol "%" refers to the expression on the immediately preceding line, beginning with "(d<number>)".

```
(c71) gradef(f(x), fprime(x));
   /* user-defined gradient of fnc., f is fprime */
(d71)                                     f(x)

(c72) match(td(f(y**2),y));
                                                 2
(d79)                              2 y fprime(y )

(c81) match(pd(f(y**2)));
  /* pd(f(y**2)) is different from td(f(y**2),y) */
                                                2
(d81)                                 fprime(y )

(c82) depends(y,[eps,k1,k2],eps,[k1,k2],k1,3);
  /* y depends on eps, k1, and k2; etc. */
(d82)          [y(eps, k1, k2), eps(k1, k2), k1(3)]

(c89) match(td(y,e));
  /* chain rule for partial derivatives  */
                               dk1 dy    deps dk1  dy
(d92)                          --- --- + ---- --- ----
                               de  dk1   dk1  de   deps

(c93) match(pd(y,k1));
  /* single partial derivative */
                                         dy
(d93)                                    ---
                                         dk1
```

4. Previous Work

The matchvariable concept used here is inspired by the matchdeclare concept in Fateman's matcher for general MACSYMA expressions. [3] Many newer features have been added in order to allow more complicated bodies of mathematical knowledge to be easily expressed in a rule-based manner. The previous examples are some cases in point.

Michael Genesereth [5] had previously written a matcher using Fateman's matchdeclare concept in which one could declare addition and multiplication to have any combination of the properties commutative, associative, identity, and general field axioms. This is similar to our attributes, but they were not extended to apply to arbitrary functions. The previous examples stand here, also, as cases where the newer features described in this paper would be desirable.

SMP [2] and McIsaac [7] use properties that perform some of the same functions as our attributes. Their properties include Flat, Comm, and Dist (associative/symmetric, commutative, and distributive). Some of SMP's properties, such as dist, are implemented as rules in our system. McIsaac contains ellipsis and SMP contains multi-generic symbols of the form $$var, that serve a similar purpose to our rest arguments. Certain of our features (e.g.: matchfunction, match_eval, rule-sets) have no analogue in those systems.

An early rule-based computer algebra system had been designed by J. Fenichel. [4] He used a purely syntactic matcher and rules for commutativity, associativity, and more complex axioms, in order to algebraically simplify expressions. Fenichel's system performed correctly but slowly, because the basic axioms (our attributes) were expressed by rules. Thus simplifying (a+(b+(c+d))) to (((a+b)+c)+d) required several applications of the associative rule, possibly including backtracking. The philosophy of this matcher differs in incorporating the commonly used axioms in our system as separate attributes, and leaving only the "special-case" rules for the matcher.

A more recent example of a rule-based computer algebra program is Bundy's PRESS program for elementary algebra. [1] This is of special interest since it also has the goal of expressing a large body of mathematical knowledge in a manner depending heavily on rules. Some of the rules of PRESS are being implemented in this matcher to gain experience on the strengths and weaknesses of using this matcher. The more important test, to be carried out, is to emulate in this matcher some of PRESS's higher level methods, which had not been implemented in rule-based form. The tendency of PRESS has been to use rules at the lower levels, and use hard-wired code at the higher levels which are able to use the lower level rules. The interest in this work is to incorporate higher constructs in the rule-based matcher, so as to directly express higher-level knowledge in a rule-based form.

5. A Matcher as a Programming Language

The combination of the rule-based and procedural style can be more advantageous than either one alone for programming mathematical algorithms. Where the rule-based paradigm is natural, its extreme modularity causes the resulting program to be easier to understand, debug, and maintain. By making use of the matcher, a rule-based program will often be shorter than its procedural counterpart. Yet procedural programs are more natural in contexts where the algorithm is most naturally expressed in a procedural manner, or the greater efficiency of direct coding is required.

The coexistence of the two styles is especially important for low-level manipulations which are more efficiently done by standard LISP code. Any LISP program may call the matcher, and the matcher may call any LISP program. An example follows to demonstrate how a LISP program may be incorporated in the rule-based system.

```
defrule(universal_expr, match_eval(replacement))

(matchvariable '$universal_expr #'universal-program)
(declare (special replacement))
(defun universal-program (arg)
        (cond ((eq *total-target* arg)
               (setq replacement
                  (arbitrary-program arg)) t)
              (t nil)))
```

The condition `(eq *total-target* arg)` is required to guarantee that the subexpression being matched by universal_expr is the full top-level expression. With that sole condition satisfied, `arbitrary-program` is applied to the target and the result bound to the special variable, `replacement`. The `match_eval` then returns the value of `replacement`.

The effect of subroutines can be obtained by the matcher changing the active rule-set, and declaring a rule in the new active rule-set which deactivates it, and reactivates the old rule-set when no more matches apply in the new rule-set. An alternative technique is for the matcher to recursively call itself. Procedural algorithms with several steps can also be emulated by having each rule-set deactivate itself, and activate the next rule-set when no more rule applications are possible.

6. Future Work and Conclusions

Experience with incorporating other bodies of mathematical knowledge will determine the strengths and the limitations of this style of programming. The two days to program and debug the differentiation package discussed here, demonstrates the usefulness of the matcher for a domain in which the rule-based paradigm is natural. One would ultimately like to use the matcher to create large mathematical expert systems to accomplish some of the same tasks for mathematics that traditional expert systems have accomplished for other domains. One is not restricted to using only the rule-based approach since the two programming paradigms can coexist.

Certain restrictions are in effect in the current matcher. The interaction of multiple rest arguments with matchvariable properties of the rest arguments can be ill-defined. Matchvariable_assoc has not been fully tested with commutative and associative functions. These restrictions may be lifted in future versions.

A possible future application involves keeping a subset of the rules in restricted form so as to allow automatic derivation of useful, new rules to be installed. The Knuth-Bendix completion algorithm is one such technique which has been used for rewrite rules. The feature of interchangeability of program and data is already familiar to users of LISP and PROLOG.

In line with the expert systems paradigm, this matcher is expected to be most useful in mathematical domains with numerous algorithms, heuristics, or rules-of-thumb, which are not expressed easily in the language of an existing computer algebra system. Examples might include asymptotic analysis, transforming expressions to self-adjoint form, transformation of differential equations to numerical FORTRAN programs, or specialized domains for which the effort of writing a full computer algebra package is not worthwhile.

7. Acknowledgements

The author gratefully acknowledges the programming assistance of Jeffrey Higginson and Bradley Turner. He also wishes to thank Jeffrey Golden for many helpful discussions on the design of a matcher for computer algebra. He wishes to thank Bernard Silver for fruitful discussions on the techniques and architecture used by PRESS. Finally, he wishes to thank the reviewers for their helpful suggestions.

References

[1] A. Bundy and B. Welham, "Using Meta-Level Inference for Selective Application of Multiple Rewrite Rules in Algebraic Manipulation", *Artificial Intelligence* **16**, #2, 1981

[2] C. Cole, S. Wolfram, et. al., *SMP: a Symbolic Manipulation Program* (Manual, Version one), Cal. Inst. Technology, 1981

[3] R. Fateman, "The User-Level Semantic Matching Capability in MACSYMA", *A.C.M. Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, Cal., March, 1971

[4] R. Fenichel, *FAMOUS*, M.I.T. Ph.D. Thesis, Cambridge, Mass., 1969

[5] M. Genesereth, no title (unpublished manuscript), M.I.T., 1979

[6] *MACSYMA Reference Manual*, (Version 10), The Mathlab Group, Laboratory for Computer Science, M.I.T., Jan., 1983

[7] K. McIsaac, "Pattern Matching Algebraic Identities", *SIGSAM Bulletin*, **19**, #2, May, 1985

[8] J. Moses, "Symbolic Integration", Chapter 3, MAC-TR-47, Project MAC, Dec., 1967 (available from Defense Document. Center, AD # 662666)