

Design for a mathematical transformation system for symbolic expressions

Richard J. Fateman
Computer Science Division
Electrical Engineering and Computer Sciences
University of California at Berkeley

December 15, 2011

Abstract

One kind of “transformation system for symbolic expressions” is a program that supports rules: a search for patterns and subsequent replacement based on the pattern matching. That is the pattern produces identifications of subparts (binding names to them) in that selection, and then a replacement expression is produced. If the new expression is equivalent but “simpler”, then this program, applied repeatedly, may be a simplifier. If the new expression is different, but related in value, it may represent a step in some sequence of manipulations. We discuss the issue primarily in terms of the Mathematica system which already has a version of this idea: indeed, almost all of Mathematica’s programming language semantics is based on this mechanism, a situation that sometimes confuses users who often think they are defining “functions.” The Mathematica design is interesting but flawed; there are some remedies, however. We present the issues and remedies.

1 Introduction

Mathematica’s basic functionality is expressed with `Rules` and `ReplaceAll`, where selection (or matching) and replacement are indicated by the left and right sides of a `Rule`. The program usually used to apply rules is `ReplaceAll`, but there are many variants; furthermore many users who think they are “defining functions” are in actuality defining rules for the Mathematica evaluator.

The most obvious flaws in the Mathematica system arise because the transformations are based on patterns matching the underlying internal forms of expressions in Mathematica. These are familiar to computer scientists who are schooled in compiler and data representation technology: they will recognize this form as an “intermediate expression tree.” Unfortunately, parts of this form may be unfamiliar to Mathematica users, since it does not necessarily conform to the more familiar surface syntax used by Mathematica for input and output of mathematical expressions, and there is no particular reason to expect a user to be aware of the nuances.

For an example where the process of applying rules can result in puzzling results, consider a rule to replace `I` by `-I` in an expression¹ The obvious rule will transform `a+b*I` to `a-b*I` but will leave `3+ 4*I` unchanged. The discrepancy can best be explained via the `FullForm` printouts which are a more faithful rendition of the internal form: `Plus[a,Times [b, Complex[0,1]]]` and `Complex[3,4]`, respectively. Note that the latter has no occurrence of the subexpression for `I`, which is `Complex[0,1]`, and so the pattern `I` is nowhere to be found.

We address the question of how one might implement a system for defining and applying rules so that the rules are interpreted in a mathematically meaningful way, especially when they fall short of the syntactic specificity currently required for use by Mathematica’s `ReplaceAll`. In fact we note that more “semantically oriented” rules can be expanded to sets of rules or otherwise altered to conform to the syntactically specific requirements of Mathematica.

¹ “I” is $\sqrt{-1}$ in Mathematica.

The argument sometimes offered, that Mathematica *already does the right thing* by “literally” matching expressions, loses much of its force by observing how frequently mathematicians report its behavior as erroneous. Users are easily fooled into thinking that “literal” matches should work on what they literally typed in to the system. Furthermore, a more semantically-oriented rule system can be implemented on top of the primitive rule structure, and can coexist with it through adding *another* pattern-replacement operation: an alternative to `ReplaceAll`.

It is also worth noting that historically, others have suggested such a move in the past to change what Mathematica does, as shown by Harris², and other computer algebra systems have been doing a different (and arguably, in places) better, job for years.

2 Matching

The *basic* idea is simple, and can, in some sense be made extraordinarily simple without loss of “computational completeness”.

Axel Thue (1914) described a formal mathematical string transformation mechanism as a theoretical encoding for computation; eventually this was shown to be equivalent to a “Turing machine”, a subsequent theoretical invention which is now the standard mathematical basis for proofs of non-computability of certain functions.

Subsequent to this mathematical theorizing, it turns out we can define string transformation computer program implementations, which are (subject to limits of time and space) capable of computing anything that can be computed. Not necessarily fast.

Extending the Thue-system formal system for defining rules (substantially) provides a far more convenient framework for programmers. One form or another of string pattern matching has been a foundational basis for classes of programming languages (notably, SNOBOL). In the form of “regular expressions” string patterns are still in widespread though perhaps somewhat esoteric use in editing and search applications. For a period of time, rule-based transformation systems became popular as vehicles for “expert systems” as well.

Another kind of matching of patterns is that used for logical unification, where the idea, loosely speaking, is to take two patterns or formulas $p(x)$, $q(x)$, and find a value v for x , (x is in general a vector of names, v a vector of values or patterns), such that $p(v) = q(v)$. This is widely used in logic and theorem proving, and a version of this became popular as defined for the programming language Prolog.

In the setting of a symbolic mathematical system such as Mathematica, the strictly string model falls short as a suitable representation for general mathematical expressions. Since the common operators of addition and multiplication are commutative with identities that “disappear” from the internal structure, looking for a literal pattern such as “a+b” excludes the pattern “b+a”, and well as “a”, a possibility when $b=0$, etc. Thus to teach a system that “a+0” should be transformed to “a” is insufficient to transform “0+a”.

3 Why do we see so much pattern “syntax”?

Presenting a semantic concept by encoding it as a collection of simple patterns ($a + b$ OR $b + a$) etc. is generally bulky and difficult to compose, at least for realistically complicated mathematics. Computer algebra systems tend to embroider the pattern-matching process to cut down on the proliferation of rules that are semantically equivalent. This cuts down on the bulk, reduces the likelihood of missing some cases, and probably speeds up processing. That is, the embroidery will allow for missing operands in sums to match 0, in products to match 1, as well as allowing single variables in a pattern to match more than one object, and many more tricks.

This topic has been discussed in numerous papers since at least the mid-1960s. (Fenichel, Moses, Jenks, Fateman, Hearn, Cooperman, Bundy, Greif, McIsaac, Harris).

²<http://www.mathematica-journal.com/issue/v7i3/features/harris>

4 Why programming via patterns is initially attractive

Writing simple programs via patterns is simple, but the simplicity can be illusory. That is, the simple program may not work, for reasons that cannot be explained simply. Going beyond simple transformations, it is apparent that complex programs require careful design and an understanding of the subtle ways in which multiple rules interact. Just as it is possible to design and code incorrect or inefficient conventional programs, so it is possible to write poor rule-based systems. Many novices in rule-writing will not detect inefficiencies or redundancies; indeed it is plausible that collections of rules will cause infinite loops. Sometimes match programs introduce subtle strategies to try to head off such processing situations. Let's try a simple rule set.

```
fact[0]:=1
fact[n_]:=n*fact[n-1]
```

This implements a factorial function in Mathematica. Note that if the rules are always applied last-in first-used, it might define an infinite loop: Consider `fact[0]->0*fact[-1]`. Whether this becomes a loop depends on whether `fact[-1]` etc is computed, or if the (implicit) rule `0* anything -> 0` is computed.

We might want to add some error checking as shown in the next set of rules which print messages if the argument to `fact` is negative or not an integer. The syntax involving `"/;"` adds a condition, so that `n_/;!IntegerQ[n]` matches `n` only if it is not an integer.

```
fact::badarg="The argument '1' is not a positive integer."
fact[n_/;!IntegerQ[n]]:= Message[fact::badarg,n]
fact[n_/;n<0]:= Message[fact::badarg,n]
```

As indicated earlier, we are already treading (conceptually) on thin ice; we are depending on the system to apply rules in a particular order: it must check for error conditions ($n < 0$) *before* computing $n - 1$ etc. How did it know to do that? Will it always be able to do that? What if we defined the rules in a different order?

(Answers: It figured out which was the more general rule *in this case*, and can test the more specialized "exception" rules first. No, it can't always do that, but a rule with *no* constraints is at least as general as one with constraints. Mathematica applies the rules in a first-in, first-used order, *if it can't tell which is more general*. As for when it can tell, it isn't telling.)

A careful designer may aim for a scheme where the patterns consist of mutually exclusive tests whose union entirely covers the domain. In the `fact` definition we have covered all integers and all non-integers, for example. It may not be obvious, but symbolic arguments such as x will fall into the non-integer category. Even if the system is told `Element[x,Integers]`, the expression `tt x` is not an integer. Perhaps for an argument z which is not a number at all we should provide yet another message, or perhaps leave the result as `fact[z]`. Somehow the rule-based program doesn't seem quite so simple any more.

Before proceeding further, we provide a simple where Mathematica clearly can't tell which rule is more general:

```
Clear[f]
testa[a_]:=True
testb[a_]:=True
f[a_/;testa[a]]:= 1 (* this is used first *)
f[b_/;testa[a]]:= 2 (* this is never used *)
```

Just as it is possible to write conventional programs that have inaccessible statements or do not terminate or return unintended results under certain input conditions or wastefully compute things that are never used, it is possible to write poor sets of rules. We believe that people with some experience using conventional languages³ find learning a similar subsequent run-of-the-mill programming language relatively easier than the first. We believe that the rule-based paradigm is sufficiently different that such experience does not transfer easily. Especially for developing significant programming projects, the kinds of hazards are relatively novel. For writing a large collection of interacting rules, one must master the techniques for unambiguous pattern

³Say ordinary sequential state-based programming, or functional programming

definition, understand the binding times used in match processing, and understand rule-ordering heuristics. Most people do not master these completely, and are reduced to trial and error.⁴

5 Why this paper?

This paper describes some aspects of the rule-based situation, often specifically with respect to Mathematica, a system whose dependence on pattern matching is substantial, and whose matching mechanism appears to incorporate nearly every feature that has occurred in previous programs, and more. Yet it does not do what some people (reasonably) expect. How can a system design satisfy the expectations of users?

To anticipate one direction of this paper, let us point out that matching to a somewhat obscure internal form leads to problems; users can plausibly expect matching to something closer to the form that Mathematica displays. If you type `Exp[x] + 1/Exp[x]` into Mathematica you might believe that the result has two occurrences of the pattern `Exp[x]`, but it has none. Instead it has powers of `E` corresponding to e^x and e^{-x} , as can be seen in its “FullForm”: `Plus[Power[E, Times[-1, x]], Power[E, x]]`.

Another theme is that people do not have a clear idea of how much work a pattern matcher will do. Will a pattern `n*(n+1)` match `12` with `n=3`? How hard would that be, really⁵?

After writing an initial draft of this paper we came across the (earlier) work of Jason Harris on Semantica, and consider that to be a step in the right direction for most people. Unfortunately, the details are tricky and Semantica does not solve the particular problems that first prompted us to write this essay (mostly, matching constants!). On the other hand it works for patterns that are more sophisticated in some respects.

6 An example: A quadratic form

Quadratic forms are widely used in mathematical applications. Recognizing them leads us to uses of the quadratic formula, many integration formulas, and much more.

To be specific let us try to define a pattern to implement this notion: If you see a quadratic $p = av^2 + bv + c$ in the variable v , call `qq[c,b,a]` else `notqq[p]`. (We use `qq` and `notqq` as stubs for programs that would define whatever else you might wish to compute. For example, if you wanted to “complete the square” of a quadratic in v , you could define `qq[c_,b_,a_] := a*((v + b/2/a)^2 + c/a - b^2/4/a^2)`.)

To see how one might write this *entirely without patterns*, consider this program:

```
quad=Function[{ex,var}, Module[{ans=CoefficientList[ex,var]},
  If [Length[ans]==3 && FreeQ[ans,var], Apply[qq,ans], notqq[ex]]]]
```

The above program works correctly for all the examples we use below, including examples which present difficulties for potential patterns. We start with the obvious pattern version `quadp` below.

```
quadp[a_ x_^2+b_ x_+c_,x_] := qq[c,b,a]
```

This works fine for `quadp[3 x^2 + 4 x + 5, x]`, but it fails for the quadratic `quadp[x^2 + 4 x + 5, x]`

There is a simple fix for this: a default match for `a` of 1. We can also make `c` match anything that is left over, too,, using

```
quadp[a_. x_^2+b_ x_+c___,x_] := qq[c,b,a]
```

There is a subtle error here because `c___` returns a `Sequence`. If we try `quadp[3 x^2+4 x+1+z,x]` we get `qq[1,z,4,3]` instead of `qq[1+z,4,3]`. Here’s one fix, using `BlankNullSequence` or three `_` underscore characters, one elaboration in Mathematica’s matcher.

```
quadp[a_. x_^2+b_ x_+c___,x_] := qq[Plus[c],b,a]
```

Here’s another

⁴Some people skip the first step and go directly to the error.

⁵You might not have noticed that the match `n=-4` also works, and that in general to match `n*(n+1)` to x we are implicitly solving $n^2 + n - x = 0$

```
quadp[a_. x_^2+b_ x_+c_.,x_]:= qq[c,b,a]
```

where c is matched to the default within a sum (zero), if it is not otherwise found. Since “+” is “Flat” the pattern variable c can match a sum.

But there are yet more problems:

It fails for `quadp[x^3 + 5 x, x]` which should not be recognized as quadratic. It fails for `quadp[x^2 + 5 x Sin[x], x]` which should not be a quadratic.

Here are some more fixes

```
quadp[a_. x_^2+b_ x_+c_.,x_]:= qq[c,b,a] /; FreeQ[{a,b,c},x]
```

This fails for `quadp[x^2 + 5, x]` not recognized as quadratic.

We must make a match of `c=0` when there is no constant term. There is an `Optional` syntax marker (`:`) to do this. Also we need to make the binding `b=0` when there is no linear term. There may be another way of doing this using `Optional` and/or `Alternatives` but I could get this to work and instead propose the solution below. Duplicating the first rule omitting the linear term does it. Finally, we add a rule to deal with the non-quadratic case.

```
Clear[quadp]
```

```
quadp[a_. x_^2+ b_. x_+c_., x_]:= qq[c,b,a] /; FreeQ[{a,b,c},x]
```

```
quadp[a_. x_^2 +c_., x_]:= qq[c,0,a] /; FreeQ[{a,c},x]
```

```
quadp[w_,x_]:=notqq[w]
```

Still the program is not very good at embodying quadratic-ness.

It fails for `quadp[x^2 +a x^2+ 5 x, x]` which is a quadratic. It fails for `quadp[x*(x+1), x]` which is a quadratic. It fails for `quadp[(x^3+x)/x, x]` which is a quadratic.

In some circles it is acceptable for a program to be wrong, so long as it is fast. As it happens, the pattern version is not only wrong, but slow.

We thought that matching the variable first would make the system run much faster, so we define a “backwards” `quadp`, and some test cases:

```
h100= Sum[z[i]^2+z[i],{i,0,99}]
```

```
h1000= Sum[z[i]^2+z[i],{i,0,999}];
```

```
Clear[bquadp]
```

```
bquadp[x_, a_. x_^2+ b_. x_+c_.] := qq[c,b,a] /; FreeQ[{a,b,c},x]
```

```
bquadp[x_, a_. x_^2 +c_.] := qq[c,0,a] /; FreeQ[{a,c},x]
```

```
bquadp[x_,w_] :=notqq[w]
```

If we have a quadratic in many variables $x^2 + y^2 + z^2 + \dots$, it seems less efficient to determine if the expression is a quadratic in some variable and then afterwards specify the variable, requiring the matcher to cycle through all possibilities. If the pattern matcher goes strictly left-to-right, it would be faster to place the variable on the left, as in `bquadp` above. The Mathematica rule definition process when applied to function definitions (using “`:=`”) may spend some time to optimize the rules (see the description of `Dispatch`); unconstrained match for x, matching it faster, but we are merely speculating here.

If time is important, the original non-pattern based program, `quad` is about 160 times faster than `quadp` in comparing `quad[h100,z[99]]` `quadp[h100,z[99]]` and as mentioned earlier, `quad` gets the answer right.

We are puzzled as to why `bquadp` is almost as slow as `quadp`, and in particular, why *EACH* of them is much faster at identifying a quadratic in `z[0]` than `z[99]`. For `bquadp` it should not matter which index is used, at least if the x argument is matched first and then the pattern match for the second argument is conditioned on that. Apparently that optimization is not available regardless of the order of arguments. If we change the target `h100` expression so that it contains only simple atoms like `z99` instead of `z[99]`, and

search for a particular variable identified in the pattern (not by matching, but also as a constant name like `z99`), we eliminate all backtracking and the program is much faster; almost instantaneous. This suggests that for Mathematica, it pays whenever possible, to separate the binding of names of specialized variables (e.g. let `v` be the variable of integration), well before doing pattern matching.

Here's a constant-matching program

```
Clear[cquadp,z]
z[i_]:=StringJoin["z",ToString[i]]
cquadp[ a_. z[100]^2+ b_. z[100]+c_.]:= qq[c,b,a] /; FreeQ[{a,b,c},z[100]]
cquadp[ a_. z[100]^2      +c_.]:= qq[c,0,a] /; FreeQ[{a,c},z[100]]
cquadp[w_]:=notqq[w]
```

6.1 Patterns? Wasn't quadp a function?

As we have suggested earlier, users (as a useful mental shortcut) often think that *function definition* is done in Mathematica by rules. For example, something like `f[x_]:=x+1` and thus the examples above are not rules at all, and the definition of `quad` using `Function` was some kind of strange and unnecessary part of Mathematica.

Really that locution of “:=” or `SetDelayed` is just defining a rule known to the evaluator, the rule being `f[x_]>x+1`. Roughly speaking, the evaluator will try to apply that rule whenever possible. When confronted with an expression `expr`, Mathematica will do this: `Replace[expr, f[x_]>x+1]`.

If you need to define a function in Mathematica, not via simulating a function by rules try something like `f=#+1&` or the less abstruse equivalent `f=Function[{x},x+1]` used in defining `quad`.

7 Tricky bits

Before we get on to the meat of the subject, we feel some obligation to point out some of the tricky bits. The reader is, however, under no obligation to read this section in this order, or perhaps ever.

There are (potentially) separate scope issues and execution time issues for rule definition. When a Rule `x->y` is defined, are `x` and `y` evaluated? (Answer: yes, both.)

When a Rule `x_>y` is defined, are `x` and `y` evaluated? (Answer: only `y`.)

As for `x_`, the name `x` is a variable name that will be temporarily bound to some matching object during the evaluation of `y` again. That's right, `y` is evaluated as part of the rule, and then again when the rule is applied, but with `x` bound.

There is another way of defining rules: In `x_>f[x]`, the form `f[x]` is not evaluated at rule-definition time, but only at application time, since the rule definition is via

`RuleDelayed, :>`. What about other items named in the rule? Let us define

```
GreaterThan[q_]:= #>q&
gt2=GreaterThan[2]
```

This could be used as `GreaterThan[3][4]` which returns `True` since `4 > 3`.

In `x?(GreaterThan[r]) -> f[x]`, when is the predicate `GreaterThan[r]` accessed? In particular, when can we change the value of `r`? Is the value of `r` fixed at the time the rule is defined or is it unset until application time? What if `GreaterThan` is redefined between the time the rule is defined and the time of rule application?⁶

All these, and more, issues are necessarily resolved concretely in the implementation of the pattern match facility. Yet, given its potential for confusing the customer, the documentation tends to de-emphasize or even ignore such issues. Fortunately, it often it doesn't matter, at least if symbols mentioned here and there

⁶For Mathematica, the only values and function definitions that matter on the left-hand side of the rule appear to be those set at application time. On the right-hand side, it depends on `:>` versus `->`, where the latter evaluates the right-hand side when the rule is defined, and then again when the rule is applied.

are uniquely mapped to variables, placing them all in some global environment may be just fine. If function definitions never change, and scope is never nested, it almost doesn't matter. However, if you build upon someone else's program and accidentally re-use a name, you may unintentionally stumble into a morass.

There are at least two circumstances in which these issues must be addressed:

- Writing the pattern matcher or reverse-engineering one;
- Writing programs that take *patterns* as input, and produce *new patterns* as output.

We did the first with `MockMMA[]`, using a fairly standard base for a pattern matcher and modifying it to accommodate the features needed for Mathematica. The remainder of this paper describes a situation in which the second motivation prevails. It seems that one can use `HoldPattern` and `Verbatim` in Mathematica for some of this, but the descriptions and examples are unclear.

How can one write a pattern that matches and dissects patterns? (Yes, this can be done, but with some difficulty). For example, must one generate new names for pattern variables, or will the scoping rules keep you from getting into trouble? (so-called hygienic macro expansion).

We will generally ignore discussing the tricky bits in later sections, since our point is largely orthogonal to these issues. A description of some of these bits is available to anyone with access to documentation for Mathematica (version 7 is what I looked at). This presents a particular implementation standpoint. The document explains binding and scope as renaming. Mathematica is probably the only programming system to use the inefficient idea of truly instantiating “new symbols” for this purpose, but you can read about it in “tutorial/VariablesInPureFunctionsAndRules”. Warning: an understanding of the documentation may require a very careful reading.

8 How is it that Mathematica's Rule Replacement Facility Falls Short?

8.1 Recognizing explicit constants with substructure

The first set of examples are rules whose left-hand side (LHS) is free of pattern variables. That is, from the perspective of pattern transformations these are constant expressions: there are no “pattern match variable” components in the LHS.

Pragmatically, for the Mathematica user using the usual syntax, we mean that no LHS has an item named with one or more trailing underscores like `a_`, `a_ _`, etc. Viewed in `FullForm`, these items are not truly names but expressions that look like `Pattern[a,Blank[]]`, `Pattern[a,BlankSequence[]]`, etc.

Here are the examples of rules whose property may be surprising. Each has a constant LHS expressions for patterns.

`I -> -I` will not change `3+4I` to `3-4I` , but will change `a+b I` to `a-b I`.

`2 -> 4` will not change `1/2` to `1/4` , but will change `f[2]` to `f[4]`

`1/4 -> fourth` will not change `3/4` to `3 fourth` but will change `x/4` to `fourth x`.

`s^2 ->t` will not change `1/s^2` to `1/t` but will change `s^2` to `t`.

`1/s -> t` will not change `(1/s)^2` to `t^2`

`Exp[x]-> t` will not change `Exp[-x]` to `1/t` but will change `Exp[x]` to `t`

`s^2+c^2-> 1` will not change `s^2+2*c^2`, but will change `c^2+d^2+s^2` to `1+d^2`

While not all such behaviors will be surprising to all users, it does give one pause. We are cognizant of the possibility that a user might really truly know what is going to match and is exactly willing to work with that. In fact we do so ourselves to show how better tools can be built, even on top of Mathematica.

That is, we can broaden the scope of matches to eliminate the surprises above: by using multiple patterns or other modifications to the syntactic tools in Mathematica.

8.2 Non-constant patterns

There are also issues with Mathematica's default which includes pattern variables. Programming around these issues is more challenging, and our first program does not currently massage rules that have pattern variables.

Here are some examples WITH pattern variables:

`x_/y` will not match `1/y` but will match `2/y`.

`x_+ I y_` will not match `3+4 I` but will match `3+b I`.

`Complex[x_,4]` will not match `a+4` but will match `3+4 I` or `Complex[a,4]`

`x^n_` will not match `x` but will match `1/x`. but see next two patterns.

`x^(n_:1)` will match `x` with `n=1`. The parens are required.

`x_^n_` will match `x_` But with `n=1` the pattern simplifies to `x_`, which matches anything.

This is almost always going to lead to unintentional matches.

`a^2+x /. s_^n_ -> f[s,n]` will return `f[a^2+x,1]`, not `f[a,2]+f[x,1]`, because `1+x` is an instance of `s_`.

A more probably-intended result will appear in this example:

`1+x+x^2 /. x^n_ -> f[n]` will return `1+f[1]+f[2]`.

Note that the symbol `x` (not `x_`) acts as an anchor in the pattern to increase the prospect that the human user and the computer system agree on the intended targets.

While an explicit anchor helps, here's a pattern that works by excluding unintended matches of `s_^n_` which includes `s_^1` which is really just `s_`. But `s_` matches anything, including the whole expression.

`1 + x + y + z^2 /. s_?(MemberQ[{a, b, c, x, z}, #] &)^n_ -> f[s, n]`

This keeps the system from searching for matches unintended by the user.

`x_Integer/y_Integer` will never match anything. but ..

`Rational[x_,y_]` will match any rational constant.

8.3 Recognizing solvable matches

(This is addressed by Harris' Semantica package, dating back to 1995.) My speculation and sample programs addressing this matter are in some cases subsumed in his work.

`f[Semantic[n_*(n-1)]]:= g[n]`

provides a solution to `f[12]` of `MultipleSolutions[g[4],g[-3]]` which seems to be a thoughtful way out of the fact that `n` is not unique. Since most of Mathematica other than Semantica will know nothing about how to deal with this `MultipleSolutions` construction, there are option settings for returning only one solution.

By providing a wrapper of `Semantic[]`, the new kinds of patterns can be intermixed with the more syntactic version.

Unfortunately, the tools available within Semantica cannot handle the quadratic form recognition. One failure is that conditions like `FreeQ[a,b,c,x]` cannot be imposed, and another is that (at least in my testing in version 7 Mathematica), the equations that need to be solved are apparently too complex for `Solve`. Thus the direction of improvement to a solution may be another “bite of the apple” and an attempt to figure out common pattern schemes – such as picking out coefficients in a sum of products.

9 Making the substitution

Finding a pattern sometimes only gives a hint as to what to do with evaluation. For example, how much work should be done in some of these cases to find occurrences of patterns?

Example.

```
s                /. s^2->t   could be Sqrt[t]   or just s.
s+2*c           /. s+c->1   could be 1+c or 2-s or no change.
s^2+2*s*c+c^2  /. s+c->1   could be 1 or no change.
```

Macsyma’s `ratsubst` command provides an alternative to simple syntactic substitution. Essentially consider “substitute A for B in C ” as equivalent to “By long division, compute R and Q such that $C = Q * B^n + R$. The result of the substitution is $Q * A^n + R$ ”. Choice of n and main variable in division makes it possible to achieve different results.

10 Checking and optimizing

There are claims in the documentation that Mathematica can transform sets of patterns into a dispatch table or hash table, at least when the rules are associated with a global symbol (a “function name” informally). This is indicated as an optimization feature. There is presumably some advantage to be gained by exploring this pre-processing time for checking the validity, consistency, and coverage of a rule or rule set. (This general topic has been explored in the CAS literature.) For example, it is undesirable to have a pattern that flails around *unnecessarily*. A typical situation might be a pattern including `x___+y_____`. This is not ‘wrong’ but for an expression of length n would have 2^n possible matches; was this intentional?⁷

11 Other methods of applying transformations

Some rules, especially those that are expressed as polynomial side relations, can be applied more thoroughly through algebraic methods, using Mathematica’s `Reduce` or `Eliminate` programs. Should this kind of transformation be provided through a kind of rule interface? Are there in fact good uniform methods for substitution? (the “With” facility is not suitable, and “Conjugate” merely replaces I by $-I$ with a special command. If you used J as $\sqrt{-1}$ as is common in some electrical engineering contexts, you might have less trouble!)

⁷It is possible the user intended to set up a combinatorial search. Perhaps a warning should be issued when such a search is implied, if there is an appropriate time, as when explicitly constructing a dispatch table, to do so.

12 Writing programs to transform programs

The programs we wrote can be extended to handle `RuleDelayed` at the cost of some readability, either in carrying around which version of rule definition is used, or in bulkiness: writing out some nearly-identical code segments. We found it impossible to merge these near-identical segments because Mathematica's semantics of `Hold` and `Release` are occasionally unsuitable for dealing with partially evaluating expressions to produce patterns. Perhaps `HoldPattern` and `Verbatim`, which we learned about only after beginning our programming, will make more compact programs possible. We found `Hold` to be useless.

A simple transformation pattern example:

The user is looking for

```
rule1= f[n_*(n-1)] -> g[n]
```

which does not match `f[12]`. This next rule does:

```
rule1a=Module[{n}, f[x_] /; (n = 1/2*(1 + Sqrt[1 + 4 x])); True) -> g[n]]
```

matching with `g[4]`;

```
rule1b=Module[{n}, f[x_] /; (n = 1/2*(1 - Sqrt[1 + 4 x])); True) -> g[n]]
```

matches the other root, returning `g[-3]`, and collecting these results is exactly what `Semantica` does.

A sample transformation example follows:

```
BetterRules[(ruletype : Rule | RuleDelayed)[
  base_?FreeOfPat^sup_?BlankPatternQ, rhs_] := {
  ruletype[base^sup?Positive, rhs],
  ruletype[base^sup?Negative, (1/rhs /. sup[[1]]->-sup[[1]])],
  ruletype[base, (rhs /. sup[[1]] -> 1)]}
```

This takes a pattern like `s^n_ ->f[n]` and converts it into three:

```
{s^(n_)?Positive -> f[n], s^(n_)?Negative -> f[-n]^(-1), s -> f[1]}
```

which may or may not be what you intend, but this is how it looks. `Semantica` seems to go into an infinite loop on `f[Semantic[s_ ^n_]]:=h[s,n]`

The `BetterRules` program could, without change, support `Replace` and `ReplaceRepeated`, and perhaps other variants of the pattern replacement model.

6. The first section of program we provide does not include the above transformation, or any transformation on rules whose LHS contains patterns. We can, perhaps on a case-by-case basis, add features. Consider a rule like this (modified from an example supplied by David Park): `x_ ^n_ :>ToString[x]<>ToString[n]` which converts `s^2+t^3` to `s2+t3`. Given `|1/s^2|`, do we want to see `1/s2` or `s-2`? We could easily transform the rule to provide `1/s2`. (as Parks suggests). Given `s+t` should we transform it to `s1 + t1`?

Note that the simple `n_` is `Pattern[n,Blank[]]` in `FullForm`, and can be easily detected/matched.

To illustrate limits to this case-by-base approach, consider if the rule pattern were `x_ ^n_?Positive :>`. In this case the expression `1/s^2` *should not be changed at all*. The pattern associated with the form `n_?predicate` looks like `PatternTest[Pattern[n,Blank[]],predicate]` in “`FullForm`”, and it is clear that full analysis for `BetterRules` is not generally possible for arbitrary predicates associated with `PatternTest`.

13 A narrative version of patterns

It is clear that a program to execute a pattern match follows a certain recipe to achieve a match (or a failure to match). We propose a kind of compile-time natural-language version of the pattern-plus-program process be considered as a pattern-design and debugging aid. It would work like this:

A pattern $p_1 = f[a]$ would be narrated as (something like) `Pattern p1 matches only a single occurrence of the expression e which, after evaluation consists of exactly f[a]`.

A pattern $p_2 = f[a_]$ would be narrated as (something like) Pattern p_2 matches only a single occurrence of an expression e_1 which, after evaluation consists of $f[e_2]$, where the named variable a is bound to the expression e_2 .

A pattern $p_3 = a_ x^2$ would be narrated as (something like) Pattern p_3 matches only a single occurrence of an expression e_1 which, after evaluation consists of a product of two or more terms $e_2 x^2$, where the named variable a is bound to the term or the product of terms of the expression e_2 .

A pattern $p_4 = a_ . x^2$ would be narrated as (something like)

Pattern p_4 matches only a single occurrence of an expression e_1 which, after evaluation consists of one of two possibilities:

- A product of two or more terms $e_2 x^2$, where the named variable a is bound to the term or the product of terms of the expression indicated by e_2 .
- One term namely x^2 , where the named variable a is bound to the (identity under multiplication), 1.

A pattern $p_5 = a_ . x_^2$ would be narrated as (something like)

Pattern p_4 matches only a single occurrence of an expression e_1 which, after evaluation consists of one of two possibilities:

- A product of two or more terms $e_2 e_3$, where e_3 matches **Power** e_4 2 and where the named variable x is bound to e_4 . The named variable a is bound to the term or the product of terms indicated by the expression e_2 .
- One term where e_3 matches **Power** e_4 2 and where the named variable x is bound to e_4 . The named variable a is bound to the term 1.

A pattern $p_6 = a_ +b_$ would be narrated as (something like)

Pattern p_6 matches an occurrence of an expression e_1 which, after evaluation consists of a sum of n items where $n > 1$. There are $2^n - 2$ possibilities which will potentially be tried in some order, and the first which passes all additional conditions will set the bindings for the pattern variables a and b . In each of the possibilities, a consists of a single term or a sum of k terms, for $0 < k < n$ and b consists of the remaining $n - k$ terms, possibly only one term.

A pattern $p_7 = a_ ?p$ would be narrated as (something like)

Pattern p_7 matches an occurrence of an expression e_1 which, after evaluation consists of an expression e_2 such that the predicate $p[e_2]$ returns **True**.

This, and most other constructs can be extended “recursively” to any pattern.

A pattern $p_8 = q / ; p[a, b, \dots]$ where q is a pattern would be narrated as (something like)

Pattern p_8 matches an occurrence of an expression e_1 which, after evaluation consists of an expression e_2 such that the pattern q matches e_2 and predicate $p[a, b, \dots]$ returns **True**. The arguments to p may include named pattern variables or other data available at the time the match is computed.

These examples only scratch the surface of possible narratives. We have not provided examples for **BlankSequence** or **BlankNullSequence** matches or interactions with expressions whose **Heads** are declared to have various attributes in common with **Plus** and **Times**, specifically **Flat** and **Orderless**. The narratives are inherently descriptions of the pattern matching process; it is clear that they can be produced. What is less clear is that they will continue to be usefully readable in the face of much additional complexity.

14 Conclusions

Pattern matching can be a useful tool for guiding computations in a computer algebra system. It can be used to mimic ordinary programming, and to that extent is subject to approximately the same kinds of difficulties in design and debugging as ordinary programming. More elaborate and intensive use of pattern matching requires careful attention to detail, and a clear understanding of the semantics of the matching (and for Mathematica, a clear understanding of the elaborate syntax as well). Sometimes the complexity of the matching methods and (accidentally) poorly-composed patterns interact in unexpected ways to produce

unwanted matches or miss matches that were expected by the programmer. Even skilled and experienced programmers are sometimes surprised by pattern matching results.

There is no easy way to check that syntactically and semantically “valid” patterns happen to coincide with the semantic intentions of the pattern composer. Patterns can be composed which are ambiguous and work only on a subset of the intended domain, and/or have unwarranted actions elsewhere, or are potentially expensive by triggering some combinatorial search.

One route to more expressive patterns, sometimes, is to transform the requested match into an algebraic or analytic calculation, as is done by Semantica. Another is to recognize patterns with typical motives (like picking out coefficients in an expression), and translate them into programs accessing features like `CoefficientList`. This latter view was implemented in Macsyma over 40 years ago [?]. Thus the Macsyma pattern for quadratic essentially compiles into the fast program we offered as the “right” solution here.

15 Thanks

For the further discussion of matters raised in this paper, see the `comp.soft-sys.math.mathematica` newsgroup, January, 2010, and again in late December, 2010 The relevant early threads include `{I->-I}` in the subject line; the later ones have “pattern” in the subject line. The particular issues include whether the observed behavior of Mathematica is a bug or a feature; how should persons actually achieve the desired behavior, is it advisable to to change the behavior, or provide an alternative (as here), should such system programs as might be written to provide such behavior be automatically installed and more. For example what are the the role(s) of system builders and users in determining future directions for software? I thank each of the contributors, even the ones who strenuously hold views at odds with mine. This paper was reviewed and \TeX errors fixed in December, 2011.

16 Draft of a partial solution: a Mathematica program

(*Here are some Mathematica programs. (c) 2010 Richard Fateman *)

```
(
Clear[BetterRules,BRr,BRi,BRc,BRreal,BRnopat,BRpat,BRnopat1];
BetterRules[x_List]:=Flatten[ BetterRules /@ x]; (*For a List of Rules, process each. *)

FreeOfPat= FreeQ[#,_Pattern]&;
ContainsPat = Count[#,_Pattern,Infinity]>0&;
NotOneQ = #!=1& (* not equal to integer 1 *)

BetterRules[lhs_Rational->rhs_]:= BRr[lhs,rhs];
BetterRules[lhs_Integer ->rhs_]:= BRi[lhs,rhs];
BetterRules[lhs_Complex ->rhs_]:= BRc[lhs,rhs];
BetterRules[lhs_Real ->rhs_]:= BRreal[lhs,rhs]; (* no ideas here *)
BetterRules[lhs_?FreeOfPat->rhs_]:= BRnopat[lhs,rhs];
BetterRules[lhs_?ContainsPat -> rhs_]:=BRpat[lhs,rhs];
(* handle delicately or not at all*)

(* Fill in some of the details *)
(* 1/3 -> rhs should change n/3 to n*rhs. Even 5/3 -> 5*rhs*)
BRr[Rational[1,m_],rhs_] := (Rational[k_,m]->k*rhs);
BRr[Rational[n_?NotOneQ,m_],rhs_] := (n/m->rhs); (*no change if n!=1*)

BRpat[lhs_,rhs_] := (lhs->rhs); (*no change*)
```

```

(*possibly*
  BRpat[Power[Pattern[x_,Blank[]

(* 3*I -> zz could change to a+3*I-> a+zz. [Or, maybe a+I -> a +zz/3 as here] *)
BRc[Complex[0,m_],rhs_] := (Complex[k_,j_]->k+j/m*rhs);

(* What to do with Real? Just the same old thing?*)
BRreal[m_,rhs_] := (m->rhs);

(* for integer 2->x also do 1/2->1/x ; for 2->0 do 1/2->Infinity..*)

BRi[m_,rhs_] := {m->rhs,BRr[1/m,Quiet[1/rhs,Power::infy]]};

(* x^3 -> s also forces x^(-3)-> 1/s *)
BRnopat1[Power[x_,n_],rhs_] := { x^n->rhs, x^(-n)->Quiet[1/rhs,Power::infy]} ;

Findvars[expr_] :=
(* make a list of the symbols in an expression. *)
Block[{varlist = {}},
  Map[If[AtomQ[#] && Not[NumericQ[#]], AppendTo[varlist, #];] &,
    expr, -1];
  DeleteDuplicates[varlist]];

(* There is a decision point based on whether something with no
pattern variables involves exactly one symbolic variable, in which
case we solve for it and choose first solution. *)

BRnopat[lhs_,rhs_] :=
  Block [{v=Findvars[lhs]},
    Switch[Length[v],
      0, (* Constant Expression. what to do?*) lhs->rhs,
      1, If [Head[lhs]==Power, BRnopat1[lhs,rhs], (* simple case *)
          Block[{RHSx,newrule},
            newrule=((Solve[lhs==RHSx,v[[1]]][[1]])/. RHSx->rhs);
            Message[BetterRules::solve, lhs->rhs, newrule]; newrule]],
      _, (* more than one unknown, can't invert in some obvious way *)
        lhs->rhs
    ]];

BetterRules::solve = "Converting rule '1' into rule '2'"

(* I suppose all the conversions could be incorporated into Messages *)
)
(* end of programs *)

(* EXAMPLES *)

3+0.1 I /. I->0
3+0.1 I /. BetterRules[I->0]

```

```
3+12 I /. 3 I -> zz
3+12 I /. BetterRules[3 I -> zz]
```

```
3+4 I /. I -> -I
3+4 I /. BetterRules[I -> -I]
```

```
x^2+1/x^2 /. x^2-> s
x^2+1/x^2 /. BetterRules[x^2-> s]
```

```
5/3 /. 1/3 -> Third
5/3 /. BetterRules[1/3 -> Third]
```

```
Exp[x]+Exp[-x] /. Exp[x]->s
Exp[x]+Exp[-x] /. BetterRules[Exp[x]->s]
```

```
1+Cos[x]^2 /. Sin[x]->s
1+Cos[x]^2 /. BetterRules[Sin[x]->s]
```

```
e^2 + f^2 /. {1/e^2 -> q, f^2 -> r}
e^2 + f^2 /. BetterRules[{1/e^2 -> q, f^2 -> r}]
```

Can we use a better syntax?

Mathematica has a package that allows for syntax change. One can essentially insert a statement (via palette selection in the `Notation` package) of the form

```
Notation[x__ =/. y__ <==> ReplaceAll[x__,BetterRules[y__]]]
```

after which time,

```
3+4 I /. BetterRules[I -> -I]
```

can be written more simply, as

```
3+4 I =/. I -> -I
```