Rule-based Simplification of Expressions

Jürgen Billing and Stefan Wehmeier
MuPAD Research Group, Universitity of Paderborn
mailto:bij@mupad.de, mailto:stefanw@mupad.de

We present a new, rule-based method for the simplification of expressions. This will be the basis for the `simplify` *function in future versions of MuPAD.*

The goal

MuPAD provides many functions that rewrite expressions in an equivalent form, such as `expand`, `rewrite`, `combine`, `normal`, `simplify`, `radsimp`, and many more. While this helps the user to obtain many different expressions equivalent to his input, it still forces the user to try many of these functions manually until some output seems acceptable to him. Some automated search would be nice.

Simplification functions have to find a compromise between strength and running time. It is a pity if *MuPAD* does not find out that a given input is equivalent to zero; but it is also a pity if a call to `simplify` does not return for two days. The user should be able to tell *MuPAD* how much time he wants to spend waiting for a result.

Many users find it hard to foresee what functions like `combine` or `simplify` will do to the input. A good simplification function must provide complete control, letting the user select what rules to apply.

Sometimes users may want to have a list of equivalent expressions, in order to be able to choose on their own. Sometimes users may want a mathematical proof that the expressions are equivalent.

Summing up, some intelligent, highly configurable search is needed.

Some theory

We may consider different–looking expressions equivalent if we find a proof that they are equivalent. An equivalence proof consists of finitely many steps; each step is the application of a rule (axiom, theorem, ...) to the result of the previous step. Two expressions are equivalent if one is the input to the first step and the other is the output of the last step of an equivalence proof.

Finding an equivalence proof is difficult (in fact, undecidable). We might apply all rules we know to the input to obtain all expressions the equivalence of which can be proved in just one step; then apply all rules to all of these to obtain all expressions that have an equivalence proof of length two; and so on. In fact, this is how to prove that the problem is recursively enumerable. If we have a definition of complexity, the same argument works in order to show that we are able to find the simplest expression that can be proved to be equivalent to the input in at most n steps, with n fixed.

We may visualize the problem as a graph where the vertices represent expressions, and an edge between two vertices indicates that one expression may be derived from the other in one step.

All of these statements are true in a well-known context, the theory of formal languages and grammars. Formal grammars are a standard concept of computer science: they consist of an initial word and a set of replacement rules.

Sequences of symbols that can be obtained, starting from the initial word, by successive application of replacement rules, are called words of the language defined by the grammar. Our problem can then be rephrased as searching for the shortest word in a formal language defined by a grammar.

Some ideas for heuristics

So far, we have learnt that we are searching a vertex in an infinite graph, but no idea how to do it. This is no surprise since we have talked about general grammars, using no knowledge about arithmetical expressions.

Let us start with symmetry: we have not used it, should we try to use it somehow? In principle, the converse of every rule will hold: we will rewrite $\sin(x)^2 + \cos(x)^2$ as 1, and we would be allowed to do it the other way around as well. Intuitively, we feel that this is a bad idea: we should not use all rules that are mathematically valid, and the set of rules we actually use will not be closed under exchanging left and right hand side. This means that we are walking in a directed graph.

Now suppose that we can assign a “valuation” to every expression which we want to minimize. How can we search?

One idea would be to go from a vertex to its “best” neighbour, and continue this until some vertex is reached that is a local minimum. However, good results in mathematics are often achieved through complicated intermediate results.

Should we then select a neighbour vertex at random, with the probabilty weighted by valuation, and allow increase in valuation during the early steps, in a simulated–annealing manner? But since we are in a directed graph, we can never correct a false decision.

Just the contrary idea is to follow our theoretical proof from above (width-first search), stop the enumeration after a while, and return the best result found so far. The drawback to this approach is (of course) that only shallow depths can be reached within a reasonable amount of time.

We have chosen to do a best-first search instead, as we will describe below.

Local vs. global simplification

We have not yet discussed another point that is important both in general grammars as well as in our special case of mathematical expressions. Grammar rules (or mathematical rules) say that a given substring (or subexpression, respectively) may be replaced by another one.

If the left/right hand sides of the rules are small compared to the word size, much of the previous word remains. If some other substitution step applying to another part of the word was possible before, it might still be possible now.

On the other hand, we might try to find a rule that only replaces new parts of the word. In this way, we may hope to be able to manipulate different parts of the word in a nearly context-free way.

A good algorithm should try to do this, and collect information how every subword can be locally transformed. We do this, storing all derivations we have made in a large table.

The algorithm

Let an expression A be given. The main part of our algorithm can be described in a very simple way:

- create a toDo-list of simplification steps for the simplification of A
- while no stopping criterion is reached
  - do the next simplification - step for A
- return the simplest expression found so far

Now this tells almost nothing, we have to explain what a simplification step is, how the *next* step is determined, what our stopping criteria are, and what the simplicity of an expression is.

A simplification step produces an equivalent expression, and/or it produces new tasks that have to be done. Possible steps are:

`findRules`. Given an expression, find out which rules can be applied. This is the only item in every newly created toDo-list. It returns nothing, and creates one `applyRule` task for every rule found. It also initializes to-do lists for every operand, and creates one `recurseOnOperand` task for every operand of the expression.

`applyRule`. Given an expression and a rule, apply the rule to that expression. This returns an equivalent expression, and creates a `findRules` task for that new expression.

`recurseOnOperand` do a local simplification step on a given operand; replace the operand by the equivalent expression found. We explain this on page 38.

We will explain below on page 39 how rules are organized and what applying a rule exactly means.

The tasks are sorted according to the expected complexity of the output they are going to produce. On page 39, we will explain how the priority of tasks is computed. Stopping criteria are explained on page 40. We will describe on page 40 how the complexity of an expression is measured.

The recursion step

Suppose we want to simplify an expression with many operands (for simplicity, let us consider a sum) $a_1+\ldots+a_n$. It is clear that we may apply rules with n-element sums on their left hand side. On the other hand, we may substitute some a_i by an equivalent one; or apply some rule for sums with two summands to $a_i + a_j$, for some i, j.

Some optimization must be done here: since simplified versions of the same subexpression are needed many times, we have to store them. Now suppose that for each a_i, some equivalent expressions $a_{i,j}$, $1 \leq j \leq m_i$, are already available. One might suspect that trying all $m_1 \cdots m_n$ possible substitutions was too much effort. Experiments show that this is not the case; therefore we form all combinations involving a particular a_{ij} and a's obtained earlier as soon as we compute $a_{i,j}$.

Technically, a recursion step for simplifying one a_i looks as follows:

- Do one simplification step for a_i
- Combine the expression obtained thereby with all a_k, $k \neq i$, and expressions equivalent to them, in all possible ways.

- Store the results as expressions equivalent to the original sum of a_i's.
- For each newly obtained result, put the task to simplify it on the to-do list. Put the task to simplify a_i another time on the to-do list.

A recursion step for a partial sum $a_{i_1} + \ldots + a_{i_k}$ looks similarly.

The first step means that our simplifier must be able to call itself in that way that only one step is done, but the to-do list is not lost afterwards. However, *MuPAD* has no static variables; we discuss this problem in a separate article.

Rules

A rule is a prescription how to rewrite an expression into another expression and consists of three parts:

1. the left hand side: a pattern
2. the goal or right hand side: an expression to substitute
3. some conditions for the pattern variables (not necessary)

If the pattern matches a given expression, then we say that the rule is applicable to that expression.

An example for a rule is: `Rule(sin(X)^2 + cos(X)^2, 1)`

The pattern (part 1) is the expression `Rule(sin(X)^2 + cos(X)^2` where `X` matches every *MuPAD* expression (because no conditions for `X` are given), the second part `1` is the goal of this rule. In this case no conditions for the pattern variable `X` are necessary.

A more complex rule is:
```
Rule(X^p, hold(I^p * op(X, 2)^p),
     {type(X) = DOM_COMPLEX and op(X, 1) = 0 and op(X, 2) > 0})
```

The rule rewrites powers when the base is an imaginary number with positive imaginary part. To prevent evaluating the right hand side of a rule, one should use `hold`.

It's also possible to use a special function to simplify an expression:

```
Rule(X, hold(rewrite(X, ln)), {hold(has(X, exp))})
```

A rule base is a collection of rules, e.g., a list. For large collections of rules, it is useful to split the list into several smaller ones, one per pattern type. The selection of rules is done by the function `Simplify::selectRules` which is called with an expression and returns all rules that could be applicable the given expression.

This select function can be replaced by a user defined function to consider user defined rule bases (see p. 41).

The priority of tasks

We use the convention that small numbers stand for high priorities. The priority of a step equals the expected complexity of its output. In principle, one should multiply this by the exponential of some constant times the expected running time. Since it is difficult to foresee both complexity of the output and running time, we have to enter some arbitrary estimates anyway. In detail, we assign the following priorities:

- To find applicable rules for a given expression has the priority given by the complexity of the expression, times a constant factor smaller than one which represents the self-estimate of the simplifier (the average factor by which an expression gets better per simplification step).
- The priority of applying a given rule to a given expression equals the complexity of the expression times a factor specified inside the rule; by default, the ratio between the complexity of the right hand side and the complexity of the left hand side is used.
- The priority of doing a simplification step for a subexpression is determined based on a guess how much the given subexpression will be simplified in that step, and how much the valuation of the whole expression will change thereby. Here, the expected valuation of the result of simplifying the subexpression can be read off from the priority of the first element of the to-do list for that subexpression. It is assumed that the decrease of the complexity of the subexpression will cause the same amount of decrease of complexity of the whole expression.

Stopping criteria

It may happen that no tasks are left at some point, such that all equivalent expressions have been found. Mathematically, it cannot happen that there are only finitely many expressions equivalent to a given one; but a sensible rule base will have no rule e.g. with left hand side 0, such that trying to simplify 0 will immediately return.

Except for this case, several stopping criteria can be imagined. We may do a fixed number of simplification steps; or stop if the next step has a too bad priority; or stop if a given goal has been reached.

In our current implementation, we use 150 simplification steps. The user may customize this value using the option `MaxSteps = n`, where `n` is a positive integer. The user may define a particular expression *as goal* using the option `Goal = expression`. The simplification stops until the goal is reached and the number of steps is shown. We did not implement a priority-based stopping criterion yet.

Complexity

One important point is to find a measure to compare expressions and find out the simplest expression.

Unfortunately, there exists no generally useful solution, although most users have a certain opinion what simplicity means. Therefore, we allow the user to set his valuation (measure of complexity) by an option.

We define a default valuation recursively. Atomic *MuPAD* objects get a constant valuation only depending on their type, e.g., rational numbers are more complicated than integers, and complex numbers are even more complicated. An expression gets the sum of the valuation of its operands plus a value for the operation as valuation. Thus operations gets different values for their complexity, e.g., `+` and `*` are simple operations, and `cot` and `arcsin` are more complex.

Let us look at some examples:

Expression	**valuation**
`0`	1
`-1523`	1
`1/4`	2
`x`	2

Expression	**valuation**
`x + 1`	5
`x + y`	6
`2 + I*7.1`	10
`x^2`	10

Expression	**valuation**
`PI^(1/2)*(2*x)^(1/2)`	29
`sin(x)`	32
`sin(2*x)`	36
`int(sin(2*x)^2, x)`	146

With the recursive definition it cannot happen, that parts of expressions get a greater valuation than the expression itself.

Configurability

One of our goals is to give the user maximum control over the simplification. Our implementation has many optional arguments to achieve this:

- `selectRules`: the user may provide a function that returns rules possibly matched by a given expression
- `Valuation`: the user may provide a function that assigns a complexity to each expression
- `MaxSteps`: the user may limit the number of steps and thereby the running time
- `Goal`: the user may choose to stop when a given goal expression is found

The user may also specify the kind of output he wants; currently, there are three kinds of output.

- By default, one expression equivalent to the input is returned.
- If the option `OutputType = ''Proof''` is given, the return value is a proof of the equivalence between the input and the expression that it would return by default.
- With option `All`, a list of all equivalent expressions and their valuations is returned.

Examples

The new mechanism maintains most results of the standard `simplify` function. Differences are due to another normal form and notion of simplicity. We do not give a list of expressions that can be simplified; rather, we want to show some ways to influence simplification.

Strength vs. speed

Our current version happens to find out that $\cos(x)\tan(x) = \sin(x)$ in the 12th simplification step (expanding $\tan$ makes an expression more complicated, therefore it is not chosen as the very first option). 11 steps do not suffice:

MuPAD

```
>> Simplify(cos(x)*tan(x), MaxSteps = 11)
```

Output

$$\cos(x)\tan(x)$$

but 19 steps do:

MuPAD

```
>> Simplify(cos(x)*tan(x), MaxSteps = 12)
```

Output

$$\sin(x)$$

Hence the default (150 steps) would suffice but take more time than necessary.

Controlling the valuation

To many people, expanding means to minimize the number of brackets in the output. Hence, let us define

```
NumberOfBrackets:=
  proc(ex)
  begin
    nops(stringlib::contains(expr2text(ex), "(", IndexList))
  end_proc
```

Suppose we want to use the default valuation, but impose an additional penalty of 2 for each bracket in the output:

```
valuation:= Simplify::valuation + 2*NumberOfBrackets:
```

To look at the difference, it is best to choose the output type `All`:

```
MuPAD
>> Simplify(x*(y + z), All)
   Output
                        [[x (y + z), 13], [x y + x z, 14]]
```

such that by default, the expression is not expanded because the input is simpler. With the penalty, the order is reversed:

```
MuPAD
>> Simplify(x*(y + z), Valuation = valuation, All)
   Output
                        [[x y + x z, 14], [x (y + z), 15]]
```

It seems a matter of personal taste indeed which result is better.

Using the new simplify to write special simplification modules

One could wish to simplify powers “unsophisticated” as described in school books, however, none of the many rewriting procedures of *MuPAD* can do it.

Here the new simplification project used with an own rule base leads to the desired “simplification modul”:

```
PowerRules:=
  [Rule(A^m*A^n, hold(A^(m + n))),    Rule(A^m/A^n, hold(A^(m - n))),
   Rule(A^n*B^n, hold((A*B)^n)),      Rule(A^n/B^n, hold((A/B)^n)),
   Rule(A^n/B^n, hold((B/A)^-n)),     Rule(A^m^n,   hold(A^n^m))]
```

We have defined a small rule base to simplify powers as learned in school. We need also a selection procedure; it simply returns the whole rulebase, independently of expression.

```
powerRules:= () -> PowerRules:
```

Now we can use the new `Simplify` with our own rulebase to simplify powers as expected:

MuPAD

```
>> a := T^(1/2)*(R*T)^(-1/2)
```

Output

$$\frac{\sqrt{T}}{\sqrt{RT}}$$

```
>> Simplify(a, SelectRules = powerRules)
```

Output

$$\frac{1}{\sqrt{R}}$$

Equivalence proofs

For many reasons, it may be interesting to get the result together with a proof that it is correct.

In the example before, one obtains

MuPAD

```
>> Simplify(a, SelectRules = powerRules, OutputType = Proof")
```

Output

```
Input was T^(1/2)/(R*T)^(1/2).
Applying the rule A^n/B^n  ->  (A/B)^n
  to T^(1/2)/(R*T)^(1/2) gives (1/R)^(1/2)
Applying the rule (A^m)^n  ->  (A^n)^m
  to (1/R)^(1/2) gives 1/R^(1/2)
END OF PROOF
```

Future perspectives

The new simplification package is still under development; because of its modular structure, many parts of it can be fine-tuned independently of each other, and this fine-tuning has still to be done. We did not talk about running times in this article; in fact, analysing the connection between the different input parameters and the resulting running time has still to be done, too. We are going to present the final version in a future version of *MuPAD*.