

An Introduction to Term Rewriting Systems

Jeff Clark

November 5, 2003

All animals are created equal, but some are more equal than others.

George Orwell, *Animal Farm*

- If $1/2 = 2/4$, why do we prefer to work with $1/2$?
- If $x + x = 2x$, why do we prefer to work with $2x$?
- If $(x^3)' = 3x^2$, why do we always replace $(x^3)'$ by $3x^2$ when we can?

The reason is that $1/2$ is not identical to $2/4$; we use different symbols in the expressions. Instead, they are *equivalent*: we always use $1/2$ and $2/4$ to refer to the same number.

Similarly, $x + x$ is not identical to $2x$, but is equivalent to it, and $(x^3)'$ is not identical to $3x^2$, but is equivalent to it.

Two things are equivalent if we always use them the same way.

Equivalence should satisfy the following properties:

- Every x should be equivalent to itself.
- If x is equivalent to y , then y should be equivalent to x .
- If x is equivalent to y and y is equivalent to z , then x should be equivalent to z .

Simplification is different.

- No x is simpler than itself.
- If x is simpler than y , then y shouldn't be simpler than x .
- We still want transitivity: if x is simpler than y , and y is simpler than z , then x should be simpler than z .

It would literally take forever to list all of the pairs of things that are equivalent.

It would literally take forever to list all of the ways that things can be simplified.

To be practical: we need to find finite ways of describing equivalences and finite ways of describing simplifications.

Before we can do that, we still need to specify what things we are talking about.

We will focus on algebraic formulas. In order to do that, we need to work with variables (and an unlimited supply of them at that) as well as operators on those variables. We will only need a finite number of operators at any given time, such as addition, multiplication, negation, and so forth.

Let $V = \{v_1, v_2, \dots\}$ be an infinite supply of variables. Let $O = \{f_1, f_2, \dots, f_n\}$ be a finite list of operators.

Each operator will have a degree associated with it that refers to how many inputs it takes. Multiplication has degree two since we multiply two numbers; negation has degree one since we negate one number at a time. Constants like 0 and 1 are operators with degree zero since they aren't variables, but they don't take any inputs.

In C++, the *test ? true-value : false-value* operator has degree three. There are not many commonly used operators with degree more than two.

We will define the *term algebra* $T(O, V)$ recursively: $T(O, V)$ is the smallest set such that

- Every variable v in V is a term in $T(O, V)$.
- If f is an operator in O of degree n and t_1, t_2, \dots, t_n are terms in $T(O, V)$, then $f(t_1, t_2, \dots, t_n)$ is also a term in $T(O, V)$.

$T(O, V)$ consists of every possible combination of variables and operators that makes sense, paying attention to the degrees of the operators.

In our definition every operator is written in prefix notation.

Example. Let $O = \{a, s, m\}$ where a , s , and m stand for addition, subtraction, and multiplication, respectively. Each operator has degree two.

Then v_1 and v_2 are terms, $a(v_1, v_2)$ is a term (meaning $v_1 + v_2$), v_3 is a term, $m(a(v_1, v_2), v_3)$ is a term (meaning $(v_1 + v_2) \cdot v_3$), and so on. $a(v_1, v_2, v_3)$ is not a term since a has degree two.

Notice that a , s , m have interpretations as addition, subtraction, and multiplication, but as far as the terms are concerned they could mean something else. The term algebra only cares about getting the syntax right, i.e., making sure that each operator has enough inputs.

We will want our variables to stand for anything, including other terms.

A *substitution* σ is a function that sends variables in V to terms in $T(O, V)$ that only moves a finite number of variables: $\sigma(v) = v$ for all but a finite number of variables.

Example. Let $\sigma(v_1) = a(v_1, v_2)$ and $\sigma(v_2) = m(v_3, a(v_1, v_2))$ in the previous example, with $\sigma(v) = v$ for all other variables v .

We can extend any substitution to be a function defined on terms in $T(O, V)$ recursively: given any term t in $T(O, V)$

- If t is a variable v , the $\sigma(t) = \sigma(v)$.
- If t is $f(t_1, \dots, t_n)$ for some operator f of degree n , then

$$\sigma(f(t_1, \dots, t_n)) = f(\sigma(t_1), \dots, \sigma(t_n))$$

Example.

$$\begin{aligned}\sigma(m(a(v_1, v_2), v_3)) &= m(\sigma(a(v_1, v_2)), \sigma(v_3)) \\ &= m(a(\sigma(v_1), \sigma(v_2)), v_3) \\ &= m(a(a(v_1, v_2), \\ &\quad m(v_3, a(v_1, v_2))), v_3)\end{aligned}$$

Once we have substitutions we are ready to discuss how to define an equivalence in a finite amount of time.

Suppose that we would like to enforce associativity for addition, i.e.

$$(a + b) + c = a + (b + c)$$

in *all* sums. We can now do it with the single equation

$$a(a(v_1, v_2), v_3) = a(v_1, a(v_2, v_3))$$

with the interpretation that for any substitution σ ,

$$\begin{aligned} & a(a(\sigma(v_1), \sigma(v_2)), \sigma(v_3)) \\ & \equiv a(\sigma(v_1), a(\sigma(v_2), \sigma(v_3))) \end{aligned}$$

If we let z be the constant zero (of degree zero), then we have the following equations defining zero as the additive identity.

$$a(z, v_1) = v_1$$

$$a(v_1, z) = v_1$$

Similarly, if we let n be the negation operator (of degree one), then we have the following additive inverse.

$$a(n(v_1), v_1) = z$$

$$a(v_1, n(v_1)) = z$$

Do we need to define identities and inverses twice?

It is an old question in mathematics: if we start just with the equations

$$a(a(v_1, v_2), v_3) = a(v_1, a(v_2, v_3))$$

$$a(z, v_1) = v_1$$

$$a(n(v_1), v_1) = z$$

does it necessarily follow that $a(v_1, z) = v_1$?
Or that $a(v_1, n(v_1)) = z$?

We have simplifications going on in these equations.

v_1 is simpler than $a(z, v_1)$ since it is a part of the latter term.

We can choose to have z be simpler than any other term; since it has degree zero it will never be affected by substitutions. Then z is simpler than $a(n(v_1), v_1)$.

We can choose to view simplification lexicographically in $a(a(v_1, v_2), v_3) = a(v_1, a(v_2, v_3))$: both sides start with the addition operator a . We start comparing inputs: the first input on the left side is $a(v_1, v_2)$ and the first input on the second side is v_1 , which is definitely simpler.

If we view these equations as simplifications, we should write them differently:

$$a(a(v_1, v_2), v_3) \rightarrow a(v_1, a(v_2, v_3))$$

$$a(z, v_1) \rightarrow v_1$$

$$a(n(v_1), v_1) \rightarrow z$$

We now call them *rewrite rules*.

Whenever we encounter a term (or even part of a term) that is of the form of the left-side of a rewrite rule, we can replace it by the corresponding right side.

For example:

$$\begin{aligned} a(z, a(n(v_1), v_1)) &\rightarrow a(n(v_1), v_1) \\ &\rightarrow z \end{aligned}$$

where we first use the second rewrite rule about z being an additive identity, and then we use the third rewrite rule about n being an additive inverse.

Given a finite set of equations that we want to always hold true, we would like to construct a finite set of rewrite rules that define our equivalence in the following way: if we want to check if two terms x and y are equivalent, we would simplify x as far as possible, simplify y as far as possible, and then check if the results are the same. If so, $x \equiv y$, and if not they are not equivalent.

There are two potential problems:

- Could the simplification go on forever? We want our system of rewrite rules to be *terminating*.
- Even if the system is terminating, does it make a difference what order we apply our rules in? We want our system of rewrite rules to be *confluent*, i.e., to always yield the same simplest form for an expression no matter what order of simplifications we use.

We won't deal with the terminating issue here; the trick is to define simplification in a way that forces termination.

Confluence is already a problem. Suppose we try to apply our system of three rules (associativity, left identity, and left inverse) to the term $a(a(z, v_1), v_2)$.

$$\begin{aligned} a(a(n(v_1), v_1), v_2) &\rightarrow a(z, v_2) \\ &\rightarrow v_2 \end{aligned}$$

or

$$a(a(n(v_1), v_1), v_2) \rightarrow a(n(v_1), a(v_1, v_2))$$

We have two simplest forms for $a(a(z, v_1), v_2)$ that differ.

Do we give up?

No: we have a new equation:

$$v_2 = a(n(v_1), a(v_1, v_2))$$

The left hand side is definitely simpler (since v_2 appears in $a(n(v_1), a(v_1, v_2))$), so we also can create a fourth rewrite rule.

$$a(n(v_1), a(v_1, v_2)) \rightarrow v_2$$

which translates into $-v_1 + (v_1 + v_2) \rightarrow v_2$.

We have found a term that has two different simplifications, and used it to create a new rewrite rule. We don't have to look through all the terms to find such things: we can take the left-hand sides of any pair of rewrite rules and combine them to create such a term. When we finish simplifying that term, if the other rewrite rules simplify the left and the right to the same thing, we are done. If not, we have a new equation and a new rewrite rule.

This process need not terminate; but if it does, we end up with a finite rewrite system that can be used to define our equivalence.

Input: equations, ordering for which terms are simpler

Output (if terminates): list of rewriting rules that generate the same equivalence as the list of equations.

Start with an empty set of rewrite rules.

While there are any equations left in the list:

- Pick an equation from the input list.
- Simplify it as far as possible with the existing rules.
- If the sides of the equation are identical, return to the start of the loop.

- If not, see if one side is simpler than the other; if not, exit—failure with this definition of simpler.
- If one side is simpler, create a new rule.
- Find all “overlaps” with existing rules and itself; use to create new equations to add to the list.
- Add new rule to set of rewrite rules.
- Return to start of loop.

Inputs:

- $a(z, v_1) = v_1$
- $a(n(v_1), v_1) = z$
- $a(a(v_1, v_2), v_3) = a(v_1, a(v_2, v_3))$

Each equation becomes a rule.

- Rule #1: $a(z, v_1) \rightarrow v_1$
- Rule #2: $a(n(v_1), v_1) \rightarrow z$
- Rule #3: $a(a(v_1, v_2), v_3) \rightarrow a(v_1, a(v_2, v_3))$

Overlap Rule #2 with Rule #3:

$$\begin{aligned} a(a(n(v_1), v_1), v_2) &\rightarrow a(n(v_1), a(v_1, v_2)) \\ a(a(n(v_1), v_2), v_2) &\rightarrow a(z, v_2) \\ &\rightarrow v_2 \end{aligned}$$

This yields a new equation and from that a new rule.

$$\text{Rule \#4: } a(n(v_1), a(v_1, v_2)) \rightarrow v_2.$$

Overlap Rule #1 with Rule #4 to get:

$$\text{Rule \#5: } a(n(z), v_1) \rightarrow v_1.$$

Overlap Rule #2 with Rule #4 to get:

$$\text{Rule \#6: } a(n(n(v_1)), z) \rightarrow v_1.$$

Overlap Rule #6 with Rule #3 to get:

Rule #7: $a(n(n(v_1)), v_2) \rightarrow a(v_1, v_2)$.

Apply Rule #7 to Rule #6 to get:

Rule #8: $a(v_1, z) \rightarrow v_1$.

We have now proven that given associativity, a left identity, and a left inverse, that the left identity is also a right identity.

Continuing (at length), eventually there is a complete set of ten rewrite rules, including $a(v_1, n(v_1)) \rightarrow z$, and negation is also a right inverse.

Applications:

- Rewrite rules are at the heart of computer algebra systems such as *Mathematica* and *Maple*, where they are used both for rules of differentiation but also for algebraic simplifications.
- This process is also used in solving systems of polynomial equations using Gröbner basis theory.
- The appropriate Rewrite System can also be used for automated theorem-proving; a theorem claiming that an equation is always true can be proven by simplifying both sides until they are identical.

I have been working off-and-on for a while on implementing the process for converting a set of equations into a set of rewrite rules.

The simplification that I have been using is called the Knuth-Bendix ordering.

For speed of development, I have been using the Python programming language.

```

knuthBendix = KnuthBendix(log = 'example1.log',
                          maxEquations=1000)
knuthBendix.declareVariables('a', 'b', 'c')
knuthBendix.declareOperator(name = 'E', degree = 0, weight = 1)
knuthBendix.declareOperator(name = 'M', degree = 2, weight = 0)
knuthBendix.declareOperator(name = 'I', degree = 1, weight = 0)

equationString = """
M M a b c = M a M b c
  M E a = a
  M I a a = E
"""

knuthBendix.setEquations(equationString)

output = open('example1.out', 'w')

knuthBendix.computeRewriteRules()

if knuthBendix.hasCompleted():
    print >>output, knuthBendix.getRewriteRulesAsString()
else:
    print >>output, 'Knuth Bendix procedure has not completed'

```

```

def __init__(self, log=None, maxEquations=1000):
    """Constructor

    log:  string containing name of log file to write to

    maxEquations:  maximum number of equations to create

    """
    self.__equations = Equations(maxEquations=maxEquations)
    self.__log = Log(log)
    self.__reductionResult = None
    self.__rewriteRules = RewriteRules(self.__log)
    self.__symbolTable = SymbolTable()
    return

def computeRewriteRules(self, maxRewriteRules=1000):
    """Mutator:  computes rewrite rules from equations

    loop bounded by number of rewrite rules
    created using maxRewriteRules

    raises IncomparableTerms if comes
    across two wff's that can't be ordered

    maxRewriteRules:  positive integer

    """
    self.__log.beginning(self._getEquations())
    while self._getEquations().shouldContinue():
        self._processEquation()
    self.__log.ending()
    return

```

```

def _processEquation(self):
    """Mutator: pulls out next Equation, reduces it, result is
    tautology to be discarded or new RewriteRule

    raises IncomparableTerms if Equation, upon reduction,
    can not be ordered
    """
    self._chooseNextEquation()
    self._reduceNextEquation()
    if self._nextEquationIsTautology():
        self.__log.tautology(self._getCurrentEquation())
        return
    else:
        self._convertNextEquationToRule()
        self._reduceExistingRules()
        self._ensureEnoughVariables()
        self._createSyzygies()
        self._addNewRule()
        self._writeLog("Equations:\n%s\nRules:\n%s\n" \
            % (self._getEquations(), self._getRewriteRules()))
    return

```

Python is an object-oriented interpreted language. It has much in common with Perl, but prides itself on simplicity and clarity.

It's most obvious distinguishing feature is that there are no braces or semicolons; indentation and whitespace determine blocks.

It is immediately clear in looking at Python code what its structure is.

Python caches byte-code created from source code, and can produce Java byte-code if so desired.

Once I can tear myself away from writing more test code (350 test cases so far) I will create a GUI to facilitate use using Java Swing classes.