Lazy Math Instructor

Testing the equivalence of arbitrary terms

Ludwig Kolesch
ludwig.kolesch@student.uni-tuebingen.de
Eberhard Karls Universität
Tübingen, Baden-Württemberg, Germany

ABSTRACT

A mathematical term is an instruction on how a result can be calculated. It can involve numbers, variables, or other types of expressions such as mathematical functions. Under a broad definition of *expression*, the problem of determining whether two terms *mean the same thing*, that is, whether they always evaluate to the same result, is generally undecidable. This means that there cannot be a general algorithm that can determine whether two given arbitrary expressions mean the same thing. By taking a very narrow definition of *expression* instead, we will investigate how to determine the equivalence of terms under certain restrictions. In this seminar paper, a general algorithm is developed that can test the equivalence of mathematical terms with only a small set of allowed operations.

KEYWORDS

C#, terms, testing for equivalence, parsing, polynomials, object-oriented

1 INTRODUCTION

Given two arbitrary mathematical terms T_1 and T_2 it is not immediately clear how to determine whether or not they are equivalent, that is, whether $T_1(a, b, ..., z) = T_2(a, b, ..., z)$ for all possible values of $a, b, ..., z$. Allowing for mechanisms like recursion in the terms, this problem cannot be decided at all, similarly to how the equivalence of Turing machines is undecidable by Rice's theorem [4]. Even when restricting the problem to basic mathematical operations on integer constants and variables, there are still questions that arise: For example, a term might not be defined for all possible variable assignments. Dividing by zero, calculating 0^0, roots of negative numbers or logarithms with base 1 all share the property of being undefined. Being interested in the numbers that the terms evaluate to, such cases should be avoided. We are going to investigate this problem with restrictions that ensure that such cases cannot happen: Limiting ourselves to only the three basic operations of addition, subtraction and multiplication, all performed operations are well-defined in the real numbers at all times. Still, it is not clear how a general algorithm can test the equivalence of such terms. A naive approach might be to brute-force the problem by evaluating the terms under various variable assignments. However, with 26 possible variables and infinitely many possible integers, there is no obvious point at which said algorithm could terminate for equivalent input terms. Without further considerations, testing only a finite subset of the infinitely many possible variable assignments can never prove that the terms really are equivalent.
However, the restriction to the three forenamed operators brings another advantage: Any possible term T is a polynomial, and can be transformed into a normalized polynomial form P_T. Exploiting the structure of the terms and the properties of real-valued polynomials, we can write an efficient algorithm to test the equivalence.

This problem [1] is from the 24th ACM International Collegiate Programming Contest [3] in 2000. Originally, there are a few more simplifications like a limit of 80 characters per term and a restriction to single-digit integers. These are not enforced by my implementation, making it a little more general than the original task requires it to be.

2 EXACT SPECIFICATIONS AND POLYNOMIALS

Firstly, the problem needs exact boundaries under which the algorithm is going to operate. Terms may consist of non-negative integer constants, variables, brackets and the operators `+`, `-` and `*`. Brackets are allowed to be nested and the three operators are treated with equal precedence. This means that multiplication does not go before addition and subtraction. Instead, the term is evaluated strictly from left to right. For example, the final `*8` in the example term in Figure 1 does not apply to the `(b*b)` bracket only, but also to the entire term before it. This specification is going to simplify the algorithm we use later.

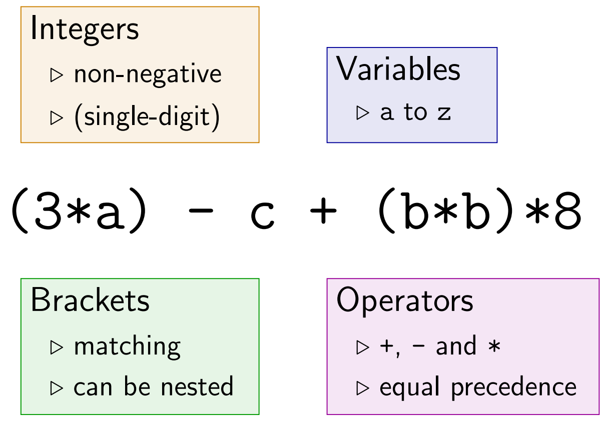

Figure 1: Example input term with components

Formally, the syntax of a term under these restrictions can be defined recursively as follows:

- A non-negative integer (0, 1, 2, ...) is a term.
- A variable (a, b, ..., z) is a term.
- If T is a term, then (T) is also a term.
- If T_1, T_2 are terms, then $T_1 + T_2$, $T_1 - T_2$ and $T_1 * T_2$ are also terms.

Integers and variables form the base case of the recursive definitions and are the *atoms* of a term.
An important observation under these restrictions is the following: Since brackets only change the order of operations but do not add new operations, everything that happens inside a term is the addition, subtraction and multiplication of atoms and more complex terms built of atoms. An integer is a constant polynomial of degree 0 and a variable is a linear polynomial of degree 1, so all atoms are polynomials. Since the real numbers form a ring with addition and multiplication, and polynomials over a ring form a ring themselves [2], they are closed under addition, subtraction and multiplication. Hence, any sums, differences and products inside the term remain polynomials and therefore the entire term is in fact a polynomial.
This is helpful because polynomials over the real numbers can be simplified by grouping matching terms using laws of the real numbers, and equivalent simplified polynomials are identical except for their ordering. Using an algorithm that can transform an arbitrary given term T into its simplified polynomial form P_T, we can easily solve our problem: Given the two input polynomials T_1 and T_2, we construct P_{T_1} and P_{T_2}. From $T_1 \equiv P_{T_1}$ and $T_2 \equiv P_{T_2}$, it follows that $T_1 \equiv T_2 \Leftrightarrow P_{T_1} \equiv P_{T_2}$. This equivalence of simplified polynomials can be tested for by verifying that P_{T_1} and P_{T_2} are identical, ignoring the ordering of sub-terms. Altogether, the equivalence of arbitrary terms can be tested this way, given that a term can be transformed into its simplified polynomial representation.

3 REPRESENTING POLYNOMIALS

A polynomial P is a finite sum of products, each of which is finite and contains an integer constant, the *coefficient* and any number of variables. If the same variable appears multiple times in the product, the multiplicity can be expressed in terms of a power, where the multiplicity forms the *exponent* of the variable. In constructing a polynomial, the smaller parts of a term interact with each other in addition, subtraction and multiplication operations. This led me to an object-oriented approach written in C# to represent normalized polynomials where sub-terms are represented as objects and interact with each other in mathematical operations. Normalization here includes simplifying the polynomial as far as possible, that is decreasing the number of summands and factors as far as possible, and ordering the sub-terms in a unique manner. In a normalized polynomial, the variable products of each summand are unique, because if they were not, they could be grouped together, simplifying the polynomial as a whole. For example, in $4a^2b + 2b^3 + 3a^2b$, the a^2b parts can be grouped together (using commutativity, associativity and distributivity), forming the simpler but equivalent $7a^2b + 2b^3$. Similarly, inside each variable product of a normalized polynomial, each variable is unique, because if they were not, they could be grouped together, again simplyfying the polynomial. For

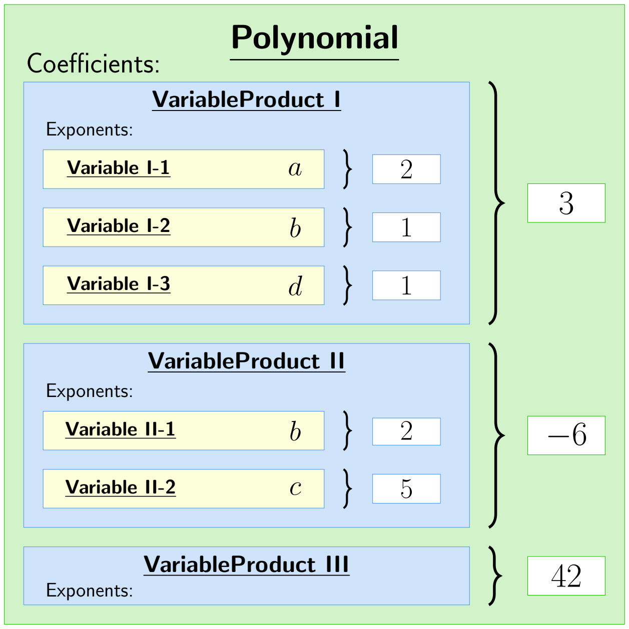

Figure 2: Object-oriented polynomial representation

example, in $7a^2ba^3$, the two a parts can be grouped together (using commutativity, associativity and the product rule for exponents), forming the simpler but equivalent $7a^5b$. These two uniqueness properties led me to a dictionary-based approach: A polynomial is represented as a dictionary called `Coefficients` which maps each variable product that appears in it to the corresponding coefficient. Each variable product in itself is also represented as a dictionary called `Exponents` which maps each variable inside the product to its exponent. An exemplary representation of the polynomial $3a^2bd - 6b^2c^5 + 42$ can be seen in Figure 2. In JSON notation, the `Coefficients` dictionary may be represented as follows:

$$\{\{a:2, b:1, d:1\}:3, \quad \{b:2, c:5\}:-6, \quad \{\}:42\}$$

The representation also takes care of establishing a unique ordering of the summands in the polynomial, as well as the factors in each variable product by making use of *sorted* dictionaries that are internally ordered by their keys. How exactly the dictionaries are ordered is irrelevant, as long as the ordering is unique. One possible way of doing this is to sort the variables inside a variable product alphabetically ($a < b < ... < z$) and sort the variable products in the sum by decreasing a-exponent, then by decreasing b-exponent, etc., and lastly by decreasing z-exponent. This leads to $3a^2bd - 6b^2c^5 + 42$ for the example polynomial and resembles the ordering I have chosen in my implementation.

This way, a unique ordering of the summands and of the factors is established. Since a term in this representation can never be simplified further, two polynomials are equivalent iff they are represented equally. The equivalence of polynomials in this representation can therefore be tested in linear time by only iterating once over all entries.

In my C# implementation, a variable is an enumeration type of the 26 possible values a, ..., z. A variable product is an object with the sorted `Exponents` dictionary and custom methods for equality checks and arithmetics. Likewise, a polynomial is an object with the sorted `Coefficients` dictionary and custom methods for equality checks and arithmetics. Altogether, this allows for readable code for arithmetics and equality checks since two polynomial objects can for example be added with the C# `+` operator, or tested for equivalence by comparing them with the default C# `.Equals(...)` method.

4 CALCULATING WITH POLYNOMIALS

Having a unique representation of polynomials, we are now in need of an algorithm that can parse terms into their equivalent normalized polynomial form. Since the chosen polynomial representation is always completely simplified, the desired form can be calculated by continuously simplifying the term until it reached a fixpoint where it does not change anymore. This is done by evaluating the term from left to right, always using the chosen polynomial representation and implementing addition, subtraction and multiplication for polynomials in this representation. Since atoms are trivially simplified, they can be transformed into the simplified polynomial representation as follows:

- An integer constant, for example 42, is the coefficient of an empty variable product, so the `Coefficients` dictionary looks as follows:

$$\{\{\} : 42\}$$

- A variable, for example `x`, is a variable product with exactly one entry that has a coefficient of 1 and an exponent of 1. The `Coefficients` dictionary looks as follows:

$$\{\{x : 1\} : 1\}$$

To perform operations on these simplified polynomials, we need general methods to add, subtract and multiply two of them. Adding two polynomials could be done by a simple union of their both `Coefficients` dictionaries, but that would not maintain the key uniqueness: Adding the two polynomials $2x$ and $4x$, we would get $2x + 4x$, or in our representation:

$$\{\{x : 1\} : 2\} + \{\{x : 1\} : 4\} = \{\{x : 1\} : 2,\ \ \{x : 1\} : 4\}$$

This is of course not a valid dictionary and defeats the point of maintaining *simplified* polynomials. The result should instead be $2x + 4x = (2 + 4)x = 6x$, using distributivity in the real numbers. Since x could be replaced by any variable product in this example, coefficients of matching variable products need to be added. Similarly for subtraction, coefficients of matching variable products need to be subtracted. If a variable product only occurs in one of the terms, the other term can be treated as having said product with a coefficient of 0:

$$(2x) + (3y) = (2x + 0y) + (0x + 3y) = (2 + 0)x + (0 + 3)y = (2x + 3y)$$

In the object representation, this case corresponds to an union of the `Coefficients` dictionaries of both terms:

$$\{\{x : 1\} : 2\} + \{\{y : 1\} : 3\} = \{\{x : 1\} : 2,\ \ \{y : 1\} : 3\}$$

All of this leads to following implementation in pseudocode for adding two polynomials, assuming that failing dictionary lookups return 0:

Adding polynomials

```
1  function p1 + p2: Poly Poly -> Poly
2      result <- Poly()
3
4      for varProd in (p1.Keys ∪ p2.Keys) do
5          coefficient <- p1[varProd] + p2[varProd]
6
7          if coefficient != 0 then
8              result[varProd] <- coefficient
9          end if
10     end for
11
12     return result
13 end function
```

Figure 3: Pseudocode for adding two polynomials

If the new coefficient is zero, it is not included in the resulting polynomial, since a product with 0 is just 0. The polynomial is therefore simpler when 0 is left out of the sum.

Subtracting two polynomials can be accomplished the same way by simply subtracting the coefficients instead of adding them in line 5. Implementing the multiplication is slightly more complex as multiplying two sums involves multiplying each term of the first sum with each term of the second sum. Doing so naively would again not result in simplified polynomials:

$$(2x) * (4x^2) = 2x * 4x^2 = (2 * 4)xx^2 = 8x^1x^2$$

The coefficients are simply multiplied, but inside the variable products, exponents of matching variables need to be added to get $8x^3$, the shortest possible form. This addition of exponents is a direct application of the product rule for exponents. Therefore, implementing the multiplication first requires a multiplication for variable products:

Multiplying variable products

```
function vp1 * vp2: VarProd VarProd -> VarProd
    result <- VarProd()

    for variable in (vp1.Keys ∪ vp2.Keys) do
        exponent <- vp1[variable] + vp2[variable]
        result[variable] <- exponent
    end for

    return result
end function
```

Figure 4: Pseudocode for multiplying two variable products

Since exponents of 0 do not appear in the product, all exponents are strictly positive and thus there is no need to add a zero-check for the resulting exponent. With the multiplication of variable products implemented, polynomials can be multiplied as follows:

Multiplying polynomials

```
1  function p1 * p2: Poly Poly -> Poly
2      result <- Poly()
3
4      for varProd1 in p1.Keys do
5          for varProd2 in p2.Keys do
6              resVarProd <- varProd1 * varProd2
7
8              result[resVarProd] += p1[varProd1] *
9                                    p2[varProd2]
10
11             if result[resVarProd] == 0 then
12                 result.Remove(resVarProd)
13             end if
14         end for
15     end for
16
17     return result
18 end function
```

Figure 5: Pseudocode for multiplying two polynomials

Note the multiplication of the variable products in line 6, and that in line 8 a failing dictionary lookup is again assumed to return 0.

5 PARSING TERMS TO POLYNOMIALS

Finally, we need to apply these operations in correct order to parse any term string into its normalized polynomial representation. In order to adhere to the left-to-right operation order, a recursive algorithm can go through the string from right to left, removing one sub-term at a time. The algorithm to parse a term works as follows:

(1) If the term is an atom, return its polynomial representation.
(2) Split off the rightmost subterm connected to the rest of the term with an operator. This subterm can either be an atom or any term inside brackets.
(3) Recursive call on both this rightmost term and the rest of the term to get polynomial representations for both.
(4) Add, subtract or multiply both polynomials depending on the operator they were connected with.

To demonstrate how this works, here is how the algorithm operates on the example term `a + b * c`. The textual input is represented in `blue` and parsed polynomial objects in *green*:

Parse(`"a + b * c"`)	(2) + (3)
Parse(`"a + b"`) * Parse(`"c"`)	(1)
Parse(`"a + b"`) * c	(2) + (3)
(Parse(`"a"`) + Parse(`"b"`)) * c	(1)
(Parse(`"a"`) + b) * c	(1)
(a + b) * c	(4)
$(a + b)$ * c	(4)
$ac + bc$	

Figure 6: Example of the parsing algorithm

This example demonstrates how by splitting off one piece from the right at all times, a left-deep structure inside the term is created through the recursive calls. This way, $a + b$ is evaluated first because a recursive Parse call is made on the rest term `"a + b"` after splitting off the rightmost `"c"`. Only after having calculated $a + b$, that entire term is multiplied with the split off c, giving the final result of $ac+bc$. This corresponds to the left to right evaluation order defined in the specification. In the chosen representation, that polynomial corresponds to

$$\{\{a : 1, b : 1\} : 1, \quad \{b : 1, c : 1\} : 1\}$$

and any other term is equivalent to the input term `"a + b * c"` iff it has the exact same representation.

6 ANALYSIS AND LIMITS

Given two input terms of length at most n, what runtime can be expected to test their equivalence? The parsing algorithm visits each atom once (recursive base case) and performs two operations for each operator: Splitting off the right term, and calculating the operation after the children calls have finished. Each pair of brackets is also visited exactly once, since the term in brackets is split off and the brackets themselves are discarded. Therefore, no character of the input string leads to more than two steps for the parsing algorithm, so it performs at most $O(2n) = O(n)$ steps. Splitting a sub-term off can be performed in constant time, and so can parsing an atom. Addition and subtraction of polynomials involve running a loop over all variable products in either of the polynomials. Since there cannot be more than n different variable products in a string with entire length n, that loop has at most $O(n)$ iterations. Similarly, the nested loop required to multiply two polynomials has a runtime of at most $O(n^2)$. These observations lead to a theoretical worst-case runtime of $O(n) \cdot O(n^2) = O(n^3)$ for parsing a term of length n. Doing so twice is still $O(n^3)$ and testing if the parsed polynomials are equivalent is $O(n)$, so the entire algorithm performs in $O(n^3)$ worst-case runtime. In practice, the algorithm will often perform faster, since not all operations can be multiplications of n different variable products if the entire term has just length n. The $O(n^3)$ is therefore just an upper bound for the worst-case runtime. Testing randomly generated terms with nested brackets and all operators revealed a runtime of about 60 seconds for parsing 50 terms with 400 characters each. For larger input terms, the time increased drastically, indicating the overall superlinear runtime of the algorithm. These tests were performed on a laptop with an Intel i7 processor of the 12th generation.

Since it is implemented recursively, the parsing algorithm is restricted by the stack size of C#: Terms with too many operators lead to a stack overflow in the recursive call tree. This happens at around 10,000 operations, depending on how the term is structured. Other than that, it can correctly test any terms for equivalence by setting the stack size in accordance to the size of the terms to test.

The specification of having no operator precedence is of course a simplification of the task. If multiplication is supposed to be performed before addition and subtraction, like it's done in conventional mathematical notation, the algorithm can be slightly adjusted by adding a pre-processing step to the terms: Each multiplication operator is visited and the two parts (atoms or terms in brackets) with the `*` symbol in between are surrounded by brackets. This converts the implicit precedence of the multiplication to an explicit one which the algorithm in its current form can work with.

7 ALTERNATIVE ALGORITHMS

A few alternatives to the presented algorithm might come to mind:

Instead of doing the parsing recursively, it is of course possible to write an iterative algorithm, since any recursive algorithm can also be expressed iteratively. Going through the term from left to right, parsing atoms to polynomials at all times, and immediately performing the arithmetic operations works just as long as there are no (nested) brackets involved. To get them working, a data structure to store parts that are already parsed and parts that are not is probably required.

Yet another idea could be an enhancement of the naive algorithm presented in the beginning: Knowing that all terms are polynomials and there is no *power* operator in the terms, there are properties that could be exploited. For example, a term that contains no variable other than `x`, and contains `x` exactly twice, is a polynomial in one variable of degree at most 2. Assuming this property applies to both input terms, the equivalence can be tested just by substituting values for `x`: Since the graphs of polynomials of degree 2 are uniquely determined by three points, the both terms are equivalent iff they evaluate to the same value for three different assignments of `x`. If the degree of the polynomial increases, the number of variable assignments to test, increases as well: Polynomials of degree d need $d + 1$ assignments. While this could still work, the approach will get harder to follow when multiple variables are involved.

8 CONCLUSION

The presented algorithm is more than satisfactory for the ACM contest problem: It can handle any non-negative integer constant instead of just single-digit ones, and it works in fractions of a second for inputs as short as 80 characters per term which is the limit set by the contest. For terms that are a lot larger, my implementation of the algorithm might not work without increasing the stack size due to its recursive nature, or it might take very long to terminate due to its superlinear runtime.

REFERENCES

[1] 2000. Lazy Math Instructor. ACM ICPC 2020. Asia Region, Tehran Site, Sharif University of Technology, Problem E.

[2] Laurenz Göllmann. 2023. *Lineare Algebra im algebraischen Kontext* (3. ed.). Springer Spektrum, 21–22. https://doi.org/10.1007/978-3-662-67174-0

[3] icpc.foundation. 2023. The ICPC International Collegiate Programming Contest. https://icpc.global/regionals/abouticpc. Accessed: 11.09.2023.

[4] Lutz Priese and Katrin Erk. 2018. *Theoretische Informatik* (4. ed.). Springer Vieweg, 303. https://doi.org/10.1007/978-3-662-57409-6