QEDb: a database to create computer checked proofs with a focus on applied sciences

Herman Bergwerf

June 24, 2017

Abstract

This paper outlines a database for storing simple algebraic proofs from various fields in applied sciences. The goal of this project is to create a tool that can be used to write down computer checked proofs. The data about these proofs is stored in a relational database. This allows a computer system to break down mathematical proofs into their fundamental steps. It is not the intention to find new proofs. This system is designed with little prior knowledge about computer aided proof checking, the design and implementation are from a programmer perspective. The database and its underlying rule based engine are implemented in the Dart programming language.[1] Implementation details are not discussed in this paper, instead it focuses on the conceptual design and some related questions that are not directly relevant for the implementation.

1 Overview

Writing down a proof starts with an initial mathematical expression. This expression is manipulated according to a sequence of steps that will produce the final expression. There are only two expression manipulation methods that are used in this system: rearrangement and substitution. A proof is a sequence of such manipulations and shows in a rigorous way that the final expression can be formed from the initial one. A proof can be re-used in other proofs as single substitution, this way increasingly complex proofs can be constructed.

To provide a user interface where proofs can be written down easily, a set of algorithms has been developed that can take the difference between two expressions and finds a set of manipulations that connect the two expressions. The user does not have to tell the system what substitutions are used in a given proof, instead it is possible to write down a series of mathematical expressions that relate to consecutive points in the proof and the system can do the rest of the work.

[1] The code can be found at `https://github.com/qedb`

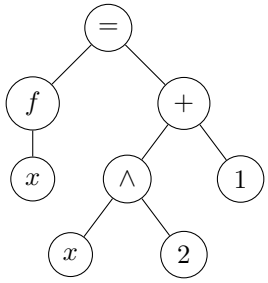


Figure 1: Tree representation for the expression $f(x) = x^2 + 1$

Although an important goal is a high level of internal consistency and mathematical correctness, this cannot be guaranteed. In a number of cases mathematical integrity is sacrificed for simplicity and usability. More concretely, the checking algorithm cannot validate all statements, instead is relies on the initial rules that are entered into the system. It is possible to enter incorrect rules and compromise the system. Aside from this, the system is also designed with loose constraints on variables. When a proof starts with an expression that is invalid in itself, such as $\sin \hat{e}_2$, there is no mechanism that will raise an error. More about this is explained in section 2.1. Finally, when the initial expression is incomplete, the order in which substitutions are applied can affect the result, more about this in section 2.2.3.

2 Expressions

Mathematical expressions are the cornerstone of this database. These expressions follow a very simple tree like structure that reminds of an Abstract Syntax Tree. Expressions are composed of functions and integers. A function is a node with an associated ID and a number of arguments. The argument count is fixed for each function ID, and is configured globally. From this a tree can be constructed like in figure 1. Symbols, such as π, can be expressed as functions with zero arguments.

Functions can also be generic, which means they can be replaced with an arbitrary expression later on. Generic functions play an important role in the formation of general rules. Variables such as position, or speed, can be defined as a generic function with zero arguments (a generic symbol). To avoid too much complexity, generics do not have any additional constraints. The variable x in $f(x) = x^2 + 1$ can be substituted with any expression, not just a scalar. This raises problems when the set of expressions a variable can be substituted with is limited. In this case x might need to be in the set of real numbers, and some proofs only work for integers. However these is no way to assign a class to a generic function.

2.1 Why not use classes for generic functions?

It would be possible to expand the previously described problem by introducing a classification system. For example, we could define x to be a 1 dimensional, real vector: $x \in \mathbb{R}^1$. Such a mechanism is avoided because determining the class of an expression, and whether is can be substituted, quickly becomes complicated. For example, to know that $1 + \vec{u} \cdot \vec{v}$ reduces to $\mathbb{R}^1$ would require the proof checker to have (vector) algebra knowledge built-in. Even though a minimal implementation might seem trivial, more rules need to be introduced quickly. For example $a + b \in \mathbb{R}$ for $a \in \mathbb{R} \wedge b \in \mathbb{R}$. In the end the primary reason to keep classes away from generic functions is to avoid building algebra knowledge into the core algorithms as much as possible. Basically any function (including a generic function) is meaningless outside the scope of rules.

2.2 Expression manipulation

Expression manipulation happens in two ways: rearrangement and substitution (either with rules or conditions). The manipulation can take place at any point in the expression tree.

2.2.1 Rearrangement

Rearrangement is the first type of manipulation that can be applied to expressions. It is a combination of the commutative[2] and associative[3] property of some binary operators (in particular addition and multiplication). A function can be defined to be rearrangeable, meaning that they have these two properties (i.e. the order of operations does not matter). Rearranging the children then means changing the expression tree (applying the operators in a different order), but keeping the same children as illustrated in figure 2. Instead of using fundamental rules to apply such manipulation, a custom algorithm can resolve rearrangements. This is done in order to avoid dealing with the vast number of substitutions that would result from such a common manipulation.

[2] $f(a, b) \to f(b, a)$ where f is the operator

[3] $f(f(a, b), c) \to f(a, f(b, c))$ and $f(f(a, b), f(c, d)) \to f(a, f(f(b, c), d))$

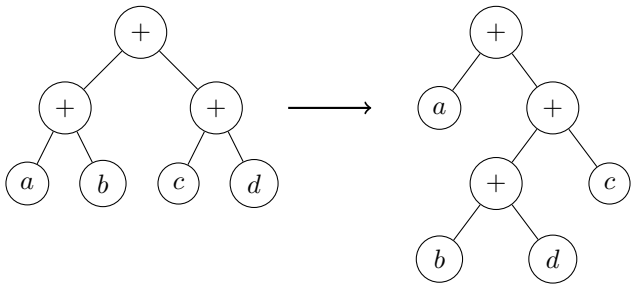


Figure 2: Rearrangement from $(a + b) + (c + d)$ into $a + ((b + d) + c)$

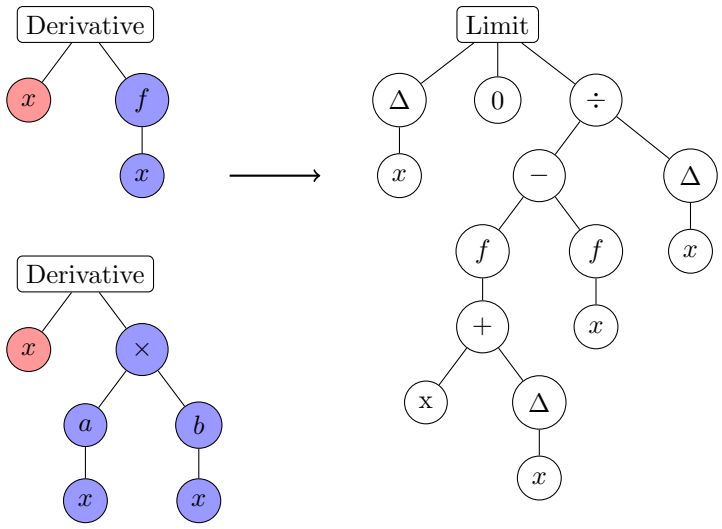
One possible sequence of five substitutions that would be required for the rearrangement shown in figure 2 is as follows (only the fundamental properties may be used). It quickly becomes clear why using plain substitutions for this is a burden.

1. $(a + b) + (c + d)$
2. $a + ((b + c) + d)$ which is a fundamental property
3. $a + ((c + b) + d)$ by using $x + y \to y + x$
4. $a + (c + (b + d))$ by using $(x + y) + z \to x + (y + z)$
5. $a + ((b + d) + c)$ by using $x + y \to y + x$

2.2.2 Substitution

Substitution is the second type of manipulation that can be applied to expressions. Substitutions are executed for a given equality (a *left* and *right* expression) and is applied at a certain point in the target expression tree (the substitution point). The left expression must match the expression at the substitution point, which will then be replaced with the right expression. Each generic function in the left expression will be mapped to the expression at the same point in the target expression[4]. Naturally, a specific generic function can only be mapped to one expression for the left expression to match. This mapping is then substituted in the right expression. The expression at the substitution position is replaced with the resulting expression. This process is illustrated in figure 3.

[4]This is sometimes called unification



Generic function	Mapped to
x	x
$f(x)$	$a(x)b(x)$

$$\frac{\partial}{\partial x} a(x)b(x) \tag{1}$$

$$\frac{\partial}{\partial x} f(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{2}$$

$$\lim_{\Delta x \to 0} \frac{a(x + \Delta x)b(x + \Delta x) - a(x)b(x)}{\Delta x} \tag{3}$$

Figure 3: Eq. 2 is substituted in Eq. 1, resulting in Eq. 3. The trees again show how the program sees the expression. The mapping of $f(x)$ is achieved by substituting $f(x) = a(x)b(x)$ into the right expression

2.2.3 Substitution order can matter

When function dependencies are not explicitly pointed out in expressions, different substitution sequences can lead to different results. In some cases this can even lead to a discrepancy, in particular when a generic function is substituted with an expression that again contains this generic function. To illustrate this, lets look at an example similar to the one in figure 3.

$$b = 2x \tag{4}$$

$$g(x) = a(x)b \tag{5}$$

$$\frac{\partial}{\partial x}g(x) = \lim_{\Delta x \to 0} \frac{g(x + \Delta x) - g(x)}{\Delta x} \tag{6}$$

$$\frac{\partial}{\partial x}a(x)b = \lim_{\Delta x \to 0} \frac{a(x + \Delta x)b - a(x)b}{\Delta x} \tag{7}$$

$$\frac{\partial}{\partial x}a(x)2x \neq \lim_{\Delta x \to 0} \frac{a(x + \Delta x)2x - a(x)2x}{\Delta x} \tag{8}$$

$$\frac{\partial}{\partial x}a(x)2x = \lim_{\Delta x \to 0} \frac{a(x + \Delta x)2(x + \Delta x) - a(x)2x}{\Delta x} \tag{9}$$

Substituting the definition of the derivative to function $g(x)$ produces Eq. 6. Now substitute the definition for $g(x)$ from Eq. 5 to produce Eq. 7. Note that the result is similar to first substituting $g(x) = a(x)b$ in the derivative and then applying the limit definition. If the definition of b is substituted to produce Eq. 8, there is a serious problem. Since b was defined without function argument, the delta expression is not substituted in $2x$. Meanwhile, if $g(x)$ *and* b is substituted *before* the limit definition, the produced equation (Eq. 9) is as expected. To fix this, it would be sufficient to define $b(x) = 2x$ instead of $b = 2x$. In this case that seems pretty trivial, however other cases may arise where the dependencies are not always obvious ahead of time, or are not explicitly written down. For example, in classical mechanics it is quite common to write $\vec{a}$ instead of $\vec{a}(t)$, even when the acceleration depends on the time variable. Obviously the initial rule set is the weak link here, and the database has no means of verifying the fundamental rules it is given.

It is possible to add some constraints to substitutions that restrict this, and at least prevent the case described here. To do this the substitution algorithm has to discard any substitution where a generic function is mapped to an expression that depends on any other variable but the first argument of the generic function. This means the generic function can have 1 argument at most (this issue does not apply to generic functions with no arguments). For this to work, all functions in the database have to be symbols or pure functions. This implies that all functions are either defined with no arguments, and can depend on any

other function in the database, or *only* depend on the arguments given. Such a constraint is enforced in QEDb.

3 Rules

A rule is an equation of a left and right expression, and can be used for substitutions. QEDb has a global set of rules that can be used in all contexts. Experiments with subject bounded rules have been done, however this approach did not reveal any significant advantages. Fundamental rules can be entered into the database, and from this more rules can be derived. The global rule-set has to be kept at a reasonable size to allow relatively quick difference resolving (more in section 4.1).

3.1 Semi-generic rules

Often, rules have to be semi-generic. For example Newtons second law, $F = ma$, describes a general rule for all kinds of different forces. However this rule cannot be substituted when a different symbol is used for F (e.g. F_g) because F, m and a are not generic. Making them generic would cause obvious problems. One way to solve this issue is to use a non-generic, single argument function for each context-dependent symbol. The argument of these function indicates the context. This way the rule can be created using a new generic symbol ctx: $F(ctx) \to m \cdot a(ctx)$. In this case a subscript might look more familiar: $F_{ctx} \to m \cdot a_{ctx}$. For the gravitational force a new non-generic symbol g can be defined and used with the semi-generic rule to get: $F_g \to m \cdot a_g$.

3.2 Condition

Creating a proof in QEDb is tedious work compared to writing out the proof on paper; every single assumption and algebraic operation has to be defined or proved before it can even be substituted. This means some very trivial steps may have to be worked out multiple times. The limitations of generic functions to aid this become very obvious when looking at the following simple case.

$$\frac{\partial g}{\partial x} = 0 \tag{10}$$

$$\begin{aligned} \frac{\partial (g \cdot f)}{\partial x} &= \frac{\partial g}{\partial x} \cdot f + g \cdot \frac{\partial f}{\partial x} && (11)\\ &= 0 \cdot f + g \cdot \frac{\partial f}{\partial x} && (12)\\ &= g \cdot \frac{\partial f}{\partial x} && (13) \end{aligned}$$

The fact that it is possible to move everything out of the derivative that is constant with respect to x is something that is often taken for granted. But since this database does not have any built-in mathematics knowledge, rules like this one have to be proven from first principles as well. There is one big inconvenience though, since Eq. 13 relies on Eq. 10, it has to be proven separately for every single case using the steps shown here (introducing Eq. 10 as a generic rule would obviously not be possible, then all possible derivatives would suddenly be equal to zero). To solve this inconvenience rules can have conditions. Using conditions this case can be expressed as:

$$\frac{\partial(g \cdot f)}{\partial x} = g \cdot \frac{\partial f}{\partial x} \quad \text{for} \quad \frac{\partial g}{\partial x} = 0 \tag{14}$$

When substituting the rule, another rule has to be referenced that satisfies the condition. There are some built-in conditions for special cases. For example the derivative of an integer is always zero, and the database must handle this automatically since it is not possible to express a rule for this (there is no way to limit a generic symbol to integers).

Conditions are limited to rule-like structures; it has to be possible to look them up in the rule-set. Conditions are added by substituting them at some point in the proof. This means that while creating a proof, any arbitrary equation that is not in the rule-set can be substituted as a condition. For now it seems that rules, and therefore conditions, have to be limited to equalities. Introduced other types of conditions (such as $\leq$ or $\in$) is appealing but also complicates the system making development more difficult.

3.3 Avoiding conflicts within the rule set

To prevent conflicting or ambiguous rules, it is necessary to check any new rules against all existing rules. When a new rule is submitted, the database will check if a rule already exist that directly proves the new rule, or if the new rule directly proves any existing rules. If this is the case the new rule is rejected.

4 Proofs

The final piece of the puzzle are proofs. Proofs have been mentioned before, but there are a few things left to say about them. Proofs can be started with any arbitrary expression. If the proof is the manipulation of one side of an equation, for example the derivation of centripetal acceleration by working out the second derivative of the position vector, then it is possible to start with a_c and end up at v^2/r by applying a range of manipulations. However when an equation has to be solved for one variable, for example when solving $2x+10=5$ for x, it is necessary to start with the entire equation as first expression. The equals operator is just another 2-argument function in the database. Just like before, fundamental rules can be added to rearrange this expression such as $a+b=c \rightarrow a=c-b$.

A few basic functions, including addition, subtraction and multiplication, will be automatically evaluated after each step. Other functions are left unevaluated and are, by convention, part of the algebraic solution. Evaluation is done when all arguments are integers, and should always produce an integer. So for example, applying the rule $a^b a^c \rightarrow a^{b+c}$ to $a^2 a^3$ will produce a^5 as resulting expression since addition belongs to the set of automatically evaluated functions.

4.1 Automatic difference resolving

QEDb can compare two expressions and find a tree of manipulations that satisfy their difference. This is achieved by recursively comparing both expressions, their arguments, etc., and matching them against the rule-set. Before looking for rules, an algorithm will determine whether the difference is due to a rearrangement. Rearrangements can be nested, so when comparing $a+(b+(c \cdot e))$ with $(e \cdot c)+b+a$, the algorithm is able to determine that two rearrangements took place. This automatic resolving helps to create an intuitive interface for building proofs, which is critical to adding content to the database.

5 Similar software

There are many projects that implement a computer aided proof checker, and some have used it to build some sort of database. When I started out building this database I was not aware of most projects that have been done in the past (in part because I did not know what exactly to look for). The design of QEDb is not directly inspired by other projects, although the concepts that are used are pretty generic.

5.1 ProofWiki

The ProofWiki is basically a very well organized MediaWiki containing articles for a wide range of proofs. The proofs presented in the ProofWiki go far beyond what QEDb can do. In fact QEDb is mostly limited to analytic and algebraic proofs.

5.2 MetaMath

MetaMath employs a similar proof checking system as QEDb to construct a huge database of theorems that prove fundamental mathematical properties starting with ideas from set theory.