It is possible to commit no mistakes and still lose.
That is not a weakness, that is life.

Jean-Luc Picard - *Star Trek: The Next Generation*

CHAPTER 5

Qualitative and Quantitative Evaluations

Contents

This chapter primarily contributes to the research task **V**, i.e., evaluating the effectiveness of the semantification and translation system LaCASt. In Section 5.1, we also extend LaCASt semantic LaTeX translations to support more mathematical operators, including sums, products, integrals, and limit notations. Hence, this chapter secondarily also contributes to research task **IV**, i.e.,

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-658-40473-4_5.

A. Greiner-Petter, *Making Presentation Math Computable*,
https://doi.org/10.1007/978-3-658-40473-4_5

implementing an extension of the semantification approach to provide translations to CAS. We evaluate LaCASt on two different datasets: the DLMF and Wikipedia.

First, we evaluate LaCASt on the DLMF to estimate the capabilities and limitations of our rule-based translator on a semantic enhanced dataset. Translating formulae from the DLMF to CAS can be considered simpler primarily for three reasons. First, the formulae are manually enhanced and can be considered unambiguous in most cases. Second, the constraints of formulae are directly attached to equations and therefore accessible to LaCASt. Lastly, parts of equations in the DLMF are linked to their definitions which allow to resolve substitutions and fetch additional constraints. This meta information is either not available or given in the surrounding context in Wikipedia articles which greatly harms the accessibility of this crucial data. Hence, we presume that we achieve the best possible translations via LaCASt on the DLMF. For evaluating the capabilities of LaCASt, we perform numeric and symbolic evaluation techniques to evaluate a translation [3, 13]. We will further use these evaluation approaches to identify flaws in the DLMF and CAS computations.

Next, we evaluate LaCASt on Wikipedia as the direct successor of the previous Chapter 4. Here, we use the full and final version of LaCASt, including every improvement that has been discussed throughout the thesis. Specifically, it actively uses all common knowledge pattern recognition techniques discussed in Section 4.2.6.1, all heuristics for detecting math operators introduced in Section 5.1.2, and the enhanced symbolic and numeric evaluation pipeline first outlined in [3] and finally elaborated in Section 5.1.3. In combination with the automatic evaluation, we are able to perform plausibility checks on complex mathematical formulae in Wikipedia.

This chapter is split in two parts following two main motivations behind them. In Section 5.1, we elaborate the possibility to use LaCASt translations to automatically verify entire DML and CAS with one another. We specifically focus on the DLMF for our DML and Mathematica and Maple for our general-purpose CAS. In Section 5.2, we use the final context-sensitive version of LaCASt introduced in Chapter 4, including every improvement introduced in the first Section 5.1 of this chapter, with the goal to verify equations in Wikipedia articles. This chapter finalizes the improvements of LaCASt for semantic LaTeX expressions (Section 5.1) and general LaTeX expressions (Section 5.2).

The content of Section 5.1 was published at the TACAS conference [8]. Some parts in Section 5.2 have also been previously published at the CICM conference [2]. Section 5.2, as the direct successor of Chapter 4, is part of the aforementioned submission to the TPAMI journal [11].

5.1 Evaluations on the Digital Library of Mathematical Functions

Digital Mathematical Library (DML) gather the knowledge and results from thousands of years of mathematical research. Even though pure and applied mathematics are precise disciplines, gathering their knowledge bases over many years results in issues which every digital library shares: consistency, completeness, and accuracy. Likewise, CAS[1] play a crucial role in the modern era for pure and applied mathematics, and those fields which rely on them. CAS can be used to simplify, manipulate, compute, and visualize mathematical expressions. Accordingly,

---

[1] In the sequel, the acronyms CAS and DML are used, depending on the context, interchangeably with their plurals.

modern research regularly uses DML and CAS together. Nonetheless, DML [2, 10] and CAS [20, 100, 180] are not exempt from having bugs or errors. Durán et al. [100] even raised the rather dramatic question: "*can we trust in [CAS]*?"

Existing comprehensive DML, such as the DLMF [98], are consistently updated and frequently corrected with errata[2]. Although each chapter of the DLMF and its print analog *The NIST Handbook of Mathematical Functions* [276] has been carefully written, edited, validated, and proofread over many years, errors still remain. Maintaining a DML, such as the DLMF, is a laborious process. Likewise, CAS are eminently complex systems, and in the case of commercial products, often similar to black boxes in which the magic (i.e., the computations) happens in opaque private code [100]. CAS, especially commercial products, are often exclusively tested internally during development.

An independent examination process can improve testing and increase trust in the systems and libraries. Hence, we want to elaborate on the following research question.

> **Research Question**
>
> How can digital mathematical libraries and computer algebra systems be utilized to improve and verify one another?

Our initial approach for answering this question is inspired by Cohl et al. [2]. In order to verify a translation tool from a specific LaTeX dialect to Maple , they perform symbolic and numeric evaluations on equations from the DLMF. This approach presumes that a proven equation in a DML must be also valid in a CAS. In turn, a disparity in between the DML and CAS would lead to an issue in the translation process. However, assuming a correct translation, a disparity would also indicate an issue either in the DML source or the CAS implementation. In turn, we can take advantage of the same approach proposed by Cohl et al. [2] to improve and even verify DML with CAS and vice versa. Unfortunately, previous efforts to translate mathematical expressions from various formats, such as LaTeX [3, 10], MathML [18], or OpenMath [152], to CAS syntax show that the translation will be the most critical part of this verification approach.

In this section, we elaborate on the feasibility and limitations of the translation approach from DML to CAS as a possible answer to our research question. We further focus on the DLMF as our DML and the two general-purpose CAS Maple and Mathematica for this first study. This relatively sharp limitation is necessary in order to analyze the capabilities of the underlying approach to verify commercial CAS and large DML. The DLMF uses semantic macros internally in order to disambiguate mathematical expressions [260, 403]. These macros help to mitigate the open issue of retrieving sufficient semantic information from a context to perform translations to formal languages [10, 18]. Further, the DLMF and general-purpose CAS have a relatively large overlap in coverage of special functions and orthogonal polynomials. Since many of those functions play a crucial role in a large variety of different research fields, we focus in this first study mainly on these functions.

In particular, we extend the first version of LaCASt [3] to increase the number of translatable functions in the DLMF significantly. Current extensions include a new handling of constraints, the support for the mathematical operators: sum, product, limit, and integral, as well as overcoming

---

[2] https://dlmf.nist.gov/errata/ [accessed 2021-05-01]

semantic hurdles associated with Lagrange (prime) notations commonly used for differentiation. Further, we extend its support to include Mathematica using the freely available WED[3] (hereafter, with Mathematica, we refer to the WED). These improvements allow us to cover a larger portion of the DLMF, increase the reliability of the translations via LACAST, and allow for comparisons between two major general-purpose CAS for the first time, namely Maple and Mathematica. Finally, we provide open access to all the results contained within this paper[4].

The section is structured as follows. Section 5.1.1 explains the data in the DLMF. Section 5.1.2 focus on the improvements of LACAST that had been made to make the translation as comprehensive and reliable as possible for the upcoming evaluation. Section 5.1.3 explains the symbolic and numeric evaluation pipeline. We will provide an in-depth discussion of that process in Section 5.1.3. Subsequently, we analyze the results in Section 5.1.4. Finally, we conclude the findings and provide an outlook for upcoming projects in Section 5.1.5.

Related Work Existing verification techniques for CAS often focus on specific subroutines or functions [45, 58, 107, 148, 180, 185, 225, 228], such as a specific theorems [218], differential equations [153], or the implementation of the `math.h` library [224]. Most common are verification approaches that rely on intermediate verification languages [45, 148, 153, 180, 185], such as *Boogie* [29, 225] or *Why3* [41, 185], which, in turn, rely on proof assistants and theorem provers, such as *Coq* [37, 45], *Isabelle* [153, 167], or *HOL Light* [146, 148, 180]. Kaliszyk and Wiedijk [180] proposed on entire new CAS which is built on top of the proof assistant HOL Light so that each simplification step can be proven by the underlying architecture. Lewis and Wester [228] manually compared the symbolic computations on polynomials and matrices with seven CAS. Aguirregabiria et al. [20] suggested to teach students the known traps and difficulties with evaluations in CAS instead to reduce the overreliance on computational solutions.

We [2] developed the aforementioned translation tool LACAST, which translates expressions from a semantically enhanced LACAST dialect to Maple. By evaluating the performance and accuracy of the translations, we were able to discover a sign-error in one the DLMF's equations [2]. While the evaluation was not intended to verify the DLMF, the translations by the rule-based translator LACAST provided sufficient robustness to identify issues in the underlying library. To the best of our knowledge, besides this related evaluation via LACAST, there are no existing libraries or tools that allow for automatic verification of DML.

5.1.1 The DLMF dataset

In the modern era, most mathematical texts (handbooks, journal publications, magazines, monographs, treatises, proceedings, etc.) are written using the document preparation system LaTeX. However, the focus of LaTeX is for precise control of the rendering mechanics rather than for a semantic description of its content. In contrast, CAS syntax is coercively unambiguous in order to interpret the input correctly. Hence, a transformation tool from DML to CAS must disambiguate mathematical expressions. While there is an ongoing effort towards such a process [14, 214, 329, 402, 408], there is no reliable tool available to disambiguate mathematics sufficiently to date.

---

[3] https://www.wolfram.com/engine/ [accessed 2021-05-01]
[4] https://lacast.wmflabs.org/ [accessed 2021-10-01]

The DLMF contains numerous relations between functions and many other properties. It is written in LaTeX but uses specific semantic macros when applicable [403]. These semantic macros represent a unique function or polynomial defined in the DLMF. Hence, the semantic LaTeX used in the DLMF is often unambiguous. For a successful evaluation via CAS, we also need to utilize all requirements of an equation, such as constraints, domains, or substitutions. The DLMF provides this additional data too and generally in a machine-readable form [403]. This data is accessible via the i-boxes (information boxes next to an equation marked with the icon ⓘ). If the information is not given in the attached i-box or the information is incorrect, the translation via LaCASt would fail. The i-boxes, however, do not contain information about branch cuts (see Section 5.1.4.1) or constraints. Constraints are accessible if they are directly attached to an equation. If they appear in the text (or even a title), LaCASt cannot utilize them. The test dataset, we are using, was generated from DLMF Version 1.0.28 (2020-09-15) and contained 9,977 formulae with 1,505 defined symbols, 50,590 used symbols, 2,691 constraints, and 2,443 warnings for non-semantic expressions, i.e., expressions without semantic macros [403]. Note that the DLMF does not provide access to the underlying LaTeX source. Therefore, we added the source of every equation to our result dataset.

5.1.2 Semantic LaTeX to CAS translation

The aforementioned translator LaCASt was first developed by Greiner-Petter et al. [3, 10]. They reported a coverage of 53.6% translations [3] for a manually selected part of the DLMF to the CAS Maple. Afterward, they extended LaCASt to perform symbolic and numeric evaluations on the entire DLMF and reported a coverage of 58.8% translations [2]. This version of LaCASt serves as a baseline for our improvements. They reported a success rate of $\sim 16\%$ for symbolic and $\sim 12\%$ for numeric verifications.

Evaluating the baseline on the entire DLMF result in a coverage of only 31.6%. Hence, we first want to increase the coverage of LaCASt on the DLMF. To achieve this goal, we first increasing the number of translatable semantic macros by manually defining more translation patterns for special functions and orthogonal polynomials. For Maple, we increased the number from 201 to 261. For Mathematica, we define 279 new translation patterns which enables LaCASt to perform translations to Mathematica. Even though the DLMF uses 675 distinguished semantic macros, we cover $\sim 70\%$ of all DLMF equations with our extended list of translation patterns (see Zipf's law for mathematical notations [14]). In addition, we implemented rules for translations that are applicable in the context of the DLMF, e.g., ignore ellipsis following floating-point values or `\choose` always refers to a binomial expression. Finally, we tackle the remaining issues outlined by Cohl et al. [2] which can be categorized into three groups: (i) expressions of which the arguments of operators are not clear, namely sums, products, integrals, and limits; (ii) expressions with prime symbols indicating differentiation; and (iii) expressions that contain ellipsis. While we solve some of the cases in Group (iii) by ignoring ellipsis following floating-point values, most of these cases remain unresolved.

In the following, we first introduce the constraint handling via blueprints[5]. Next, we elaborate our solutions for (i) in Section 5.1.2.2 and (ii) in Section 5.1.2.3.

---

[5] This subsection 5.1.2.1 was previously published by Cohl et al. [2].

5.1.2.1 Constraint Handling

Correct assumptions about variable domains are essential for CAS systems, and not surprisingly lead to significant improvements in the CAS ability to simplify. The DLMF provides constraint (variable domain) metadata for formulae, and we have extracted this formula metadata. We have incorporated these constraints as assumptions for the simplification process (see Section 5.1.3.1). Note however, that a direct translation of the constraint metadata is usually not sufficient for a CAS to be able to understand it. Furthermore, testing invalid values for numerical tests returns incorrect results (see Section 5.1.3.2).

For instance different symbols must be interpreted differently depending on the usage. One must be able to interpret correctly certain notations of this kind. For instance, one must be able to interpret the command `a,b\in A`, which indicates that both variables `a` and `b` are elements of the set `A` (or more generally `a_1,\dots,a_n\in A`). Similar conventions are often used for variables being elements of other sets such as the sets of rational, real or complex numbers, or for subsets of those sets.

Also, one must be able to interpret the constraints as variables in sets defined using an equals notation such as `n=0,1,2,\dots`, which indicates that the variable `n` is a integer greater than or equal to zero, or together `n,m=0,1,2,\dots`, both the variables `n` and `m` are elements of this set. Since mathematicians who write LaTeX are often casual about expressions such as these, one should know that `0,1,2,\dots` is the same as `0,1,\dots`. Consistently, one must also be able to correctly interpret infinite sets (represented as strings) such as `=1,2,\dots`, `=1,2,3,\dots`, `=-1,0,1,2,\dots`, `=0,2,4,\dots`, or even `=3,7,11,\dots`, or `=5,9,13,\dots`. One must be able to interpret finite sets such as `=1,2`, `=1,2,3`, or `=1,2,\dots,N`.

An entire language of translation of mathematical notation must be understood in order for CAS to be able to understand constraints. In mathematics, the syntax of constraints is often very compact and contains textual explanations. Translating constraints from LaTeX to CAS is a compact task because CAS only allow precise and strict syntax formats. For example, the typical constraint $0 < x < 1$ is invalid if directly translated to Maple, because it would need to be translated to two separate constraints, namely $x > 0$ and $x < 1$.

We have improved the handling and translation of variable constraints/assumptions for simplification and numerical evaluation. Adding assumptions about the constrained variables improves the effectiveness of Maple's `simplify` function. Our previous approach for constraint handling for numerical tests was to extract a pre-defined set of test values and to filter invalid values according to the constraints. Because of this strategy, there often was no longer any valid values remaining after the filtering. To overcome this issue, instead, we chose a single numerical value for a variable that appears in a pre-defined constraint. For example, if a test case contains the constraint $0 < x < 1$, we chose $x = \frac{1}{2}$.

A naive approach for this strategy, is to apply regular expressions to identify a match between a constraint and a rule. However, we believed that this approach does not scale well when it comes to more and more pre-defined rules and more complex constraints. Hence, we used the POM-tagger to create blueprints of the parse trees for pre-defined rules. For the example LaTeX constraint `$0 < x < 1$`, rendered as $0 < x < 1$, our textual rule is given by

```
0 < var < 1 ==> 1/2.
```

The parse tree for this blueprint constraint contains five tokens, where `var` is an alphanumerical token that is considered to be a placeholder for a variable.

We can also distinguish multiple variables by adding an index to the placeholder. For example, the rule we generated for the mathematical LaTeX constraint `$x,y \in \Real$`, where `\Real` is the semantic macro which represents the set of real numbers, and rendered as $x, y \in \mathbb{R}$, is given by

```
var1, var2 \in \Real ==> 3/2,3/2.
```

A constraint will match one of the blueprints if the number, the ordering, and the type of the tokens are equal. Allowed matching tokens for the variable placeholders are Latin or Greek letters and alphanumerical tokens.

5.1.2.2 Parse sums, products, integrals, and limits

Here we consider common notations for the sum, product, integral, and limit operators. For these operators, one may consider Mathematically Essential Operator Metadata (MEOM). For all these operators, the MEOM includes *argument(s)* and *bound variable(s)*. The operators act on the arguments, which are themselves functions of the bound variable(s). For sums and products, the bound variables are referred to as *indices*. The bound variables for integrals[6] are called *integration variables*. For limits, the bound variables are continuous variables (for limits of continuous functions) and indices (for limits of sequences). For integrals, MEOM include precise descriptions of regions of integration (e.g., piecewise continuous paths/intervals/regions). For limits, MEOM include limit points (e.g., points in $\mathbb{R}^n$ or $\mathbb{C}^n$ for $n \in \mathbb{N}$), as well as information related to whether the limit to the limit point is independent or dependent on the direction in which the limit is taken (e.g., one-sided limits).

For a translation of mathematical expressions involving the LaTeX commands `\sum`, `\int`, `\prod`, and `\lim`, we must extract the MEOM. This is achieved by (a) determining the argument of the operator and (b) parsing corresponding subscripts, superscripts, and arguments. For integrals, the MEOM may be complicated, but certainly contains the argument (function which will be integrated), bound (integration) variable(s) and details related to the region of integration. Bound variable extraction is usually straightforward since it is usually contained within a differential expression (infinitesimal, pushforward, differential 1-form, exterior derivative, measure, etc.), e.g., $\mathrm{d}x$. Argument extraction is less straightforward since even though differential expressions are often given at the end of the argument, sometimes the differential expression appears in the numerator of a fraction (e.g., $\int \frac{f(x)\mathrm{d}x}{g(x)}$). In which case, the argument is everything to the right of the `\int` (neglecting its subscripts and superscripts) up to and including the fraction involving the differential expression (which may be replaced with 1). In cases where the differential expression is fully to the right of the argument, then it is a *termination symbol*. Note that some scientists use an alternate notation for integrals where the differential expression appears immediately to the right of the integral, e.g., $\int \mathrm{d}x f(x)$. However, this notation does not appear in the DLMF. If such notations are encountered, we follow the same approach that we used for sums, products, and limits (see Section 5.1.2.2).

---

[6] The notion of integrals includes: antiderivatives (indefinite integrals), definite integrals, contour integrals, multiple (surface, volume, etc.) integrals, Riemannian volume integrals, Riemann integrals, Lebesgue integrals, Cauchy principal value integrals, etc.

Extraction of variables and corresponding MEOM The subscripts and superscripts of sums, products, limits, and integrals may be different for different notations and are therefore challenging to parse. For integrals, we extract the bound (integration) variable from the differential expression. For sums and products, the upper and lower bounds may appear in the subscript or superscript. Parsing subscripts is comparable with the problem of parsing constraints [2] (which are often not consistently formulated). We overcame this complexity by manually defining patterns of common constraints and refer to them as blueprints (see Section 5.1.2.1). This blueprint pattern approach allows LaCASt to identify the MEOM in the sub- and superscripts.

For our MEOM blueprints, we define three placeholders: `varN` for single identifiers or a list of identifiers (delimited by commas), `numL1`, and `numU1`, representing lower and upper bound expressions, respectively. In addition, for sums and products, we need to distinguish between including and excluding boundaries, e.g., $1 < k$ and $1 \leq k$. An excluding relation, such as $0<k<10$, must be interpreted as a sum from 1 to 9. Table 5.1 shows the final set of sum/product subscript blueprints.

Standard notations may not explicitly show infinity boundaries. Hence, we set the default boundaries to infinity. For limit expressions we need different blueprints to capture the limit direction. We cover the standard notations with ‘`var1 \to numL*`’, where `*` is either `+`, `-`, `^+`, `^-` or absent and the different arrow-notations where `\to` can be either `\downarrow`, `\uparrow`, `\searrow`, or `\nearrow`, specifying one-sided limits. Note that the arrow-notation (besides `\to`) is not used in the DLMF and thus, has no effect on the performance of LaCASt in our evaluation. Note further that, while the blueprint approach is very flexible, it cannot handle every possible scenario, such as the divisor sum $\sum_{(p-1)|2n} 1/p$ [98, (24.10.1)]. Proper translations of such complex cases may even require symbolic manipulation, which is currently beyond the capabilities of LaCASt.

Table 5.1: The table contains examples of the blueprints for subscripts of sums/products including an example expression that matches the blueprint.

Blueprints	**Example**
`numL1 \leq var1 < var2 \leq numU1`	$0 \leq n < k \leq 10$
`-\infty < varN < \infty`	$-\infty < n < \infty$
`numL1 < varN < numU1`	$0 < n, k < 10$
`numL1 \leq varN < numU1`	$0 \leq k < 10$
`numL1 < varN \leq numU1`	$0 < n, k \leq 10$
`varN \leq numU1`	$n, k \leq N + 5$
`varN \in numL1`	$n \in \{1, 2, 3\}$
`varN = numL1`	$n, k, l = 1$

Identification of operator arguments Once we have extracted the bound variable for sums, products, and limits, we need to determine the end of the argument. We analyzed all sums in the DLMF and developed a heuristic that covers all the formulae in the DLMF and potentially a large portion of general mathematics. Let x be the extracted bound variable. For sums, we consider a summand as a part of the argument if (I) it is the very first summand after the operation; or (II) x is an element of the current summand; or (III) x is an element of the following summand (subsequent to the current summand) and there is no termination symbol between the current summand and the summand which contains x with an equal or lower depth according to the parse tree (i.e., closer to the root). We consider a summand as a single logical construct since addition and subtraction are granted a lower operator precedence than multiplication in mathematical expressions. Similarly, parentheses are granted higher precedence and, thus, a sequence wrapped in parentheses is part of the argument if it obeys the rules (I-III). Summands, and such sequences, are always entirely part of sums, products, and limits or entirely not.

A termination symbol always marks the end of the argument list. Termination symbols are relation symbols, e.g., $=$, $\neq$, $\leq$, closing parentheses or brackets, e.g., $)$, $]$, or $>$, and other operators with MEOMs, if and only if, they define the same bound variable. If x is part of a subsequent operation, then the following operator is considered as part of the argument (as in (II)). However, a special condition for termination symbols is that it is only a termination symbol for the current chain of arguments. Consider a sum over a fraction of sums. In that case, we may reach a termination symbol within the fraction. However, the termination symbol would be deeper inside the parse tree as compared to the current list of arguments. Hence, we used the depth to determine if a termination symbol should be recognized or not. Consider an unusual notation with the binomial coefficient as an example

$$\sum_{k=0}^{n} \binom{n}{k} = \sum_{k=0}^{n} \frac{\prod_{m=1}^{n} m}{\prod_{m=1}^{k} m \prod_{m=1}^{n-k} m}. \tag{5.1}$$

This equation contains two termination symbols, marked red and green. The red termination symbol $=$ is obviously for the first sum on the left-hand side of the equation. The green termination symbol $\prod$ terminates the product to the left because both products run over the same bound variable m. In addition, none of the other $=$ signs are termination symbols for the sum on the right-hand side of the equation because they are deeper in the parse tree and thus do not terminate the sum.

Note that `varN` in the blueprints also matches multiple bound variable, e.g., $\sum_{m,k\in A}$. In such cases, x from above is a list of bound variables and a summand is part of the argument if one of the elements of x is within this summand. Due to the translation, the operation will be split into two preceding operations, i.e., $\sum_{m,k\in A}$ becomes $\sum_{m\in A}\sum_{k\in A}$. Figure 5.1 shows the extracted arguments for some example sums. The same rules apply for extraction of arguments for products and limits.

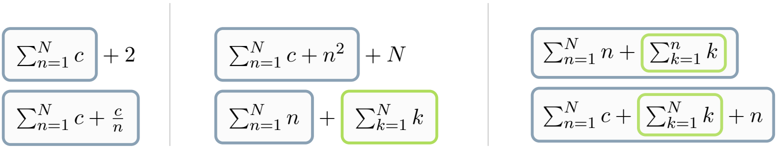


Figure 5.1: Example argument identifications for sums.

5.1.2.3 Lagrange's notation for differentiation and derivatives

Another remaining issue is the Lagrange (prime) notation for differentiation, since it does not outwardly provide sufficient semantic information. This notation presents two challenges. First, we do not know with respect to which variable the differentiation should be performed. Consider for example the Hurwitz zeta function $\zeta(s,a)$ [98, §25.11]. In the case of a differentiation $\zeta'(s,a)$, it is not clear if the function should be differentiated with respect to s or a. To remedy this issue, we analyzed all formulae in the DLMF which use prime notations and determined which variables (slots) for which functions represent the variables of the differentiation. Based on our analysis, we extended the translation patterns by meta information for semantic macros according to the slot of differentiation. For instance, in the case of the Hurwitz zeta function, the first slot is the slot for prime differentiation, i.e., $\zeta'(s,a) = \frac{\mathrm{d}}{\mathrm{d}s}\zeta(s,a)$. The identified variables of differentiations for the special functions in the DLMF can be considered to be the standard slots of differentiations, e.g., in other DML, $\zeta'(s,a)$ most likely refers to $\frac{\mathrm{d}}{\mathrm{d}s}\zeta(s,a)$.

The second challenge occurs if the slot of differentiation contains complex expressions rather than single symbols, e.g., $\zeta'(s^2,a)$. In this case, $\zeta'(s^2,a) = \frac{\mathrm{d}}{\mathrm{d}(s^2)}\zeta(s^2,a)$ instead of $\frac{\mathrm{d}}{\mathrm{d}s}\zeta(s^2,a)$. Since CAS often do not support derivatives with respect to complex expressions, we use the inbuilt substitution functions[7] in the CAS to overcome this issue. To do so, we use a temporary variable `temp` for the substitution. CAS perform substitutions from the inside to the outside. Hence, we can use the same temporary variable `temp` even for nested substitutions. Table 5.2 shows the translation performed for $\zeta'(s^2,a)$. CAS may provide optional arguments to calculate the derivatives for certain special functions, e.g., `Zeta(n,z,a)` in Maple for the n-th derivative of the Hurwitz zeta function. However, this shorthand notation is generally not supported (e.g., Mathematica does not define such an optional parameter). Our substitution approach is more lengthy but also more reliable. Unfortunately, lengthy expressions generally harm the performance of CAS, especially for symbolic manipulations. Hence, we have a genuine interest in keeping translations short, straightforward and readable. Thus, the substitution translation pattern is only triggered if the variable of differentiation is not a single identifier. Note that this substitution only triggers on semantic macros. Generic functions, including prime notations, are still skipped.

Table 5.2: Example translations for the prime derivative of the Hurwitz zeta function with respect to s^2.

System	$\zeta'(s^2,a)$
DLMF	`\Hurwitzzeta'@{s^2}{a}`
Maple	`subs(temp=(s)^(2),` `diff(Zeta(0,temp,a),temp$(1)))`
Mathematica	`D[HurwitzZeta[temp,a],{temp,1}]` `/.temp->(s)^(2)`

A related problem to MEOM of sums, products, integrals, limits, and differentiations are the notations of derivatives. The semantic macro for derivatives `\deriv{w}{x}` (rendered as $\frac{\mathrm{d}w}{\mathrm{d}x}$) is

---

[7] Note that Maple also support an evaluation substitution via the two-argument `eval` function. Since our substitution only triggers on semantic macros, we only use `subs` if the function is defined in Maple. In turn, as far as we know, there is no practical difference between `subs` and the two-argument `eval` in our case.

often used with an empty first argument to render the function behind the derivative notation, e.g., `\deriv{}{x}\sin@{x}` for $\frac{\mathrm{d}}{\mathrm{d}x}\sin x$. This leads to the same problem we faced above for identifying MEOMs. In this case, we use the same heuristic as we did for sums, products, and limits. Note that derivatives may be written following the function argument, e.g., $\sin(x)\frac{\mathrm{d}}{\mathrm{d}x}$. If we are unable to identify any following summand that contains the variable of differentiation before we reach a termination symbol, we look for arguments prior to the derivative according to the heuristic (I-III).

Wronskians With the support of prime differentiation described above, we are also able to translate the Wronskian [98, (1.13.4)] to Maple and Mathematica. A translation requires one to identify the variable of differentiation from the elements of the Wronskian, e.g., z for $\mathscr{W}\{\mathrm{Ai}(z), \mathrm{Bi}(z)\}$ from [98, (9.2.7)]. We analyzed all Wronskians in the DLMF and discovered that most Wronskians have a special function in its argument—such as the example above. Hence, we can use our previously inserted metadata information about the slots of differentiation to extract the variable of differentiation from the semantic macros. If the semantic macro argument is a complex expression, we search for the identifier in the arguments that appear in both elements of the Wronskian. For example, in $\mathscr{W}\{\mathrm{Ai}(z^a), \zeta(z^2, a)\}$, we extract z as the variable since it is the only identifier that appears in the arguments z^a and z^2 of the elements. This approach is also used when there is no semantic macro involved, i.e., from $\mathscr{W}\{z^a, z^2\}$ we extract z as well. If LaCASt extracts multiple candidates or none, it throws a translation exception.

5.1.3 Evaluation of the DLMF using CAS

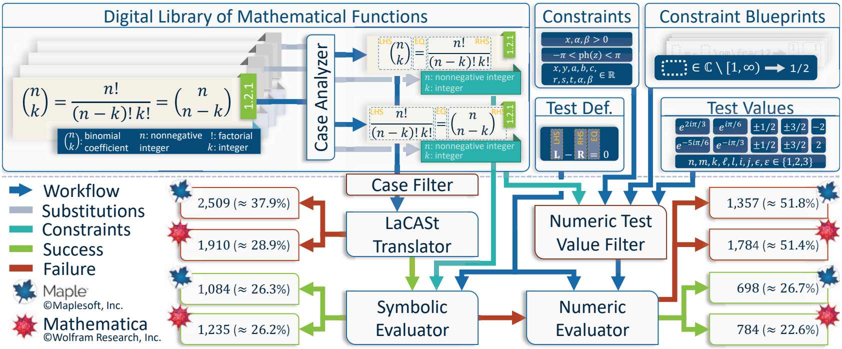


Figure 5.2: The workflow of the evaluation engine and the overall results. Errors and abortions are not included. The generated dataset contains $9,977$ equations. In total, the case analyzer splits the data into $10,930$ cases of which $4,307$ cases were filtered. This sums up to a set of $6,623$ test cases in total.

For evaluating the DLMF with Maple and Mathematica, we symbolically and numerically verify the equations in the DLMF with CAS. If a verification fails, symbolically and numerically, we identified an issue either in the DLMF, the CAS, or the verification pipeline. Note that an issue does not necessarily represent errors/bugs in the DLMF, CAS, or LaCASt (see the discussion about branch cuts in Section 5.1.4.1). Figure 5.2 illustrates the pipeline of the evaluation engine.

First, we analyze every equation in the DLMF (hereafter referred to as test cases). A case analyzer splits multiple relations in a single line into multiple test cases. Note that only the adjacent relations are considered, i.e., with $f(z) = g(z) = h(z)$, we generate two test cases $f(z) = g(z)$ and $g(z) = h(z)$ but not $f(z) = h(z)$. In addition, expressions with $\pm$ and $\mp$ are split accordingly, e.g., $i^{\pm i} = e^{\mp\pi/2}$ [98, (4.4.12)] is split into $i^{+i} = e^{-\pi/2}$ and $i^{-i} = e^{+\pi/2}$. The analyzer utilizes the attached additional information in each line, i.e., the URL in the DLMF, the used and defined symbols, and the constraints. If a used symbol is defined elsewhere in the DLMF, it performs substitutions. For example, the multi-equation [98, (9.6.2)] is split into six test cases and every ζ is replaced by $\frac{2}{3}z^{3/2}$ as defined in [98, (9.6.1)]. The substitution is performed on the parse tree of expressions [10]. A definition is only considered as such, if the defining symbol is identical to the equation's left-hand side. That means, $z = (\frac{3}{2}\zeta)^{3/2}$ [98, (9.6.10)] is not considered as a definition for ζ. Further, semantic macros are never substituted by their definitions. Translations for semantic macros are exclusively defined by the authors. For example, the equation [98, (11.5.2)] contains the Struve $\mathbf{K}_\nu(z)$ function. Since Mathematica does not contain this function, we defined an alternative translation to its definition $\mathbf{H}_\nu(z) - Y_\nu(z)$ in [98, (11.2.5)] with the Struve function $\mathbf{H}_\nu(z)$ and the Bessel function of the second kind $Y_\nu(z)$, because both of these functions are supported by Mathematica. The second entry in Table E.2 in Appendix E available in the electronic supplementary material shows the translation for this test case.

Next, the analyzer checks for additional constraints defined by the used symbols recursively. The mentioned Struve $\mathbf{K}_\nu(z)$ test case [98, (11.5.2)] contains the Gamma function. Since the definition of the Gamma function [98, (5.2.1)] has a constraint $\Re z > 0$, the numeric evaluation must respect this constraint too. For this purpose, the case analyzer first tries to link the variables in constraints to the arguments of the functions. For example, the constraint $\Re z > 0$ sets a constraint for the first argument z of the Gamma function. Next, we check all arguments in the actual test case at the same position. The test case contains $\Gamma(\nu + 1/2)$. In turn, the variable z in the constraint of the definition of the Gamma function $\Re z > 0$ is replaced by the actual argument used in the test case. This adds the constraint $\Re(\nu + 1/2) > 0$ to the test case. This process is performed recursively. If a constraint does not contain any variable that is used in the final test case, the constraint is dropped.

In total, the case analyzer would identify four additional constraints for the test case [98, (11.5.2)][8]. Note that the constraints may contain variables that do not appear in the actual test case, such as $\Re\nu + k + 1 > 0$. Such constraints do not have any effect on the evaluation because if a constraint cannot be computed to true or false, the constraint is ignored. Unfortunately, this recursive loading of additional constraints may generate impossible conditions in certain cases, such as $|\Gamma(iy)|$ [98, (5.4.3)]. There are no valid real values of y such that $\Re(iy) > 0$. In turn, every test value would be filtered out, and the numeric evaluation would not verify the equation. However, such cases are the minority and we were able to increase the number of correct evaluations with this feature.

To avoid a large portion of incorrect calculations, the analyzer filters the dataset before translating the test cases. We apply two filter rules to the case analyzer. First, we filter expressions that do not contain any semantic macros. Due to the limitations of LaCASt, these expressions most likely result in wrong translations. Further, it filters out several meaningless expressions

---

[8] See Table E.2 in Appendix E available in the electronic supplementary material for the applied constraints (including the directly attached constraint $\Re z > 0$ and the manually defined global constraints from Figure 5.3).

that are not verifiable, such as $z = x$ in [98, (4.2.4)]. The result dataset flag these cases with '*Skipped - no semantic math*'. Note that the result dataset still contains the translations for these cases to provide a complete picture of the DLMF. Second, we filter expressions that contain ellipsis[9] (e.g., `\cdots`), approximations, and asymptotics (e.g., $\mathcal{O}(z^2)$) since those expressions cannot be evaluated with the proposed approach. Further, a definition is skipped if it is not a definition of a semantic macro, such as [98, (2.3.13)], because definitions without an appropriate counterpart in the CAS are meaningless to evaluate. Definitions of semantic macros, on the other hand, are of special interest and remain in the test set since they allow us to test if a function in the CAS obeys the actual mathematical definition in the DLMF. If the case analyzer (see Figure 5.2) is unable to detect a relation, i.e., split an expression on $<$, $\leq$, $\geq$, $>$, $=$, or $\neq$, the line in the dataset is also skipped because the evaluation approach relies on relations to test. After splitting multi-equations (e.g., $\pm$, $\mp$, $a = b = c$), filtering out all non-semantic expressions, non-semantic macro definitions, ellipsis, approximations, and asymptotics, we end up with $6,623$ test cases in total from the entire DLMF.

After generating the test case with all constraints, we translate the expression to the CAS representation. Every successfully translated test case is then symbolically verified, i.e., the CAS tries to simplify the difference of an equation to zero. Non-equation relations simplifies to Booleans. Non-simplified expressions are verified numerically for manually defined test values, i.e., we calculate actual numeric values for both sides of an equation and check their equivalence.

5.1.3.1 Symbolic Evaluation

The symbolic evaluation was performed for Maple as described in the following (taken from [2]). Originally, we used the standalone Maple `simplify` function directly, to symbolically simplify translated formulae. See [26, 28, 148, 190] for other examples of where Maple and other CAS simplification procedures has been used elsewhere in the literature. Symbolic simplification is performed either on the difference or the division of the left-hand sides and the right-hand sides of extracted formulae. Thus the expected outcome should be respectively either a `0` or `1`. Note that other outcomes, such as other numerical outcomes, are particularly interesting, since these may be an indication of errors in the formulae.

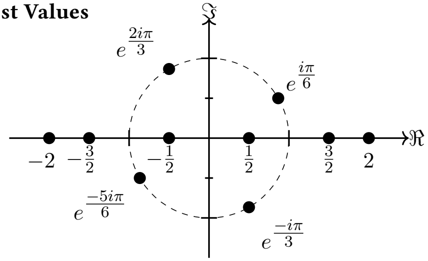
In Maple, symbolic simplifications are made using internally stored relations to other functions. If a simplification is available, then in practice it often has to be performed over multiple defined relevant relations. Often, this process fails and Maple is unable to simplify the said expression. We have adopted some techniques which assist Maple in this process. For example, forcing an expression to be converted into another specific representation, in a pre-processing step, could potentially improve the odds that Maple is able to recognize a possible simplification. By trial-and-error, we discovered (and implemented) the following pre-processing steps which significantly improve the simplification process:

- conversion to exponential representation;
- conversion to hypergeometric representation;
- expansion of expressions (for example `(x+y)^2`); and
- combined expansion and conversion processes.

---

[9] Note that we filter out ellipsis (e.g., `\cdots`) but not single dots (e.g., `\cdot`).

Test Values



Special Test Values

$n, m, k, \ell, l, i, j, \epsilon, \varepsilon \in \{1, 2, 3\}$

Global Constraints

$x, \alpha, \beta > 0$

$-\pi < \mathrm{ph}(z) < \pi$

$x, y, a, b, c, r, s, t, \alpha, \beta \in \mathbb{R}$

Figure 5.3: The ten numeric test values in the complex plane for general variables. The dashed line represents the unit circle $|z| = 1$. At the right, we show the set of values for special variable values and general global constraints. On the right, i is referring to a generic variable and not to the imaginary unit.

In comparison to the original approach described in [2], we use the newer version Maple 2020 now. Another feature we added to LACAST is the support of packages in Maple. Some functions are only available in modules (packages) that must be preloaded, such as `QPochhammer` in the package `QDifferenceEquations`[10]. The general `simplify` method in Maple does not cover q-hypergeometric functions. Hence, whenever LACAST loads functions from the q-hypergeometric package, the better performing `QSimplify` method is used. With the WED and the new support for Mathematica in LACAST, we perform the symbolic and numeric tests for Mathematica as well. The symbolic evaluation in Mathematica relies on the full simplification[11]. For Maple and Mathematica, we defined the global assumptions $x, y \in \mathbb{R}$ and $k, n, m \in \mathbb{N}$. Constraints of test cases are added to their assumptions to support simplification. Adding more global assumptions for symbolic computation generally harms the performance since CAS internally uses assumptions for simplifications. It turned out that by adding more custom assumptions, the number of successfully simplified expressions decreases.

5.1.3.2 Numerical Evaluation

Defining an accurate test set of values to analyze an equivalence can be an arbitrarily complex process. It would make sense that every expression is tested on specific values according to the containing functions. However, this laborious process is not suitable for evaluating the entire DML and CAS. It makes more sense to develop a general set of test values that (i) generally covers interesting domains and (ii) avoid singularities, branch cuts, and similar problematic regions. Considering these two attributes, we come up with the ten test points illustrated in Figure 5.3. It contains four complex values on the unit circle and six points on the real axis. The test values cover the general area of interest (complex values in all four quadrants, negative and positive real values) and avoid the typical singularities at $\{0, \pm 1, \pm i\}$. In addition, several variables are tied to specific values for entire sections. Hence, we applied additional global constraints to the test cases.

---

[10] https://jp.maplesoft.com/support/help/Maple/view.aspx?path=QDifferenceEquations/QPochhammer [accessed 2021-05-01]

[11] https://reference.wolfram.com/language/ref/FullSimplify.html [accessed 2021-05-01]

The numeric evaluation engine heavily relies on the performance of extracting free variables from an expression. Maple does not provide a function to extract free variables from an expression. Hence, we implemented a custom method first. Variables are extracted by identifying all `names` [36][12] from an expression. This will also extract constants which need to be deleted from the list first. Unfortunately, inbuilt functions in CAS, if available, and our custom implementation for Maple are not very reliable. Mathematica has the undocumented function `Reduce‘FreeVariables` for this purpose. However, both systems, the custom solution in Maple and the inbuilt Mathematica function, have problems distinguishing free variables of entire expressions from the bound variables in MEOMs, e.g., integration and continuous variables. Mathematica sometimes does not extract a variable but returns the unevaluated input instead. We regularly faced this issue for integrals. However, we discovered one example without integrals. For `EulerE[n,0]` from [98, (24.4.26)], we expected to extract $\{n\}$ as the set of free variables but instead received a set of the unevaluated expression itself {`EulerE[n,0]`}[13]. Since the extended version of LaCaST handles operators, including bound variables of MEOMs, we drop the use of internal methods in CAS and extend LaCaST to extract identifiers from an expression. During a translation process, LaCaST tags every single identifier as a variable, as long as it is not an element of a MEOM. This simple approach proves to be very efficient since it is implemented alongside the translation process itself and is already more powerful as compared to the existing inbuilt CAS solutions. We defined subscripts of identifiers as a part of the identifier, e.g., z_1 and z_2 are extracted as variables from $z_1 + z_2$ rather than z.

The general pipeline for a numeric evaluation works as follows. First, we replace all substitutions and extract the variables from the left- and right-hand sides of the test expression via LaCaST. For the previously mentioned example of the Struve function [98, (11.5.2)], LaCaST identifies two variables in the expression, ν and z. According to the values in Figure 5.3, ν and z are set to the general ten values. A numeric test contains every combination of test values for all variables. Hence, we generate 100 test calculations for [98, (11.5.2)]. Afterward, we filter the test values that violate the attached constraints. In the case of the Struve function, we end up with 25 test cases (see also Table E.2 in Appendix E available in the electronic supplementary material).

In addition, we apply a limit of 300 calculations for each test case and abort a computation after 30 seconds due to computational limitations. If the test case generates more than 300 test values, only the first 300 are used. Finally, we calculate the result for every remaining test value, i.e., we replace every variable by their value and calculate the result. The replacement is done by Mathematica’s `ReplaceAll` method because the more appropriate method `With`, for unknown reasons, does not always replace all variables by their values. We wrap test expressions in `Normal` for numeric evaluations to avoid conditional expressions, which may cause incorrect calculations (see Section 5.1.4.1 for a more detailed discussion of conditional outputs). After replacing variables by their values, we trigger numeric computation. If the absolute value of the result is below the defined threshold of 0.001 or true (in the case of inequalities), the test calculation is considered successful. A numeric test case is only considered successful if and only if every test calculation was successful. If a numeric test case fails, we store the information on which values it failed and how many of these were successful.

---

[12] A *name* in Maple is a sequence of one or more characters that uniquely identifies a command, file, variable, or other entity.

[13] The bug was reported to and confirmed by Wolfram Research Version 12.0.

5.1.4 Results

The translations to Maple and Mathematica, the symbolic results, the numeric computations, and an overview PDF of the reported bugs to Mathematica are available online[14]. In the following, we mainly focus on Mathematica because of page limitations and because Maple has been investigated more closely by [2]. The results for Maple are also available online. Compared to the baseline ($\approx 31\%$), our improvements doubled the amount translations ($\approx 62\%$) for Maple and reach $\approx 71\%$ for Mathematica. The majority of expressions that cannot be translated contain macros that have no adequate translation pattern to the CAS, such as the macros for interval Weierstrass lattice roots [98, §23.3(i)] and the multivariate hypergeometric function [98, (19.16.9)]. Other errors (6% for Maple and Mathematica) occur for several reasons. For example, out of the 418 errors in translations to Mathematica, 130 caused an error because the MEOM of an operator could not be extracted, 86 contained prime notations that do not refer to differentiations, 92 failed because of the underlying LaTeX parser [402], and in 46 cases, the arguments of a DLMF macro could not be extracted.

Out of $4{,}713$ translated expressions, $1{,}235$ (26.2%) were successfully simplified by Mathematica ($1{,}084$ of $4{,}114$ or 26.3% in Maple). For Mathematica, we also count results that are equal to 0 under certain conditions as successful (called `ConditionalExpression`). We identified 65 of these conditional results: 15 of the conditions are equal to constraints that were provided in the surrounding text but not in the info box of the DLMF equation; 30 were produced due to branch cut issues (see Section 5.1.4.1); and 20 were the same as attached in the DLMF but reformulated, e.g., $z \in \mathbb{C} \backslash (1, \infty)$ from [98, (25.12.2)] was reformulated to $\Im z \neq 0 \vee \Re z < 1$. The remaining translated but not symbolically verified expressions were numerically evaluated for the test values in Figure 5.3. For the $3{,}474$ cases, 784 (22.6%) were successfully verified numerically by Mathematica (698 of $2{,}618$ or 26.7% by Maple[15]). For $1{,}784$ the numeric evaluation failed. In the evaluation process, 655 computations timed out and 180 failed due to errors in Mathematica. Of the $1{,}784$ failed cases, 691 failed partially, i.e., there was at least one successful calculation among the tested values. For $1{,}091$ all test values failed. The Appendix E, available in the electronic supplementary material, provides a Table E.2 with the results for three sample test cases. The first case is a false positive evaluation because of a wrong translation. The second case is valid, but the numeric evaluation failed due to a bug in Mathematica (see next subsection). The last example is valid and was verified numerically but was too complex for symbolic verifications.

5.1.4.1 Error Analysis

The numeric tests' performance strongly depends on the correct attached and utilized information. The example [98, (1.4.8)] from the DLMF

$$\frac{\mathrm{d}^2 f}{\mathrm{d}x^2} = \frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{\mathrm{d}f}{\mathrm{d}x}\right), \tag{5.2}$$

illustrates the difficulty of the task on a relatively easy case[16]. Here, the argument of f was not explicitly given, such as in $f(x)$. Hence, LaCASt translated f as a variable. Unfortunately,

---

[14] `https://lacast.wmflabs.org/` [accessed 2021-10-01]

[15] Due to computational issues, 120 cases must have been skipped manually. 292 cases resulted in an error during symbolic verification and, therefore, were skipped also for numeric evaluations. Considering these skipped cases as failures, decreases the numerically verified cases to 23% in Maple.

[16] This is the first example in Table E.2

this resulted in a false verification symbolically and numerically. This type of error mostly appears in the first three chapters of the DLMF because they use generic functions frequently. We hoped to skip such cases by filtering expressions without semantic macros. Unfortunately, this derivative notation uses the semantic macro `deriv`. In the future, we filter expressions that contain semantic macros that are not linked to a special function or orthogonal polynomial.

As an attempt to investigate the reliability of the numeric test pipeline, we can run numeric evaluations on symbolically verified test cases. Since Mathematica already approved a translation symbolically, the numeric test should be successful if the pipeline is reliable. Of the $1{,}235$ symbolically successful tests, only 94 (7.6%) failed numerically. None of the failed test cases failed entirely, i.e., for every test case, at least one test value was verified. Manually investigating the failed cases reveal 74 cases that failed due to an `Indeterminate` response from Mathematica and 5 returned `infinity`, which clearly indicates that the tested numeric values were invalid, e.g., due to testing on singularities. Of the remaining 15 cases, two were identical: [98, (15.9.2)] and [98, (18.5.9)]. This reduces the remaining failed cases to 14. We evaluated invalid values for 12 of these because the constraints for the values were given in the surrounding text but not in the info boxes. The remaining 2 cases revealed a bug in Mathematica regarding conditional outputs (see below). The results indicate that the numeric test pipeline is reliable, at least for relatively simple cases that were previously symbolically verified. The main reason for the high number of failed numerical cases in the entire DLMF (1,784) are due to missing constraints in the i-boxes and branch cut issues (see Section 5.1.4.1), i.e., we evaluated expressions on invalid values.

Bug reports Mathematica has trouble with certain integrals, which, by default, generate conditional outputs if applicable. With the method `Normal`, we can suppress conditional outputs. However, it only hides the condition rather than evaluating the expression to a non-conditional output. For example, integral expressions in [98, (10.9.1)] are automatically evaluated to the Bessel function $J_0(|z|)$ for the condition[17] $z \in \mathbb{R}$ rather than $J_0(z)$ for all $z \in \mathbb{C}$. Setting the `GenerateConditions`[18] option to `None` does not change the output. `Normal` only hides $z \in \mathbb{R}$ but still returns $J_0(|z|)$. To fix this issue, for example in (10.9.1) and (10.9.4), we are forced to set `GenerateConditions` to false.

Setting `GenerateConditions` to false, on the other hand, reveals severe errors in several other cases. Consider $\int_z^\infty t^{-1}e^{-t}\,\mathrm{d}t$ [98, (8.4.4)], which gets evaluated to $\Gamma(0,z)$ but (condition) for $\Re z > 0 \wedge \Im z = 0$. With `GenerateConditions` set to false, the integral incorrectly evaluates to $\Gamma(0,z) + \ln(z)$. This happened with the 2 cases mentioned above. With the same setting, the difference of the left- and right-hand sides of [98, (10.43.8)] is evaluated to 0.398942 for $x, \nu = 1.5$. If we evaluate the same expression on $x, \nu = \frac{3}{2}$ the result is `Indeterminate` due to `infinity`. For this issue, one may use `NIntegrate` rather than `Integrate` to compute the integral. However, evaluating via `NIntegrate` decreases the number of successful numeric evaluations in general. We have revealed errors with conditional outputs in (8.4.4), (10.22.39), (10.43.8-10), and (11.5.2) (in [98]). In addition, we identified one critical error in Mathematica. For [98, (18.17.47)], WED (Mathematica's kernel) ran into a *segmentation fault (core dumped)* for $n > 1$. The kernel of the full version of Mathematica gracefully died without returning an output[19].

---

[17] $J_0(x)$ with $x \in \mathbb{R}$ is even. Hence, $J_0(|z|)$ is correct under the given condition.

[18] https://reference.wolfram.com/language/ref/GenerateConditions.html [accessed 2021-05-01]

[19] All errors were reported to and confirmed by Wolfram Research.

Besides Mathematica, we also identified several issues in the DLMF. None of the newly identified issues were critical, such as the reported sign error from the previous project [2], but generally refer to missing or wrong attached semantic information. With the generated results, we can effectively fix these errors and further semantically enhance the DLMF. For example, some definitions are not marked as such, e.g., $Q(z) = \int_0^\infty e^{-zt} q(t)\,\mathrm{d}t$ [98, (2.4.2)]. In [98, (10.24.4)], ν must be a real value but was linked to a *complex parameter* and x should be positive real. An entire group of cases [98, (10.19.10-11)] also discovered the incorrect use of semantic macros. In these formulae, $P_k(a)$ and $Q_k(a)$ are defined but had been incorrectly marked up as Legendre functions going all the way back to DLMF Version 1.0.0 (May 7, 2010). In some cases, equations are mistakenly marked as definitions, e.g., [98, (9.10.10)] and [98, (9.13.1)] are annotated as local definitions of n. We also identified an error in LaCASt, which incorrectly translated the exponential integrals $E_1(z)$, $\mathrm{Ei}(x)$ and $\mathrm{Ein}(z)$ (defined in [98, §6.2(i)]). A more explanatory overview of discovered, reported, and fixed issues in the DLMF, Mathematica, and Maple is provided in Appendix D available in the electronic supplementary material.

Branch cut issues Problems that we regularly faced during evaluation are issues related to multi-valued functions. Multi-valued functions map values from a domain to multiple values in a codomain and frequently appear in the complex analysis of elementary and special functions. Prominent examples are the inverse trigonometric functions, the complex logarithm, or the square root. A proper mathematical description of multi-valued functions requires the complex analysis of Riemann surfaces. Riemann surfaces are one-dimensional complex manifolds associated with a multi-valued function. One usually multiplies the complex domain into a many-layered covering space. The correct properties of multi-valued functions on the complex plane may no longer be valid by their counterpart functions on CAS, e.g., $(1/z)^w$ and $1/(z^w)$ for $z, w \in \mathbb{C}$ and $z \neq 0$. For example, consider $z, w \in \mathbb{C}$ such that $z \neq 0$. Then mathematically, $(1/z)^w$ always equals $1/(z^w)$ (when defined) for all points on the Riemann surface with fixed w. However, this should certainly not be assumed to be true in CAS, unless very specific assumptions are adopted (e.g., $w \in \mathbb{Z}$, $z > 0$). For all modern CAS[20], this equation is not true. Try, for instance, $w = 1/2$. Then $(1/z)^{1/2} - 1/z^{1/2} \neq 0$ on CAS, nor for w being any other rational non-integer number.

In order to compute multi-valued functions, CAS choose branch cuts for these functions so that they may evaluate them on their principal branches. Branch cuts may be positioned differently among CAS [84], e.g., $\mathrm{arccot}(-\frac{1}{2}) \approx 2.03$ in Maple but is ≈ -1.11 in Mathematica. This is certainly not an error and is usually well documented for specific CAS [108, 171]. However, there is no central database that summarizes branch cuts in different CAS or DML. The DLMF as well, explains and defines their branch cuts carefully but does not carry the information within the info boxes of expressions. Due to complexity, it is rather easy to lose track of branch cut positioning and evaluate expressions on incorrect values. For example, consider the equation [98, (12.7.10)]. A path of $z(\phi) = e^{i\phi}$ with $\phi \in [0, 2\pi]$ would pass three different branch cuts. An accurate evaluation of the values of $z(\phi)$ in CAS require calculations on the three branches using analytic continuation. LaCASt and our evaluation frequently fall into the same trap by evaluating values that are no longer on the principal branch used by CAS. To solve this issue, we need to utilize branch cuts not only for every function but also for every equation in the DLMF [10]. The positions of branch cuts are exclusively provided in the text

---

[20] The authors are not aware of any example of a CAS which treats multi-valued functions without adopting principal branches.

but not in the i-boxes. Adding the information to each equation in the DLMF would be a laborious process because a branch cut position may change according to the used values (see the example [98, (12.7.10)] from above). Our result data, however, would provide beneficial information to update, extend, and maintain the DLMF, e.g., by adding the positions of the branch cuts for every function. An extended discussion about branch cut issues is available in Appendix A available in the electronic supplementary material.

5.1.5 Conclude Quantitative Evaluations on the DLMF

We have presented a novel approach to verify the theoretical digital mathematical library DLMF with the power of two major general-purpose computer algebra systems Maple and Mathematica. With LACAST, we transformed the semantically enhanced LaTeX expressions from the DLMF to each CAS. Afterward, we symbolically and numerically evaluated the DLMF expressions in each CAS. Our results are auspicious and provide useful information to maintain and extend the DLMF efficiently. We further identified several errors in Mathematica, Maple [2], the DLMF, and the transformation tool LACAST, proving the profit of the presented verification approach. Further, we provide open access to all results, including translations and evaluations[21].

The presented results show a promising step towards an answer for our initial research question. By translating an equation from a DML to a CAS, automatic verifications of that equation in the CAS allows us to detect issues in either the DML source or the CAS implementation. Each analyzed failed verification successively improves the DML or the CAS. Further, analyzing a large number of equations from the DML may be used to finally verify a CAS. In addition, the approach can be extended to cover other DML and CAS by exploiting different translation approaches, e.g., via MathML [18] or OpenMath [152].

Nonetheless, the analysis of the results, especially for an entire DML, is cumbersome. Minor missing semantic information, e.g., a missing constraint or not respected branch cut positions, leads to a relatively large number of false positives, i.e., unverified expressions correct in the DML and the CAS. This makes a generalization of the approach challenging because all semantics of an equation must be taken into account for a trustworthy evaluation. Furthermore, evaluating equations on a small number of discrete values will never provide sufficient confidence to verify a formula, which leads to an unpredictable number of true negatives, i.e., erroneous equations that pass all tests.

After all, we conclude that the approach provides valuable information to complement, improve, and maintain the DLMF, Maple, and Mathematica. A trustworthy verification, on the other hand, might be out of reach.

5.1.5.1 Future Work

The resulting dataset provides valuable information about the differences between CAS and the DLMF. These differences had not been largely studied in the past and are worthy of analysis. Especially a comprehensive and machine-readable list of branch cut positioning in different systems is a desired goal [84]. Hence, we will continue to work closely together with the editors of the DLMF to improve further and expand the available information on the DLMF. Finally, the numeric evaluation approach would benefit from test values dependent on the actual functions involved. For example, the current layout of the test values was designed to avoid

---

[21] `https://lacast.wmflabs.org/` [accessed 2021-10-01]

problematic regions, such as branch cuts. However, for identifying differences in the DLMF and CAS, especially for analyzing the positioning of branch cuts, an automatic evaluation of these particular values would be very beneficial and can be used to collect a comprehensive, inter-system library of branch cuts. Therefore, we will further study the possibility of linking semantic macros with numeric regions of interest.

Finally, we used LACAST to perform translations solely on semantic LaTeX expressions. Real-world mathematics, however, is not available in this semantically enriched format. In the previous chapter, we already developed and discussed a context-sensitive extension for LACAST. This enables LACAST to translate not only semantic LaTeX formulae from the DLMF but, considering an informative textual context, also general mathematical expressions to multiple CAS. In the following section, we will evaluate this new extension of LACAST on Wikipedia articles.

5.2 Evaluations on Wikipedia

In the following, resulting from our motivation outlined in Chapter 4 - improving Wikipedia articles - we use Wikipedia for our test dataset to evaluate our context-sensitive extension of LACAST. More specifically, we considered every English Wikipedia article that references to the DLMF via the `{{dlmf}}` template[22]. This should limit the domain to OPSF problems that we are currently examining. The English Wikipedia contains 104 such pages, of which only one page did not contain any formula (Spheroidal wave function)[23]. For the entire dataset (the remaining 103 Wikipedia pages), we detected $6,337$ formulae in total (including potential erroneous math).

So far, one of our initial three issues from Section 4.2.3 still remains unsolved: how can we determine if a translation was appropriate and complete? We called a translation appropriate, if the intended meaning of a presentational expression $e \in \mathcal{L}_P$ is the same as the translated expression $\mathfrak{t}(e, X) \in \mathcal{L}_C$. However, how can we know the intended semantic meaning of a presentational expression $e \in \mathcal{L}_P$? In natural languages, the BLEU score [282] is widely used to judge the quality of a translation. The effectiveness of the BLEU score, however, is questionable when it comes to math translations due to the complexity and high interconnectedness of mathematical formulae. Consider, a translation of the arccotangent function $\operatorname{arccot}(x)$ was performed to $\arctan(1/(x))$ in Maple. This translation is correct and even preferred in certain situations to avoid issues with so-called branch cuts (see [13, Section 3.2]). Previously, we developed a new approach that relies on automatic verification checks with CAS [2, 11] to verify a translation. This approach is very powerful for large datasets. However, it requires a large and precise amount of semantic data about the involved formulae, including constraints, domains, the position of branch cuts, and other information to reach high accuracy. In turn, we perform this automatic verification on the entire 103 Wikipedia pages but additionally created a benchmark dataset with 95 entries for qualitative analysis. To avoid issues like with the BLEU score, we manually evaluated each translation of the 95 test cases.

---

[22] Templates in Wikitext are placeholders for repetitive information which get resolved by Wikitext parsers. The DLMF-template, for example, adds the external reference for the DLMF to the article.

[23] Retrieved from `https://en.wikipedia.org/wiki/Special:WhatLinksHere` by searching for *Template:Dlmf* [accessed 2021-01-01]

5.2.1 Symbolic and Numeric Testing

The automatic verification approach makes the assumption that a correct equation in the domain must remain valid in the codomain after a translation. If the equation is incorrect after a translation, we conclude a translation error. As we have discussed in the previous Section 5.1, we examined two approaches to verify an equation in a CAS. The first approach tries to symbolically simplify the difference of the left- and right-hand sides of an equation to zero. If the simplification returned zero, the equation was symbolically verified by the CAS. Symbolic simplifications of CAS, however, are rather limited and may even fail on simple equations. The second approach overcomes this issue by numerically calculating the difference between the left- and right-hand sides of an equation on specific numeric test values. If the difference is zero (or below a given threshold due to machine accuracy) for every test calculation, the equivalence of an equation was numerically verified. Clearly, the numeric evaluation approach cannot prove equivalence. However, it can prove disparity and therefore detect an error due to the translation.

In the previous Section 5.1, we saw that the translations by LACAST [13] were so reliable that the combination of symbolic and numeric evaluations was able to detect errors in the domain library (i.e., the DLMF) and the codomain systems (i.e., the CAS Maple and Mathematica) [2, 11]. Unfortunately, the number of false positives, i.e., correct equations that were not verified symbolically nor numerically, is relatively high. The main reason is unconsidered semantic information, such as constraints for specific variables or the position of branch cuts. Unconsidered semantic information causes the system to test equivalence on invalid conditions, such as invalid values, and therefore yields inequalities between the left- and right-hand sides of an equation even though the source equation and the translation were correct. Nonetheless, the symbolic and numeric evaluation approach proofs to be very useful also for our translation system. It allows us to quantitatively evaluate a large number of expressions in Wikipedia. In addition, it enables continuous integration testing for mathematics in Wikipedia article revisions. For example, an equation previously verified by the system that fails after a revision could indicate a poisoned revision of the article. This automatic plausibility check might be a jump start for the ORES system to better maintain the quality of mathematical documents [359]. For changes in math equations, ORES could trigger a plausibility check through our translation and verification pipeline and adjust the score of good faith of damaging an edit accordingly.

5.2.2 Benchmark Testing

To compensate for the relatively low number of verifiable equations in Wikipedia with the symbolic and numeric evaluation approach, we crafted a benchmark test dataset to qualitatively evaluate the translations. This benchmark includes a single equation (the formulae must contain a top-level equality symbol[24], no `\text`, and no `\color` macros) randomly picked from each Wikipedia article from our dataset. For eight articles, no such equation was detected. Hence, the benchmark contains 95 test expressions. For each formula, we marked the extracted descriptive terms as irrelevant (0), relevant (1), or highly relevant (2), and manually translated the expressions to semantic LaTeX and to Maple and Mathematica. If the formula contained a function for which no appropriate semantic macro exists, the semantic LaTeX equals the generic (original) LaTeX of this function. In 18 cases, even the human annotator was unable to appropriately

---

[24] This excludes equality symbols of deeper levels in the parse tree, e.g., the equality symbols in sums are not considered as such.

Table 5.3: The symbolic and numeric evaluations on all 6, 337 expressions from the dataset with the number of translated expressions (**T**), the number of started test evaluations (**Started**), the success rates (**Success**), and the success rates on the DLMF dataset for comparison (DLMF). The DLMF scores refer to the results presented in the previous Section 5.1.

Symbol Evaluation

	T	**Started**	**Success**	**DLMF**
Maple	4, 601	1, 747	.113	.264
Mathematica	4, 678	1, 692	.158	.262

Numeric Evaluation

	T	**Started**	**Success**	**DLMF**
Maple	4, 601	1, 627	.181	.433
Mathematica	4, 678	1, 516	.236	.429

translate the expressions to the CAS, which underlines the difficulty of the task. The main reason for a manual translation failure was missing information (the necessary information for an appropriate translation was not given in the article) or it contained elements for which an appropriate translation was not possible, such as contour integrals, approximations, or indefinite lists of arguments with dots (e.g., $a_1, \ldots, a_n$). Note that the domain of orthogonal polynomials and special functions is a well-supported domain for many general-purpose CAS, like Maple and Mathematica. Hence, in other domains, such as in group, number, or tensor field theory, we can expect a significant drop of human-translatable expressions[25]. Since Mathematica is able to import LaTeX expressions, we use this import function as a baseline for our translations to Mathematica. We provide full access to the benchmark via our demo website and added an overview to Appendix F.4 available in the electronic supplementary material.

5.2.3 Results

First, we evaluated the 6, 337 detected formulae with our automatic evaluation via Maple and Mathematica. Table 5.3 shows an overview of this evaluation. With our translation pipeline, we were able to translate 72.6% of mathematical expressions into Maple and 73.8% into Mathematica syntax. From these translations, around 40% were symbolically and numerically evaluated (the rest was filtered due to missing equation symbols, illegal characters, etc.). We were able to symbolically verify 11% (Maple) and 15% (Mathematica), and numerically verify 18% (Maple) and 24% (Mathematica). In comparison, the same tests on the manually annotated semantic dataset of DLMF equations [403] reached a success rate of 26% for symbolic and 43% for numeric evaluations [11] (see the previous Section 5.1). Since the DLMF is a manually annotated semantic dataset that provides exclusive access to constraints, substitutions, and other relevant information, we achieve very promising results with our context-sensitive pipeline. To test a theoretical continuous integration pipeline for the ORES system in Wikipedia articles, we also analyzed edits in math equations that have been reverted again. The Bessel's function contains

---

[25] Note that there are numerous specialized CAS that would cover the mentioned domains too, such as GAP [177], PARI/GP [283], or Cadabra [290].

such an edit on the equation

$$J_n(x) = \frac{1}{\pi}\int_0^\pi \cos(n\tau - x\sin\tau)\, d\tau. \tag{5.3}$$

Here, the edit[26] changed $J_n(x)$ to $J_ZWE(x)$. Our pipeline was able to symbolically and numerically verify the original expression but failed on the revision. The ORES system could profit from this result and adjust the score according to the automatic verification via CAS.

5.2.3.1 Descriptive Term Extractions

Previously, we presumed that our update of the description retrieval approach to MOI would yield better results. In order to check the ranking of retrieved facts, we evaluate the descriptive terms extractions and compare the results with our previously reported F1 scores in [330]. We analyze the performance for a different number of retrieved descriptions and different depths. Here, the depth refers to the maximum depth of in-going dependencies in the dependency graph to retrieve relevant descriptions. A depth value of zero does not retrieve additional terms from the in-going dependencies but only the noun phrases that are directly annotated to the formula itself. The results for relevance 1 or higher are given in Table 5.4a and for relevance 2 in Table 5.4b. Since we need to retrieve a high number of relevant facts to achieve a complete translation, we are more interested in retrieving any relevant fact rather than a single but precise description. Hence, the performance for relevance 1 is more appropriate for our task. For a better comparison with our previous pipeline [330], we also analyze the performance only on highly relevant descriptions (relevance 2). As expected, for relevant noun phrases, we outperform the reported F1 score (.35). For highly relevant entries only, our updated MOI pipeline achieves similar results with an F1 score of .385.

5.2.3.2 Semantification

Since we split our translation pipeline into two steps, semantification and mapping, we evaluate the semantification transformations first. To do this, we use our benchmark dataset and perform tree comparisons of our generated transformed tree $\mathrm{t}_s(e, X)$ and the semantically enhanced tree using semantic macros. The number of facts we take into account affects the performance. Fewer facts and the transformation might be not complete, i.e., there are still subtrees in e that should be already in $\mathcal{L}_C$. Too many facts increase the risk of false positives, that yield wrong transformations. In order to estimate how many facts we need to retrieve to achieve a complete transformation, we evaluated the comparison on different depths D and limit the number of facts with the same MOI, i.e., we only consider the top-ranked facts f for an MOI according to $\mathrm{s}_{\mathrm{MLP}}(f)$. In addition, we limit the number of retrieved rules r_f per MC. We observed that an equal limit of retrieved MC per MOI and r_f per MC performed best. Consider we set the limit N to five, we would retrieve a maximum of 25 facts (five r_f for each of the five MC for a single MOI). Typically, the number of retrieved facts f is below this limit because similar MC yield similar r_f. In addition, we found that considering replacement patterns with a likelihood of 0% (i.e., the rendered version of this macro never appears in the DLMF), harms performance drastically. This is because semantic macros without any arguments regularly match single letters, for example, Γ representing the gamma function with the argument (z)

---

[26] https://en.wikipedia.org/w/index.php?diff=991994767&oldid=991251002&title=Bessel_function&type=revision [accessed 2021-06-23]

Table 5.4: Performance of description extractions via MLP for low (5.4a) and high (5.4b) relevance. In all tables, **D** refers to the depth (following ingoing dependencies) in the dependency graph, **N** is the maximum number of facts and r_f for the same MOI, TP are true positives, and FP are false positives.

(a) Relevance 1 or higher.

Description Extraction

D	N	TP	FP	Prec	Rec	F1
0	1	59	32	.648	.184	.286
0	3	136	95	.589	.424	.493
0	**6**	**155**	**150**	**.508**	**.483**	**.495**
0	15	167	190	.468	.520	.493
1	1	123	211	.368	.383	.376
1	3	179	602	.229	.558	.325
1	6	210	1107	.159	.654	.256
2	1	122	210	.367	.379	.373
2	3	179	600	.230	.556	.325

(b) Relevance 2.

Description Extraction

D	N	TP	FP	Prec	Rec	F1
0	1	41	59	.451	.210	.287
0	**3**	**82**	**149**	**.355**	**.421**	**.385**
0	6	90	215	.295	.462	.360
0	15	95	262	.266	.487	.344
1	1	82	252	.246	.421	.310
1	3	106	675	.139	.544	.217
1	6	124	1193	.094	.636	.164
2	1	56	227	.198	.287	.234
2	3	88	661	.117	.451	.186

being omitted. Hence, we decided to consider only replacement patterns that exist in the DLMF, i.e., $\mathrm{s}_{\mathrm{DLMF}}(r_f) > 0$.

Since certain subtrees $\tilde{e} \subseteq e \in \mathcal{L}_P$ can be already operator trees, i.e., $\tilde{e} \in \mathcal{L}_C$, we calculate a baseline (`base`) that does not perform any transformations, i.e., $e = \mathrm{t}(e, X)$. The baseline achieves a success rate of 16%. To estimate the impact of our manually defined set of common knowledge facts $\mathcal{K}$, we also evaluated the transformations for $X = \mathcal{K}$ and achieve a success rate of 29% which is already significantly better compared to the baseline. The full pipeline, as described above, achieves a success rate of 48%. Table 5.5 compares the performance. The table shows that depth 1 outperforms depth 0, which intuitively contradicts the F1 scores in Table 5.4a. This underlines the necessity of the dependency graph. We further examine a drop in the success rate for larger N. This is attributable to the fact that $g_f(e)$ is not commutative and large N retrieve too many false positive facts f with high ranks. We reach the best success rate for depth 1 and $\mathrm{N} = 6$. Increasing the depth further only has a marginal impact because, at depth 2, most expressions are already single identifiers, which do not provide significant information for the translation process.

5.2.3.3 Translations from LaTeX to CAS

Mathematica's ability to import TeX expressions will serve as a baseline. While Mathematica does allow to enter a textual context, it does recognize structural information in the expression. For example, the Jacobi polynomial $P_n^{(\alpha,\beta)}(x)$ is correctly imported as `JacobiP[n,\[Alpha],\[Beta],x]` because no other supported function in Mathematica is linked with this presentation. Table 5.6 compares the performance. The methods `base`, `ck`, `full` are the same as in Table 5.5, but now refer to translations to Mathematica, rather than semantic LaTeX. Method `full` uses the optimal setting as shown in Table 5.5. We consider a

Table 5.5: Performance of semantification from LaTeX to semantic LaTeX. **D** refers to the depth (following ingoing dependencies) in the dependency graph, **N** is the maximum number of facts and r_f for the same MOI. The methods `base` refers to no transformations $\mathrm{t}(e, X) = e$, `ck` where $X = \mathcal{K}$, and `full` use the full proposed pipeline. ✔ matches the benchmark entry and ✘ does not match the entry.

Semantic LaTeX Comparison

Method	**D**	**N**	✔	✘
`base`	-	-	.16	.84
`ck`	-	-	.29	.71
`full`	0	3	.36	.64
	0	6	.40	.60
	0	15	.40	.60
	1	3	.43	.57
	1	**6**	**.48**	**.52**
	1	15	.45	.55
	1	20	.44	.56

translation a *match* (✔) if the returned value by Mathematica equals the returned value by the benchmark. The internal process of Mathematica ensures that the translation is normalized.

We observe that without further improvements, LACAST already outperforms Mathematica's internal import function. Activating the general replacement rules further improved performance. Our full context-aware pipeline achieves the best results. The relatively high ratio of invalid translations for `full` is owed to the fact that semantic macros without an appropriate translation to Mathematica result in an error during the translation process. The errors ensure that LACAST only performs translations for semantic LaTeX if a translation is unambiguous and possible for the containing functions [13]. Note that we were not able to appropriately translate 18 expressions (indicated by the human performance in Table 5.6) as discussed before.

5.2.4 Error Analysis & Discussion

In this section, we briefly summarize the main causes of errors in our translation pipeline. A more extensive analysis can be found in Appendix F.3 (available in the electronic supplementary material) and on our demo page at: `https://tpami.wmflabs.org`. In the following, we may refer to specific benchmark entries with the associated ID. Since the benchmark contains randomly picked formulae from the articles, it also contains entries that might not have been properly annotated with math templates or math-tags in the Wikitext. Four entries in the benchmark (28, 43, 78, and 85) were wrongly detected by our engine and contained only parts of the entire formula. In the benchmark, we manually corrected these entries. Aside from the wrong identification, we identified other failure reasons for a translation to semantic LaTeX or CAS. In the following, we discuss the main reasons and possible solutions to avoid them, in order of their impact on translation performance.

Table 5.6: Performance comparison for translating LaTeX to Mathematica. A translation was successful (**ST**) if it was syntactically verified by Mathematica (otherwise: **FT**). ✔ refers to matches with the benchmark and ✘ to mismatches. The methods are explained in Section 5.2.3.3.

LaTeX Translations to Mathematica

Method	**ST**	**FT**	✔	✘
`MM_import`	57 (.60)	38 (.40)	9 (.09)	48 (.51)
LaCASt`_base`	55 (.58)	40 (.42)	11 (.12)	44 (.46)
LaCASt`_ck`	**62 (.65)**	**33 (.35)**	19 (.20)	43 (.45)
LaCASt`_full`	53 (.56)	42 (.44)	**26 (.27)**	**27 (.26)**
`Theory_def`	-	-	+18 (.19)	-18 (.19)
`Theory_ck`	-	-	+3 (.03)	-3 (.03)
Human	95 (1.0)	0 (.00)	77 (.81)	18 (.19)

5.2.4.1 Defining Equations

Recognizing an equation as a definition would have a great impact on performance. As a test, we manually annotated every definition in the benchmark by replacing the equal sign $=$ with the unambiguous notation $:=$ and extended LaCASt to recognize such combination as a definition of the left-hand side[27]. This resulted in 18 more correct translations (e.g., 66, 68, and 75) and increased the performance from .28 to .47. The accuracy for this manual improvement is given as `Theory_def` in Table 5.6.

The dependency graph may provide beneficial information towards a definition recognition system for equations. However, rather than assuming that every equation symbol indicates a definition [214], we propose a more selective approach. Considering one part of an equation (including multi-equations) as an extra MOI would establish additional dependencies in the dependency graph, such as a connection between $x = \mathrm{sn}(u, k)$ and $F(x; k) = u$. A combination with recent advances of definition recognition in NLP [111, 134, 183, 370] may then allow us to detect x as the defining element. The already established dependency between x and $F(x; k) = u$ can finally be used to resolve the substitution. Hence, for future research, we will elaborate on the possibility of integrating existing NLP techniques for definition recognition [111, 134] into our dependency graph concept.

5.2.4.2 Missing Information

Another problem that causes translations to fail is missing facts. For example, the gamma function seems to be considered common knowledge in most articles on OPSF because it is often not specifically declared by name in the context (e.g., 19 or 31). To test the impact of considering the gamma function as common knowledge, we added a rule r_f to $\mathcal{K}$ and attached a low rank to it. The low rank ensures the pattern for the gamma function will be applied late in the list of transformations. This indeed improved performance slightly, enabling a successful translation of three more benchmark entries (`Theory_ck` in Table 5.6). This naive

---

[27] The DLMF did not use this notation, hence LaCASt was not capable of translating $:=$ in the first place.

approach, emphasizes the importance of knowing the domain knowledge for specific articles. In combination with article classifications [320], we could activate different common knowledge sets depending on the specific domain.

5.2.4.3 Non-Matching Replacement Patterns

An issue we would more regularly faced in domains other than OPSF is non-standard notations. As previously mentioned, without definition detection, we would not be able to derive transformation rules if the MOI is not given in a standard notation, such as $p(a, b, n, z)$ for the Jacobi polynomial. This already happens for slight changes that are not covered by the DLMF. For six entries, for instance, we were unable to appropriately replace hypergeometric functions because they used the matrix and array environments in their arguments, while the DLMF (as shown in Table 4.5) only uses `\atop` for the same visualization. Consequently, none of our replacement patterns matched even though we correctly identified the expressions as hypergeometric functions. A possible solution to this kind of minor representational changes might be to add more possible presentational variants m for a semantic macro $\widetilde{m}$. Previously [14], we presented a search engine for MOI that allows searching for common notations for a given textual query. Searching for Jacobi polynomials in arXiv.org shows that different variants of $P_n^{(\alpha,\beta)}(x)$ are highly related or even equivalently used, such as p, H, or R rather than P. There were also a couple of other minor issues we identified during the evaluation, such as synonyms for function names, derivative notations, or non-existent translations for semantic macros. This is also one of the reasons why our semantic LaTeX test performed better than the translations to Mathematica. We provide more information on these cases on our demo page.

Implementing the aforementioned improvements will increase the score from $.26$ (26 out of 95) to $.495$ (47 out of 95) for translations from LaTeX to Mathematica. We achieved these results based on several heuristics, such as the primary identifier rules or the general replacement patterns, which indicates that we may improve results even further with ML algorithms. However, a missing properly annotated dataset and no appropriate error functions made it difficult to achieve promising results with ML on mathematical translation tasks in the past [1, 15]. Our translation pipeline based on LaCASt paves the way towards a baseline that can be used to train ML models in the future. Hence, we will focus on a hybrid approach of rule-based translations via LaCASt on the one hand, and ML-based information extraction on the other hand, to further push the limits of our translation pipeline.

5.2.5 Conclude Qualitative Evaluations on Wikipedia

We presented LaCASt, the first context-sensitive translation pipeline for mathematical expressions to the syntax of two major Computer Algebra Systems (CAS), Maple and Mathematica. We demonstrated that the information we need to translate is given as noun phrases in the textual context surrounding a mathematical formula and common knowledge databases that define notation conventions. We successfully extracted the crucial noun phrases via part-of-speech tagging. Further, we have shown that CAS can automatically verify the translated expressions by performing symbolic and numeric computations. In an evaluation with 104 Wikipedia articles in the domain of orthogonal polynomials and special functions, we verified 358 formulae using our approach. We identified one malicious edit with this technique, which was reverted by the community three days later. We have shown that LaCASt correctly translates about 27% of mathematical formulae compared to 9% with existing approaches and a 81% human baseline.

Further, we demonstrated a potential successful translation rate of 46% if LᴬCᴀST can identify definitions correctly and 49% with a more comprehensive common knowledge database.

Our translation pipeline has several practical applications for a knowledge database like Wikipedia, such as improving the readability [17] and user experience [150], enabling entity linking for mathematics [320, 17], or allowing for automatic quality checks via CAS [2, 11]. In turn, we plan to integrate [401] our evaluation engine into the existing ORES system to classify changes in complex mathematical equations as potentially damaging or good faith. In addition, the system provides access to different semantic formats of a formula, such as multiple CAS syntaxes and semantic LaTeX [260]. As shown in the DLMF [260], the semantic encoding of a formula can improve search results for mathematical expressions significantly. Hence, we also plan to add the semantic information from our mathematical dependency graph to Wikipedia's math formulae to improve search results [17].

In future work, we aim to mitigate the issues outlined in Section 5.2.4, primarily focusing our efforts on definition recognitions for mathematical equations. Advances on this matter will enable the support for translations beyond OPSF. In particular, we plan to analyze the effectiveness of associating equations with their nearby context classification [111, 134, 183, 370], assuming a defining equation is usually embedded in a definition context. Apart from expanding the support beyond OPSF, we further focus on improving the verification accuracy of the symbolic and numeric evaluation pipeline. In contrast to the evaluations on the DLMF, our evaluation pipeline currently disregards constraints in Wikipedia. While most constraints in the DLMF directly annotate specific equations, Wikipedia contains constraints in the surrounding context of the formula. We plan to identify constraints with new pattern matches and distance metrics, by assuming that constraints are often short equations (and relations) or set definitions and appear shortly after or before the formula they are applied to. While we made math in Wikipedia computable, the encyclopedia does not take advantage of this new feature yet. In future work, we will develop an AI [401] (as an extension to the existing ORES system) that makes use of this novel capability.