Generating Expressions in MMT

Christin Cerny, 21996009

Supervisor: Florian Rabe

10.12.2020

Abstract

Mathematical expressions are an important aspect in many fields of science, engineering and teaching. As many applications can require a large number of expressions or expressions can be of high complexity an automated approach to generate those expressions is desirable. Such approaches are however often application specific. This paper proposes a multi purpose generator and explores how expressions can be defined as interesting for an application, how that interestingness could be formalized and how this can be translated into criteria a generator can utilize to achieve a greater degree of application and grammar independence, as well as a possible early implementation in the MMT environment.

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ort, Datum

1 Introduction

Expressions are the foundation of many fields, from engineering over physics to mathematics and logic. At their core, they are syntactic entities, consisting of variables, constants, literals and function symbols, which can be evaluated and convey a meaning within a context. For example, $a^2 + b^2 = c^2$, the Pythagoran equation, consists of the variables a, b and c, the literal 2 in three positions and the function symbols for addition and equality. It is a fundamental relation within the context of Euclidean geometry, stating that in a triangle the sum of the squares of the cathetes is equal to the square of the hypotenuse.

Expressions like this can be handcrafted. However, the quickly increasing requirements of applications to the number and complexity of expressions makes the manual construction of those tedious and long. As computers are good at quickly processing a large number of operations, the computer based automatisation has become an established practice, with programms like PASSAROLA [Alm+13] for exercise generation in mathematical education or QuickSpec [SMA+17] to conduct theory exploration in mathematical research having become a common sight. It is clear that, most of the time, implemented generators are purpose driven and application specific, as the context of an application defines the usefulness and interestingness of an expression.

The main contribution of this thesis will be an attempt to clarify the term interestingness in context of expressions in regard to applications, the formalization and categorization of expression properties as well as generation criteria and methods in an attempt to build a expression generation system with increased application independence. For this purpose, any discussion as well as the implementation will use the MMT system as a basis, as it is foundation-independent and allows for the usage of any kind of grammar.

Chapter 2 will first introduce and explain important aspects of the MMT system as our system of choice, as well as shortly discuss existing generators and their applications. In chapter 3 we will examine several selected applications in order to find possible properties that can function as criteria for expression generation. These properties will then be further analyzed in chapter 4 to determine which of them are useful and implementable as actual criteria in a generation algorithm. In chapter 5 we will explain several design decisions that where made during developement, and in chapter 6 explain the impact of the criteria on generation and show a possible implementation for MMT using Scala at the end.

2 Preliminaries

2.1 MMT

For this thesis we will utilize MMT [Rabb], a formal generic module system for mathematical knowledge, as the framework for the implementation of the generation algorithm [RK12]. MMT has the advantage of being foundation independent, which allows us a grammar and application independent approach to the generation of expressions [Rab16]. For this thesis we will however utilize a predefined logical framework LF and a predefined logic SFOL, which we will justify later in the design decisions. The most important aspect of MMT for us is therefore the concept of a theory.

Theories are formalized structures used to define a formal language. They take the form of ordered lists, containing uniquely named typed declarations. Theories can also contain inclusions of other theories. The following definition of a theory has been adapted from [RM19]

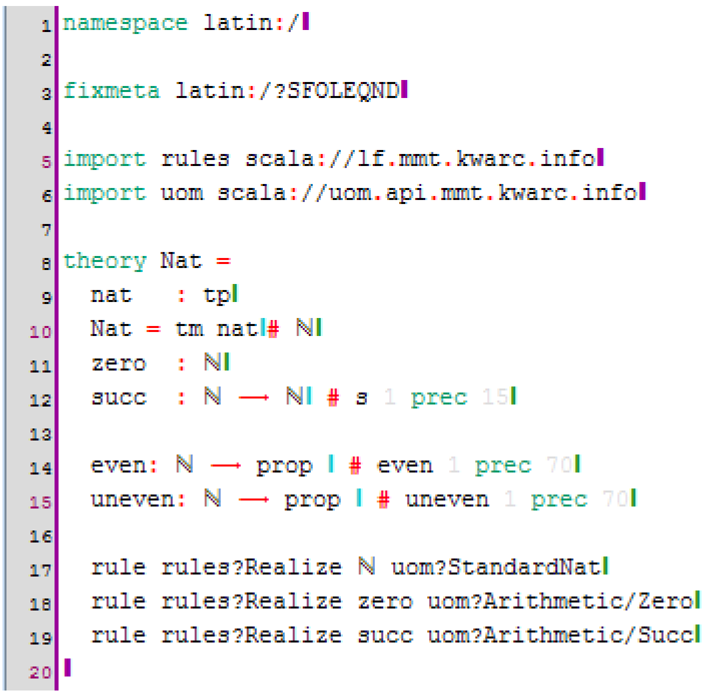
Definition 1. *The grammar for theories and expressions is*

TDec ::== T = (Dec,...,Dec)	*theory declaration*
Dec ::== c: E[=E]|include T	*constant or include declaration*
E ::== c|...	*expression built from constants*

Especially of interest for our purposes are constant declarations for variables, literals, function symbols and quantifiers, as defined in [Raba].

Definition 2. *The grammar for constants is*
$c ::= OMID(c)|OMV(x)|OMLIT(E,value)|OMA(c,E^*)|OMBIND(c,Dec^*,E)$

Declarations in a theory can include types, function and predicate symbols, as well as axioms and theorems [Raba]. For the generation of expressions, only types and function/predicate symbols are of importance. For further elaboration, consider a theory Nat for natural numbers.



```
namespace latin:/❚

fixmeta latin:/?SFOLEQND❚

import rules scala://lf.mmt.kwarc.info❚
import uom scala://uom.api.mmt.kwarc.info❚

theory Nat =
  nat   : tp❙
  Nat = tm nat❘# ℕ❙
  zero  : ℕ❙
  succ  : ℕ ⟶ ℕ❘ # s 1 prec 15❙

  even: ℕ ⟶ prop ❘ # even 1 prec 70❙
  uneven: ℕ ⟶ prop ❘ # uneven 1 prec 70❙

  rule rules?Realize ℕ uom?StandardNat❙
  rule rules?Realize zero uom?Arithmetic/Zero❙
  rule rules?Realize succ uom?Arithmetic/Succ❙
❚
```

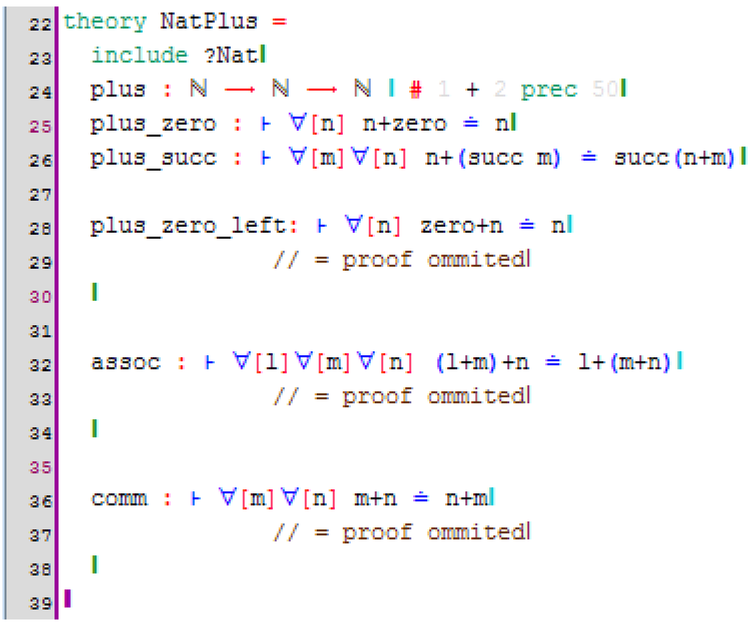
First a type nat is declared, which is of the type type. This declaration only states that there is a type nat, from which we can declare a typed term Nat, here named ℕ. Any natural number is now treated as a typed MMT term, allowing for the declaration of function symbols. Function symbols can have any number of inputs and have always one output.

Due to this, the declaration of 0 and the successor function can be achieved by declaring a zero as a typed term ℕ, and then declaring a successor function succ that takes a ℕ and returns a new ℕ. From MMT perspective, both zero and succ are now function symbols, with zero having no inputs and succ having exactly one input.

For the purpose of this thesis, we additionally defined two predicate symbols, even and uneven. Both take a typed term ℕ and return a proposition, allowing us to build atomic formulas. The prop type is provided by a prebuild sorted first order logic (SFOL), which will also be used for the generator.

Realization rules can then be imported to use standardized implementations of known types and functions. While expressions can be build without realization rules, some aspects like literals, which require a realized type, and later computation requires such rules.

With this theory as a basis, we can now of course expand it and include for example addition.



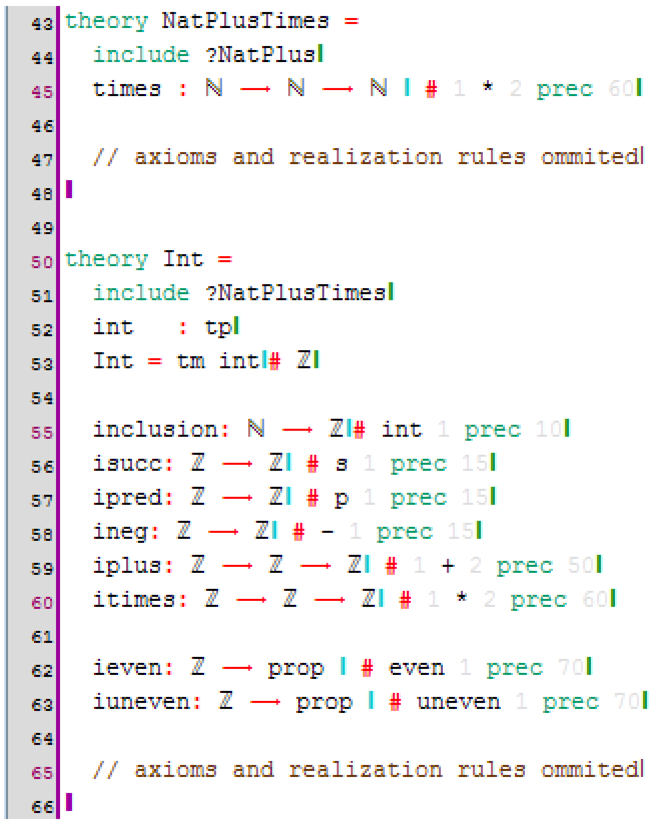
```
theory NatPlus =
  include ?Nat❙
  plus : ℕ ⟶ ℕ ⟶ ℕ ❘ # 1 + 2 prec 50❙
  plus_zero : ⊦ ∀[n] n+zero ≐ n❙
  plus_succ : ⊦ ∀[m]∀[n] n+(succ m) ≐ succ(n+m)❙

  plus_zero_left: ⊦ ∀[n] zero+n ≐ n❘
                 // = proof ommited❘
  ❙

  assoc : ⊦ ∀[l]∀[m]∀[n] (l+m)+n ≐ l+(m+n)❘
                 // = proof ommited❘
  ❙

  comm : ⊦ ∀[m]∀[n] m+n ≐ n+m❘
                 // = proof ommited❘
  ❙
❚
```

Here, we did it by declaring a new theory which includes the theory of natural numbers, meaning that the theory NatPlus will include all declarations of Nat. Of course, the theory can be flattened into a single theory.

We declare a new function symbol +, a binary operator that takes two ℕ to make a new ℕ. The naming of it as ”1 + 2” means that the first and second ℕ are taken and connected through the + symbol. This allows the MMT presenter module to show the generated expression in a more natural form and output for example $x_1 + x_2$ instead of $plus(x_1)(x_2)$.

In this theory we also declared several axioms regarding the addition. Axioms are declared as proofs, as indicated by ⊢, of a proposition. The proofs those axioms, as well as axioms and realization rules in NatPlusTimes and Int are ommited as they are not of further importance for the purpose of this thesis.

The inclusion of the multiplication operator in a theory NatPlusTimes requires no additional explanation. A theory for integer numbers ℤ however has a minor inconvience. Integer numbers include natural numbers, meaning we want to include the theory for natural numbers.



```
theory NatPlusTimes =
  include ?NatPlus❙
  times : ℕ ⟶ ℕ ⟶ ℕ ❘ # 1 * 2 prec 60❙

  // axioms and realization rules ommited❙
❚


theory Int =
  include ?NatPlusTimes❙
  int    : tp❙
  Int = tm int❘# ℤ❙

  inclusion: ℕ ⟶ ℤ❘# int 1 prec 10❙
  isucc: ℤ ⟶ ℤ❘ # s 1 prec 15❙
  ipred: ℤ ⟶ ℤ❘ # p 1 prec 15❙
  ineg: ℤ ⟶ ℤ❘ # - 1 prec 15❙
  iplus: ℤ ⟶ ℤ ⟶ ℤ❘ # 1 + 2 prec 50❙
  itimes: ℤ ⟶ ℤ ⟶ ℤ❘ # 1 * 2 prec 60❙

  ieven: ℤ ⟶ prop ❘ # even 1 prec 70❙
  iuneven: ℤ ⟶ prop ❘ # uneven 1 prec 70❙

  // axioms and realization rules ommited❙
❚
```

However, a current limitation of the MMT system makes it necessary to newly declare operator symbols that are used by both $\mathbb{Z}$ and $\mathbb{N}$, meaning we now have two versions of the same operation - one for each number type. We can enable natural numbers to use the integer operators by defining a function inclusion that produces an integer number from a natural number. The duplication problem and how to potentially deal with it is something we have to keep in mind when implementing the generator, however.

It is important to note that within the MMT system the word "term" has a slightly different meaning then in mathematical logic. In logic, term denotes simply a mathematical object, for example the $x_1 + x_2$, while formula denotes a mathematical proposition like $x_1 + x_2 = x_3$. In MMT, term denotes any syntactic construction using variables or function symbols. To avoid confusion, in this thesis the word **expression** will be used in cases when term is used in the context of MMT, and term and formula will be used to denote the usage in mathematical logic.

The MMT system supports implementation of a potential expression generator on four levels [Rab16], each with a number of advantages and disadvantages.

Namely, these levels are, from the most abstract to the most specific, the MMT system level, the implementation of a logical framework, the implementation of a logic and a domain theory. In general, the more abstract the level of implementation is, the less function symbols are predetermined and can to be imported. This obviously increases the freedom of generation at the price of more difficult ensurance of generation integrity. Following is an overview of the levels.

The **MMT level** is the most abstract implementation level in the system. Here everything from the theory over the logic to the logical framework has to be imported, meaning there are no predetermined function symbols. While this gives a potential generator the advantage of the widest range of possibilities for term generation, as no restrictions exist and everything can be user defined, this lack of restrictions also makes it hard to ensure the generation of sensical terms. All operators, logical and theory dependend, are seen as on the same level, a distinction is hardly enforcable. This can lead to the creation of random and illegal terms.
On the **logical framework level** a logical framework, like LF [HHP01] for example, is utilized. While this sacrifices some freedom it also ensures the generation of legit terms, but still has less control over the term form then the implementation on logic level.
On **logic level** we implement a generator for a specific logic, like predicate logic or SFOL. The advantages of LF level apply - freedom of generation is sacrificed for better controlled term forms, except that we now can assume logical operators.
Domain level is the most specific implementation level and takes advantage on generation by using a specific theory, for example the natural numbers, and generate terms for that specific theory only. This allows for hard coding of the entire generation process, as all function symbols are predetermined. While this ensures that all terms are legal and is the easiest to implement, as no imports are required and all function symbols are known, it also means that every new theory requires its own generator.

2.2 Related Work

The automated generation of expressions has become a well established pratice to support applications in computer algebra. For example, on the front of computer-assisted mathematics one of the largest applications are theory exploration systems, a term coined by Buchberger as an expansion to automated theory proving in the Theorema project [Buc00]. In systems like this, generation algorithms are used to build terms and conjectures in order to find lemmas and definitions in mathematical theories. There is a large number of systems available. Aside of the allready mentioned Theorema, there are also for example Hipster [Joh+14] [Joh17], which utilizes QuickSpec [SMA+17] for conjecture generation, or IsaCoSy [JDB11] for Isabelle.

However, other fields also get increasing attention. For mathematical education proposals were made to introduce generation system to build mathematical exercises. Tomás [TL03] suggested the application of Constraint Logic Programming and the description of exercises in grammars, with an emphasis on the solving algorithms. Milani [MHP18] later proposed a rule-based generation of mathematical expressions using a new context-free like grammar, in which application rules for function symbols are codified, that can be used to construct templates from which expressions can be generated.

Further, a possible implementation for the generation of test cases for SAT-solvers was presented by Klimek [KGK19], constructing logical formulae with randomized structure, using parameters to customize syntactical properties of generated formulas.

Expression generation is not restricted to mathematical and logical expressions, however. XtraGen [Ste02] for example uses conditions, parameters and constraints for real time generation of natural language with application specific orientation in mind. YAG [MCA02] tries a similiar approach, but with a general purpose in mind.

In general, most of those generators are restricted and implemented with a specific application in mind or concentrate on a specific grammar to build more or less specialized expressions. Constraints and parameters are mainly used to further application specific goals and enable customization options for the generation. We believe that those constraints and parameters can be used to implement a generator system that allows for greater application and grammar independence.

3 Applications and Interestingness

The generation of expressions can not be entirely random. Aside from obvious and most basic restriction, like ensuring that all generated expressions are well-formed within the language from which they are built, even a preliminary glance at possible applications that can benefit from a generator reveal differences as to when a term is useful, or interesting, for an application. Consider, for example, the generation of a term for a mathematical exercise compared to the generation of a test case for a SAT solving algorithm. It becomes immediately clear that expressions interesting for one application can have quite different properties when compared to an expression of interest for the other application.

The interestingness of an expression is therefore defined by its properties, structural and computational, and has to be considered during the generation process. In order to do so, the possible applications for which we might want to generate expressions for have to be examined and the properties collected to find possible critera that can be applied to the process. For this thesis, five potential applications have been selected and the expressions they require or yield examined.

3.1 Theory Exploration

Theory exploration, as introduced by Buchberger [Buc00], is a process in which a mathematical theory, a collection of formulae in some logical formal language [Buc04], is explored to find rules and definitions to build up the mathematical knowledge base. Since its introduction many proof assistants, like Theorema and IsaCoSy, and fully automated systems like Hipster were developed.

Hipster [Joh17], using QuickSpec2 [SMA+17] for conjecture generation, follows for example roughly the following procedure. The system generates and tests terms and conjectures to find definitions and lemmas. Conjectures that are tested positive are then proven, if possible. Conjectures are generated by first building terms and categorizing them through testing in equivalence classes, from which the conjectures are then constructed.
Consider within this approach, for example, the theory of natural numbers as shown in chapter 2. A possible goal could be the discovery of the lemma of associativity for addition. If this approach is used, first the terms would have to be generated.

Term 1: $x_1 + (x_2 + x_3)$
Term 2: $(x_1 + x_2) + x_3$

After testing, those terms would then be in the same equivalence class, and the conjecture would be built.

Example Conjecture: $x_1 + (x_2 + x_3) = (x_1 + x_2) + x_3$

This conjecture, through further testing and finally proving, would yield the lemma for associativity.

Example: Additive associativity of natural numbers
$\forall x_1 \in \mathbb{N}. \forall x_2 \in \mathbb{N}. \forall x_3 \in \mathbb{N}. x_1 + (x_2 + x_3) = (x_1 + x_2) + x_3$

Of course, other approaches could aim to directly generate conjectures or lemmas. A general purpose generator naturally would have to be able to produce any expression within those steps, mathematical terms and formulae.

With all this considered, we can now find a number of properties that can define the usefulness and interestingness of expressions.

- The terms of many mathematical lemma have a very limited number of variables, making a limitation of the variable pool during generation desireable.
- The syntactic depth of such expressions is often quite limited.
- To ensure a sufficient search of different syntactic depths, an escalation mechanism for generation could be considered.
- In formulae, limiting the alterations between quantifiers might be of interest, as well as enabling and disabling them entirely.
- Satisfiability can be considered, as we might want to sort out conjectures that are obviously wrong. [JDB11]

3.2 Mathematical Exercises

In educational environments exercises and examples are generally used to help students understand the methodology and algorithms required to solve mathematical problems. These problems generally come in very specific forms to allow students to concentrate on a limited number of methods to build up confidence and skill in using those. Ideally, a student should have access to a large number of exercises as to prevent simple memorization of solutions of problems [TL03], which often sets in by repeating a limited number of available ecercises. Handcrafting a large number of exercises is, however, time consuming as exercises can have a large number of constraints to fulfill to be considered useful.

Consider, for example, curve sketching, in which often a polynom or a rational function is examined to determine a number of properties its graph has, e.g. symmetries, extrema and roots to name a few. Exercises for such problems are often representable in a standard form, where f(x) is a polynom in the form of f(x) = $\sum(c_i * x^i)$, where c_i is either a literal, a constant or a parameter

or a rational function of the form

f(x) = $\frac{g(x)}{h(x)}$, g(x), h(x) are polynoms.

Quite a number of properties can apply on such an expression.

- The polynomial nature, as well as the restriction of the power function to often quadratic or cubic functions limits again the number of variables required in generation and defines the form.
- The rather strict form of the structure also allows for the consideration of a template based generation system.
- Depending on the grade or educational level of the student, restrictions to the availability of function operators such as exponential or logarithmic functions apply.
- Any generated exercise has to be solvable by both the student and the program that offers the exercise. [TL03]
- Often, it also has to be solvable in a reasonable number of steps, where reasonable is a debatable term and depends on the educational level of the student in question. [TL03]
- Special conditions such as the restriction of partial solutions to rational numbers to increase student comfort while solving a problem may apply. [TL03]

3.3 Test Cases

For many problems automated solving algorithms are developed in order to determine solvability, satisfiability and equaly complicated properties of expressions. Especially the satisfiability problem has recieved a lot of attention and SAT solvers are becoming ever more powerful [Bie+09]. Determining such properties is not easy. The satisfiability problem, for example, is the problem of determining if there exists an interpretation that satisfies a given Boolean formula and is NP-complete as shown by Cook. Problem instances are often in conjunctive normal form (CNF), as every propositional formula is, according to the laws of Boolean algebra, representable in CNF. In order to test those solvers, new test cases have to be created, which is a very time consuming process. Modern problem instances consist of on average 50,000 variables in a logical formula, while large problems can have 1,000,000 variables or more [KGK19], making the automization of this process very beneficial.

Example for an expression in CNF:
$X_1 \wedge (X_1 \vee \neg X_2 \vee X_3) \wedge (X_2 \vee X_3)$

- The syntactic depth is very high because of the large number of variables and clauses.
- Potential for application of template based generation, as forms like CNF are highly predictable.
- Very limited number of function symbols, CNF contains for example only conjunctions and disjunctions.
- The satifiability of a generated formula is important as the tested property has to be known beforehand with test cases

3.4 Machine Learning

Machine learning, as pioniered by Arthur Samuel [Sam59], is a part of AI studies in which computer algorithms improve through experience [MRT12]. A computer is given a specific task, for example parsing mathematical expressions provided in string form into a term in MMT format, and the computer has to develop an algorithm to achieve that task on its own. In order for the machine to learn such a process, a large number, ranging in the millions to billions, of highly diverse expressions has to be provided such that the computer has a sufficient set to experiment with. A sufficent amount of those expressions has to exist in both forms, so that the computer has comparison objects to verify its solution. This makes automated generation of such expressions imperative.

In the given example the computer would have to correctly identify function symbols in string form, as well as their structure and create an MMT term with that information. Consider for example the following two very simililar but unequal expressions in the context of natural numbers.

a) $x_1 * x_2 + 3$
b) $x_1 + x_2 * 3$

The computer now has to recognize that

- x_1 and x_2 are variables.
- 3 is a literal of the type of natural numbers.
- There are two different function symbols: $+$ and $*$.
- Those function symbols are binary
- The function symbol $*$ has a higher precedence then the function symbol $+$, meaning that $+$ is in both cases the top level function symbol

Having learned that, the computer then has to generate an algorithm that returns the correct MMT terms

a) OMA(+, {OMA($*$, {OMV(x_1), OMV(x_2)}), OMLIT(3, nat)})
b) OMA(+, {OMV(x_1), OMA($*$, { OMV(x_2), OMLIT(3, nat)})})

In order to gain all of these informations and to test the computer generated algorithms for this problem, expressions have to be of a wide variety of complexities, ranging from very easy expressions like a single variable or literal, to very deep expressions with many function symbol alterations and both variables and literals to provide a sufficiently diverse set to analyse.

3.5 Autocomplete

Autocompletion is a function offered by a wide range of applications, from spreadsheets like Microsoft Excel [Cor] to search engines like Google [LLC] and more. The goal is to offer possible syntactic constructs a user might be interested to use for their current purpose.

Since programs like this are commercial, the inner workings of the involved generator systems that supply the application with the expressions it requires are not readily available. However, some assumption can be made based on observation.

Offered expressions, whether they are mathematical or lingual, are often context sensitive, meaning that the application often uses some heuristics to generate expressions that are likely useful for the user. This can include, for example, previously used variables and function symbols, previous searches, so called trending items, always within the context of the application itself.
Further, offered expressions are often generated starting from simpler expressions which slowly escalate towards higher complexities.

4 Properties and Criteria

Through the exploration of possible applications we gained a number of properties that can make a term interesting in different situations. Properties and their derived criteria can be classified in two categories - semantic and structural. Not all properties and their potential criteria are suitable for or enforcable during the generation process. In order to learn which properties can yield applicable criteria they have to be examined, which we will do in this chapter. Additionally, we will explain shortly the concept of templates as a special form of applied criteria, and how they can be used in a generator in the MMT environment.

4.1 Semantic Criteria

Semantic properties and criteria cover conceptual questions regarding a term and are generally of binary nature, e.g. whether or not a term is satisfiable or not, or observe abstract questions like if the solution of a term can be considered odd. As we will see are semantic properties mostly not suitable for a generator.

Satisfiability was one semantic property of interest for test cases for sat solvers. A formula is satisfiable if there exists an interpretation that makes the formula true. [Ben12]

Definition 3. *Let A be a structure, ψ a formula, and a an collection of elements from the structure that satisfy ψ. Then it holds, that*
$A \models \psi[a]$
$A \models \psi$*, if* ψ *has no free variables*

Satisfiability is not a trivial matter and hard to ensure during formula generation. One exemplary way to enforce satisfiability during generation of a formula in CNF could involve a check during the construction of the clauses - each variable used as well as their negated variation is counter tested with the already existing clause and thrown out and rerolled if the resulting clause becomes unsatisfiable. This, however, requires testing in almost every generation step, increasing the time until a formula is build exponentialy. Considering the sheer amount of variables and clauses in average and larger cases, satisfiability is unsuitable as a generation criterion.

Solvability was identified as potentially interesting in the case of mathematical exercises. Solvability, or the decision problem, in general asks yes or no questions about mathematical expressions. In the context of mathematical exercises for example we could be interested whether or not a mathematical term can be computed at all, or if a partial or end solution to a problem is a rational number. Questions like those can only be answered for an existing expression, through testing and solving of said expression. Just like satisfiability, that makes this unsuitable for a generation criteria.

4.2 Structural Criteria

Structural criteria consider properties pertaining the composition and structure of a term. Such criteria often come in the form of restrictions regarding compositional complexity and restrictions regarding available components for terms.

The number of variables available for generation can be defined as a set of variables a generated term can contain.

Definition 4. *Let n be a natural number. Then*
$V := \{OMV(x_i)|1 \leq i \leq n\}$
is the set of all variables a term can potentially contain.

Equally, the **number of literals** available for the generation process can be defined.

Definition 5. *Let n be a natural number, rt the realized type of a data type contained in a theory T and v a value for the literal. Then*
$L := \{OMLIT(v_i, rt)|1 \leq i \leq n\, rt \in T\}$
is the set of all literals a term can potentially contain.

It is easy to observe that limiting the number of available variables is desireable in most cases, and that this limitation is dependend on the purpose of the generation. Consider the examples of generation for theory exploration, which often only requires a single digit number of variables, and the generation of SAT-solver test cases, which can contain thousands of variables. Considering this, allowing to customise this set of available variables to the specific needs of the application is a basic necessity and quite easy to ensure as variables have to be instantiated anyway.

Limiting the number of available literals is at first not as obvious, though this is less about limitation and more about saving specific literals and their values. Numerical literals can be generated on demand, but either iterate with each generation, or are generated on random. Instantiating a number of literals during initialization of the generator allows us to consistently apply literals from a set pool, which can be preferable.

The syntactic depth of a term, as we have discovered in all example application, is another almost universal criteria to apply. Any term can be depicted as a concrete syntax tree, a hierachical representation of the syntactic structure of a term. Such a tree is normally rooted in an n-ary top level function, with the inner vertexes also being function symbols and the leafs being generally variables or literals.

The longest path from the root to a leaf is the syntactic depth of a term, with a leaf itself having a syntactic depth of 0.

Definition 6. *The syntactic depth D(c) is defined as*
$D(OMVAR(x)) := 0$
$D(OMLIT(v, rt)) := 0$
$D(OMA(c, E^*)) := max(D(E_1), ..., D(E_n)) + 1$ *iff* E^* *non-empty, 0 else*
$D(OMBIND(C, Dec*, E)) := D(E) + 1$

More often then not we might be interested in limiting the maximum depth of a term. In the example of theory exploration, many definitions and lemmas contain terms of a rather low syntactic depth, making a limitation or control of depth escalation of generated terms beneficial. On the other end of the spectrum in the example of SAT-solver test cases, we can be interested in limiting both minimal and maximal depth of clauses, for example. Consider the average SAT-problem, containing 50,000 variables and an undetermined but multiplicative higher number of clauses. In such a case, we might want to ensure that several throusand clauses are contained in a formula, each consisting of several thousand variables, to ensure a certain minimal level of test case complexity.

The syntactic depth can easily be saved and counted up during the generation process, and limits of this kind are easily enforcable, making this a definite choice for a criteria.

Data type restrictions concern the output type of a function symbol, the definition of which has already been been clarified in chapter 2. In theories with several data types we might be interested in concentrating generation on a specific type. Consider our theory in integer numbers which has natural numbers included. Since we have duplications of function symbols that are technically the same but implemented for different data types, we now might be interested to limit our generation to integer numbers in order to test our theory implementation.

This option has been implemented since we can easily check the output type of any generated term and function symbol, filtering both as needed.

Function symbol restrictions can apply if a theory contains function symbols that, if applied during generation, aren't expedient towards the goals of the application. This restriction can be defined as a set of function symbols, where we

Definition 7. *Let O be the set of all function symbols in a theory, and N be a set of function symbols to be excluded. Then*
$A := O/N$
is the set of for generation available function symbols.

Excluding specific function symbols can be very useful. Consider the theory of natural numbers from chapter 3. It includes definitions of zero and the successor, which are handled by the MMT system as function symbols. If we now, for example, wanted to generate terms using variables and instantiated standard literals the usage of the definitional zero and successor function might be considered redundent and could hence be excluded from the generation process. Another example would be the generation specific forms of formulas - like horn formulas which, as seen in theory exploration, contain only forall and no other quantifiers and are limited to conjunction and implication as logical function symbols.

Considering this, the implementation of such a restriction can also be rather easy as we simply have to create a new set from two existing sets, which in most programming languages can be achieved by using lists. As such, this restriction has been chosen for implementation purposes.

Function symbol alterations is the number of changes in function symbols in a term from level to level. An alteration exists if at least one top level function symbol of the involved sub expressions is different from the function symbol in the currently examined level.

Definition 8. *The number of function symbol variations FA(E) of an expression E is defined as*
$FA(OMVAR(x)) = 0$
$FA(OMLIT(v, rt)) = 0$
$FA(OMA(c, E^*)) = 0$ *if E is empty, else*
$FA(OMA(c, E^*)) = 1 + \sum FA(E_i)$ *if* $\exists E_i : E_i \neq OMA(c, E_i^*)$,
else $\sum FV(E_i) + 1$

Function symbol alterations can potentially be of interest in several cases. In the example of machine learning we noted that we might be interested in generating complex terms containing many different function symbols and alterations between them to ensure a computer learn how to handle the parsing of differently weighted symbols. In such a case, a minimal number of function symbol alterations could ensure a sufficient complexity for the machine to work with. However, even in this case we want both terms with many and little alterations as to provide the machine with different difficulty levels to work with and learn from. Considering the huge number of expressions generated for this example, even with random generation a sufficient number of both simple and complex terms should be generated without appliance of an alteration criterion.

Special forms of formulae can also contain alteration limits. Horn formulae, for example, are very limited in the number of alterations. Ignoring potential alterations in the terms the formula consists of, a Horn formula has a maximum of two alterations, one alteration when going from the conjunctions to the implication, and one alteration when going to the forall quantifiers. This however can potentially be ensured by using a template.

One general case where alteration limitations are useful are applied quantifiers to formulas. We might be interested in limiting the alterations within an quantifier block to avoid constant jumps between $\forall$, $\exists$ and $\exists!$.

Depending on the chosen generation method the number of alterations is not particularly easy to enforce. This is especially noticable in the case of backward generation, a process in which an expression is build from top to bottom. Due to this and the limited actual usefulness the alteration number has been discarded as a criteria except in the case of quantifer alterations, in which case alteration handling is easier to enforce.

The number of free variables is the number of variables within a formula that are not bound by quantifiers. The free variables of an expression can be defined as a set of variables of an expression that are not bound by a quantifier, and the number of free variables as the cardinality of that set.

Definition 9. *The set of free variables FV(E) of an expression E*
$FV(OMVAR(x)) = x$
$FV(OMLIT(v, rt)) = \emptyset$
$FV(OMA(c, E^*)) = \bigcup FV(E_i)$ *if E nonemtpy, else* $\emptyset$
$FV(OMBIND(c, Dec^*, E)) = FV(E) \backslash \{Dec\}$

The number of free variables is
$nFV(E) = |FV(E)|$

The specification of the number of free variables can be very useful to increase the chance of generation of special formulas. A horn formula, for example, has no free variables. If now especially horn formulas have to be generated, setting this number too zero in combination with restrictions to quantifiers can ensure this.

Probabilistic criteria is a collective term for any application of a function symbol by chance. In general, with the exception of templates, function symbols are picked from the set of available function symbols at random. However, in some cases additional probabilistic influence on the generation process might be desirable. For example, during the generation of mathematical atomic formulae the user might wish to specify a desired ratio of applied predicate symbols to equality, or the user might want to specify a desired ratio of variables to literals.

Using such a ratio directly is overtly complicated, as every application of a function symbol and every use of a variable and literal would have to be tracked and saved. Instead, probabilities can be used to increase the chance of the generation of an expression with the desired ratio. Expressions that fall outside of a user or application given range of deviation can then be filtered out.

This approach has been considered and chosen for ratios of atomic formulas containing a predicate symbol to equality, variables to literals in terms, quantified to unquantified formulas.

Positional information of function symbols can be important when building special forms of expressions or when a template is to be used. Essentially, information about the exact function symbol on an exact position within an expression is given, and that function symbol is then applied on that position. While there are possible approaches on how to achieve this with the use of lists, flags and more, this criterion has been dropped rather early in developement as templates have an easier alternative implementation, special expression forms can often be expressed as templates and this information is therefore rarely if ever used outside of templates.

4.3 On templates

Templates can be seen as a form of applied criteria and are used in many of the earlier noted generators. They provide exact informations about positions of function symbols, variables and literals. As an example, consider the linear function $f(x) = mx + t$, the general function that describes lines in two dimensional geometry. By substituting m and t with numerical literals we gain specific instances. Since MMT supports substitutions templates are easy to realize. An MMT expression can be given, e.g.

OMA(+, {OMA{*, {OMV(m), OMV(x)}},OMV(t)})

for the linear function. With the help of a substitution list that defines which variable has to be substituted and what type of substitute, e.g. substitution with literals, should be used, we then can very easily achieve randomized substitution within a template. We are not even restricted to substitution with literals and variables but can use entire randomly generated expressions.

Templates are not restricted to such an absolute form, however, and can contain continuous parts. Consider, for example, this general CNF formula

$\wedge_{min}^{max} \vee_{min}^{max} x$

where a number of disjunctions are applied on propositional variables to build clauses, which are then connected with conjunctions. The conjunctions and disjunctions can be seen as continuous parts of this formula, which can be restricted through the appliance of minima and maxima, while the variable x is an absolute template part and can be used for substitution. If for the substitution another template is used, it can allow for a rather flexible tool to build many different kinds of expressions and special forms.

This approach was followed and implemented with relative success, albeit with a few unsolved problems. The implementation of the template system can be seen in chapter 6.

5 Design Decisions

Aside from identifying the criteria we use for the generator, several decisions regarding the features of the generator, as well as specific handling of some criteria, had to be made during developement. These decisions will be discussed in this chapter.

General purpose vs application dependance The first decision that had to be made was whether the generator should concentrate on the generation of terms for a specific application, or if an attempt should be made to forgo application dependance and go for a general purpose generator. Tailoring a generator for a specific application has a number of advantages which can be summed up as streamlining of the generation process. In application specific generation possible criteria of other applications can, naturally, be ignored, limiting the number of criteria. This can be amplified if specific term forms are required, like the CNF which is often used for SAT-solver test cases. The generation of those can be hard coded, eliminating the need for criteria further. This additionally increases generation efficiency as less checks to ensure the criteria have to be run during generation.

However, a general purpose generator which is usable as a plug-and-generate system has its merrits. Such a generator could be used for testing purposes of applications while the specific generator is still in developement, or where an available if less efficent generator is preferable to complete new development. Additionally, since the developement of such a generator requires the crossexamination of several applications, the continued work on such a system can provide additional insights into potential criteria for term generation. This is especially interesting for machine learning, where an algorithm could take the terms provided by a general purpose generator and identify patterns that could yield new criteria for known applications, potentially helping with the developement of application specific systems. For those reasons, general purpose generation was ultimately chosen.

Level of Generation As explained in the chapter about the MMT system, term generation can be implemented on several levels of the MMT system. The advantages and disadvantes of those levels have been noted already.

From the beginning we decided against an implementation on MMT and domain level. The MMT-system level, while having the largest amount of freedom for the generation of terms also provides significant difficulties regarding the consistency of terms and makes it harder to apply a number of criteria. Domain level, on the other hand, is to restrictive and would require seperate generators for each theory. This left us with either an implementation on LF-level or logic level.

In the end we decided to implement the generator on Logic-Level. While LF-Level implementation would also have worked almost exactly the same, the usage of SFOL is widely enough applicable for our purpose and allows for a seperation of generation of mathematical terms and formulas. We deemed this beneficial as for terms and formulas now seperate criterias can easily be applied, enabeling greater customisation options.

Exhaustive vs randomized generation Expression generation can be exhaustive or at random. Exhaustive generation produces consecutively all terms that can be generated in an escalating matter. Consider for example the theory of natural numbers, which has four function symbols - zero, succ, plus and mult - and imagine a generation case with three variables. Exhaustive generation would first generate the three possible atomic terms, consisting only of the variables, followed by all possible terms with a single applied function symbol.

This is rarely, even almost never, advantageous due to the sheer number of possible terms. More often it is better to examine a smaller partial set of those terms, for which randomized generation is a better alternative. Instead of consecutively generating all possible terms, new terms are generated by randomly selecting function symbols and sub terms. Through the appliance of criteria, the generation space can then be further restricted, increasing the chance to generate interesting expressions. Due to this, the randomized approach was selected.

Types of generation The generation of terms can be criteria based, template based or hard coded. Considering this and the goal of the generator, providing a general purpose generation system, criteria based generation was chosen from the beginning. We also already discussed templates as a special form of criteria application and the potential benefits, and decided based on this to include this option as well.

Hard coding is the easiest way to produce expressions of a specific form. Any method build that way is however naturally limited to generating this specific form only, or would require a flag to choose from implemented forms. At large, the same effect can be achieved by utilizing templates, which is why hard coding was not chosen in the end.

Forward vs Backward generation The generation of expressions can further be defined as either forward and backward.

Forward generation generates new terms in an escalating way. Starting with simple atomic terms, new terms are produced by determining at random a new function symbol that is to be used for it. Then, already generated terms are taken to fulfill the role of the inputs. It is a quick and easy way to generate a large amount of terms. However, depending on the criteria many generated terms can be unsuitable for return, but might have to be generated anyway as they are required to build other expressions. Consider, for example, a case in which a high minimal syntactic depth is required. In order to generate those

terms, a large number of terms below that depth have to be generated, leading to a preperation phase as many reject terms are created and saved until terms meeting the minimal requirement are generated.
At the same time generation does escalate relatively quickly. While that means that the preperation phase is shortened, it also means that the chance of generation of lower depth terms is reduced over time. Since already generated terms are used to build new terms, larger terms quickly begin to outnumber smaller terms. This, however, can be countered as we will see with the consideration of escalating the syntactic depth during generation.

Backward generation operates in the other direction. Instead of simply using already generated terms for new ones, a new term is instead generated by firstly determining the depth of the new term. Next, an operator is chosen and the depths of input terms for the operator are determined. All of those determinations are random within the constraints given by the criteria. Those terms are then again generated through the backward generation process, down to the atomic term. Since every subterm is newly generated at demand, generation of a new term can take significantly longer then in forward generation. However, even in long run times the generation or more evenly distributed then with forward generation, which can be quite advantageous.

Cascading syntactic depth The randomized generation approach and the possibility of usage of forward generation a quickly escalating syntactic depth can be problematic if the application requires smaller depth values or an even distribution of those values over a longer amount of time. This can however be partially circumnavigated by allowing the slow incrementation of the syntactic depth of expressions over time, enforcing specific depth values until a counter. Since this is relatively easy to achieve we decided to allow this in out implementation.

Syntactic depth handling The syntactic depth, as discussed before, is a basic, almost universally applicable criteria. However, in the case of formulae the depth of the entire formula is often not important, but can be considered in layers. The depth of a formula can be partioned into three layers - the depth of terms, if atomic formulas consist of terms, the depth of the formula, and the depth of quantifiers.

The depth added by quantifiers often is not particularly interesting since they are limited by the number of variables in a formula anyway and are often also further limited by restrictions regarding the number of free variables and alterations. As such the depth added by quantifiers is not added to the overall depth.

The depth of the formula body and the terms are more important. We have decided however to seperate the syntactic depth of terms and formulae since we might want to use different criteria for both and it allows for greater customization.

6 Algorithm

In this chapter we will first present how the generation criteria are used and how they influence the generation of terms and provide example generations where benefitial. Then, the nessecary informations the system has to know and save about terms will be listed and explained, followed by an explanation of the implementation of templates. Lastly, the implementation of the algorithm methods and data structures will be explained. The implementation unfortunately lacks an independent UI. All example generations have been extracted through console printouts.

6.1 Generation Criteria

Generation Criteria The criteria object allows embedding a second criteria object inside of it to allow the seperate customization of formula level and term level of an expression. The formula criteria contains the term criteria object. This is required in the case that formulae are generated without logic mode, which enables the usage of propositional variables instead of generated atomic formulae, or if a template is included that utilizes term generation.

Logic Mode Flag allows to generate logical formulas using propositional variables as atomic formulae instead atomic formulae build from terms. Quantifiers are not usable in this mode.

Variable number specifies the number of unique variables available for the generation process and is used during the criteria initialisation of the generator. Unique naming should of course be used, for example x_i where i is the number of the generated variable. A value of 3 therefore would generate three unique variables OMV(x_1), OMV(x_2) and OMV(x_3).

Literal number specifies the number of unique literals available for the generation process. Literals are generated for every data type in the theory and in a sequence. Generating 5 literals for a theory that contains natural numbers would yield the literals OMLIT(v, nat), where $v \in \{0, 1, 2, 3, 4\}$. Generating the same number of literals for a theory that contains both natural numbers and integer numbers would yield the afformentioned literals, as well as the integer literals OMLIT(v, int) where $v \in \{0, 1, -1, 2, -2\}$. Note that in order to use literals, a theory has to include realization rules for all types the theory contains.

Formula flag notifies the system that the criteria used are intended for formulas and call the formula generation method accordingly. It also allows for the usage of an embedded additional generation criteria object, allowing seperate customization of formulas and terms. Formulae use standard SFOL logic operators, namely negations, conjunctions, disjunctions, implications, equivalences and quantifiers, as well as equality and theory specific predicates to generate atomic formulas from terms.

Generation Mode determines whether forward or backward generation is used by the generation methods. Forward generation can be expected to quickly escalate the syntactic depth during generation, wheras backward generation allows for a more randomized depth within a given range. If formulae and term criteria are used, the generation mode of both has to match.

Ratio is used differently in terms and formulae, but always a number between 0 and 100. In terms it determines the ratio between literals and variables, where 0 deactivates literal usage, 100 deactivates variable usage, and 50 will attempt an evenly usage of both. As the usage is randomized, with the ratio used as a chance to pick one or the other, accumulations are possible.
In Formulas, the ratio determines the chance whether equality or a predicate is applied to a term. Ratio values are simmilar to the values used in terms, with 0 deactivating equality usage, and 100 deactivating predicates.

Function symbol exclusion list contains function symbols that are not to be used in the generation process. The exclusion list has to contain the symbols of the appropriate level - SFOL symbols for formula criteria, and theory specific symbols for term criteria. The list is optional - if no list is given, an empty list is automatically generated and no restrictions are applied.

Minimal and maximal syntactic depth specify the range of depth in which expressions are to be returned. Both are optional - if no values are given, no restrictions apply. If for example a minimal depth of 0 and a maximum depth of 4 is requested, only terms with a depth of 2, 3 or 4 are returned. As noted in the design decisions, the syntactic depth of formulae and terms are handled seperately.

Following is an example generation, using the theory NatPlusTimes and backward generation, in which only atomic formulae where generated, with an even application of predicate symbols and $\doteq$. For the terms, 3 variables where instantiated and no literals where used and the function symbols zero and successor where excluded. The term depth was set between 0 and 3.

Examples for generated atomic formulae:
uneven x1*(x1*x2)
even x2*x3
x1+x3$\doteq$x1
x3*((x2+x1)*x2)$\doteq$x1*x3+x1
(x2*x2)*x3$\doteq$(x2+x1*x3)*x1
uneven x1+x3
uneven x2*x1

Number of atomic formulas is required in the case formula generation is called using forward generation and determines how many atomic formulas are initialized for the generation of more complex formulas. In the case of backward generation, this can be skipped as new atomic formulae are generated on demand.

Type restriction list allows the exclusion of all function symbols of a specific type at once. Note that this excludes functions by output type - if a function symbol has the excluded data type as an input but a different output type, it still will be used and has to be excluded seperately. This is intentional as to allow the usage of e.g. variables and literals of one type and use such an inclusion function for type changes.

Escalation flag notifies the system that the syntactic depth of terms has to be escalated during generation. This ensures that for a while only terms or formulas of a specific depth are generated. The number of **Escalation steps** determines how many terms of a syntactic depth have to be returned before escalation. Since generation may yield no usable terms, for example if forward generation with a minimal syntactic depth is called, the **Escelation security steps** determine how many attempts should be made before escalating the depth.

Following is an example generation of integer terms in an escalating matter using forward generation. 3 variables and no literals where instantiated. The minimal depth was set to 2 and the type of natural numbers was excluded. The depth was escalated after 2 generations to keep the example clear.

```
Examples for generated integer terms:
(s x2)*x2
p s x1
(p x3)*((s x2)*x2)
-p s x1
p -p s x1
((p x3)*((s x2)*x2))*-x1
```

Quantifier flag decides whether or not Quantifiers are applied to a formula. Currently, quantifiers are only applied to the top level of a function. Quantifiers can not be applied in logic mode.

Quantifier alterations can be specified maximal and minimal number of quantifier alterations a formula can contain.

Number of free variables specifies how many variables have to remain unbound in a formula and can be set with a minimal and maximal value. This criterion trumps the alteration criterion, meaning that if not enough free variables exist to ensure the minimal number of alterations, only as many alterations as possible are applied.

Below is an example generation in the theory of natural numbers using backward generation. To keep it readable, terms have been limited to simple variables. Quantifier alterations where disallowed by setting both minimum and maximum to 0, and exactly 1 free variable was enforced.

Examples for generated quantified terms:

∀[x3:tm nat]∀[x2:tm nat]x1≐x2 ⟹ (even x3)

∃[x1:tm nat]∃[x3:tm nat]((uneven x2)⇔x3≐x1)∧(x1≐x3 ⟹ (uneven x3))

∀[x1:tm nat]x1≐x2

even x1

∃[x3:tm nat](x2≐x3 ⟹ (even x3))⇔¬(uneven x3)

∃[x2:tm nat]∃[x1:tm nat]x2≐x1∨¬(even x3)

Template contains a template object from which an expression is built. The template object is clarified later in this chapter.

6.2 Complexities

In order to correctly generate expressions and to ensure the criteria, some informations have to be saved. For this purpose a new complexity object has been created. The generator creates the complexity object alongside the expression and saves both in a tuple, which is then written in a set. Upon generation of a new expression from other expressions, both the old expression and the complexity object are accessed. Following is a short overview which informations are saved and an explanation why.

Syntactic depth is a convience save for easy access to the syntactic depth of a term, which is required to calculate syntactic depth during forward generation. This is necessary since in the case of a minimal syntactic depth criterion as we have to generate expressions of lower syntactic depth first, which are then used to build deeper expressions.

Variables is a list of all used variables in an expression, as well as their type. A similiar list for all **bound variables** is saved as well. This allows easy access to variables during the quantification process in formula generation and allows to derive the number and list of unbound variables for quantifier application.

Last used quantifier and the **number of quantifier alterations** are saved to make it easier to count quantifier alterations during generation and ensure that the number of alterations lies within the bounds given in the criteria.

6.3 Template

For the purpose of template usage, a new template object has been implemented. This object accounts for both continuous and absolute templates and allows for the limited combination of both.

The **continuous template** a list of triple containing the GlobalName and 2 int values. Int values indicate minimal and maximal number of applications of the level symbol. This allows for the building of terms with several levels of continuous functions. A template for a CNF formula with 10 to 20 clauses containing 5 to 10 atomic formulas can for example be expressed as the List $\{(\wedge,10,20),(\vee,5,10)\}$ and an absolute template OMV(x), where the variable x is substituted with either a generated atomic formula or a propositional variable if logic mode is active.
The continuous template is however limited in terms of quantifier application. Only a single quantifier can be in such a template at the moment. It has to be at the first list position and is applied to all free variables - the int values are unused in that case. This is due to the fact that we can't effectively enforce limits and bounds regarding quantifier application here.

Absolute template is an expression on which substitutions according to a substitution list are applied. As explained in chapter 3, any MMT expression can serve as an absolute template. For any variable used in such a template a substitution has to be provided in the substitution list.

Substitution list contains a list of quadruples with a variable reference OMV(x), the type of the variable symbol, an int to indicate the substitution mode and an optional template object. Four modes are supported - substitution with a literal, with a generated term, a generated formula, the provided template. If no mode is selected the variable will not be substituted. The substitution with another template allows for the recursive definition of more complex expressions.

Consider for example a template for a horn formula $\forall x \wedge_0^2 f \implies f$ with 1 to 3 horn clauses implying another horn clause, where x are bound variables and f randomly generated atomic formulas. This can be translated into a generator template consisting of the continuous template $\{(\forall,0,0)\}$, the absolute template OMA($\implies$,OMV(t),OMV(f)) and the substitution list {(OMV(t),3,TO), (OMV(f),2,null)}.
TO is then another template object with the continuous template $\{(\wedge,0,5)\}$, the absolute template OMV(f) and the substitution list {(OMV(f),2,null)}

By using the generation criteria for formulas to enforce the exclusive generation of atomic formulas, and term criteria as one wished, now horn formulas can be generated. An example generation using this template and limited depth of terms for readability is provided below.

Examples for generated Horn formulas:
$\forall$[x1:tm nat]$\forall$[x3:tm nat]$\forall$[x2:tm nat]x2$\doteq$x3$\wedge$(x2$\doteq$x1$\wedge$(even x2)) $\Longrightarrow$ x1$\doteq$x1
$\forall$[x3:tm nat]$\forall$[x2:tm nat](even x2) $\Longrightarrow$ (even x3)
$\forall$[x3:tm nat]$\forall$[x2:tm nat]x3$\doteq$x2 $\Longrightarrow$ x2$\doteq$x3
$\forall$[x3:tm nat]$\forall$[x2:tm nat](even x2) $\Longrightarrow$ (uneven x3)
$\forall$[x1:tm nat]$\forall$[x3:tm nat]$\forall$[x2:tm nat](even x1)$\wedge$(x3$\doteq$x1$\wedge$x2$\doteq$x1) $\Longrightarrow$ (even x1)

Unfortunately, aside from the limitations in quantifier application, we also can't express the conditional aspect of horn formulas here. Horn formulas can also come in the form of horn clauses $\forall x f(x)$. If we wanted to enable this in a template, we would have to implement a conditional application, which we did not manage in the time frame.

6.4 Implementation

SFOLTermGenerator(controller: Controller, mp: MPath) is called first to instantiate the generator. A Controller and MPath, the path to the theory for which terms are to be generated, are required inputs. The theory is read out through an adapter, yielding type, function and predicate symbols, as well as literals. All necessary data structures to save generated terms and functions are initialized, as are counters necessary for escalating generation.

Generator(Criteria) is the top level function that is called when new terms or formulas have to be generated. The Criteria object is tested for inconsitencies and then saved globally. Variables and literals are instantiated according to the criteria and saved as terms, and globally saved data structures reinitialized. This allows the user to call the generator to partially reset it, allowing the generation of new terms with new criteria on the same theory. Generator then calls the StreamGenerator(mode) to generate a stream of terms.

TermGenerator(), **FormulaGenerator()** and **TemplateGenerator()** are recursive helper functions to build the expression streams. In case of TermGenerator and FormulaGenerator, the escalation of syntactic depth is checked and executed here and the depth of a new term is randomly determined if backward generation is executed without depth escalation.

generatebyTemplate() generates a term according to a template and substitution list in the criteria. Since the structure is set by the template, general criteria are ignored. The template is a term in which the variables are substituted according to the substitution list. Variables can be substituted with other variables, literals or entire terms and formulae, determined by flags within the list. The substitution process is random within their categories. In the case of terms and formulae, the appropriate criteria have to be given in the criteria objects.

generateTerm(requested depth) generates a single term according to the generation criteria. Both forward and backward generation are supported. If forward generation is called, a function symbol is chosen at random from a list of available symbols. This list can be reduced by an exclusion list contained in the criteria. For the chosen function symbol the required input terms are chosen randomly from a list of already generated terms and their complexities. Only terms below the maximum depth are usable, if a maximum depth is applicable. A new complexity object is generated alongside the term, using the known complexities of the used input terms. The term is saved in the list of terms and complexities, if the depth is either below the maximum depth, or if it is the same depth as the current escalation level is escalation is enabled. The term is then checked against the rest of the criteria. If it fulfilles them, it is returned, otherwise a new term is returned. For escalation, a safety block counts up the number of unsuccessful generations. If this number reaches the number given in the complexities, the depth is escalated with a warning.

Backward generation uses the requested depth to directly generate a new term within the depth. A random operator is chosen in the same way as in forward generation, with the additional requirement that a data type has to be chosen in the criteria. For each required input, the method is called again recursively with a randomly determined depth up to one below the current depth. One of the inputs is chosen to have exactly the depth minus one to ensure the requested depth is met. In the case that the requested depth is 0, an atomic term is chosen from either the list of variables or the list of literals. The chance with which a variable or a literal is chosen is a ratio in the Criteria, from 0 to 100. A ratio of 70 means that there is a 70 percent chance a variable is chosen. A new complexity object is generated accordingly. The new term and its complexity is saved, and the term returned. In backward generation we can assume that all generated terms meet the criteria.

generateFormula(requested depth) functions analogue to the generateTerm() method, but uses the SFOL function symbols $\neg, \wedge, \vee, \implies, \Leftrightarrow, \forall, \exists, \exists!$. The building of function bodies and quantification is seperated, and the system is currently designed to apply quantifiers only on the top level of functions.

FormulaInit() initializes a number of atomic formulas. As forward generation requires existing expressions to build new expressions from, those atomic formulas have to be pre generated from terms.

generateAtomicFormula() generates a single atomic formula. The method applies predicates and typed equality according to the ratio given in the criteria object for formulae.

requestDepth(Criteria) is a helper function that generates a new depth according to the minimal and maximal syntactic depth from a criteria object.

Since this function is used by both term and formula generation methods, the criteria object has to be handed over at method call to ensure the usage of the correct criteria.

requestNumber(mod, min) is another helper function that returns a random positive number. mod is an integer value, which gets applied on a modulo operation, min is an optional integer number if the number has to use a different minimum then 0.

CriteriaCheck(Criteria) is called upon initialization of the generator method and checks the criteria object for inconsistencies. It will abort in the case of such a inconsistency and return an error message on the console clarifying which values where responsible for the abort.

TemplateCheck(Template) is called if the criteria object contains a template object and checks it.

escalate(Criteria) is a helper function that manages the counter for automated syntactic depth escalation. Different counters are used for terms and formulae. The main counter determines the current generation level. In the case of backward generation, it is automatically set ot the minimal syntactic depth as given in the criteria. If forward generation is called, the counter is initiated at 0 as lower levels have to be generated for the higher syntactic depths. Any generated expression below the minimum is saved, but not returned in that case.

The user has to declare the number of expressions per syntactic depth level that have to be generated. The generator will automatically increment the syntactic depth when those expressions are returned. Additionally a security number has to be provided. It defines the number of failed generations on a level that are allowed. If that number is reached, the generator will provide a warning and increment the generation level. This is again important for forward generation and cases in which criteria might hinder successful generation on specific depth levels.

7 Conclusion

By investigating applications and their respective expressions we have identified properties that can define an expression as interesting for an application and, through further evaluation of these properties, created criteria usable for the early implementation of a multi purpose generator system. Some of those criteria, especially the semantic criteria, ended up as either to complex to implement for a multi purpose system or where impossible to ensure during the generation process and had to be filtered out in the end. Furthermore we have discussed several generation methods and presented a flexible template system that allows for the combination of absolute and continuous template formats as well as criteria based generation in an MMT environment.

During the developement of the generator, it became increasingly clear that criteria based and application independent generation of expressions, while possible, is not an optimal solution. Constraints in time and resources as well as growing algorithm complexity always lead to a situation in which a cost-benefit consideration has to be made, leading to the selective filtering of less used and less important criteria which might have been implemented in a specialized generator system. This leads to a loss of generation accuracy in regard to the amount of useful expressions generated, which has to be minimized to ensure the usefulness of the generator.

Nontheless an application independent generator can remain useful in situations in which a specialized generation system has not been implemented yet, or where the loss of generation accuracy is sufficiently small to neglect and expression filtering is an option.

Future work

Expansion of the generator Naturally, being an early implementation, thereis still has a number of improvements that can be made on the generator. UI support for the creation of criteria and template object code and an added possibility of parsing of criteria from external files would greatly increase usability and integratability into other systems. A reexamination of found but rejected criteria may also lead to possible future inclusion of those criteria, improving generation accuracy for the already investigated applications.

Changing the level of implementation Currently, the generator is implemented in SFOL level, utilizing a single specified logic. While the generator has still a large amount of freedom reimplementing it on the level of a logical framework would allow even greater freedom by gaining independance from the predetermined logic, opening the generator for more applications.

Examination In this thesis, we only examined a small number of applications to gain generation criteria. To further decrease the application dependence of the generator, more applications have to be investigated to find more criteria that can be applied to the process. The investigation of those applications can further yield additional insights as to when an expression is useful, potentially benefiting specialized generator systems as well and improving the understanding of the problems solving algorithms and generators are written for.

Automatization Ideally, this investigation would not be manual. Considering the constant improvements in pattern recognition and machine learning, the examination process could be automized to not only find properties that increase the chance of an expression to be useful for an application, but also to suggest potential implementations to realize those properties as criteria.

References

[Sam59] Arthur L. Samuel. “Some studies in machine learning using the game of Checkers”. In: *IBM JOURNAL OF RESEARCH AND DEVELOPMENT* (1959), pp. 71–105.

[Buc00] Bruno Buchberger. “Theory Exploration with Theorema”. In: *Analele Universitatii Din Timisoara, Seria Matematica-Informatica* XXXVIII (Jan. 2000), pp. 9–32.

[HHP01] Robert Harper, Furio Honsell, and Gordon Plotkin. “A Framework for Defining Logics”. In: *Journal of the ACM* 40 (July 2001). DOI: 10.1145/138027.138060.

[MCA02] Susan Mcroy, Songsak Channarukul, and Syed Ali. “YAG: A Template-Based Generator for Real-Time Systems”. In: (Dec. 2002). DOI: 10.3115/1118253.1118293.

[Ste02] Holger Stenzhorn. “XtraGen - A Natural Language Generation System Using XML- and Java-Technologies”. In: (Oct. 2002). DOI: 10.3115/1118808.1118821.

[TL03] Ana Tomás and José Leal. “A CLP-Based Tool for Computer Aided Generation and Solving of Maths Exercises”. In: vol. 2562. Jan. 2003, pp. 223–240. ISBN: 978-3-540-00389-2. DOI: 10.1007/3-540-36388-2_16.

[Buc04] Bruno Buchberger. “Algorithm Supported Mathematical Theory Exploration: A Personal View and Stragegy”. In: Jan. 2004, pp. 236–250.

[Bie+09] Armin Biere et al. “Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications”. In: (Jan. 2009).

[JDB11] Moa Johansson, Lucas Dixon, and Alan Bundy. “Conjecture Synthesis for Inductive Theories”. In: *J. Autom. Reasoning* 47 (Oct. 2011), pp. 251–289. DOI: 10.1007/s10817-010-9193-y.

[Ben12] Mordechai Ben-Ari. *Mathematical Logic for Computer Science*. 3rd. Springer Publishing Company, Incorporated, 2012. ISBN: 978-1-4471-4128-0.

[MRT12] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012. ISBN: 978-0-262-01825-8.

[RK12] Florian Rabe and Michael Kohlhase. “A Scalable Module System”. In: (Dec. 2012).

[Alm+13] José Almeida et al. “PASSAROLA: High-order exercise generation system”. In: Jan. 2013, pp. 1–5.

[Joh+14] Moa Johansson et al. “Hipster: Integrating Theory Exploration in a Proof Assistant”. In: May 2014. ISBN: 978-3-319-08433-6. DOI: 10.1007/978-3-319-08434-3_9.

[Rab16] Florian Rabe. “The Future of Logic: Foundation-Independence”. In: *Logica Universalis* 10 (Mar. 2016). DOI: 10.1007/s11787-015-0132-x.

[Joh17] Moa Johansson. “Automated Theory Exploration for Interactive Theorem Proving: An introduction to the Hipster system”. In: Aug. 2017, pp. 1–11. ISBN: 978-3-319-66106-3. DOI: 10.1007/978-3-319-66107-0_1.

[SMA+17] NICHOLAS SMALLBONE et al. “Quick specifications for the busy programmer”. In: *Journal of Functional Programming* 27 (July 2017). DOI: 10.1017/S0956796817000090.

[MHP18] Muhammed Milani, Sahereh Hosseinpour, and Hüseyin Pehlivan. “Rule-Based Production of Mathematical Expressions”. In: *Mathematics* 6 (Nov. 2018), p. 254. DOI: 10.3390/math6110254.

[KGK19] Radoslaw Klimek, Katarzyna Grobler-Debska, and Edyta Kucharska. “System for automatic generation of logical formulas”. In: *MATEC Web of Conferences* 252 (Jan. 2019), p. 03005. DOI: 10.1051/matecconf/201925203005.

[RM19] Florian Rabe and Dennis Müller. “Structuring Theories with Implicit Morphisms”. In: June 2019, pp. 154–173. ISBN: 978-3-030-23219-1. DOI: 10.1007/978-3-030-23220-7_9.

[Cor] Microsoft Corporation. *Excel Formula Autocomplete*. URL: https://support.microsoft.com/en-us/office/use-formula-autocomplete-6d13daa5-e003-4431-abab-9edef51fae6b. (accessed: 10.12.2020).

[LLC] Google LLC. *Google search prediction*. URL: https://support.google.com/websearch/answer/106230?hl=en. (accessed: 10.12.2020).

[Raba] Florian Rabe. *MMT Online Documentation*. URL: https://uniformal.github.io//doc/. (accessed: 10.12.2020).

[Rabb] Florian Rabe. *UniFormal/MMT – The MMT Language and System*. URL: https://github.com/UniFormal/MMT. (accessed: 10.12.2020).