A Meta Language for Mathematical Reasoning

Michael Junk
University of Constance
78457 Konstanz, Germany
michael.junk@uni-konstanz.de

Stefan Hölle
University of Constance
78457 Konstanz, Germany
stefan.hoelle@uni-konstanz.de

Sebastian Sahli
University of Constance
78457 Konstanz, Germany
sebastian.sahli@uni-konstanz.de

Abstract

We present a formal system which establishes a meta-language for the description and the subsequent reasoning in general mathematical models and theories. The basic expressions of the language act like functions and types at the same time. As crucial ingredient, the evaluation operation is only admissible for compatible arguments. The initial function objects and base expressions allow list formation, restriction of functions, formulation of theorems, equality and existence statements. We will present examples of the expressiveness of the language, the basic syntactic structure and inferential rules.

1 Starting Point

Our approach to formal mathematics was initiated by requirements arising from research projects and teaching in applied mathematics and scientific computing. The idea was to establish a formalism which allows to specify mathematical models intuitively and consistently in the whole range from word problems in elementary school up to full fledged models in industrial projects, while being easy to convey even to first year students.

In the past four years, we have mainly concentrated on the design of syntactical elements and rules which are easily comprehensible for math students attending the preparatory course, the basic math lectures and the modeling lecture. With consistency, conciseness and student-acceptance playing the role of environmental pressure, an evolutionary process has led to the meta-language $\mathbb{M}$ATh as acronym for ***m**eta-language for **m**odels, **a**lgorithms and **th**eories* [1]. Now we are interested in discussions about similarities and differences to existing approaches and hope to contribute some ideas which resulted from our particular starting point.

2 Mathematical Models

Our understanding of mathematical models is quite broad. In particular, we do not assume a particular set theory or logic from the beginning but rather consider these as mathematical models themselves.

In general, a model (or *theory* according to [2]) comprises of a finite number of parameters which represent abstract mathematical objects whose meaning emerges from a list of axioms which describe their mutual relations.

In: O. Hasan, C. Kaliszyk, A. Naumowicz (eds.): Proceedings of the Workshop Formal Mathematics for Mathematicians (FMM), Hagenberg, Austria, 13-Aug-2018, published at http://ceur-ws.org

Once the model assumptions are fixed, consequences are drawn from these assumptions according to precisely stated rules.

While concluding from the axioms can be viewed as an *interior* aspect of a model, there are also *exterior* aspects because models may be used as building blocks within other models. As example, we consider the concept *probability space* as a mathematical model of uncertainty which we call *pSpace* here. It starts out with parameters $\Omega, \mathcal{F}, P$ which represent a sample space Ω, a set $\mathcal{F}$ of events and a probability measure P. Important consequences are related, for example, to the notion *random variable, expectation* and *independence* which are defined within the model.

Similarly, the model *pSpace* is a consequence of measure theory which belongs to the standard model of real numbers which again belongs to the consequences of set theory and classical logic. But *pSpace* is also used sideways in many specific models which deal with uncertainty. Simple textbook examples (coin-flipping, fair dice, lottery etc.) and numerous statistical models specify concrete sample spaces, event sets and probability measures which satisfy the axioms of *pSpace* once they replace $\Omega, \mathcal{F}, P$. By *applying* the model in this way, all its true statements translate into the more specific scenario and are ready to be used to derive other consequences.

This aspect of using *pSpace* is very similar to applying a theorem in order to access its implication. However, it is quite likely that also the expectation functional E or the adjective *independent* defined in *pSpace* need to be translated to the specific scenario. To achieve this, one could think of *pSpace* as a class-constructor which assigns to an admissible triple $(D, \mathcal{A}, Q)$ an object $S := pSpace(D, \mathcal{A}, Q)$ which contains the specific versions of E and *independent* as named components, i.e. $S.E$ and S.independent. Naturally, one would consider S of *type pSpace* in this case, i.e. $S : pSpace$.

From this example we derive the important observation that (1) models are *built* within other models (proofs are required to ensure that the consequences are valid) and (2) models are *used* within other models and produce return values for that purpose (proofs are required to check the model assumptions for the arguments).

To reflect these different purposes syntactically, we introduce so called *frames* which consist of a list of parameters followed by a list of axioms, a list of consequences and a resulting expression where each of these blocks may be empty. Using an exemplary grammar, a sketch of the *pSpace*-description may look like

```
pSpace := (Omega, F, P) with
  (Omega, F, P) : measureSpace;
  P(Omega)=1;
consequences
  rV(G with G:sigmaAlgebra) := X with X:measurable(F,G) end;
  E(X with X in L1(P)) := integral(X,P);
  prove forall X with X:bounded holds X in L1(P) directly ...qed
  ⋮
result
  ('randVar' >> rV, 'Expectation' >> E, ...);
end
```

Here the result is a record with field names `randVar`, `Expectation` and so on. It turns out, that the same syntactic structure can be used for various other purposes like definig plain functions

```
tan := x with not(cos(x)=0) result sin(x)/cos(x) end
```

Here, the consequences are often empty (unless well-definition cannot be determined automatically and needs proof steps from the user). Of course, one would generally recast this definition in a more intuitive form with the same content

```
tan(x with not(cos(x)=0)) := sin(x)/cos(x)
```

Since functions are special frames, the type-statement `0:tan` is valid (being of type `tan` means being formed by `tan`, i.e. being in the range of `tan`). To access the argument condition of a frame, we use the expression `Def(tan)` which is equal to the frame

```
D := x with not(cos(x)=0) end
```

In this case, no result is provided which means implicitly that the frame parameter itself is used, i.e. `D` is an identity function. Also here, being of type `D` means being in the range of `D`, but since range and domain of definition coincide, it also means satisfying the frame condition.

For this reason, frames returning simply the frame parameters take over the role of collections (like sets) and are used to define all standard mathematical notions, e.g.

```
injective := f with forall x,y with f(x)=f(y) holds x=y end
```

Altogether, models, theories, notions, functions, sets, classes and types can be summarized as frames which again are essentially functions enriched with statements derived from the assumptions on the arguments.

Since every frame is defined within another frame, a hierarchical organization of mathematical content follows naturally. To initialize this hierarchy, the MATh-interpreter provides an initial context in which frames can be built based on rules which are presented in the following section.

3 Formal Aspects of MATh

A MATh-text essentially describes a sequence of actions which modify a context represented in the MATh-interpreter. Such actions may be definitions, opening and closing of sub-contexts, claim checking, adding new objects and axioms etc. which generally take MATh-expressions as arguments. The context is essentially characterized by a list of parameters and axioms, a list of defined names with the expressions they point to and a list of true statements which have been deduced from the axioms. If a sub-context is opened, names and true statements are inherited, although local names may mask parent names.

Expressions in MATh can be *admissible* and if they are, they can *hold* (i.e. *be true*). These attributes are controlled by corresponding rules which are realized as strategies within the MATh-interpreter. If statements are to be proved, references to these strategies are required (like in many proof assistants[4, 5]). Due to automatic strategy searches and recursive strategy calls, some sort of automatic theorem proving can be achieved, although this is not our primary concern. Currently, the interpreter is only a tool to check that the proposed language rules are practicable.

It should be noted that MATh expects its input in terms of expression trees which are built by combining elementary expression data-structures of an intermediate language. In this way, the realization of the initial context is not tied to any comfortable syntax and can be kept rather slim. The actual human readable syntax of a full implementation will offer more constructs which are mapped to the base expressions after parsing (our current implementation uses the ANTLR parser[3] - the essential grammar description is online [1]).

We will now introduce the base expressions of the intermediate language in a textual form (table 1).

Table 1: Base Expressions: `x` can be replaced by a name or a list of pairwise different names and $\texttt{f,a,y,A}_\texttt{i}\texttt{,B}_\texttt{j}$ can be replaced by expressions

map	`x->y`
evaluation	`f(a)`
list	$\texttt{(A}_1\texttt{,...,A}_\texttt{n}\texttt{)}$
condition	$\texttt{x with A}_1\texttt{;...;A}_\texttt{n}\texttt{ end}$
theorem	$\texttt{forall x with A}_1\texttt{;...;A}_\texttt{n}\texttt{ hold B}_1\texttt{;...;B}_\texttt{m}\texttt{ end}$

Initially admissible expressions are given by seven parameter names which we discuss in the sequel.

```
indicator, Def, equal, example, list, restriction, frame
```

The parameter `indicator` is used to realize the type relation `x:T` which is only a shorthand of `indicator(T)(x)`. It parallels the idea of indicator functions $\mathbb{1}_U$ of sets U which evaluate to $\mathbb{1}_U(y) = 1$ if and only if $y \in U$ holds. In this spirit, we say that `x` is of type `T` if `indicator(T)(x)` holds.

The parameter `Def` controls function evaluation in the sense that `f(x)` is admissible if `f,x` are admissible and if `x:Def(f)` holds. Since admissibility of `f(x)` depends on admissibility of three other evaluations in `indicator(Def(f))(x)`, additional rules ensure that certain combinations of `Def` and `indicator` evaluations are admissible.

The parameter `equal` realizes the equality expression `x=y` as abbreviation of `x:equal(y)`. Rules ensure that arbitrary admissible expressions can be combined in equality statements.

The expression `example` is used to formulate existence statements. Here the idea is that `example` assigns a witness to each non-empty type, i.e. rules ensure that `example(T):T` holds whenever it is admissible. With

this usage in mind, `Def(example)` discriminates between empty and non-empty types so that `T:Def(example)` is reasonably abbreviated as `exists T`, meaning that examples of type `T` exist. Since the availability of such a choice function entails the axiom of choice if sets are modeled as special types, the explicit usage of `example` has to be suppressed if the axiom of choice is not desired. In this case, access is only possible in proof steps which allow working with witnesses of valid existence statements. Another usage of `example` is its restriction to unique types which gives rise to a definite description operator. This is important when working with types that possess exactly one example.

A list like `L:=(a,b)` with admissible components is considered as a function whose arguments are `1,2` and whose corresponding values are `a,b`. In particular, strategies ensure that `Def(L)=(1,2)`, `L(1)=a`, `L(2)=b` are true as well as and `a:L`, `b:L`. More generally, lists of length `n` are examples of type `list(n)` so that `Def(list)` can be viewed as the type of unsigned integers. In this way, `n`-ary functions are functions on subtypes of `list(n)` which can be naturally combined with the syntax convention that `f(a,b,c)` abbreviates `f((a,b,c))`.

The base expression `x->y` is admissible if the expression `y` is partially admissible, meaning that the `Def`-constraints in building up `y` are neglected. Together with the rule, that `u:Def(x->y)` holds if `y` is admissible after replacing `x` by `u`, this allows the construction of partial functions.

Next, we consider the base expression `E := x with` $A_1;...;A_n$ `end` which essentially describes an identity function and therefore acts set-like. It accepts a list of expressions $A_1;...;A_n$ and is admissible if all `x->`A_i are. Rules ensuring the admissibility of `u:E` (i.e. the truth of `u:Def(indicator(E))`) for certain types of `u` or `E` are typically formulated as axioms which have to be chosen carefully to avoid contradictions. We comment on this in the last section. Once the admissibility is assured, `u:E` holds iff each `(x->`A_i`)(u)` holds. In this case, `E(u)` is identical to `u`. Syntactically, we allow also definitions `N:=z` in the list $A_1;...;A_n$. When forming the actual base expression, this is tranformed to `z:admissible` while the meta-information about `N` as name for `z` is extracted and used to update the state of the interpreter.

With `restriction`, the domain of `x->y` can be modified. More specifically, for `F:=restriction(D,R)` the rules assure that `u:Def(F)` holds iff `u:D` and `u:Def(R)` are true with the result `F(u)=R(u)`. In this way, general function definitions like those in the previous section are possible. For such general functions, the type relation `y:F` is equivalent to `exists x with y=F(x) end`.

The base expression `forall x with` $A_1;...;A_n$ `hold` $B_1;...;B_m$ `end` introduces the theorem-concept in MATh. The statement is true if in a sub-context with additional parameter `x` and axioms $A_1;...;A_n$ the truth of $B_1;...;B_m$ can be shown (which is the direct proof strategy). With a modus ponens strategy, theorems can be applied. As example, the subtype relation `A<B` is an abbreviation of `forall x with x:A holds x:B` where a simplified syntax is used in the case of theorems with a single conclusion.

Finally, `F:=frame(D,C,R)` encodes a frame with condition block `D`, conclusion block `C` and result `R`. It is admissible if the properties `D<C` and `D<Def(R)` hold. `F` attains the values `R(x)` whenever `x:D` is true.

4 Examples

In our first example, we formulate Peano's axioms for a successor-function `succ` defined on `nat:=Def(list)`. Assuming that the logical negation function `not` has already been defined, the condition that `0` is no successor can be stated as `not(0:succ)`. The axiom that successors are again natural numbers can be cast in the form `succ<nat` with the subtype relation and `succ:injective` states that different numbers have different successors. In the higher order version, the induction-axiom can be formulated by quantification over all possible types `X` which contain `0` and satisfy the induction-step condition.

```
peano := succ with
  Def(succ)=nat; not(0:succ); succ<nat; succ:injective;
  step := X-> forall n with n:nat; n:X holds succ(n):X;
  forall X with 0:X; step(X) holds nat<X;
end
```

In a corresponding first order version, `X` would be restricted by the condition `X:formula` where `formula` needs to be defined by suitable recursive relations on the formation of admissible logical formulas. The induction-axiom then plays the role of an axiom schema

```
step := X-> forall n with n:nat; X(n) holds X(succ(n));
forall X with X:formula; X(0); step(X) holds forall n with n:nat holds X(n);
```

The way in which the `Def`-concept is introduced, obvious contradictions in the spirit of Russell's antinomy are avoided. In a model where `not` is the classical negation function, a Russell-type expression would be `R:= x with not(x:x) end` which is admissible. However, there is no rule ensuring `R:Def(indicator(R))` so that we don't know whether `R:R` as abbreviation of `indicator(R)(R)` is admissible. In particular, the dangerous tertium-non-datur statement `(R:R) or not(R:R)` cannot be formed. If we assume that `R:R` is admissible, then the standard argument leads to a contradiction which finally proves that `not(R:Def(indicator(R)))` holds which means that `R:R` is not admissible.

While the `Def`-concept avoids obvious contradictions it also blocks a lot of (seemingly) uncritical arguments. In particular, axioms on the structure of `Def(indicator(T))` are needed to allow constructive conclusions of the form `x:T`. As example, we mention the axiom that `x:T` is admissible for any `x` if `T` is a list or a condition of the particular form `x with x:A end`.

Similarly, the usual ZF set theory results, if the ZF set constructions are formulated as condition valued mappings and if `x:T` is postulated admissible for all `x` of type `set`.

References

[1] 𝕄ATh-Homepage, `http://www.math.uni-konstanz.de/mmath`. (currently in German)

[2] A. Tarski. *Introduction to Logic and to the Methodology of Deductive Sciences.* Oxford University Press, 1941.

[3] T. Parr. *The Definitive ANTLR 4 Reference.* The Pragmatic Programmers, 2012.

[4] Nuprl-Homepage, `http://www.nuprl.org/`.

[5] Isabelle-Homepage, `https://isabelle.in.tum.de/`.