Syntactic Metaprogramming for Type Theories

Andre Knispel

April 7, 2023

Abstract

This paper works towards the goal of creating a proof assistant that is capable of formalizing all of mathematics, without bias towards certain foundations or notations. It is motivated by the problem of interpreting arbitrary written mathematical texts, ignoring the issue of natural language processing. The central argument is that to translate such texts into terms of a type theory requires capabilities that are traditionally provided by the reader, as mathematical texts may require them to perform arbitrary and context-aware processing steps. We attempt to model the reader as part of a proposed proof assistant, enabling a high degree of control over syntax and semantics. This allows for interactions that are common in written mathematics but have not yet been possible in proof assistants, such as the ability to implement an object logic without syntactic overhead or the ability to implement, and thus also modify any aspect of the system that is not part of the core in the system itself, like the module system or surface syntax.

1 Introduction

While in principle proof assistants are sufficiently expressive to cover the entirety of mathematics [Bar10], in practice the translation effort to convert mathematical definitions, theorems and proofs into such systems is often non-trivial and forces users to consider issues that are not present in written mathematics, such as choosing a certain style based on performance considerations [GCS14] or carefully tweaking code to behave favorably with respect to language features. If one hopes to convince mathematicians to adopt theorem provers, one should try to match the 'feature set' of written mathematics. Proof assistants extend a formal system with some convenience functionality, so they model a human reader that would follow a specific set of rules to derive terms of the formal system from a text. Freek Wiedijk writes in [Wie07]: "if we want to make some progress of getting people actually to use formal mathematics, it has to be close to the way mathematics already is being done for centuries. Improving on tradition is good, but ignoring tradition is stupid". While this is an argument for classical and declarative systems, it applies just as much to the flexibility of mathematics.

Proof assistants are generally built on some formal system, but are much larger than just that system. This arrangement could be described as a formal system with bells and whistles that make it easier to work in that system, but they don't provide any additional power. This is very different from mathematics, where the formal system is usually not even fixed. Instead, we are usually satisfied with a proof if it is evident that it could be translated into some formal system. But that system could depend on the very text one is reading or could even be left up to the reader (to some extent). On the other hand, any equivalent of the syntactic sugar and convenience features of proof assistants can be introduced by a mathematical text or assumed to be known from some previous text, but none of them are actually built into mathematics itself. Thus, to arrive at a system that behaves more like mathematics in this regard, it seems natural to attempt to mirror that behavior.

In Section 2, two concrete properties of mathematics are described and arguments for why these properties were selected as requirements for the proposed systems are presented. In Section 3, these properties are expressed in a way that is directly applicable to proof assistants, and a system that satisfies these requirements is derived informally. A formal description of this system and its interaction with a given type theory is provided in 4, and implementation details in Section 5. Section 6 demonstrates how such a system can address the issues raised in Section 2 in a way that is akin to how we would naturally solve them if we were to treat mathematical texts more formally.

As a side effect, this system also provides the user with a programming language whose metaprogramming system can supply a large number of features that are traditionally built into the system, similar to variants of Lisp such as Racket. This ranges from simple notational features, such as notation for strings, integers and lists, to more complicated features such as a module system and quotation syntax. These examples and a sketch of their implementation are also presented in Section 6. An important aspect of implementing these systems using metaprogramming is that competing implementations for the same system can coexist and be loaded by the user on demand. This could for example allow the user to use legacy code whose features aren't supported by the system anymore to be loaded side-by-side to newer code.

The system described here has been implemented with a type theory based on Cedille-core [Stu18] which is a minimalistic type theory without a datatype system. It was chosen because of its minimalistic implementation, Turing completeness, and simple proof of consistency. Because the additions are also fairly small, the resulting system is very small, its implementation consists of roughly 3000 lines of code. It can be found at `https://github.com/WhatisRT/meta-cedille`. Lean 4 also follows a similar path and has a metaprogramming system of comparable strength that is used to implement many of its surface-level features.

2 Comparing mathematics and proof assistants

This section proposes two "features" that the reader (or listener, etc.) provides to mathematical communication that are not commonly found in proof assistants:

1. The reader can be required to generate arbitrary mathematical language as part of the text, for example by leaving something to the reader explicitly, and they can

2. make arbitrary changes to existing syntax and its semantics, including replacing them with different ones entirely.

These properties will be translated into requirements applicable to a QED system in the next section. First however, note that point 1 is not about the ability of a reader to find the correct piece of language to be inserted in a text, but rather the ability to leave arbitrary holes in the text that the reader could fill, if they arrived at the correct piece of text to plug in the hole in some way, for example by following some instructions given in the text.

To go into detail as to why these two properties will be taken as requirements, first consider the act of leaving something "up to the reader". This can be explicit or implicit, and there is no restriction as to what can be left to the reader. There is also no requirement that every reader who is able to understand the rest of the text be able to fill the hole by themselves. For example, a text can say "A C^k manifold is defined similarly to a C^0 manifold, except that continuous is replaced by k-times continuously differentiable", leaving the definition to the reader, but providing some guide. This leads naturally to the first requirement.

The second requirement is less obvious at first, even though it is clear that mathematics provides the ability to change syntax at will. To convince the reader that it is indeed necessary, we will give some examples of where this is used in written mathematics.

First, consider that, as alluded to in the introduction, written mathematics does not directly correspond to the syntax of a formal system but instead can be *translated* into such a system. Importantly, the formal system does not need to be fixed before we can start writing mathematics, but it can instead be chosen by the reader. Consider this example from [Uni13], after Lemma 2.1.1:

Since this is our first time stating something as a "Lemma" or "Theorem", let us pause to consider what that means. Recall that propositions [...] are identified with types, whereas lemmas and theorems [...] are identified with inhabited types. Thus, the statement of a lemma or theorem should be translated into a type, [...], and its proof translated into an inhabitant of that type.

While for most of mathematics this is an uncommonly deep intrusion in the meaning of mathematical writing, a QED system attempting to support the entirety of mathematics needs to be able to express the equivalent of this paragraph in some way, thus at least being able to change the semantics of common mathematical language.

Additionally, compared to the syntax of proof assistants or programming languages, the syntax of mathematics is a wild west, and syntax can have arbitrary semantics. Consider these examples:

- Mathematics can express every syntax and semantic of every past and future programming language (or pseudo-code).

- There are some higher-dimensional notations, like commutative diagrams or string diagrams from category theory.

- Mathematical texts regularly introduce new syntax, for example:
  By $b_k \cdots b_0$ with $b_i \in \{0,1\}$, we mean the natural number $\sum_{i=0}^{k} b_i 2^i$.
  For mathematics, **any** way of reading natural numbers can be considered as being introduced in a similar fashion instead of being native to the system, as evidenced for example by a long history of mathematics without arabic numerals.

3 Motivation

We translate the requirements from the previous section as follows:

1. The system has a metaprogramming system with the property that any code fragment that could be written by the user can be generated by a meta-program.

2. The grammar of the system can be arbitrarily replaced with a different grammar (from a certain fixed class) and the function that interprets the grammar can be arbitrarily assigned by the user.

We now give an informal overview of the proposed system, that skips many details. It serves as motivation and lets us compare the above wish list to the system from a higher level perspective. A detailed description and formal definitions can be found in the next section. For the remainder of this section, fix some dependent type theory $\mathbf{T}$. The system consists of $\mathbf{T}$, extended with a *metaprogramming monad* $\mathbf{M}$ and a set of corresponding primitives, similar to the commonly used `IO` monad, and a *meta-environment* consisting of:

- A list of tuples (R, t) where R is a parsing rule for a context-free grammar and t a term,

- a starting non-terminal for the rules of that grammar and
- an *evaluator*, which is a Kleisli arrow of the metaprogramming monad between some types, i.e. of type $A \to \mathbf{M}\, B$ for some types A and B.

We require that every operation that modifies the environment of the type theory can be invoked from the metaprogramming monad via corresponding primitives and such that there is a primitive to assign values to the meta-environment.

The type A is akin to the type of abstract syntax trees, i.e. the type of results of the parser. The terms associated to the parsing rules are used to build a value of type A from an input string and can be thought of as constructors of A corresponding to the parsing rules. A description of how exactly an input string corresponds to a value of type A is given in the next section.

Processing input will execute the following steps:

1. Generate some a of type A from the input, using the parsing rules and associated terms.
2. Apply the evaluator to a to obtain a b of type $\mathbf{M}\, B$.
3. Try to execute the meta-program b, which may change the current (meta-)environment.

Steps 1 and 3 in this list might fail. A failure in Step 1 can be a result of a parsing error or a type checking error (if the term does not check as having type A), and a failure in Step 3 is a failure in one of the primitives, for example because the primitive tried to define a term that does not type check.

Let us compare this system to the above requirements. As we include primitives for every possible modification of the environment and meta-environment, any such modification can be carried out within $\mathbf{M}$, which means that arbitrary language can be generated during the elaboration step, which is our first requirement. Assuming that we have a parser that can handle our chosen class of grammars, the user can indeed replace the grammar and the function interpreting it, simply by replacing the meta-environment.

3.1 The parsing system

The parsing system is derived from the observation that the production rules for a context-free grammar have a similar structure as constructors for inductive datatypes. This is similar to [Atk12]. To make this more precise, we first associate a category to a context-free grammar:

Definition 1. Let G be a context-free grammar. Define $\mathcal{C}_G$ to be the free monoidal category whose objects are generated by the non-terminal symbols of G, and whose morphisms are generated by the rules of G, where R induces a morphism

$$m_R : N_1 \otimes \cdots \otimes N_l \to N$$

where N is the non-terminal that R expands and $N_1, \dots, N_l$ are the non-terminals of the string R expands N to (with duplications and in order). If this results in the empty string, we use the empty tensor product, i.e. the unit object I.

Then, if $\mathcal{T}$ is the category of contexts of our type theory and the type theory has some notion of inductive types and constructors, there exists a functor $\alpha : \mathcal{C}_G \to \mathcal{T}$ that sends each non-terminal to the type of abstract syntax trees that this non-terminal can parse, and each morphism to a constructor of that type. This α is faithful and an isomorphism onto its image.

If S is the starting non-terminal for G, then a derivation amounts to specifying a morphism $I \to S$ in $\mathcal{C}_G$ and applying α results in a morphism $\mathrm{Unit} \to \alpha(S)$, i.e. an element of $\alpha(S)$. This element is the AST of that derivation.

This results in a very strict form of AST though: it reflects the entire derivation, which might include optional whitespace or comments that might be undesirable. Additionally, there are some advantages to not having to work with ASTs directly. For example, one might want to use existing types that don't directly support all the features of the grammar, but that can be desugared from the grammar instead. To relax the strictness of system, we instead use an arbitrary (monoidal) functor $F : \mathcal{C}_G \to \mathcal{T}$. This lets us generate an element of $F(S)$ from a derivation, but instead of only applying constructors, F can perform preprocessing on the data. Because $\mathcal{C}_G$ is free, specifying F amounts to specifying $F(N)$ for all the non-terminal symbols N of G and $F(m_R)$ for all rules R of G (in such a way that they are compatible).

As long as every non-terminal has at least one rule associated with it, the images of the non-terminal symbols can also be inferred by the images of the rules. And as long as it is possible to syntactically distinguish terminal symbols from non-terminal symbols, to give a pair G, F of a context-free grammar and a functor $\mathcal{C}_G \to \mathcal{T}$, it is sufficient to give a list of pairs $R, F(m_R)$ such that the types of $F(m_R)$ satisfy appropriate compatibility conditions and a starting non-terminal S. Apart from the evaluator, this is exactly what the meta-environment provides.

4 Theoretical description of the system

We will now describe in detail the proposed extension to type theories and the parsing process.

4.1 Type theory

Let $\mathbf{T}$ be a type theory containing the Calculus of Constructions and $\mathbf{PrimMeta} := \{m_1, \dots, m_n\}$ a set of constants. To $\mathbf{T}$, we add the terms

$$\mathbf{M}\,t,\ \nu\ t\ t',\ \eta\ t \text{ and } \xi\ m\ t$$

where $m \in \mathbf{PrimMeta}$ and t, t' are terms. In addition, for every $m \in \mathbf{PrimMeta}$, fix two terms S_m and R_m. The new terms have the following typing rules:

$$\frac{\Gamma \vdash T : *}{\Gamma \vdash \mathbf{M}\,T : *} \qquad \frac{\Gamma \vdash t : \mathbf{M}\,T \quad \Gamma \vdash t' : \prod x{:}T.\,\mathbf{M}\,T' \quad \Gamma \vdash \mathbf{M}\,T' : *}{\Gamma \vdash \nu\ t\ t' : \mathbf{M}\,T'}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \eta\ t : \mathbf{M}\,T} \qquad \frac{\Gamma \vdash t : S_m}{\Gamma \vdash \xi\ m\ t : \mathbf{M}(R_m\ t)}$$

The additional terms and typing rules can be understood as follows: $\mathbf{M}$ is a monad, with its unit and bind operations given by ν and η respectively. Its role is similar to Haskell's `IO` monad or Agda's `TC` monad, being a wrapper for stateful interaction with the environment and operating system. In the typing rule for ν, the premises contain $\Gamma \vdash \mathbf{M}\,T' : *$ to ensure that t' is a non-dependent arrow, otherwise the type of a ν might not be valid in Γ.

Interactions with the environment are performed via ξ, which is used to combine several primitives into one. $\xi\ m$ is a primitive to interact with the system, so the second argument to ξ is an argument for that primitive. For example, if $S_{\texttt{print}} = \texttt{String}$ and $R_{\texttt{print}} t = \texttt{Unit}$,

$$\xi\ \texttt{print}\ \texttt{"Hello World!"}$$

would print the string `Hello World!` to the standard output and return the unique element of `Unit`. We will discuss how $\mathbf{PrimMeta}, S_m$ and R_m are chosen for our implementation in Section 4.2.

4.2 Parsing & evaluation

This section describes how to parse, evaluate and type check inputs. Let $\mathcal{G}$ be a subset of the set of context-free grammars.

Definition 2. A *$\mathcal{G}$-meta-environment* is triple (G, φ, e) where $G \in \mathcal{G}$, φ is a function from the set of production rules of G to terms, and e is a term. It is called *valid in* Γ if for all production rules r of G, $\varphi(r)$ has a valid type in Γ, and there exist A and B such that $\Gamma \vdash e : \prod x{:}A.\, \mathbf{M}\, B$.

The (global) state of the system is given by a tuple (Γ, E), where Γ is an environment, and E is a meta-environment valid in Γ.

To process an input, it first is parsed with the grammar contained in E and transformed into a term t using φ. Then, it is checked if there exists a type B such that $\Gamma \vdash e\ t : \mathbf{M}\, B$, and if this is the case, the statements given by $e\ t$ are executed, potentially modifying the global state (Γ, E). We assume that for every B, after normalization every term of type $\mathbf{M}\, B$ is of the form $\nu\ t\ t', \eta\ t$ or $\xi\ m\ t$ for some m, t, t'. ν and η correspond to the monadic `bind` and `pure` operations, so they are executed as usual, with the caveat that if t transforms the system from a state (Γ, E) to (Γ', E') returning some value v, $t'v$ is executed in the state (Γ', E') when executing $\nu\ t\ t'$. This leaves terms of the form $\xi\ m\ t$, corresponding to primitives, whose execution depends on a specific choice of **PrimMeta** and is an implementation detail. An example of how primitives could be chosen is described in Section 5.1.

There is one extra issue present in this system that is usually not present in type theories, which is that the initial state of the system cannot simply be some form of an empty environment. This is because if the system started with an empty grammar, it would not be able to process any input. A non-trivial meta-environment is required to bootstrap the system. See Section 5.3 for an example.

5 Implementation

Implementing the system first requires fixing the allowable set of grammars, the primitives and the initial environment. For $\mathcal{G}$, we choose the set of LL(1) grammars over some alphabet, with strings of that alphabet as non-terminal symbols[1].

5.1 Primitives

We need to give a finite set **PrimMeta**, and for each of its elements m two types S_m and R_m. Recall that if $\xi\ m\ t$ type checks, it is of type $\mathbf{M}(R_m\ t)$. For convenience, we define terms R'_m and derive R_m from R'_m via $R_m := \lambda T{:}S_m.R'_m$. Let (Γ, E) be the global state in which $\xi\ m\ t$ is executed and pr_2 the projection to the second component.

m	S_m	R'_m
check	**Term** $\times$ $*$	$pr_2\ T +$ **String**
let	**String** $\times$ **Term**	$\top +$ **String**
normalize	**Term**	**Term**
parse	**String** $\times$ $*$	$pr_2\ T +$ **String**
shell	**String**	**String**
setmeta	**Term** $\times$ **String** $\times$ **String**	$\top +$ **String**

[1]The actual implementation includes some extensions that simplify writing rules that deal with unknown characters, such as strings or comments.

- **check**: Let (t, T) be the input. If $\Gamma \vdash t : T$, return t, or an error message otherwise.
- **let**: Let (s, t) be the input. If s is a free name in Γ, and t a term for which we can synthesize a valid type, add $s := t$ to the context. If not, write an error message into the result string.
- **normalize**: Normalizes the given term and returns the result.
- **parse**: Let (s, T) be the input. Parse s with the current grammar, and return the resulting expression if it is of type T.
- **shell**: Execute the given string as a shell command, and return the result as a string.
- **setmeta**: Attempts to set the meta-environment as described in the next section. Returns an error message if it was not successful.

5.2 Setting the meta-environment

Setting the meta-environment requires deriving a meta-environment from some parameters. First, we describe how to derive a production rule from the name of a global definition.

A name that is eligible to be used as a rule consists of three parts separated by a special character σ: the namespace, a non-terminal symbol and a string of terminal and non-terminal symbols. I.e, the name should be of the form $p + \sigma + N + \sigma + r$ for some p, N and r. Within r, one has to be able to designate certain strings as non-terminal symbols and the ability to escape certain characters. For this purpose, we fix two more characters τ_1 and τ_2 that enclose non-terminal symbols and certain strings that are to be replaced by a terminal symbol, similar to how strings are enclosed by double quotes. In the examples in this paper and in the reference implementation we chose σ := `'$'`, τ_1 := `'_'` and τ_2 := `'='`. However, we avoid using the escape mechanism here, and instead use the desired symbols directly, using ␣ and ↵ to represent space and newline characters respectively.

To set the meta-environment, let e be a term, p, N strings and Γ the current environment. Let $\Gamma|_p$ be Γ restricted to definitions that have the prefix $p + \sigma$ in their name, and R be a (partial) function mapping strings to production rules as described above. Then, define $E := (G, \varphi, e)$ where G is the context-free grammar with production rules $R(n)$ for all names n in $\Gamma|_p$ and starting non-terminal N, and $\varphi(R(n)) := t$ if $n := t \in \Gamma$. If E is a valid meta-environment in Γ, change the current meta-environment to E.

5.3 Initial environment

To bootstrap the system, consider the following evaluator:

```
eval : Term → M⊤
eval t = do
  t' ← ξ check (t , M⊤)
  case t' of λ
    { (inj₁ u) → u
    ; (inj₂ s) → echo s }
```

where `echo : String → M Unit` prints a string to the standard output (which could be a primitive, or implemented with the `shell` primitive). The environment also needs to contain a grammar and constructors for the `Term` datatype. From this minimal system, the user can then define their own type and grammar for top-level statements by invoking ξ `let` and switch to that system using ξ `setmeta`.

Note that any meta-environment that gives the user access to execute arbitrary terms of $\mathbf{M}\top$ is equivalent in expressive strength to this one, as any term of type $\mathbf{M}\,T$ can be converted into a term of type $\mathbf{M}\top$ (because $\mathbf{M}$ is a monad) which can be executed.

5.4 Example: Untyped lambda calculus

As an example, consider the following grammar for an untyped lambda calculus, where V is a non-terminal symbol for variables:

$$T \to V$$
$$T \to \lambda V\,.\,T$$
$$T \to T\;T$$

To encode this grammar, assume a type `ULTerm` with the following constructors:

```
Var : String → ULTerm
Lam : String → ULTerm → ULTerm
App : ULTerm → ULTerm → ULTerm
```

This grammar, together with a syntax to invoke it from the top-level would be encoded via the following definitions:

```
lc$T$_V_ = Var
lc$T$λ_V_.␣_T_ = Lam
lc$T$_T_␣_T_ = App

lc$TOP-LEVEL$λ_name_␣=␣_T_ : String → ULTerm → M(⊤ + String)
lc$TOP-LEVEL$λ_name_␣=␣_T_ = λ n t → ξ let (n , quoteULTerm t)
```

Note that whitespace is part of this grammar, so there is no space after the λ and exactly one space after the dot. To make this more flexible, non-terminals that allow for variable amounts of whitespace can be employed. `quoteULTerm` is a function that quotes a `ULTerm` to a regular term as used by the system. After setting the meta-environment, the following two lines would be valid syntax, producing the same result:

```
λid = λx. x
id = Lam "x" (Var "x")
```

6 Applications

The description of the system up to this point is complete, save for fixing the base type theory and a few minor implementation details. This section demonstrates how to recover some common features of proof assistants in a system that implements only what has been described so far, using a Calculus of inductive Constructions as the base theory. We will also present some examples of features that are not available in current proof assistants. As before, code examples in this section are written in Agda syntax for clarity. All the applications described in this section have been tested in the system and can be found in the repository.

6.1 Applications in syntax

Before diving into more complex applications it seems sensible to give some examples of using the ability to modify syntax to imitate features a compiler front-end might provide.

6.1.1 Quasi-quotation

To write terms more easily, we want to implement a syntax for quasi-quotation. To quote a term, we want to wrap it in quotes, and to unquote parts of a quoted term, we want to wrap it in commas. Thus, the following two lines should represent the same term:

```
λ t : Term. `λ T : *. ,t,`
λ t : Term. Lambda "T" Ast t
```

To be able to unquote terms, we require the `Term` datatype to have a constructor `unquoteTerm : Term → Term`, and a function `quoteTerm : Term → Term` that replaces a term with a quoted representation of that term. That function should drop the `unquoteTerm` constructor for terms, so `quoteTerm (unquoteTerm t) = t`. On other constructors of `Term`, it should replace that constructor with a quoted representation of the constructor applied to recursive applications of `quoteTerm` to its arguments. Assuming we already had a quotation syntax, `quoteTerm` should satisfy `quoteTerm t = `t``.

The syntax is then defined as follows:

```
n$term$`_term_` = quoteTerm
n$term$,_term_, = unquoteTerm
```

6.1.2 Syntax for natural numbers, strings and lists

As alluded to earlier, we want to be able to implement common syntax constructs. For natural numbers, there are 10 parsing rules for single digits, meaning that the `digit` non-terminal symbol is parsed into a value of type `Digit`. As we have to parse at least one digit, we have two non-terminal symbols, `digits'` for zero or more digits and `digits` for one or more digits. Finally, we add a rule for the `term` non-terminal to parse `digits`. This means that the function corresponding to this rule should have type `List Digit → Term` and turn the list of digits into a term that evaluates to the natural number given by this list of digits. In summary:

```
n$digit$0 : Digit
...
n$digit$9 : Digit
n$digits'$_digit__digits'_ : Digit → List Digit → List Digit
```

```
n$digits'$ : List Digit
n$digits$_digit__digits'_ : Digit → List Digit → List Digit

n$term$_digits_ : List Digit → Term
n$term$_digits_ l = `listDigitToNat ,quoteListDigit l,`
```

All implementations of these functions are just the appropriate constructors, except for `quoteListDigit` which is a function that simply quotes a list of digits to a term that evaluates to that list.

The rules for strings and lists are done mostly in the same way, with one additional issue for the case of strings: the elementary unit that needs to be parsed for strings is a character. To allow parsing characters without requiring a separate rule for each character, we include a special syntax for parsing a single arbitrary character[2], and returning the parsed character.

6.1.3 Datatypes via lambda encodings

It is well-known that in the Calculus of Constructions, inductive types (without their induction principles) can be emulated for example via Church or Mendler encodings. This requires generating a functor from a list of constructors, which we will not repeat here - we simply assume an appropriate type `DatatypeDef` together with a function `genDefs : DatatypeDef → List (String × Term)` that converts all that data into a list of pairs of strings and terms that should be defined. This list should include the type itself, its constructors and its recursion principle. We then make the following definitions:

```
executeDefs : List (String × Term) → M ⊤
executeDefs (d :: ds) = do
  ξ let d
  executeDefs ds
executeDefs ([]) = return tt

n$TOPLEVEL$data_datadef_ : DatatypeDef → M ⊤
n$TOPLEVEL$data_datadef_ = executeDefs ∘ genDefs

n$datadef$␣_name_␣where↵_constrs_ : String → List (String × Term) → DatatypeDef
n$datadef$␣_name_␣where↵_constrs_ = ...

n$constrs$ = nil
n$constrs$|␣_constr_↵_constrs_ = cons
n$constr$_name_␣:␣_term_ s t = (s , t)
```

6.2 Environment flags and other global state

Having the ability to read and write global state whenever we are working within **M** is necessary for several of features mentioned further below and useful in general. The following describes a simple scheme of how to add such a state to an existing environment.

[2] With the capability to blacklist some characters.

Assume an existing evaluator `eval : A → M B`, a type `S` and a function `quoteS : S → Term` that quotes values of type `S`. To add a state of type `S` to the evaluator, define `evalS` as:

```
evalS : S → State S A → M B
evalS s a = let (a', s') = a s in
  do
    b ← eval a'
    ξ setmeta `evalS ,quoteS s',` p N
    return b

p$N$_OldN_ = return
```

where `p`, `N` and `OldN` are appropriate strings and `State S A` is defined as `S → A × S`. The new type of syntax trees will then be `State S A` and the old syntax is still present, embedded into `State S A`. New top-level syntax that has read and write access to the state can now be defined. We then choose an initial state `s` and run `ξ setmeta (evalS (quoteS s)) p N` to activate the new syntax. Note that this means that the meta-environment changes every time anything is evaluated.

This approach on its own does not scale well, as adding an additional state on top requires a re-implementation of the existing code. It is however straightforward to write a meta-program that generates the above from a term representing `S` and a term representing the initial state. With this, one can build a top-level command that replaces the state type `S` by `S × S'` and the current state `s` by `(s, s')`, by supplying terms for `S'` and `s'`.

6.3 Restricting features of the meta monad

Another technique that will be used below is to restrict some features of the meta monad, such as the ability to use terms with a certain prefix. This is fairly straightforward: First, define a new monad $\mathbf{M}'$ that re-implements the functions of $\mathbf{M}$ with the restrictions as desired, by evaluating a computation in $\mathbf{M}'$ to a computation in $\mathbf{M}$. Then, implement a new evaluator that uses $\mathbf{M}'$ instead of $\mathbf{M}$ and switch the meta-environment to it. This may also require re-implementing existing functions used in the syntax that use $\mathbf{M}$ to use $\mathbf{M}'$ instead.

6.4 Emulating a module system

By combining the techniques described so far, one can implement a rudimentary module system. This consists of some syntax and two additions to the state: a list of modules that can be imported, and a module state, containing a partial module that is currently under construction. While defining the module, all global definitions are given some unique prefix and are also recorded in the module state. When finishing the definition, the final module gets added to the list of modules that can be imported. Importing a module just binds every definition in the module to the same name, only with an appropriate prefix.

6.5 Reasoning in different logics

The approach used here has some similarities to [ACKS17], and we will use the same terminology of an *inner* and *outer* system here. To support another formal system, one can implement that system as one would in any programming language. This will become the inner system, and its environment can be added to the global state as described above. All that remains is then to add syntax for this system, possibly by using the existing syntax and some syntax to switch between both. Apart from performance considerations, working in the inner system would be identical to working in the same system implemented in another language.

One advantage of such an implementation over an implementation that is fully independent is that assuming the two logical systems are compatible in some way, one can automatically transfer definitions between them. If the outer system is an extension of the inner system, any closed definition done in the inner system can automatically be repeated in the outer system and vice versa. Open definitions require appropriate environments in both systems and would fail otherwise. Even if none of the systems is an extension of the other, one may still be able to implement a partial function that maps between terms of both systems, in one or both directions, and offer functionality that attempts to translate definitions if possible.

To demonstrate this, we have built a simple implementation of the Calculus of Constructions together with partial conversion functions that convert between the terms of the system itself and terms of that Calculus of Constructions. These conversion functions are used to re-use the parser for terms of the outer logic. This has the benefit that any syntax that is available in the outer logic carries over automatically, so in our implementation we get notation for lists and natural numbers for free (which requires functions like `nil`, `cons`, etc. to be defined). The statement for making a definition converts terms appropriately, type checks them and, if type checking was successful, adds them to the environment. As this implementation requires a fairly substantial amount of code that only combines an implementation of the Calculus of Constructions with concepts that already have been described, no code examples are given here.

7 Future applications

The previous section covered applications that already have been demonstrated. In this section, we discuss applications that are conjectured to be possible, and that will be investigated further in the future.

7.1 Elaboration

One can implement an inner system that is equal to the outer system and ensure that both systems have access to the same environment, by removing access to every currently defined name for the outer system as described above. Then, replace the top-level definition syntax by one that defines every term both in the inner and outer systems.

Every definition is then automatically accompanied by a quoted form of itself that cannot be accessed. However, the restricted meta monad can provide functionality to interact with the quoted terms, for example by providing a `lookup` function, that takes a name, and if that name is defined returns its quoted form. It can also provide an interface to the type checker and elaboration features of the inner system, which can then be applied to the outer system as well.

7.2 Parsing more complex grammars

The system in its current form is restricted to LL(1) grammars, simply because it is very easy to implement an LL(1) parser. There are two ways of allowing more complex grammars: either by implementing a more powerful parser for the system or by implementing a parser for the system in itself. This can be achieved by using the internal parsing rules to just accept every input until some termination symbol and return the result as a string. That string can then be parsed in the evaluator and turned into a more structured datatype before further processing. This has the advantage that there are no restrictions on the languages that can be parsed, but it comes at the cost of being difficult to integrate with the existing system.

7.3 Meta properties of algorithms

Consider the classical proof of Bézout's identity via the extended Euclidean algorithm, which gives an algorithm for how to find the pair of coefficients that the identity claims that exist. After writing this proof/algorithm, one might be interested in computational properties of the algorithm, like asymptotic runtime. In theories that are consistent with functional extensionality, it is impossible to reason about that, as it is always possible to artificially stall a computation. If we could reason about their runtime, we could specify two functions f and f' that are extensionally equal but have different asymptotic runtime, proving $f \neq f'$, thus proving the negation of functional extensionality. One way to fix this is by treating runtime characteristics as a property of the code, thus attempting to define it from its quoted representation. From that representation, one would be able to count the number of recursive calls, or the number of calls to a specific function for example.

7.4 Abstract manipulation of symbols

There are many instances where a term may or may not have proper semantics, but we care about some other property that can be derived from its denotation, like methods for approximating exact values. Consider for example the definition

$$\exp(x) := \sum_{k=0}^{\infty} \frac{x^k}{k!}.$$

While actually defining the exponential function that way requires a proof of convergence, even without it, this definition can be used to approximate e or other numbers that might be of interest. Depending on the definition, there might also be multiple methods for approximating values, such as the Monte-Carlo method for integration. Such approximations can even be useful in proofs, as they can be used as evidence for inequalities for example.

8 Related work

Many popular proof assistants include metaprogramming capabilities and methods for extending the surface syntax, most notably Coq [BBC$^+$97] and Lean 3 [dMKA$^+$15]. These systems do not include a method for applying arbitrary changes to the syntax, and instead only offer the ability to extend their syntax along predefined lines, usually with a focus only on the syntax usable in terms. It is thus easy to find examples of commonly used mathematical notations that cannot be replicated in these systems.

Stepping outside of the realm of proof assistants, there are many examples of programming languages that focus on the ability to extend syntax [OA18] [ERKO11], offering the ability to write syntax extensions for HTML or chemical notations.

MMT [RK13] and Theorema [BJK$^+$16] are examples of other systems that attempt to replicate capabilities of mathematics that are not commonly found in proof assistants. These two systems are many times more complicated than the system described here and offer large sets of features that do not yet exist in this system. It is however plausible that many of these features can be built by extending the system from the inside, without requiring any changes to the system itself.

Lean 4 [MU21] has a much more powerful metaprogramming system [dMU22], similar in strength to what is described here. It is, however, a much more complicated system and it is not clear how difficult it would be to apply it to different type theories. The system described here is very simple and should be easily adaptable to all sorts of type theories.

9 Conclusion

We have presented a system inspired by the capabilities of a reader of mathematical texts that extends a type theory with a metaprogramming and parsing system. We also have demonstrated how to replicate several types of interactions that are common in mathematical texts but that require some form of special support in many common proof assistants.

References

[ACKS17] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. *arXiv preprint arXiv:1705.03307*, 2017.

[Atk12] Robert Atkey. The semantics of parsing with semantic actions. In *2012 27th Annual IEEE Symposium on Logic in Computer Science*, pages 75–84. IEEE, 2012.

[Bar10] Bruno Barras. Sets in coq, coq in sets. *Journal of Formalized Reasoning*, 3(1):29–48, 2010.

[BBC$^+$97] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1.* PhD thesis, Inria, 1997.

[BJK$^+$16] Bruno Buchberger, Tudor Jebelean, Temur Kutsia, Alexander Maletzky, and Wolfgang Windsteiger. Theorema 2.0: Computer-assisted natural-style mathematics. *Journal of Formalized Reasoning*, 9:149–185, 01 2016.

[dMKA$^+$15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

[dMU22] Leonardo de Moura and Sebastian Ullrich. Beyond notations: Hygienic macro expansion for theorem proving languages. *Logical Methods in Computer Science*, 18, 2022.

[ERKO11] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-based syntactic language extensibility. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 391–406, 2011.

[GCS14] Jason Gross, Adam Chlipala, and David I. Spivak. Experience implementing a performant category-theory library in coq. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 275–291, Cham, 2014. Springer International Publishing.

[MU21] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on*

Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28, pages 625–635. Springer, 2021.

[OA18] Cyrus Omar and Jonathan Aldrich. Reasonably programmable literal notation. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–32, 2018.

[RK13] Florian Rabe and Michael Kohlhase. A scalable module system. *Information and Computation*, 230:1–54, 2013.

[Stu18] Aaron Stump. Syntax and typing for cedille core. *arXiv preprint arXiv:1811.01318*, 2018.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[Wie07] Freek Wiedijk. The qed manifesto revisited. *Studies in Logic, Grammar and Rhetoric*, 10(23):121–133, 2007.