Toward an Object-Oriented Structure for Mathematical Text

Fairouz Kamareddine, Manuel Maarek, and J.B. Wells

ULTRA group, Heriot-Watt University
`http://www.macs.hw.ac.uk/ultra/`

Abstract. Computerizing mathematical texts to allow software access to some or all of the texts' semantic content is a long and tedious process that currently requires much expertise. We believe it is useful to support computerization that adds some structural and semantic information, but does not require jumping directly from the word-processing level (e.g., LaTeX) to full formalization (e.g., Mizar, Coq, etc.). Although some existing mathematical languages are aimed at this middle ground (e.g., MathML, OpenMath, OMDoc), we believe they miss features needed to capture some important aspects of mathematical texts, especially the portion written with natural language. For this reason, we have been developing MathLang, a language for representing mathematical texts that has weak type checking and support for the special mathematical use of natural language. MathLang is currently aimed at only capturing the essential grammatical and binding structure of mathematical text without requiring full formalization.

The development of MathLang is directly driven by experience encoding real mathematical texts. Based on this experience, this paper presents the changes that yield our latest version of MathLang. We have restructured and simplified the core of the language, replaced our old notion of "context" by a new system of blocks and local scoping, and made other changes. Furthermore, we have enhanced our support for the mathematical use of *nouns* and *adjectives* with object-oriented features so that nouns now correspond to *classes*, and adjectives to *mixins*.

1 Introduction

From Euclid to Bourbaki, mathematicians have written their texts meticulously, in a precise, structured, and coherent form of natural language mixed with symbolic formula, which we call the Common Mathematical Language (CML). Is CML accurately reflected in current approaches to computerizing mathematics? If not, how can we make an improvement?

Approaches to computerizing mathematics. Computerizing mathematics is being done in various ways, each of which has advantages and disadvantages.

Mathematical word processing. The examples in figure 1 were included in this paper through the most basic kind of computerization. We typed the letters of the words of the text, and inserted LaTeX commands like `\begin{definition}`

Definition 20. *Of trilateral figures, an equilateral triangle is that which has its three sides equal, an isosceles triangle that which has two of its sides alone equal, and a scalene triangle that which has its three sides unequal.*

Euclid [7–Book I]

Definition 1. *A set with an associative law of composition, possessing an identity element and under which every elements is invertible, is called a group. [...] A group G is called finite if the underlying set of G is finite [...] A group [with operators] G is called* commutative *(or* Abelian*) if its group law is commutative.*

N. Bourbaki [2–Chapter I, §4]

Fig. 1. Two examples of CML

and `\end{definition}` to guide the output. In this approach, a computer program can produce a visual representation of the CML, but a computer program will have great difficulty in automatically recognizing the semantic content of the LaTeX encoding even if the TeX macros are being carefully chosen as proposed by Kohlhase [13]. Even in the best case, LaTeX can not be expected to capture the semantic content of natural language text any better than OMDoc (see below).

Semantic markup languages. A more advanced solution is computerization of CML that records more semantic content. In the semantic markup languages MathML-Content (`http://www.w3.org/Math/`) and OpenMath (`http://www.openmath.org/`), symbolic formulas are encoded using a library of predefined symbols. OMDoc (`http://www.mathweb.org/omdoc/`) adds a theory level. There are many ways to write our examples from figure 1 in OpenMath/OMDoc using a mixture of structural and symbolic XML elements and chunks of natural language. A possible encoding of our examples in OpenMath/OMDoc is sketched[1] here:

```
<!-- First example -->
<theory name="Euclid-book-1">
  <symbol id="equilateral-triangle">
    <CMP>An equilateral triangle is [...]
<!-- Second example -->
<theory name="Group">
  <symbol id="*">
  <symbol id="E">
    <CMP>A set with <OMOBJ>*</OMOBJ>, associative
         law of composition.
    <FMP>(a ∗ b) ∗ c = a ∗ (b ∗ c)
  <symbol id="e"> [...]
<theory name="FiniteGroup">
  <imports from="Group"> [...]
```

Natural language can only be stored in OMDoc in `CMP` elements as uninterpreted "blobs", while precise mathematical structure requires using symbolic encoding

[1] For readability and brevity, we show only the opening tag of each XML element for most elements; instead we use indentation to express nesting. We also use traditional mathematical output for OpenMath formulas instead of showing the XML tree.

(e.g., in `FMP` elements). Thus, for natural language mathematics, one must choose between retaining knowledge of the precise phrasing and presentation chosen by the mathematician, or capturing more of the structure via conversion to symbolic formula. Of course, one could do both like in our example above, keeping the uninterpreted natural language "blob" while adding a symbolic formula, but then the format does not support verifying they are mutually consistent. Generally, one does not expect formal checking of mathematics encoded in OMDoc.

Full formalization. Theorem Provers (TPs) such as Mizar (`http://www.mizar.org/`), Isabelle (`http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`), and Coq (`http://coq.inria.fr/`) have made a tremendous contribution to computerizing mathematics, providing frameworks in which a full formalization can be written and verified automatically. However, they do not support important issues of mathematical text, such as control over presentation and phrasing and processing of the semantic structure. Furthermore, because full formalization is very expensive in human time, most mathematical texts are unlikely to be fully formalized, but might still benefit from some form of computerization.

Semi-formalization. Lighter TPs have been proposed, such as the work by Wiedijk [17] defining Formal Proof Sketches (FPS) as light Mizar proofs. An FPS article is a basically a Mizar article with holes. This approach reduces the expense of computerization via formalization (and also loses the certainty of full formalization), but does not appear to greatly improve control over presentation and phrasing and support for semantics-based manipulation.

Computerizing the mathematical vernacular. N. G. de Bruijn, founder of the Automath project [4], suggested capturing the essence of CML through his Mathematical Vernacular (MV) [5], a language with *substantives* (*nouns*), *adjectives*, and *flags*. Weak Type Theory (WTT) [12] adapted the ideas from MV in a type-theoretical fashion. To evaluate the practicality of MV and weak types for mathematical texts, we developed MathLang-WTT [11, 10].[2] (In related work, others have investigated translating from WTT into type theory [9, 8].) MathLang-WTT improved over WTT by internalizing *flags* and *blocks* and by implementing a type checker and various automated output views of MathLang documents which are faithful to CML. See [10] for a description (which is still applicable to MathLang) of these MathLang-WTT features.

Limitations of MathLang-WTT. Despite the features of MathLang-WTT, our plan to closely follow the expressiveness of CML in a computerized language was still not fully satisfied. Limitations of MathLang-WTT (and hence also of MV and WTT) appeared during the translation of *Euclid's Elements* [7] in describing mathematical entities such as *triangles* and *lines*. Consider the example from Euclid in figure 1. A triangle is intrinsically related to the three lines it is formed by, but encoding it in MathLang-WTT was unsatisfactory. One approach was for *each* triangle to define the three lines and the triangle separately and then to state their relation. This was awkward, and more importantly there was

[2] We call the old version MathLang-WTT to distinguish it from this paper's version.

clearly a missed opportunity to do some simple type checking, like complaining if there was an attempt to define some triangle as consisting of four surfaces (like a tetrahedron) instead of three lines. Another approach was to define a triangle-constructing function, but then the type system could not check that the result was a valid triangle and could be used where a triangle was required. Description of mathematical objects needed improvement.

In trying to solve these problems, we noticed that (a) N. G. de Bruijn's informal definition of substantives and adjectives could be better formalised in MathLang and (b) work in object-oriented programming carries useful clues.

Object-oriented concepts. Some programming language research has focused on allowing organizing programs in the way that seems most natural to the programmers. *Classes* are a way of packaging definitions so that it is easy to obtain not only instances (*objects*) but also multiple distinctly modified and extended variants (*subclasses*) via *inheritance*. *Mixins* [3] are abstract subclass generators that allow reusing modifications and extensions.

Classes and objects. In object-oriented programming, a class is usually defined by a set of *fields* and *methods*. An object is an encapsulated sub-program with an internal state that is an *instance* of a particular class. Classes define the common behavior of a group of objects. Fields are named values associated with each instance, while methods are named operations on the instances.

Inheritance. Class inheritance avoids repeating the definition of fields and methods shared by several classes. A new class can be defined by inheriting from an existing *parent* class, and the child's set of fields and methods will by default contain those of the parent.

Mixins. With simple class inheritance, to make two classes share a common set of new methods without duplicating the method definitions, the classes must inherit from an ancestor class containing the new methods. This may require radical rearrangement of an existing class hierarchy. To alleviate this problem, mixins are subclass definitions that are parameterized on their superclass, and thus act as functions from classes to classes. When a mixin is applied to a class, this makes a new subclass that adds or redefines fields and methods.

Contributions of this paper. The needs of encoding mathematical texts led to the design of the following new features for MathLang reported in this paper.

1. **Lighter abstract syntax and an accessible type system.** We simplified the syntax and type system of MathLang-WTT. The new syntax of MathLang contains only one kind of identifier in contrast to the variables/constants/binders of both WTT and MathLang-WTT (section 2 and 4).
2. **Generalised reasoning structure.** This paper refines MathLang-WTT's blocks to a simpler yet more general notion and replaces MathLang-WTT's flags and contexts by a more flexible and general *local scoping*. Our new *block* and *local scoping* constructs are cases of *steps*, which are MathLang's generic structuring concept (section 2).

3. **Turning nouns into classes.** We combine MV's substantives (inherited via the nouns of WTT and MathLang-WTT) and object-oriented classes to make MathLang's *nouns* (section 3.1). Nouns are conceptually similar to classes, *terms* are objects, and *sets* can be defined from nouns.
4. **Turning adjectives into mixins.** We combine MV's mathematical adjectives with object-oriented mixins to make MathLang's *adjectives*, which can be used in different ways with nouns, adjectives, sets, and terms (section 3.2).

2 A More Generic and Structured MathLang

This section shows how MathLang improved over MathLang-WTT by defining more generic and structured constructions.

One class of identifiers. In MathLang's predecessors WTT and MathLang-WTT, identifiers are separated into three disjoint sets: *variables*, *constants*, and *binders*. The rest of this paragraph briefly describes how identifiers work in the older MathLang-WTT: All three kinds of identifiers have a weak type, and this is all that variables have. Constants also have a definition and parameters (each parameter being a variable declaration). Each use of a constant is applied to arguments of the right weak type. Binders have parameters like constants, and one additional special parameter for the bound variable. Unlike variables and constants, binders can not be defined inside a document but can only be declared in the *preface*. Binders can not be given definitions; a statement using a binder can act as a definition but there is no way to indicate this.

In encoding texts, we found these restrictions of the different identifier kinds problematic, so MathLang instead now has just one kind of identifiers and distinguishes the uses via types. To fit binders in our new scheme and to allow declaring/defining new binders in documents, we replace the old single special parameter of each binder with a new kind of parameter with a *declaration type* usable with any identifier. For example, the binder $\forall$ might be declared as `forall (dec(a), stat) : stat`, making it an identifier with output type `stat` and two parameters: a declaration of an identifier of arbitrary type `a` and an expression of type `stat` (statement). An example using this identifier is the translation `forall (n:N, >=(n,0))` of the proposition $\forall n \in \mathbb{N}. n \geq 0$ (assuming `N`, `>=` and `0` are already declared). Similarly, Russell's definite description binder ι (iota) could be declared with two parameters, a declaration and a statement. The first parameter is a variable that stands for the entire expression, and which should therefore have the same type: `iota (dec(a), stat) : a`. The expression $\iota\, n \in \mathbb{N}.\,(3 < n < 5)$ (meaning, "the unique $n \in \mathbb{N}$ s.t. $3 < n < 5$") would then be encoded as `iota (n:N, and(<(3,n),<(n,5)))`.

Simpler grouping and scoping. A fundamental idea of MathLang (inherited from MV) is capturing the grammatical and binding structure of a mathematical text. In MV and WTT, each *line* of a *book* has a context representing the set of assumptions about types of variables ("let x be a natural number") and truths ("suppose $x = y^2$ for some natural number y") used in the definition or statement made by the line. MV allows using flags as a secondary graphical 2-dimensional way of writing the current context in a book; an element repeated in the contexts of consecutive lines can be written as a flag whose *head* contains the repeated element and whose *flagstaff* goes through all the lines repeating the element. MV also has a secondary notion of blocks derived from flag nesting. (WTT could have used flags and blocks like MV, but this was never done.)

Unlike MV, MathLang-WTT directly supports flags and blocks rather than treating them as secondary syntax-sugaring notions derived from the contexts [11]. Upon careful examination of MathLang-WTT's flags and blocks, we found that they overlapped in function. MathLang-WTT's blocks allow grouping lines and sub-blocks and limiting to a block the scope of some of the constants defined in the block. MathLang-WTT's flag allow identifying a group of lines in which a context element is active.

In MathLang, we instead merged similar functionality. A *block*, written $\{step_1, \ldots, step_n\}$, is a sequence of statements. The *local scoping* construct $step_1 \rhd step_2$ makes the declarations, definitions, and assertions inside $step_1$ assumptions used by $step_2$ and restricts declarations and definitions inside $step_1$ to be visible only in $step_2$. Both blocks and local scoping constructs are *steps*, as are declarations, definitions, and assertions. Steps can be of various sizes, such as the declaration of a variable, the definition of a function, a proof, or an entire book.

Example 1. Sequences of statements in a proof are represented by a block.

```
{ x.(y+1) = x.y';
  x.y' = x.y+x;
  x.y+x = x.y+x.1 }
```

Similarly each different sub-part of a proof as well as the overall proof is represented by a block. Sections and chapter are also blocks in MathLang as they decompose the text. For example, a proof by induction could be a block with two sub-blocks (note that the second sub-block carries a local scoping which holds the inductive hypothesis):

```
{ --A proof of P by induction--
  { --Proof of the base-- [...]; P(0) };
  { --Proof of the induction--
    { n:N; P(n) } |>  { [...]; P(n+1) } } }
```

An entire proof (e.g., a proof by contradiction) can be contextualised in a local scoping.

```
{ --Proof of the contradiction-- [...] }
      |> { --Statement proved by contradiction-- [...] }
```

Note that we write a block in braces `{` and `}` , the elements of the block are separated by a semi-colon `;` . We write a local scoping with a step (which is the context), followed by the symbol `|>` (ASCII representation of $\rhd$), followed by a step (in which the elements of the context will be available). In these examples we added some comments in between `--` and cut some pieces of code ([...]). For readability, we make use of infix notation.

3 Abstraction with Nouns and Adjectives

3.1 Nouns as Classes

If we say that p is a demisemitriangle, one does not think of the set or the class of all demisemitriangles in the first place, but rather thinks of "demisemitriangle" as a type of p. It says what kind of things p is. [...] MV does not take sets as the primitive vehicles for describing elementhood but substantives (in the above example semidemitriangle is a substantive).

N. G. de Bruijn [5]

Nouns are abstractions that classify objects according to their common features. Nouns have an important place in some previous systems of representations of mathematics, such as WTT and MathLang-WTT, in which one of the weak types is *noun*. Nouns have used in translations of the first chapter of E. Landau's *Foundation of Analysis* [14] into WTT [16] and MathLang-WTT [11].

As already mentioned in section 1, we encountered limitations of the expressiveness of WTT-style nouns when we started translating Euclid's Elements [7]. Euclid starts his first chapter by defining basic geometric objects such as points, lines, figures, triangles, angles, etc. The definition of a line is as follows: *A line is breadthless length.* In MathLang-WTT, one way to write this is by defining *line* by forming a noun characterized by two statements: one that *line* "has length", the other that *line* is breadthless (does not "have breadth"). This uses a constant "has" which takes two nouns and returns a statement. This constant was unsatisfactory because it is hard to define its semantics precisely and because MathLang-WTT could not make any use of it for checking well-formedness. Because "has" deeply characterises the noun *line* and by consequence any concrete *line* — weak typed as "`term`" — created as a *line* instance, we felt it should be replaced by something that informs the language that *lines* have *length*, to allow approving of statements about the length of a line and disapproving of those about nonsense properties like its breadth, angle, weight, etc.

We found a solution in the concept of classes and objects in programming. A *line* is a class with one field *length.* Any instance of *line* is an object with a *length.* We characterise a line as breadthless in our translation with the absence of such a field. Table 1 gives more examples.

Consider the first definition in figure 1. Definition 20 of Euclid's example uses the noun *figure* (we see in section 3.2 how we encode the other nouns of

Table 1. Examples of noun definitions

Euclid's Elements	MathLang translation
A point is that which has no parts	`point := Noun`
A line is breadthless length	`line := Noun {length:term}`
A surface is that which has length and breadth only	`surface := Noun {length:term; breadth:term}`

this example). In the preceding definitions in [7], *figures* (*rectilinear figures*) are defined as *those contained by straight lines*. Therefore we define the noun `figure` with one field being the set of *straight lines* (we shorten it to *lines* in this example) and a statement precising that the figure is contained by this set of lines. The `Noun` constructor describes the noun with a step (in between braces `{` and `}`). The first unit of this step defines the field `sides`. Sides is a set of lines. The second unit of this step is a statement which uses an identifier `contained_by`. This identifier (declared earlier) takes a term and a set and returns a statement (`contained_by (term,set): stat`). The two parameters passed to this identifier are the future instance of the figure itself (encoded by the keyword `self`) and by the sides of the figure (field `sides` of `self`).

```
figure := Noun { sides : set(line);
                 contained_by(self,self.sides) }
```

Our second example in figure 1 is the definition of *group* by N. Bourbaki. We define `group` as a noun, The fields of this noun are identifiable in the text. The set E, the compositional law $*$ and the neutral element e. Two statements also define a group: the associativity of $*$ and the existence of an inverse of any element of E (we use an infix notation for `=`).

```
group := Noun { E:set; { a:E; b:E } |> *(a,b):E; e:E;
                forall (a:E, forall (b:E, forall (c:E,
                  *(*(a,b),c) = *(a,*(b,c)))));
                forall (x:E, invertible(e,x)) }
```

3.2 Adjectives as Mixins

An adjective belongs to a substantive, and serves a double purpose: (i) to form a new substantive, and (ii) to form a new sentence.

N. G. de Bruijn [5]

According to *(i)*, an adjective is a function from noun to noun. An adjective, like *isosceles*, when applied to a noun like *triangle* creates a new noun *isosceles triangle*. In our system where nouns are classes, the adjectives will be mixins [6]. Intuitively, a mixin is a function from class to class. As in mixin calculi, an adjective can also be applied to an adjective to form a new adjective, to a term to form a new term, and to a set to form a new set (mapping the adjective across all members of the set). In MathLang, we call these constructions *refinements*. Following *(ii)*, we also incorporate the possibility that an existing term has the properties held by an adjective. For example one can describe a triangle ABC and demonstrate that this triangle is isosceles. The last line of this demonstration can be written in MathLang as the statement: `ABC << isosceles` (read ABC is isosceles). In our syntax we join this *adjective statement* to a *sub-noun statement*. The sub-noun statement $A \ll B$, given by N. G. de Bruijn in MV, states that *"every A is a B"*. For example, `triangle << trilateral figure`. We kept this notation in MathLang.

Let us see the use of these notions in our two examples. In the example taken from Euclid's *Elements*, several adjectives are defined. The noun *triangle* is defined as a refinement of the noun *figure* using the adjective *trilateral*. We define the adjective `trilateral` with the constructor `Adj`. The `Adj` constructor takes as a parameter the noun to be extended to form the new noun. In the case of our example, `trilateral` could only be applied to figures as it requires the field `sides`. The body of `Adj` is a step (similarly to the `Noun` constructor). In this step two specific objects are available. `self` which refers to the instances of the noun being defined (see section 3.1) and `super` which refers to the instance of the noun being refined (only needed when a component of the old noun is hidden by a component with the same name of the new noun). After the definition of `trilateral`, `triangle` is simply defined as a `trilateral figure`. We similarly define the adjectives *equilateral*, *isosceles* and *scalene* (We use an infix notation for the identifiers `= (term,term):stat` and `!= (term,term):stat` and `and (stat,stat):stat`).

```
trilateral := Adj (figure) { card(self.sides) = 3 };
triangle := trilateral figure
equilateral := Adj (triangle) {
                 forall (side1:self.sides,
                   forall (side2:self.sides,
                     side1.length = side2.length)) }
isosceles := Adj (triangle) {
                 exists (side1:self.sides,
                   exists (side2:self.sides,
                         side1 != side2
                     and side1.length = side2.length)) }
scalene := Adj (triangle) {
               forall (side1:self.sides,
                 forall (side2:self.sides,
                   side1.length != side2.length)) }
```

3.3 Multi Adjective Refinements

With adjectives we have an operation of simple inheritance between nouns. Let us see with this last example how multi adjective refinements work.
Our *group* example defines two adjectives. These adjectives for groups are *finite* and *Abelian*. *Finite* states that the set E of the group is finite. *Abelian* (or *commutative*) states that the operator of the group is commutative. In MathLang, we write the definitions of these adjectives as follow.

```
finite := Adj (group) { finite_set(self.E) }
Abelian := Adj (group) {
  forall (x:self.E, forall (y:self.E, self.*(x,y) = self.*(y,x))) }
```

We could combine these two adjectives to obtain either `Abelian finite group` or `finite Abelian group`. In MathLang both expressions share the same type. Their meaning may differ as the statements introduced by the adjectives may overlap. It is for instance possible to define an `isosceles equilateral scalene triangle`. This expression is perfectly typable but of course would be considered as inconsistent even by pupils in primary schools. This reflects exactly the purpose of this first layer of MathLang which is to capture the structure of the text and its elements to allow, in a later stage, semantical analysis.

Table 2. Syntax of MathLang

ident, *i*	=	denumerably infinite set of identifiers	
label, *l*	=	denumerably infinite set of labels	
cvar, *v*	=	denumerably infinite set of category variables	
category, *c*	::=	`term`(*exp*) | `set`(*exp*) | `noun`(*exp*) | `adj`(*exp*, *exp*)	
	|	`stat` | `dec`(*category*) | *cvar*	
cident, *ci*	::=	*ident* | *exp*.*cident*	Identifiers anf fields
step, *s*	::=	*phrase*	Basic unit
	|	`label` *label* *step*	Labelling
	|	*step* $\triangleright$ *step*	Local scoping
	|	$\{\overrightarrow{step}\}$	Block
phrase, *p*	::=	*exp*	
	|	$cident(\overrightarrow{ident}) := exp$	Definition
	|	$ident(\overrightarrow{exp}) := exp$	Definition by matching case
	|	$ident \ll cident$	Sub-noun and adjective statement
exp, *e*	::=	$cident(\overrightarrow{exp})$	Instance
	|	$ident(\overrightarrow{category}) : exp$	Elementhood declaration
	|	$ident(\overrightarrow{category}) : category$	Declaration
	|	`Noun` {*step*}	Noun
	|	`Adj`(*exp*) {*step*}	Adjective
	|	*exp* *exp*	Refinement
	|	`self` | `super`	Self and super
	|	`ref` *label*	Referencing

$$\begin{tikzcd}
\texttt{group} \arrow[r, "\texttt{finite}"] \arrow[d, "\texttt{Abelian}"'] & \texttt{finite group} \arrow[d, "\texttt{Abelian}"] \\
\texttt{Abelian group} \arrow[r, "\texttt{finite}"'] & \begin{array}{c}\texttt{Abelian finite group} \\ \texttt{finite Abelian group}\end{array}
\end{tikzcd}$$

4 Language Description

Abstract syntax. The syntax of MathLang is given in Table 2. An arrow on top of a meta-variable represents a sequence of 0 or more meta-variables. For example $\overrightarrow{exp}$ is a sequence of *exp*. The elements of the sequence are separated with a comma "," in *ident*, *category* and *exp* and a semi-colon ";" in *step*.

Note the existence of a category constructor *noun* which describes a category expression and of a noun constructor *Noun* which describes a noun expression. In the following example, three identifiers with field `a` are defined: `p` is a noun, `p'` is a term instance of a noun, `p''` is defined as a noun.

```
{ p:noun(Noun {a:term});
  p':Noun {a:term};
  p'' := Noun{a:term} }
```

We use the following notational conventions in this document:

1. When an identifier has no parameters we omit the (). E.g., we write *ident* in place of *ident*() and *ident* : c in place of *ident*() : c.
2. We do not leave double braces in noun and adjective expressions defined with a block step. E.g., we write `Noun` $\{s_1; \dots; s_n\}$ and `Adj` (e) $\{s_1; \dots; s_n\}$ instead of `Noun` $\{\{s_1; \dots; s_n\}\}$ and `Adj` (e) $\{\{s_1; \dots; s_n\}\}$.
3. We abbreviate category expressions to shorten the syntax of some term, noun and set categories. E.g., we write `noun` (resp. `set` and `term`) in place of `noun`(`Noun` {{}}) (resp. `set`(`Noun` {{}}) and `term`(`Noun` {{}})).

Example 2. The following illustrate this syntactic sugaring:

1. We write `x` in place of `x()`.
2. We write `x:term` in place of `x():term`.
3. We write `Noun {x:term; >(x,0)}` in place of `Noun {{x:term; >(x,0)}}`.
4. We write `point:noun` in place of `point:noun(Noun{{}})`.

Type system. We now present the typing rules of our language. Each typing rule has the form: *context* ⊢ *construction* **:** *type judgement* where:

– A *context* of typing is a *step* of the language (with two additional markers that hold the type of `self` and `super` and the labels). It represents the previous steps of reasoning in which the *expression* is to be typed.
– A *type judgement* is either an *atomic type* or a *type* where:
atomic type = $Term(\mathcal{T}) \cup Set(\mathcal{T}) \cup Stat \cup Noun(\mathcal{T}) \cup Adj(\mathcal{T}, \mathcal{T}) \cup Step \cup cvar \cup Dec(type) \cup Def(type)$
$\mathcal{T}$ is the set of mappings from *ident* to *type*.
type is the set of mapping from sequences of *atomic type* to *atomic type*.
T (resp. *at* and t) ranges over $\mathcal{T}$ (resp. *atomic type* and *type*).

Here are some functions used in the derivation rules of our type system.

I : *step* ↦ *ident*	Gives the set of declared, defined and updated (sub-noun statement) identifiers in a step.
dI : *step* ↦ *ident*	Gives the set of declared identifiers in a step.
DI : *step* ↦ *ident*	Gives the set of defined identifiers in a step.
L : *step* ↦ *label*	Gives the set of defined labels in a step.
$\mathsf{dom}(f)$	Being the domain of the function f.
$T \uplus T'$	$T \cup \{\ (i, T'(i)) \mid i \notin \mathsf{dom}(T)\}$
$T \ominus T'$	$\{\ (i, T(i)) \mid i \notin \mathsf{dom}(T')\}$

And here is the subtyping relation between types and atomic types.

– $Term(T) \leq Term(T')$ *if* $\forall i \in \mathsf{dom}(T), T(i) \leq T'(i)$.
– $Set(T) \leq Set(T')$ *if* $\forall i \in \mathsf{dom}(T), T(i) \leq T'(i)$.
– $Stat \leq Stat$ *and* $Step \leq Step$.
– $Dec(t) \leq Dec(t')$ *if* $t \leq t'$.
– $Def(t) \leq Def(t')$ *if* $t \leq t'$.
– $Noun(T) \leq Noun(T')$ *if* $\forall i \in \mathsf{dom}(T), T(i) \leq T'(i)$.

$$\frac{\vdash s : \mathit{Step} \qquad s \vdash e : \mathit{Term}(T) \qquad i \in \mathsf{dom}(T)}{s \vdash e.i : T(i)}\ \textsc{ident-field}$$

$$\frac{\vdash \{\overrightarrow{s}; p\} : \mathit{Step} \qquad \{\overrightarrow{s}\} \vdash p : \mathit{Dec}(t) \qquad dI(p) = \{i\}}{\{\overrightarrow{s}; p\} \vdash i : t}\ \textsc{ident-dec}$$

$$\frac{\vdash \{\overrightarrow{s}; p\} : \mathit{Step} \qquad \{\overrightarrow{s}\} \vdash p : \mathit{Def}(t) \qquad DI(p) = \{i\}}{\{\overrightarrow{s}; p\} \vdash i : t}\ \textsc{ident-def}$$

$$\frac{\begin{array}{c}\vdash \{\overrightarrow{s}; i_1 \ll ci_2\} : \mathit{Step} \\ \{\overrightarrow{s}\} \vdash i_1 : \mathit{Term}(T_1) \qquad \{\overrightarrow{s}\} \vdash ci_2 : \mathit{Adj}(T_2, T_2') \qquad T_2 \le T_1\end{array}}{\{\overrightarrow{s}; i_1 \ll ci_2\} \vdash i_1 : \mathit{Noun}(T_1 \uplus T_2)}\ \textsc{ident-adj-term}$$

$$\frac{\vdash \{\overrightarrow{s}; s' \rhd s''\} : \mathit{Step} \qquad i \in I(s'') \qquad \{\overrightarrow{s}; s'; s''\} \vdash i : t}{\{\overrightarrow{s}; s' \rhd s''\} \vdash i : t}\ \textsc{ident-local-scoping}$$

$$\frac{\vdash \{\overrightarrow{s}; s'\} : \mathit{Step} \qquad i \notin I(s') \qquad \{\overrightarrow{s}\} \vdash i : t}{\{\overrightarrow{s}; s'\} \vdash i : t}\ \textsc{ident-skip-step}$$

These rules indicate how we retrieve the type of an identifier from the context. They decompose the *step* as context of typing to find the declaration IDENT-DEC, definition IDENT-DEF or the adjective statement IDENT-ADJ-TERM. In the case of a field of a term ($e.i$) the IDENT-FIELD rule applies first.

Fig. 2. Identifiers

- $\mathit{Adj}(T_1, T_2) \le \mathit{Adj}(T_1', T_2')$
  if $\forall i \in \mathsf{dom}(T_2), T_2(i) \le T_2'(i)$ *and* $\forall i \in \mathsf{dom}(T_1'), T_1'(i) \le T_1(i)$.
- $v \le v$.
- $(at_1, \ldots, at_n) \to at \le (at_1', \ldots, at_n') \to at'$ *if* $at \le at'$ *and* $\forall j \in [\![1 \ldots n]\!], at_j \le at'j$ *(after renaming of the category variables).*
- $T \le T'$ *if* $\forall i \in \mathsf{dom}(T), T(i) \le T'(i)$.

Figures 2, 3, 4, 5, 6 and 7 compose MathLang type system. According to these typing rules the `group` identifier defined in section 3.1 has type $\mathit{Noun}(\ \{(\mathtt{E}, \mathit{Set}),\ (*, (\mathit{Term}, \mathit{Term}) \to \mathit{Term}),\ (\mathtt{e}, \mathit{Term})\}\ )$. Similarly the noun `triangle` and the adjective `isosceles` have respective types:

$\mathit{Noun}(\ \{(\mathtt{sides}, \mathit{Set}(\{(\mathtt{length}, \mathit{Term})\}))\}\ )$ and
$\mathit{Adj}(\{(\mathtt{sides}, \mathit{Set}(\{(\mathtt{length}, \mathit{Term})\}))\},\ \{(\mathtt{sides}, \mathit{Set}(\{(\mathtt{length}, \mathit{Term})\}))\})$.

The type system prevents any misuse of identifiers' fields. For instance, let `ABC` be a declared triangle (`ABC:triangle`). This triangle is therefore a term with type $\mathit{Term}(\ \{(\mathtt{sides}, \mathit{Set}(\{(\mathtt{length}, \mathit{Term})\}))\}\ )$. According to our definition of triangle, the only defined field is `sides`, the set of lines composing a triangle. The expression `ABC.sides` refers to the sides of our triangle `ABC`. The set `ABC.sides` has type $\mathit{Set}(\{(\mathtt{length}, \mathit{Term})\})$.

The scopes of the identifiers depends on the location of the declaration or definition. Declarations could occur anywhere in an expression or could be an atomic step. We explain here the three possible cases: a declaration/definition in the flag

$$\frac{\vdash s : \mathit{Step} \qquad s \vdash e : \mathit{Noun}(T)}{s \vdash \mathtt{term}(e)/\mathtt{set}(e)/\mathtt{noun}(e) : \mathit{Term}(T)/\mathit{Set}(T)/\mathit{Noun}(T)}\ \text{CATEG-TERM/SET/NOUN}$$

$$\frac{\vdash s : \mathit{Step} \qquad s \vdash e : \mathit{Noun}(T) \qquad s \vdash e' : \mathit{Noun}(T') \qquad T \leq T'}{s \vdash \mathtt{adj}(e, e') : \mathit{Adj}(T, T')}\ \text{CATEG-ADJ}$$

$$\frac{\vdash s : \mathit{Step}}{s \vdash \mathtt{stat} : \mathit{Stat}}\ \text{CATEG-STAT} \qquad \frac{\vdash s : \mathit{Step} \qquad s \vdash c : \mathit{at}}{s \vdash \mathtt{dec}(c) : \mathit{Dec}(() \to \mathit{at})}\ \text{CATEG-DEC}$$

$$\frac{\vdash s : \mathit{Step}}{s \vdash v : v}\ \text{CATEG-VAR}$$

Category expressions are used in declarations. These category expressions set the category of the parameters and of the output of an identifier. Some category constructors (`term`, `noun`,`adj` and `set`) are parametrised by a noun expression.

Fig. 3. Categories (we use the symbol / to group the three similar rules)

part of a local scoping, a declaration/definition as atomic step in the body of a local scoping, a declaration as a parameter of an identifier. The first two are shared by definitions and declarations. The third one is declaration specific.

1. The first case is the presence of a declaration or a definition inside the flag of a local scoping. The introduced identifier is available in the step (and its sub-steps) covered by the flag-context. Here, an identifier `x` is declared in the context part of a local scoping. `x` is available in the part of this context that follows the declaration (3), and also in the body part (4) of the local scoping. But `x` is not available before being declared, in the preceding steps (1) as well as in the preceding part of the context (2) of the local scoping. `x` is also not available in the steps that follow the local scoping (5).

```
{ (1);
  { (2);
    x:term;
    (3) } |> { (4) };
  (5) }
```

2. The second case is a declaration or a definition as atomic step. The identifier is therefore available for all the following steps of the MathLang document. A declaration of a triangle is an atomic step of the sub-block of a block. The identifier `triangle` is not available before being declared (1) and (2) but is available in all what follows (3) an (4). The availability of `triangle` would have been identical if the declaration had been replaced by a definition.

```
{ (1);
  { (2);
    triangle:noun;
    (3) };
  (4) }
```

3. The last case is declarations as parameters of identifiers. If an identifier takes a declaration as a parameter, then the declared identifier is available

$$\frac{\begin{array}{c}\vdash s : \mathit{Step}\\ s \vdash ci : (at_1, \ldots, at_n) \to at \qquad \forall j \in [\![1 \ldots n]\!], \{s; e_1; \ldots; e_{j-1}\} \vdash e_j : at'_j\\ at' \notin cvar \qquad (at_1, \ldots, at_n) \to at \leq (at'_1, \ldots, at'_n) \to at' \ \mathit{satisfiable}\end{array}}{s \vdash ci(e_1, \ldots, e_n) : at'}\ \textsc{instance}$$

$$\frac{\begin{array}{c}\vdash s : \mathit{Step}\\ \{s;\ \mathit{self} : \mathit{Term}(T)\} \vdash s' : \mathit{Step} \qquad \forall i \in I(s'), \{s;\ \mathit{self} : \mathit{Term}(T);\ s'\} \vdash i : T(i)\end{array}}{s \vdash \texttt{Noun}\ \{s'\}\ : \mathit{Noun}(T)}\ \textsc{noun}$$

$$\frac{\begin{array}{c}\vdash s : \mathit{Step}\\ s \vdash e : \mathit{Noun}(T) \qquad T \leq T' \qquad \{s; \mathit{super} : \mathit{Term}(T); \mathit{self} : \mathit{Term}(T')\} \vdash s' : \mathit{Step}\\ \forall i \in I(s'), \{s; \mathit{super} : \mathit{Term}(T); \mathit{self} : \mathit{Term}(T'); s'\} \vdash i : T'(i)\end{array}}{s \vdash \texttt{Adj}\ (e)\ \{s'\}\ : \mathit{Adj}(T, T')}\ \textsc{adj}$$

$$\frac{\begin{array}{c}\vdash s : \mathit{Step} \qquad s \vdash e_1 : \mathit{Adj}(T_1, T'_1)\\ s \vdash e_2 : \mathit{Noun}(T_2)/\mathit{Set}(T_2)/\mathit{Term}(T_2) \qquad T_1 \leq T_2\end{array}}{s \vdash e_1 e_2 : \mathit{Noun}(T'_1 \uplus T_2)/\mathit{Set}(T'_1 \uplus T_2)/\mathit{Term}(T'_1 \uplus T_2)}\ \textsc{refinement}$$

$$\frac{\begin{array}{c}\vdash s : \mathit{Step} \qquad s \vdash e_1 : \mathit{Adj}(T_1, T'_1)\\ s \vdash e_2 : \mathit{Adj}(T_2, T'_2) \qquad T_1 \leq T'_2 \qquad T'_1 \leq T_2\end{array}}{s \vdash e_1 e_2 : \mathit{Adj}(T_1 \uplus (T_2 \ominus T'_1), T'_1 \uplus T'_2)}\ \textsc{adj-refinement}$$

$$\frac{\vdash s : \mathit{Step} \qquad s \vdash i_1 : \mathit{Noun}(T_1) \qquad s \vdash ci_2 : \mathit{Noun}(T_2) \qquad T_2 \leq T_1}{s \vdash i_1 \ll ci_2 : \mathit{Stat}}\ \textsc{sub-noun}$$

$$\frac{\vdash s : \mathit{Step} \qquad s \vdash i_1 : \mathit{Term}(T_1) \qquad s \vdash ci_2 : \mathit{Adj}(T_2, T'_2) \qquad T_2 \leq T_1}{s \vdash i_1 \ll ci_2 : \mathit{Stat}}\ \textsc{adj-term}$$

$$\frac{\vdash \{\overrightarrow{s}; \mathit{self} : at\} : \mathit{Step}}{\{\overrightarrow{s}; \mathit{self} : at\} \vdash \texttt{self} : () \to at}\ \textsc{self}$$

$$\frac{\vdash \{\overrightarrow{s}; \mathit{super} : at\} : \mathit{Step}}{\{\overrightarrow{s}; \mathit{super} : at\} \vdash \texttt{super} : () \to at}\ \textsc{super} \qquad\qquad \frac{\vdash s : \mathit{Step} \qquad l \in L(s)}{s \vdash \texttt{ref}\ l : \mathit{Step}}\ \textsc{ref}$$

The typing of the parameter expressions should satisfy the type of the identifier for the *instantiation of the identifier* (INSTANCE rule). In the NOUN rule, `self` is added to the context for the typing of the step defining the noun. In ADJ, both `self` and `super` are added. A *refinement* creates a noun expression from an adjective and a noun. The set of components required to use the adjective should be a subset of the set of components of the noun.

Fig. 4. Expressions (we use the symbol / to group the similar refinement rules)

for the following parameters. Let us illustrate this with the encoding of an expression with the universal quantifier. We declare an identifier `binder` with a declaration as second parameter. We also declare an identifier `operator` with three parameters. In an expression using these two identifiers, a variable `x` is declared. This identifier `x` is not available before being declared (1) and (2). `x` is available in the parameters of the `binder` that follows the declaration of `x` (3). Finally `x` is neither available in the remaining part of the expression (4) nor in the steps that follow (5).

$$\dfrac{\begin{array}{c}\vdash s \mathbin{\boldsymbol{:}} \mathit{Step} \qquad s \vdash i_1 \mathbin{\boldsymbol{:}} \mathit{Noun}(T_1) \qquad s \vdash ci_2 \mathbin{\boldsymbol{:}} \mathit{Noun}(T_2) \\ \mathsf{dom}(T_2) \subseteq \mathsf{dom}(T_1) \qquad \forall i \in \mathsf{dom}(T_2),\ T_1(i) \leq T_2(i)\end{array}}{s \vdash i_1 \ll ci_2 \mathbin{\boldsymbol{:}} \mathit{Stat}}\ \text{SUB-NOUN}$$

$$\dfrac{\begin{array}{c}\vdash s \mathbin{\boldsymbol{:}} \mathit{Step} \qquad s \vdash i_1 \mathbin{\boldsymbol{:}} \mathit{Term}(T_1) \qquad s \vdash ci_2 \mathbin{\boldsymbol{:}} \mathit{Adj}(T_2, T_2') \\ \mathsf{dom}(T_2) \subseteq \mathsf{dom}(T_1) \qquad \forall i \in \mathsf{dom}(T_2),\ T_1(i) \leq T_2(i)\end{array}}{s \vdash i_1 \ll ci_2 \mathbin{\boldsymbol{:}} \mathit{Stat}}\ \text{ADJ-TERM}$$

Fig. 5. Phrases

$$\dfrac{\begin{array}{c}\vdash s \mathbin{\boldsymbol{:}} \mathit{Step} \\ i \notin I(s) \qquad \forall j \in [\![1 \dots n]\!],\ s \vdash c_j \mathbin{\boldsymbol{:}} at_j \qquad s \vdash e \mathbin{\boldsymbol{:}} \mathit{Noun}(T)/\mathit{Set}(T)\end{array}}{s \vdash i(c_1, \dots, c_n) : e \mathbin{\boldsymbol{:}} \mathit{Dec}((at_1, \dots, at_n) \to \mathit{Term}(T))}\ \text{DEC-1}$$

$$\dfrac{\vdash s \mathbin{\boldsymbol{:}} \mathit{Step} \qquad i \notin I(s) \qquad \forall j \in [\![1 \dots n]\!], s \vdash c_j \mathbin{\boldsymbol{:}} at_j \qquad s \vdash c \mathbin{\boldsymbol{:}} at}{s \vdash i(c_1, \dots c_n) : c \mathbin{\boldsymbol{:}} \mathit{Dec}((at_1, \dots, at_n) \to at)}\ \text{DEC-2}$$

$$\dfrac{\begin{array}{c}\vdash s \mathbin{\boldsymbol{:}} \mathit{Step} \qquad i \notin DI(s) \qquad \forall j, k \in [\![1 \dots n]\!], j \neq k \Rightarrow i_j \neq i_k \\ \forall j \in [\![1 \dots n]\!], s \vdash i_j \mathbin{\boldsymbol{:}} () \to at_j \qquad \cup_{j \in [\![1 \dots m]\!]} i_j = dI(s) \setminus \{i\} \\ s \vdash e \mathbin{\boldsymbol{:}} at \qquad \mathit{if}\ \ i \in dI(s)\ \mathit{then}\ s \vdash i \mathbin{\boldsymbol{:}} (at_1, \dots, at_n) \to at\end{array}}{s \vdash i(i_1, \dots, i_n) := e \mathbin{\boldsymbol{:}} \mathit{Def}((at_1, \dots, at_n) \to at)}\ \text{DEF}$$

$$\dfrac{\begin{array}{c}\vdash s \mathbin{\boldsymbol{:}} \mathit{Step} \qquad \mathit{if}\ \ i \in I(s)\ \mathit{then}\ s \vdash i \mathbin{\boldsymbol{:}} (at_1, \dots, at_n) \to at \\ \forall j \in [\![1 \dots n]\!], s \vdash e_j \mathbin{\boldsymbol{:}} at_j \qquad s \vdash e \mathbin{\boldsymbol{:}} at\end{array}}{s \vdash i(e_1, \dots, e_n) := e \mathbin{\boldsymbol{:}} \mathit{Def}((at_1, \dots, at_n) \to at)}\ \text{DEF-CASE}$$

Declarations and definitions introduce new identifiers. For a declaration, the category of the identifier could be explicitly expressed (DEC-2 rule) or an expression could be given that represents the elementhood of the identifier (DEC-1 rule). For a definition, the parameters could either identifiers (DEF rule) or expressions for definition by matching (DEF-CASE rule).

Fig. 6. Declarations and definitions

```
{ binder (term, dec(term), term): term;
  operator (term, term, term): term;
  [...] a((1), binder ((2), x:term , (3)), (4)) [...];
  (5) }
```

5 Conclusion and Future Work

To have MathLang being adopted by mathematicians is our aspiration. We are convinced that providing yet another concrete syntax will never make a mathematical language widely used. We are therefore focusing on interfacing MathLang with user-friendly tools. We are currently embedding MathLang concepts and type checking in the scientific editor TEXmacs with the development of a MathLang-plugin. This plugin is making full reuse of the mechanisms for rendering MathLang texts in their original CML forms. These mechanisms were

$$\frac{\vdash s_1 \mathbin{\vdots} \mathit{Step} \qquad s_1 \vdash s_2 \mathbin{\vdots} \mathit{Step} \qquad \{s_1; s_2\} \vdash \{\overrightarrow{s}\} \mathbin{\vdots} \mathit{Step}}{s_1 \vdash \{s_2; \overrightarrow{s}\} \mathbin{\vdots} \mathit{Step}}\ \textsc{step-composition}$$

$$\frac{\vdash s \mathbin{\vdots} \mathit{Step} \qquad s \vdash s' \mathbin{\vdots} \mathit{Step} \qquad \{s; s'\} \vdash s'' \mathbin{\vdots} \mathit{Step}}{s \vdash s' \rhd s'' \mathbin{\vdots} \mathit{Step}}\ \textsc{local-scoping}$$

$$\frac{\vdash s \mathbin{\vdots} \mathit{Step} \qquad s \vdash p \mathbin{\vdots} \mathit{Stat}/\mathit{Dec}(t)/\mathit{Def}(t)}{s \vdash p \mathbin{\vdots} \mathit{Step}}\ \textsc{atomic-step}$$

$$\frac{\vdash s \mathbin{\vdots} \mathit{Step}}{s \vdash \mathit{self} : t \mathbin{\vdots} \mathit{Step}}\ \textsc{self-marker} \qquad \frac{\vdash s \mathbin{\vdots} \mathit{Step}}{s \vdash \mathit{super} : t \mathbin{\vdots} \mathit{Step}}\ \textsc{super-marker}$$

$$\frac{}{\vdash \{\} \mathbin{\vdots} \mathit{Step}}\ \textsc{empty-step} \qquad \frac{\vdash s \mathbin{\vdots} \mathit{Step} \qquad \{s; l : \mathit{Label}\} \vdash s' \mathbin{\vdots} \mathit{Step}}{s \vdash \texttt{label}\ l\ s' \mathbin{\vdots} \mathit{Step}}\ \textsc{label}$$

Only well typed statements, declarations or definitions could be *phrases* (ATOMIC-STEP rule). Phrases stand for atomic steps *Blocks.* Each element of a block should be a valid step in the context formed by the preceding elements of this particular element (BLOCK rule). As presented in section 2, a *local scoping* builds a step as a context for another step (LOCAL-SCOPING rule). See example 1 for examples of *steps.*

Fig. 7. Steps

presented in [10]. In parallel, we are implementing the MathLang's new features presented in this paper. These new features will be tested on already realised translations. New translations will benefit from the assistance of the editor and will gain in expressiveness with the new object oriented features of the language.

Concerning the language definition part, we believe that more flexible abstraction mechanisms could be added. For this purpose we will investigate the possibility to integrate in our system the notion of traits, a new member of the object-oriented programming. We would also like to relate our low level encoding of groups to a previous work. In the computer algebra system Focal [15], *species* and *collections* are object oriented structures that have been used to create an algebraic hierarchy. Finally we would be interested in comparing MathLang's nouns and adjectives with concepts and roles of Deductive Logics (DLs) [1] and in investigating existing research in mixin extension of DLs.

In this paper we proposed to capture the structure of mathematical with object-oriented features. We exposed the relevance of this approach with two examples and presented a type system for MathLang that incorporates these features. This work is a step of our larger aim to consider the encoding of the natural language parts when computerizing mathematical text.

References

1. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.
2. Nicolas Bourbaki. *Elements of Mathematics - Algebra*, volume II. Hermann, 1974.

3. Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proc. Int'l Conf. Computer Languages*, pages 282–290, 1992.
4. N.G. de Bruijn. The mathematical language Automath, its usage, and some of its extensions. *Lecture Notes in Mathematics*, 125:29–61, 1970.
5. N.G. de Bruijn. The mathematical vernacular, a language for mathematics with typed sets. In *Workshop on Programming Logic*, 1987.
6. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL '98*, pages 171–183, 1998.
7. Heath. *The 13 Books of Euclid's Elements*. Dover, 1956.
8. G. Jojgov and R. Nederpelt. A path to faithful formalizations of mathematics. In *MKM 2004*, volume 3119 of *LNCS*, pages 145–159, 2004.
9. G. Jojgov, R. Nederpelt, and M. Scheffer. Faithfully reflecting the structure of informal mathematical proofs into formal type theories. In *MKM Symposium 2003*, volume 93 of *ENTCS*, pages 102–117, 2004.
10. F. Kamareddine, M. Maarek, and J. B. Wells. Flexible encoding of mathematics on the computer. In *MKM 2004*, volume 3119 of *LNCS*, pages 160–174, 2004.
11. F. Kamareddine, M. Maarek, and J. B. Wells. MathLang: Experience-driven development of a new mathematical language. In *MKM Symposium 2003*, volume 93 of *ENTCS*, pages 138–160, 2004.
12. F. Kamareddine and R. Nederpelt. A refinement of de Bruijn's formal language of mathematics. *Journal of Logic, Language and Information*, 13(3):287–340, 2004.
13. Michael Kohlhase. Semantic markup for TEX/LATEX. Mathematical User-Interfaces Workshop, 2004.
14. Edmund Landau. *Foundations of Analysis*. Chelsea, 1951.
15. Virgile Prevosto, Damien Doligez, and Thérèse Hardin. Algebraic structure and dependent records. In *TPHOLs 2002*, volume 2410 of *LNCS*, pages 298–313, 2002.
16. Mark Scheffer. Formalizing Mathematics using Weak Type Theory. Master's thesis, Technische Universiteit Eindhoven, 2003.
17. F. Wiedijk. Formal proof sketches. In *TYPES 2003*, volume 3085 of *LNCS*, pages 378–393, 2004.