Supporting User-Defined Notations when Integrating Scientific Text-Editors with Proof Assistance Systems

Serge Autexier[1,2], Armin Fiedler[2], Thomas Neumann[2], and Marc Wagner[2]

[1] German Research Center for Artificial Intelligence (DFKI GmbH), Saarbrücken, Germany,
`autexier@dfki.de`

[2] FR 6.2 Informatik, Saarland University, Saarbrücken, Germany
`{autexier|fiedler|tneumann|wagner}@ags.uni-sb.de`

Abstract. In order to foster the use of proof assistance systems, we integrated the proof assistance system ΩMEGA with the standard scientific text-editor TEXMACS. We aim at a document-centric approach to formalizing and verifying mathematics and software. Assisted by the proof assistance system, the author writes her document entirely inside the text-editor in a language she is used to, that is a mixture of natural language and formulas in LATEX style. We present a basic mechanism that allows the author to define her own notation inside a document in a natural way, and use it to parse the formulas written by the author as well as to render the formulas generated by the proof assistance system. To make this mechanism effectively usable in an interactive and dynamic authoring environment, we extend it to efficiently accommodate modifications of notations, to track dependencies to ensure the right order of notations and formulas, to use the hierarchical structure of theories to prevent ambiguities, and to reuse concepts together with their notation from other documents.

1 Introduction

The vision of a powerful mathematical assistance environment that provides computer-based support for most tasks of a mathematician has stimulated new projects and international research networks in recent years across disciplinary boundaries. Even though the functionalities and strengths of proof assistance systems are generally not sufficiently developed to attract mathematicians on the edge of research, their capabilities are often sufficient for applications in e-learning and engineering contexts. However, a mathematical assistance system that shall be of effective support has to be highly user oriented. We believe that such a system will only be widely accepted by users if the communication between human and machine satisfies their needs, in particular only if the extra time spent on the machine is by far compensated by the system support. One aspect of the user-friendliness is to integrate formal modeling and reasoning tools with software that users routinely employ for typical tasks in order to promote the use of formal logic based techniques.

One standard activity in mathematics and areas that are based on mathematics is the preparation of documents using some standard text preparation system like LATEX. TEXMACS [10] is a scientific text-editor in the WYSIWYG paradigm that provides professional type-setting and supports authoring with powerful macro definition facilities like those in LaTeX. As a first step towards assisting the authoring of mathematical documents, we integrated the proof assistance system ΩMEGA into TeXMACS using the generic mediator PLATΩ [12]. In this setting the formal content of a document must be amenable to machine processing, without imposing any restrictions on how the document is structured, on the language used in the document, or on the way the document can be changed. The PLATΩ system [11] transforms the representation of the formal content of a document into the representation used in a proof assistance system and maintains the consistency between both representations throughout the changes made on either side.

Such an integrated authoring environment should allow the user to write her mathematical documents in the language she is used to, that is a mixture of natural language and formulas in LaTeX style with her own notation. To understand the meaning of the natural language parts in a mathematical document we currently rely on annotations for the document structure that must be provided manually by the user. Although it might still be acceptable for an author to indicate the macro-structures like theories, definitions and theorems, writing annotated formulas (e.g. "`\F{in}{\V{x},\F{cup}{\V{A},\V{B}}}`" instead of "`x \in A \cup B`") is definitely not. Aiming at a document-centric approach to formalizing mathematics, we present a mechanism that allows authors to define their own notation and to use it when writing formulas within the same document. Furthermore, this mechanism enables the proof assistance system to access the formal content and use the same notation when presenting formulas to the author.

The paper is organized as follows: Section 2 presents the annotation language for documents of the PLATΩ system, in particular for formulas. Inspired by notational definitions in text-books, we then present the means the author should have to define notations. The goal consists of starting from such notations to obtain an *abstraction* parser that allows to read formulas using that notation and also a corresponding *rendering* parser to render formulas generated by the proof assistance system. Section 3 describes how the notational definitions can automatically be transformed into grammar rules defining the *abstraction* and *rendering* parsers, which are created by a parser generator that allows to integrate arbitrary disambiguators. Section 4 presents a basic mechanism how to accommodate efficiently modifications of the notations. In Section 5 we extend the basic framework to restrain ambiguities, allow for the redefinition of notations and use notations defined in other documents. We discuss related works in Section 6 before concluding in Section 7.

Presentational convention: The work presented in this paper has been realized in TeXMACS. Although the TeXMACS markup-language is analogous to LaTeX-macros, one needs to get used to it: For instance a macro application like `\frac{A}{B}` in LaTeX becomes `<frac|A|B>` in TeXMACS-markup. Assuming that most readers are more familiar with LaTeX than with TeXMACS, we will use a LaTeX-syntax for sake of readability.

2 Towards Dynamic Notation

The PLATΩ system supports users to interact with a proof assistance system from inside the text-editor TeXMACS by offering service menus and by propagating changes

Element	**Arguments**
`\F`	{*name*}{`\B`?, (`\F` | `\V` | `\S`)⋆ }
`\B`	{`\V`+ }
`\V`	{*name*, (`\T` | `\TX` | `\TF`)?}
`\S`	{*name*}
`\T`	{*name*}
`\TX`	{(`\T` | `\TX` | `\TF`), (`\T` | `\TX` | `\TF`) }
`\TF`	{(`\T` | `\TX` | `\TF`), (`\T` | `\TX` | `\TF`) }

Table 1. Grammar of PLATΩ's *Formula Annotation Language*

of the document to the system and vice versa. Mediating between a text-editor and a proof assistance system requires to extract the formal content of a document, which is already a challenge in itself if one wants to allow the author to write in natural language without any restrictions. Therefore we currently use a semantic annotation language to semantically annotate different parts of a document. The annotations can be nested and subdivide the text into dependent theories that contain definitions, axioms, theorems and proofs, which themselves consist of proof steps like for instance subgoal introduction, assumption or case split. The annotations are a set of macros predefined in a TEXMACS style file and must be provided manually by the author (see [11] for details). We were particularly cautious that adding the annotations to a text does not impose any restrictions to the author about how to structure her text. Note that for the communication with the proof assistance system, also the formulas must be written in a fully annotated format whose grammar is shown in Tab. 1.

`\F{name}{args}` represents the application of the function `name` to the given arguments `args`. `\B{vars}` specifies the variables `vars` that are bound by a quantifier. A variable `name` is denoted by `\V{name}` and may be optionally typed by `\V{name,type}`. A type `name` is represented by `\T{name}`. Complex types are composed using the function type constructor $\rightarrow$ represented by `\TF{type1,type2}`, or the operator $\times$ represented by `\TX{type1,type2}` as syntactic sugar for argument types. Finally, a symbol `name` is denoted by `\S{name}`. For instance, the formula $x \in A \cap (B \cup C)$ is represented by the fully annotated form `\F{in}{\V{x},\F{cap}{\V{A},F{cup}{\V{B}, V{C}}}}` and the quantified formula $\forall x.\ x = x$ as `\F{forall}{\B{\V{x}},\F{=}{ \V{x},\V{x}}}`. In many cases type reconstruction allows to dertermine the type of a variable, and therefore typing variables is optional in our system.

Currently the macro-structures like theories, definitions, theorems, and proof steps must be annotated manually by the user. However, it is not acceptable to require to write formulas in fully annotated form. This motivates the need for an *abstraction* parser that converts formulas in LaTeX syntax into their fully annotated form. Furthermore, we also need a *rendering* parser to convert fully annotated formulas obtained from the proof assistance system into LaTeX-formulas and using the user-defined notation. In the future, we plan on the one hand to combine our approach with techniques in the tradition of using a *mathematical vernacular* (e.g. MATHLANG [4]) and on the other hand to use natural language analysis techniques for the semi-automatic annotation of the document structure, e.g. to automatically detect macro-structures.

The usual software engineering approach would be to write grammars for both directions and integrate the generated parsers into the system. Of course, this method is highly efficient but the major drawback is obvious: the user has to maintain the grammar files together with her documents. In our document-centric philosophy, the only source of knowledge for the mediator and the proof assistance system should be the document in the text-editor.

Therefore, instead of maintaining special grammar files for the parser, the idea of dynamic notation is to start from basic abstraction and rendering grammars for types and formulas, where only the base type *bool*, the complex type constructors $\rightarrow, \times$ and the logic operators $\forall, \exists, \lambda, \top, \bot, \wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$ are predefined. Based on that initial grammar the definitions and notations occurring in the document are analyzed in order to extend incrementally both grammars for dealing with new symbols, types and operators. The scope of a notation should thereby respect the visibility of its defining symbol or type, i.e. the transitive closure of dependent theories. Finally, all formulas are parsed using their theory specific *abstraction* parser.

Notations defined by authors are typically not specified as grammar rules. Therefore, we first need a user friendly WYSIWYG method to define notations and to automatically generate grammar rules from it. Looking at standard mathematical textbooks, one observes sentences like *"Let x be an element and A be a set, then we write $x \in A$, x is element of A, x is in A or A contains x."*. Supporting this format requires the ability to locally introduce the variables x and A in order to generate grammar rules from a notation pattern like $x \in A$. Without using a linguistic database, patterns like *x is in A* are only supported as pseudo natural language. Beside that, the author should be able to declare a symbol to be right- or left-associative as well as precedences of symbols.

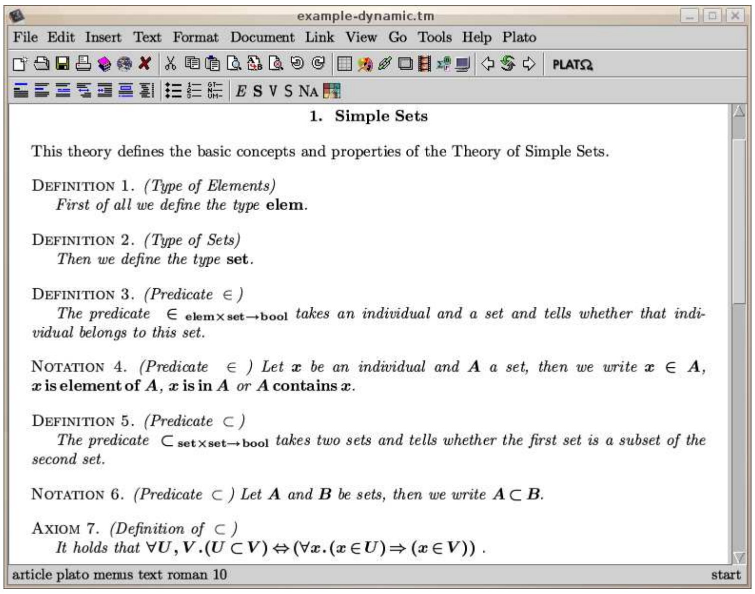
We introduce the following annotation format to define the operator $\in$ and to introduce multiple alternative notations for $\in$ as closely as possible to the textbook style.

```
\begin{definition}{Predicate $\in$}
  The predicate \concept{\in}{elem \times set \rightarrow bool}
  takes an individual and a set and tells whether that
  individual belongs to this set.
\end{definition}
```

A definition may introduce a new type by `\type{name}` or a new typed symbol by `\concept{name}{type}`. We allow to group symbols to simplify the definition of precedences and associativity. By writing `\group{name}` inside the definition of a symbol, this particular symbol is added to the group `name` which is automatically created if it does not exist. Any new concept is first introduced as a prefix symbol. This can be changed by declaring concept specific notations.

```
\begin{notation}{Predicate $\in$}
  Let \declare{x} be an individual and \declare{A} a set,
  then we write \denote{x \in A}, \denote{x is element of A},
  \denote{x is in A} or \denote{A contains x}.
\end{notation}
```

A notation may contain some variables declared by `\declare{name}` as well as the patterns written as `\denote{pattern}`. Furthermore, by writing `\left{name}` or `\right{name}` inside the notation one can specify a symbol or group of symbols to be left or right associative. Finally, precedences between symbols or groups are defined by `\prec{name1,...,nameN}`, which partially orders the precedence of these symbols


example-dynamic.tm
File Edit Insert Text Format Document Link View Go Tools Help Plato
PLATΩ
E S V S NA

1. Simple Sets

This theory defines the basic concepts and properties of the Theory of Simple Sets.

DEFINITION 1. *(Type of Elements)*
First of all we define the type **elem**.

DEFINITION 2. *(Type of Sets)*
Then we define the type **set**.

DEFINITION 3. *(Predicate* $\in$ *)*
The predicate $\in_{\mathbf{elem}\times\mathbf{set}\to\mathbf{bool}}$ *takes an individual and a set and tells whether that individual belongs to this set.*

NOTATION 4. *(Predicate* $\in$ *) Let* x *be an individual and* A *a set, then we write* $x \in A$, x **is element of** A, x **is in** A *or* A **contains** x.

DEFINITION 5. *(Predicate* $\subset$ *)*
The predicate $\subset_{\mathbf{set}\times\mathbf{set}\to\mathbf{bool}}$ *takes two sets and tells whether the first set is a subset of the second set.*

NOTATION 6. *(Predicate* $\subset$ *) Let* A *and* B *be sets, then we write* $A \subset B$.

AXIOM 7. *(Definition of* $\subset$ *)*
It holds that $\forall U, V.(U \subset V) \Leftrightarrow (\forall x.(x \in U) \Rightarrow (x \in V))$.

article plato menus text roman 10 start

Fig. 1. Annotated TEXMACS document with dynamic notation in text mode

and groups of symbols from low to high. Please note that a notation is related to a specific definition by refering its name, in our example `Predicate $\in$`.

Fig. 1 shows how the above example definition and notation appear in a TEXMACS document. Using a keyboard shortcut the author can easily switch into a so-called "box-mode" that visualizes the semantic annotations contained in the document (cf. Fig. 2, p.6). The author is free to develop the document in either view.

3 Creating Parsers from User-Defined Notations

We first present how the grammar rules for *abstraction* and *rendering* are obtained from a document and then briefly describe the parser generator that is a slight variant of standard implementations.

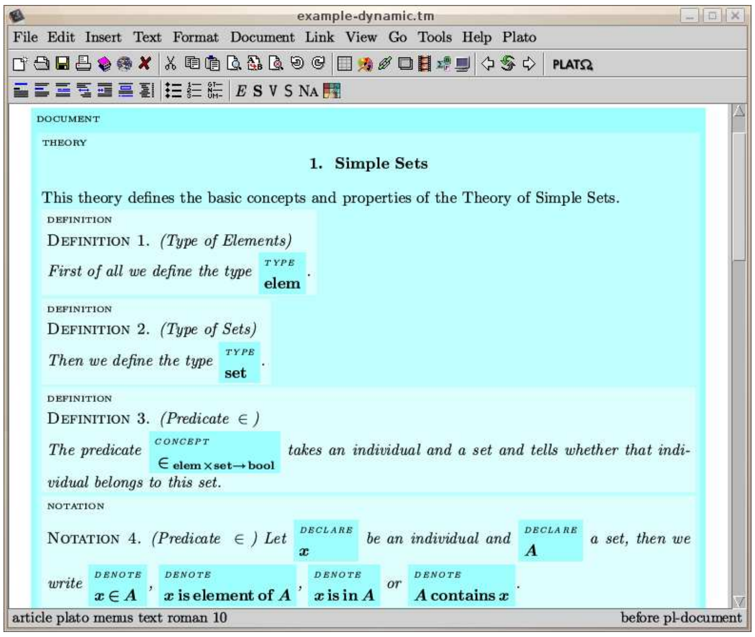
3.1 Obtaining the Grammar Rules

Starting the processing of a semantically annotated document, as for example the document shown in Fig. 1, all surrounding natural language parts in the document are removed and the *abstraction* and *rendering* parsers and scanners are initialized with the initial grammars for types and formulas. The grammar rules can be divided into rules for types, symbols and operator applications. The syntax of the rules is

```
NONTERMINAL ::= (TERMINAL|NONTERMINAL)+ --> PRODUCTION
```

and is best explained using an example rule:

```
APPLICATION ::= FORMULA.1 "\wedge" FORMULA.2
            --> \F{and}{FORMULA.1, FORMULA.2}
```


example-dynamic.tm
File Edit Insert Text Format Document Link View Go Tools Help Plato
PLATΩ
E S V S NA

DOCUMENT

THEORY

1. Simple Sets

This theory defines the basic concepts and properties of the Theory of Simple Sets.

DEFINITION

DEFINITION 1. *(Type of Elements)*

First of all we define the type TYPE **elem**.

DEFINITION

DEFINITION 2. *(Type of Sets)*

Then we define the type TYPE **set**.

DEFINITION

DEFINITION 3. *(Predicate $\in$)*

The predicate CONCEPT $\in_{\mathbf{elem}\times\mathbf{set}\to\mathbf{bool}}$ *takes an individual and a set and tells whether that individual belongs to this set.*

NOTATION

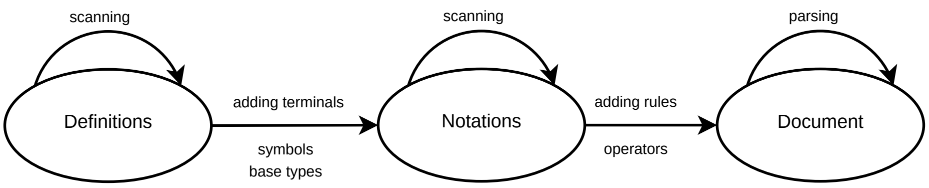
NOTATION 4. *(Predicate $\in$) Let* DECLARE x *be an individual and* DECLARE A *a set, then we write* DENOTE $x \in A$, DENOTE x **is element of** A , DENOTE x **is in** A *or* DENOTE A **contains** x .

article plato menus text roman 10 before pl-document

Fig. 2. Annotated TEXMACS document with dynamic notation in box mode

This is a rule for the non-terminal symbol `APPLICATION` used for any kind of application of an operator. If it could successfully recognize two chunks of text in the class of `FORMULA` and that are separated by `"\wedge"`, it substitutes the obtained results for `FORMULA.1` and `FORMULA.2` in `\F{and}{FORMULA.1, FORMULA.2}` to create the parsing result.

The goal of processing the document is to produce a set of grammar rules for the respective grammars from the notational definitions given in the text. A top-down approach, that processes each definition or notation on its own, extending the grammars and recompiling the parsers before processing the next element, is far too inefficient for real time usage due to the expensive parser generation process[3]. The following procedure tries to minimize the amount of parser generations as much as possible.



[3] The parser generator is currently implemented in Scheme to be part of TEXMACS and is not compiled. This causes, for instance, the parser generation for the example document to take $\approx 1min$.

1. Phase: All definitions are processed sequentially. For each definition the name of the introduced type or symbol is added to both grammars.
2. Phase: All notations are processed sequentially. For each notation the introduced patterns are analyzed to generate rules for both grammars.
3. Phase: *Abstraction* and *rendering* parsers are rebuilt and all formulas are processed.

Processing Definitions: A definition introduces either a type by `\type{name}` or a symbol by `\concept{name}{type}`. In both cases, the `name` is added as terminal to the *abstraction* scanner (if not yet included). Furthermore, we generate a fresh *internal name* which will be the name of the type (or symbol) communicated to the proof assistance system. The reason for that is that `name` can be in a syntax not accepted by the proof assistance system. The internal name is alphanumerical and is added to the *rendering* scanner. Then we extend the *abstraction* grammar by a production rule for types (or symbol) that converts `name` into that *internal name* and vice-versa for the *rendering* grammar. Overloading of symbol names is only allowed if their types are different. Please note that the scanner has always to be rebuilt when a new terminal is added to the grammar.

Example 1. In the definition of $\in$ (p.4) we have the following symbol declaration: `\concept{\in}{elem \times set \to bool}`. The `name` is `\in` and assume the internal name being `in`. These are added to the scanners. Furthermore the following rules are added to the *abstraction* and *rendering* grammars respectively:

– `SYMBOL ::= "\in" --> "in"` is added to the *abstraction* grammar
– `SYMBOL ::= "in"  --> "\in"` is added to the *rendering* grammar

The type information in the symbol declaration is only processed in the third phase because we first have to collect all type declarations. This also complies with future extensions towards dependent types. Our system currently supports only simple types.

Processing Notations: A notation defines one or more alternative notations for some symbol. The author can introduce local variables by `\declare{x1}`,...,`\declare{xN}` and use them in the patterns defining the different notations: `\denote{pattern1}`,..., `\denote{patternM}`.

Example 2. As an example consider our running example from p. 4:

```
Let \declare{x} be an individual and \declare{A} a set,
then we write \denote{x \in A}, \denote{x is element of A},
\denote{x is in A} or \denote{A contains x}.
```

We impose that the ordering in which the variables are declared by `\declare` complies with the domain of the associated operator, i.e. x is the first argument of `\in` and A the second.

First of all, the *abstraction* scanner is locally extended by the terminals for the local variables `x1`, ..., `xn`. Then each notation pattern is tokenized by the scanner, that returns a list of terminals including new terminals for unrecognized chunks. For instance, in our example above the scanner knows the terminals for the local variables `x` and `A` when tokenizing "`x is element of A`". The unknown chunks are `is`, `element` and `of`, that are added on the fly. This behavior of the scanner is non-standard, but is an essential feature to efficiently accommodate new notations. More details about the scanner are presented at the end of this section.

A notation pattern is only accepted if all declared argument variables are recognized by the scanner, namely all `x1`, ..., `xn` occur in the pattern. For every notation pattern, the *abstraction* grammar is extended by a production rule for function application that converts the notation *pattern* into the semantic function application with respect to the argument ordering. To this end we must modify the pattern by replacing the occurrences of the local variables by the non-terminals `FORMULA.1` ... `FORMULA.N` and add the *abstraction* grammar rule

```
APPLICATION ::= pattern1' --> \F{f}{FORMULA.1, ..., FORMULA.N}
```

where `pattern1'` is the modified pattern.

For instance, the above pattern `[x "is" "element" "of" A]` is transformed into `[FORMULA.1 "is" "element" "of" FORMULA.2]` and we obtain the following grammar rule:

```
APPLICATION ::= FORMULA.1 "is" "element" "of" FORMULA.2
                --> \F{f}{FORMULA.1, FORMULA.2}
```

For the *rendering* grammar we have the choice which pattern to use to render the terms. We currently just take the first possibility. The *rendering* grammar is then extended by the production rule

```
APPLICATION ::= \F{f}{FORMULA.1, FORMULA.2} --> pattern1'
```

For our running example we get the *rendering* grammar rule

```
APPLICATION ::= \F{f}{FORMULA.1, FORMULA.2}
            --> FORMULA.1 "is" "element" "of" FORMULA.2
```

Note that the notation pattern can permute the arguments of the symbol, as would be the case when using the pattern `[A contains x]`. All *abstraction* grammar rules of a symbol are grouped together in order to support dependency tracking.

Additionally the author can define the symbol `name` to be left- or right-associative by `\left{name}` or `\right{name}` as well as the precedence of operators by using `\prec{name1}`,...`{nameN}` where `namei` is the symbol name given in the definitions. This declares the relative precedence of these symbols, where `namei` is lower than `name(i+1)`.

Processing Formulas: The generated *abstraction* parser is finally used to parse the type information in symbol declarations and all formulas. The parser returns the fully annotated version of the types and formulas and also the list of grammar rules used. These rules are stored along with the type or formula and can serve to detect dependencies between definitions, notations and formulas. When the *abstraction* parsing process returns more than one possible reading, the author must advise which possibility is retained.

How these situations can be reduced to a minimum is discussed in the next Section and in Section 5.

The *rendering* parser is used to convert fully annotated formulas generated by the proof assistance system into LaTeX formulas for the text-editor. Again the grammar rules used are saved to allow for tracking the dependencies.

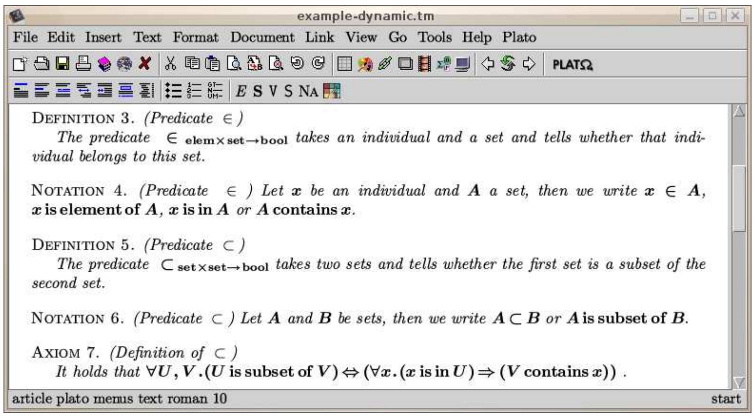
The Modified Scanner. The presented procedure to analyze the patterns in notational definitions makes extensive use of a modified scanner algorithm that returns all tokens including unknown chunks. The scanner has been implemented such that it guarantees the tokenizing of the longest possible prefixes. A hard wired scanner couldn't be used because the alphabet of the language is unknown. By using the standard scanner generation algorithm described in [14] a deterministic finite automaton is directly generated out of the given grammar without generating a non-deterministic finite automaton. Furthermore, the scanner can be built and used independently of the parser. Due to the small size of the automaton, the generation of a scanner is relatively fast with respect to grammar extensions.

3.2 Generating the Abstraction and Rendering Parsers

In this section we describe the parser generator used to create the *abstraction* and *rendering* parser from the collected grammar rules. The main features are that it returns all possible readings that are due to ambiguities in the grammar and returns for each reading the list of grammar rules used. In order to eliminate possible but incorrect readings as early as possible, an external function can be specified that is called at runtime to eliminate ambiguities.

The LALR parsers are generated using well-known algorithms [13,14] together with the usual action- and goto-tables. Since we don't restrict our input grammar too much, i.e. we allow also non-LALR grammars, we have to disambiguate the input grammar and provide a handling for non resolvable ambiguities. The disambiguation of the grammar is performed using standard methods from the BISON system[4]. In case there are still ambiguities remaining in the grammar, the parser allows to define an external callback function that is used at runtime to rule out possible readings that result from ambiguities. Thus it is possible to integrate a so-called "refiner" [3], which uses type reconstruction to filter the well-typed readings from all alternatives. Furthermore, since we are in an interactive setting, we could ask the user to resolve the remaining ambiguities in contrast to situations where there is no possibility of user feedback. However, that has still to be implemented in the PLATΩ system. If no external callback is defined, the parser splits itself by default into multiple subparsers, such that all possible readings are returned at the end. It has to be mentioned that the runtime of a splitted parser increases exponentially if the ambiguities are not completely removed. Since the formulas that need to be parsed in practice are usually not too large, we don't think this really poses a problem.

[4] `http://www.gnu.org/software/bison/`


example-dynamic.tm
File Edit Insert Text Format Document Link View Go Tools Help Plato
PLATΩ
article plato menus text roman 10
start

DEFINITION 3. *(Predicate $\in$)*
The predicate $\in_{\mathbf{elem}\times\mathbf{set}\to\mathbf{bool}}$ *takes an individual and a set and tells whether that individual belongs to this set.*

NOTATION 4. *(Predicate $\in$) Let* $\boldsymbol{x}$ *be an individual and* $\boldsymbol{A}$ *a set, then we write* $\boldsymbol{x} \in \boldsymbol{A}$, $\boldsymbol{x}$ **is element of** $\boldsymbol{A}$, $\boldsymbol{x}$ **is in** $\boldsymbol{A}$ *or* $\boldsymbol{A}$ **contains** $\boldsymbol{x}$.

DEFINITION 5. *(Predicate $\subset$)*
The predicate $\subset_{\mathbf{set}\times\mathbf{set}\to\mathbf{bool}}$ *takes two sets and tells whether the first set is a subset of the second set.*

NOTATION 6. *(Predicate $\subset$) Let* $\boldsymbol{A}$ *and* $\boldsymbol{B}$ *be sets, then we write* $\boldsymbol{A} \subset \boldsymbol{B}$ *or* $\boldsymbol{A}$ **is subset of** $\boldsymbol{B}$.

AXIOM 7. *(Definition of $\subset$)*
It holds that $\forall U, V.(U \textbf{ is subset of } V) \Leftrightarrow (\forall x.(x \textbf{ is in } U) \Rightarrow (V \textbf{ contains } x))$.

Fig. 3. Modified TEXMACS document with dynamic notation in text mode

4 Management of Change for Notations

The *abstraction* parser constructed so far is for one version of the document. When the author continues to edit the document, it may be modified in arbitrary ways, including the change of existing definitions and notations. Before the modified semantic content of the document is uploaded into the proof assistance system, we need to recompute the parsers and parse the formulas in the document. Always starting from scratch following the procedure described in the previous sections is not efficient and may jeopardize the acceptance by the author of the system if that process takes too long. Therefore there is a need for management of change for the notational parts of a document and those parts that depend on them. The management of change task has two aspects:

1. First, we must determine any modifications in the notational parts.
2. Second, we must adjust only those parts of the grammar that are affected by the determined modifications, adjust the parsers accordingly and then re-parse the formulas of the document.

Determining changes: Using the procedure from Section 3, we re-process all definitions and notations of the document and obtain a new set of grammar rules for each defined symbol. By caching these sets of rules for each symbols, we can determine how the grammar has changed using a differencing mechanism.

Adjusting the scanner and the parser: If the modification of the grammar is non-monotonic, i.e. some rules have been removed or changed, we currently have to re-compute the whole scanner and the parser from scratch using the procedure from Section 3.2. This is for instance the case if we change a notation for some symbol, e.g. if we replaced *“$A \subset B$”* by *“$B \supset A$”*, but not if we add an additional alternative notation for a symbol, like allowing *“A is subset of B”* in addition to *“$A \subset B$”* as shown in Fig. 3.

If the grammar is simply extended, we can optimize the creation of the new parser (recreating the scanner is fast anyway and there is no need to optimize that). In this case we can reuse our previous parser and extend it using a variant of the standard parser generation algorithm (cf. [13], p. 138ff.): Aside from the action- and goto-tables on which the parser operates, we also have access to the states of the automaton. Storing these data is expensive but enables the extension of the automaton. We first compute the closure of the start-state and then apply the standard algorithm with the states of the previous automaton. The computation of the algorithm produces either already existing states, modifications of existing states or completely new states. Using the identifier of a state and the entries in the goto-tables we determine if we have to create a new state or only modify an existing state. In case we have an existing, unmodified state, the algorithm returns immediately. Otherwise, we have to re-compute the automaton for the changed/new state. If an existing state is changed, then we must reuse the same identifier as before for the entries in the goto-tables, in order to guarantee the soundness of the transitions for those states that did not change.

Example 3. In our running example we add an additional alternative notation for the symbol $\subset$, i.e. we allow the notation *"A is subset of B"* in addition to *"$A \subset B$"*. Assuming the internal name for this symbol is `subset`, the following rule is added to the *abstraction* grammar:

```
APPLICATION ::= FORMULA.1 ‘‘is’’ ‘‘subset’’ ‘‘of’’ FORMULA.2
                  --> \F{subset}{FORMULA.1, FORMULA.2}
```

Re-parsing and re-rendering of formulas: Once the parser has been adjusted we need to re-parse those formulas the author has written or changed manually, e.g. the formula $\forall U,V.\ (U\ is\ subset\ of\ V) \Leftrightarrow (\forall x.\ (x\ is\ in\ U) \Rightarrow (V\ contains\ x))$ in the axiom of Fig. 3. Furthermore, we have to re-render those formulas that were generated by the proof assistance system. To this end we store the following information on formulas in the document: for each formula we have a flag indicating if it was generated by the proof assistance system, the corresponding fully annotated formula, and the set of grammar rules that were used for parsing or rendering that formula:

– If the formula was written by the author, the associated fully annotated formula and the grammar rules are the result of the *abstraction* parser.
– If a formula was generated by the proof assistance system, that formula is the result of *rendering* the fully annotated formula obtained from the proof assistance system. The stored grammar rules are those returned by the *rendering* parser.

Note that we do not prevent the author to edit a generated formula. As soon as the author edits such a formula, the flag attached to the formula is toggled to "user" and the cached fully annotated version and grammar rules are replaced during the next *abstraction* parsing of the formula.

The stored information is used to optimize the next parsing or rendering pass over the document: A formula is only parsed from scratch if at least one of the grammar rules used has been modified or deleted, or if either the user or the proof assistance system has changed the formula or the fully annotated formula.

This provides the basic mechanism that allows the author to define in a document her own notation that is used to extract the formal semantics, and that efficiently deals with modifications of the notation.

5 Ambiguities, Dependencies and Libraries

In order to enable a document-centric approach for formalizing mathematics and software, the added-values offered by the authoring environment must outweigh the additional burden imposed to the author compared to the amount of work for a non-assisted preparation of a document. In the following we present techniques to reduce the burden for the author by exploiting the theory structure contained in a document to reduce the ambiguities the author would have to deal with and also support the redefinition of notations. With respect to added-values, we provide checks if a notation is used before it has been introduced in a document and, most importantly, how we support an author to build on formalizations contained in other documents.

Ambiguities: The *abstraction* parser returns all possible readings and during the parsing process can try to make use of type-checking provided by the proof assistance system to eliminate possible readings that are not type-correct. It requires to provide type-information that depends on the context in which a formula is parsed, but even then it will require some amount of automated type-reconstruction. Therefore, there may always be situations in which we obtain more than one parsing result which requires the user to inspect the different possibilities and select the right one by menu interaction in the text-editor.

The logical context of a formula is determined by the theory it occurs in: The different parts of a document must be assigned to specific theories. New theories can be defined inside a document and build on top of other theories. The notion of theory is that of OMDOC [5] respectively development graphs [6]. One way to resolve an arising ambiguity is to provide the context of a formula to the *refiner*. The other possibility consists of having different parsers for different theories, hence avoiding some ambiguities that would arise when sticking to have a single parser.

Example 4. Consider a theory of the integers with multiplication with the notation "$x \times y$" and a completely unrelated theory about sets and Cartesian products with the same notation. This typically is a source of ambiguities that would require the use of type information to resolve the issue. Note that standard parser generators would not support the definition of two grammar rules that have the same pattern but different productions as would be necessary in this case. Using different parsers for different theories completely avoids that problem. However, when theories with overloaded notations are imported by another theory, e.g. a theory of Cartesian products of sets of integers, ambiguities may only be resolved using type information or user interaction.

From these observations we decided to not have a single parser for a whole document, but to exploit the theory structure contained in the document and allow for one parser for each theory. This entails that if we have a theory T that is included into two independent theories T_1 and T_2, then there is a parser for all three of them. Note that we could only have parsers for T_1 and T_2 and use either one when parsing a formula in the theory T. On the one hand, this would be more efficient for changes because we need only to maintain the parsers for T_1 and T_2, but on the other hand this would prevent the user to redefine in the context of say T_1 the notation for some symbol inherited from T, which is something quite common (and discussed in the next paragraph). Nevertheless, the drawback of our approach is that if some notation inside T is changed or added, all three dependent parsers must be adjusted rather than only two.

In that structured theory approach for parsers, the grammar of a parser for some theory consists of all rules obtained from notations for each symbol that is imported in that theory. The management of change mechanism from Section 4 is adapted in a straight-forward manner.

Redefining Notations. When importing a theory, we want to reuse the formal content, but possibly adapt the notation used to write formulas. This occurs less frequently inside a single document, but occurs very often when using a theory formalized in a different document. Since we linked the parsing (and hence the rendering) to the individual theories, we allow to redefine notations for symbols inherited from other theories. The grammar rules for a parser are determined by including for each imported symbol that notation that is closest in the import hierarchy of theories. If there are two such theories[5], a conflict is raised and the author aksed for advise which notation to use.

Dependencies. A parser and the associated renderer are attached to a theory and each position in the document belongs to a theory. Therefore, it is possible that within a specific theory, a formula uses the notation of some symbol although the definition of that notation only occurs afterwards in the document. We notice such situations by comparing the position in the document where grammar rules are defined and where they have been used to parse a formula. If we determine such a situation, we notify the author. The same problem can occur when rendering a formula: If the proof assistance system generates a formula with some concept c at a position that precedes the definition of the notation for c (but still in the same theory), the renderer uses the grammar rules before they are actually defined in the document. Sometimes the proof assistance system has no choice about where that formula is included: for instance if some proof steps are inserted into an existing partial proof that occurs before the definition of the notation. In other situations, for instance if the proof assistance system has used an automated theory exploration system[6] to derive new properties, we could try to determine an appropriate insertion position for these lemmas by inspecting the grammar rules used for *rendering*. However, our impression is that most authors would be upset if their document is rearranged automatically. Therefore we prefer to leave it to the user to move the text parts including surrounding descriptions into the appropriate places.

Note that another, much simpler dependency is that between the definition of a concept and the definition of its notation, that should not occur before the definition in the text. In that case we also simply notify the author.

[5] The theories can form an acyclic graph which may lead to a Nixon diamond scenario when determining grammar rules.

[6] For instance, MATHSAID [9] is connected to ΩMEGA.

Libraries. A library mechanism is the key prerequisite to support the development of large structured theories. We carry over that concept to the document-centric approach we are aiming at by extending the citation mechanism that is commonly used in documents. PLATΩ provides a macro to cite a document *semantically*, i.e. it will not only be included in the normal bibliography of the document, but the formalized content of the document is included. Currently, the document must be present in the file system and is included when the macro is evaluated and the process is recursive. Aside from the extracted formalizations that are sent to the proof assistance system to setup the background for the current document, we extract the notations contained in that document. Importing a theory defined in a semantically cited document into a theory of our current document, allows to determine the set of grammar rules using the same mechanism as described above for structured theories. Furthermore, since we allow the redefinition of notations, the author can redefine the notation for the concepts imported from cited documents, in order to adapt it to her preferences.

6 Related Work

Supporting specific mathematical notations is a major concern in all proof assistance systems. Wrt. to supporting the definition of new notations that are used for typesetting, the systems ISABELLE [7] and MATITA [2] are closest. ISABELLE comes with type-setting facilities of formulas and proofs for LaTeX and supports the declaration of the notation for symbols as prefix, infix, postfix and mixfix. Furthermore, it allows the definition of *translations* which are close to our style of defining notations. The main differences are: the notations are not defined in the LaTeX document but have to be provided in the input files of ISABELLE. Due to the batch processing paradigm of ISABELLE, there are no mechanisms to efficiently deal with modifications of the notation, which is crucial in our interactive authoring environment.

In the context of MATITA Padovani and Zacchiroli also proposed a mechanism of *abstraction* and *rendering* parsers [8] that are created from notational equations which are comparable to the grammar rules we generate from the notational definitions. Their mechanism is mainly devoted to obtain MathML representations [1] where a major concern also is to maintain links between the objects in MathML to the internal objects. Similar to ISABELLE, the notations must be provided in input files of MATITA that are separate from the actual document. Also, they do not consider the effect of changing the notations and to efficiently adjust the parsers.

7 Conclusion

In order to enable a document-centric approach for formalizing mathematics and software, the added-values offered in an assisted authoring environment must outweigh the additional burden imposed to the author compared to the amount of work for a non-assisted preparation of a document. One step in that direction is to give the freedom to define and use her own notation inside a document back to the author. In this paper we presented a mechanism that enables the author to define her own notation in a natural way in the text-editor $\text{T}_\text{E}\text{X}_\text{MACS}$ while being able to get support from the proof assistance system, such as type checking, proof checking, interactive and automatic proving, and automatic theory exploration. The notations are used to parse formulas written by the user in the LaTeX-style she is used to, as well as to render the formulas produced by the proof assistance system. Ambiguities are reduced by allowing one parser for each defined theory in the document and by integrating type-checking to resolve remaining ambiguities during the parsing process. The structure of theories also form the basis to include formalizations and notations defined in other documents. Finally, we developed maintenance techniques to accommodate the interactive and dynamic process of preparing a document by simultaneously reducing the amount of work for the user. Future work will consist of supporting LaTeX-documents and using OMDoc to exchange the formalized content and notations contained in documents of different formats.

References

1. Mathematical Markup Language (MathML) Version 2.0. W3c recommendation 21 february 2001. Technical report, `http://www.w3.org/TR/MathML2`, 2003.
2. A.Asperti, C. Sacerdoti-Coen, E.Tassi, and S.Zacchiroli. User interaction with the matita proof assistant. *Journal of Automated Reasoning, Special Issue on User Interfaces for Theorem Proving*, 2007.
3. Claudio Sacerdoti Coen and Stefano Zacchiroli. Efficient ambiguous parsing of mathematical formulae. In Andrzej Trybulec, editor, *Proceedings of the Third International Conference on Mathematical Knowledge Management*, volume 3119 of *LNCS*, pages 347–362, Bialystok, Poland, September 2004. Springer.
4. F. Kamareddine, M. Maarek, and J. B. Wells. Toward an object-oriented structure for mathematical text. In M. Kohlhase, editor, *MKM 2005, Fourth International Conference on Mathematical Knowledge Management*, LNAI 3863, pages 217–233. Springer, 2006.
5. Michael Kohlhase. *OMDOC - An Open Markup Format for Mathematical Documents [Version 1.2]*, volume 4180 of *LNAI*. Springer, August 2006.
6. Till Mossakowski, Serge Autexier, and Dieter Hutter. Development graphs - proof management for structured specifications. *Journal of Logic and Algebraic Programming, special issue on Algebraic Specification and Development Techniques*, 67(1-2):114–145, April 2006.
7. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
8. Luca Padovani and Stefano Zacchiroli. From notation to semantics: There and back again! In Jon Borwein and Bill Farmer, editors, *Proceedings of MKM'06*, volume 4108 of *LNAI*. Springer, August 2006.
9. Alan Bundy Roy McCasland. Mathsaid: a mathematical theorem discovery tool. In *Proceedings of SYNASC 2006*, 2006.
10. Joris van der Hoeven. Gnu $\text{T}_\text{E}\text{X}_\text{MACS}$: A free, structured, wysiwyg and technical text editor. Number 39-40 in Cahiers GUTenberg, May 2001.
11. Marc Wagner. Mediation between text-editors and proof assistance systems. Diploma thesis, Saarland University, Saarbrücken, Germany, 2006.
12. Marc Wagner, Serge Autexier, and Christoph Benzmüller. PlatΩ: A mediator between text-editors and proof assistance systems. In Serge Autexier and Christoph Benzmüller, editors, *7th Workshop on User Interfaces for Theorem Provers (UITP'06)*, ENTCS. Elsevier, 2006.
13. W. Waite and G. Goos. *Compiler Construction*. Springer, 1985. ISBN 0-387-90821-8.
14. Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau - Theorie, Konstruktion, Generierung, 2. Auflage*. Springer, 1997.