From notation to semantics: there and back again

Luca Padovani[1] and Stefano Zacchiroli[2]

[1] Information Science and Technology Institute, University of Urbino
padovani@sti.uniurb.it
[2] Department of Computer Science, University of Bologna
zacchiro@cs.unibo.it

Abstract. Mathematical notation is a structured, open, and ambiguous language. In order to support mathematical notation in MKM applications one must necessarily take into account presentational as well as semantic aspects. The former are required to create a familiar, comfortable, and usable interface to interact with. The latter are necessary in order to process the information meaningfully.
In this paper we investigate a framework for dealing with mathematical notation in a meaningful, extensible way, and we show an effective instantiation of its architecture to the field of interactive theorem proving. The framework builds upon well-known concepts and widely-used technologies and it can be easily adopted by other MKM applications.

1 Introduction

Mathematical formulae can be encoded at different levels of human and machine understandability [1]. Formulae at the *notational level* are encoded on the basis of their rendering, in the same spirit of the MathML Presentation markup language [8]. Interaction with the user happens on formulae at this level: the user feeds the application with formulae in some notation, the system renders the formulae in some notation.

Formulae at the *semantic level* are those which the application has the deepest understanding of and on which it can better perform *computations*. In the fields of Computer Algebra Systems and theorem provers, examples of such computations include evaluation, simplification, automatic (dis-)proving, and type-checking. This level is intrinsically application-specific.

In between is an intermediate level, which we call *content level*, whose aim is to encode the structure and, to a limited extent, the semantics of mathematical formulae. MathML Content and OpenMath [14] are examples of markup languages that encode formulae at this level. The content level is the most effective vehicle of interoperability across MKM applications not sharing semantic foundations.

A framework that deals with meaningful mathematical notation has a naturally layered architecture where the same mathematical object is encoded in

different ways according to the activities it is subjected to. The layers are connected with each other, and the encodings must be kept synchronized accordingly. In this sense we distinguish notation, which is a purely presentational tool, from *meaningful notation* that blends together both presentational and semantic aspects. From the perspective of the framework's designer, the fact that notation is extensible is a source of considerable additional complexity. It means that the layers cannot be fully described *a priori*, and that their connections must be updated dynamically as the system is enriched with new notation and new mathematical objects. It should be noted that a system supporting extensible notation in an exclusively presentational fashion is much simpler but also of limited use.

We consider the following features as characterizing such framework:

Extensibility: the framework must permit its users to define their own notation in an incremental way, using a basic set of primitive constructs along with all the notation has been defined earlier.

Remote control: notation should provide handles for enabling indirect manipulation of the (possibly hidden) information encoded at the content and semantic levels.

Ambiguity: the framework must tolerate (and encourage) ambiguity, which is common practice in traditional mathematical artifacts.

Interoperability: the framework must not hinder communication with other software.

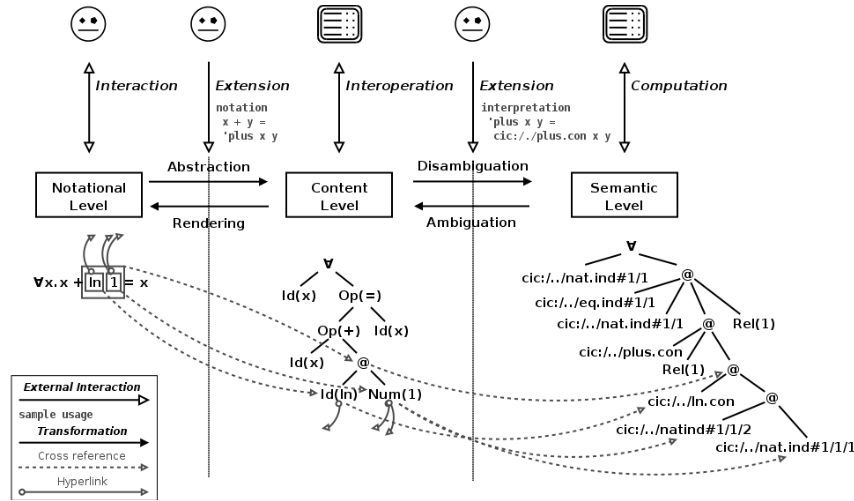

Fig. 1. Architecture of the notational framework.

In this paper we show how a framework supporting extensible, meaningful notation can be designed, and we demonstrate the effectiveness of our approach in the context of a theorem proving application. Figure 1 depicts the framework's architecture at work, where a first order logic formula is encoded at the three levels, the classical notational level on the left and the corresponding semantic encoding in the Calculus of (Co)Inductive Constructions [15] on the right. In the figure we use Helm [2] URIs as object references.

Transformations among the levels (horizontal solid arrows) are initiated by the need of interaction between the user and the application. When the need is to input a formula, *abstraction* brings the formula to the content level and *disambiguation* recovers a fully semantic encoding of the formula. When the need is to output a formula, *ambiguation* strips the formula of any application-specific semantic information and *rendering* creates a familiar representation. The transformations are driven by sets of bidirectional rules: a set of *notational equations* drives abstraction and rendering, while a set of *interpretations* drives disambiguation and ambiguation. The *extensibility* is pictorially represented as changes to these sets.

Cross references and hyperlinks account for *remote control*. Cross references relate corresponding pieces of information across the different encodings, so that the rendering engine can feature semantically driven forms of selection, cut and paste, and editing. Hyperlinks are one-to-many mappings from atomic objects to resources. Typically they link objects to their definitions.

Relevance to MKM and contribution. This paper complements [13] by investigating the technical issues related to the design of a user-extensible, interactive environment for the development and the management of mathematical knowledge in a semantically driven way. In particular, it proposes an architecture that has proven effective in mixing presentational as well as semantic aspect of the processed information. This is an improvement with respect to the currently available tools related to MKM which typically focus on one, but not both, of these equally important aspects.

Previous work [2] describing a similar architecture to that discussed in this paper did not address the issues related to extensible input support, and it only described informally how hyperlinks and cross references were propagated from the semantic to the presentation level. In this paper we describe these important features in a more abstract, but also more formal way, hoping to provide useful guidelines for future implementations.

Paper organization. The rest of the paper is organized as follows: in Section 2 we give a definition of notation by showing the relevant pieces of information that are affected by the notational equations. We do so by modelling levels with terms and transformations with functions on these terms. In Section 3 we complete the architecture by instantiating the semantic level in the particular case of a theorem proving application. Section 4 shows all the aspects of the framework at work on a concrete example. Section 5 discusses some related work and Section 6

concludes with some considerations about our implementation of the framework and some possible extensions.

2 Syntax and semantics of notation

In order to define precisely what notation is and how the information it conveys is processed during abstraction and rendering, we need a description of the languages encoding formulae at the notational and content levels.

Table 1. Syntax of presentation (E^p) and content (E^c) expressions.

	E^p ::=			E^c ::=	
	x	(identifier)		x	(identifier)
|	$l@H$	(literal)	|	$s@H$	(symbol)
|	$A\{E^\mathsf{p}\}$	(annotation)	|	$A\{E^\mathsf{c}\}$	(references)
|	$L[E^\mathsf{p}_1, \dots, E^\mathsf{p}_n]$	(layout)	|	$C[E^\mathsf{c}_1, \dots, E^\mathsf{c}_n]$	(constructor)
|	$B[E^\mathsf{p}_1 \cdots E^\mathsf{p}_n]$	(box)	|	α	(variable)
|	α	(variable)			

Table 1 shows the grammars for two streamlines languages of *presentation* and *content* expressions capturing the essence of notation. The two grammars are parametric in the following sets: a set of *layout schemata* L representing basic constructs of mathematical notation such as fractions, square roots, vectors, and so on; a set of *box schemata* B for annotating presentation expressions with line-breaking hints; a set of *identifiers* x, a set of *literals* l representing characters, numbers; a set of *symbols* s representing the basic elements in the ontology language of the content level (in MathML Content this set is predefined, in OpenMath it is completely unspecified, in either case it is open-ended and can be extended at will); a set of *constructors* C of the content level for building compound objects such as sets, lists, functions, relations. Literals and symbols are annotated with sets of *hyperlinks* H. We write l and s for $l@\emptyset$ and $s@\emptyset$ respectively. Both presentation and content expressions may be annotated with sets of cross references A. We omit the annotations p and c when it is clear that we are talking about presentation and content expressions, respectively.

A well-formed presentation pattern is a presentation expression E without identifiers, hyperlinks and cross references and such that any variable in E occurs exactly once. A presentation term is a presentation expression without variables. Content patterns and terms are defined similarly from content expressions.

A *notational equation* is a pair of well-formed patterns

$$P^\mathsf{p} \iff P^\mathsf{c}$$

that simultaneously defines (1) an *abstraction* from the notational level to the content level, and (2) a rendering from the content level to the notational level.

Example 1. The notational equation

$$\alpha = \beta \iff \texttt{apply}[\texttt{eq}, \alpha, \beta]$$

defines a notation for the infix, binary operator `=` which is represented at the content level as an `apply` constructor whose first child is the `eq` symbol followed by the two operands in order. □

2.1 Abstraction

Abstraction is the process of instantiating the content term corresponding to a presentation term. Conceptually this is done in two steps: first, the presentation term is parsed according to the notation that is available where the term occurs and its parsing tree is determined. Then, the tree is navigated and a corresponding content tree is instantiated in a bottom-up fashion.

Let us discuss parsing first. Let $\mathcal{G}_0$ be the grammar that defines the *built-in notation* of the framework and let T be the grammar nonterminal symbol producing terms. The definition of new notation causes $\mathcal{G}_0$ to be extended incrementally as follows:

$$\mathcal{G}_0 \xrightarrow{P_0^{\mathsf{p}} \iff P_0^{\mathsf{c}}} \mathcal{G}_1 \xrightarrow{P_1^{\mathsf{p}} \iff P_1^{\mathsf{c}}} \mathcal{G}_2 \xrightarrow{P_2^{\mathsf{p}} \iff P_2^{\mathsf{c}}} \cdots \xrightarrow{P_k^{\mathsf{p}} \iff P_k^{\mathsf{c}}} \mathcal{G}_k$$

where each grammar $\mathcal{G}_{i+1}$ results from $\mathcal{G}_i$ by the addition of a the production for T derived from $P_i^{\mathsf{p}} \iff P_i^{\mathsf{c}}$ and P_i^{c} is a content pattern parsed with $\mathcal{G}_i$ (this way notation can be defined incrementally on top of previously defined notation). In particular, the added production is $T \to \mathsf{exp}(P^{\mathsf{p}})$ where the function $\mathsf{exp}(P)$ converts a presentation pattern into a sequence of terminal and nonterminal grammar symbols as follows:

$$\begin{aligned}
\mathsf{exp}(l) &= l \\
\mathsf{exp}(\alpha) &= T \\
\mathsf{exp}(B[P_1 \cdots P_n]) &= \mathsf{exp}(P_1) \cdots \mathsf{exp}(P_n) \\
\mathsf{exp}(L[P_1, \dots, P_n]) &= L[\mathsf{exp}(P_1), \dots, \mathsf{exp}(P_n)]
\end{aligned}$$

Note that boxes are discarded in the expansion process as they play no role in the parsing phase and their content is juxtaposed.

A delicate technical problem related to grammars is ambiguity. An ambiguous grammar is one such that there may be multiple parse trees for the same term. In the most common cases ambiguity can be resolved by declaring precedence and associativity of productions. Thus, the language may provide additional constructs (see Section 4) so that the user can specify, for instance, that the symbol `*` has precedence over `+` and that `*` is left-associative. The remaining cases of ambiguity can be treated as errors (and the notation causing the ambiguity could be rejected or ignored), or they may be admitted provided that the implementation accommodates a form of content validation that can discriminate, among the various content terms that can be built starting from the

very same presentation term, which ones are semantically meaningful. This validation phase usually entails a deeper understanding of content terms than it is available at the content level, thus some cooperation with the semantic level becomes fundamental for settling structural ambiguities.

Now we take care of the instantiation step. Given a presentation term t, the parser yields a parsing tree for t which we denote with $\hat{t}$. In particular, it determines a pattern P_i^{p} and a substitution σ that associates variables occurring in P_i^{p} with subterms of $\hat{t}$ such that $P_i^{\mathsf{p}}\sigma = \hat{t}$ (equality here is considered up to cross references and hyperlinks). We abbreviate this writing $t \in P_i^{\mathsf{p}} \rightsquigarrow \sigma$.

Example 2. assuming that the + operator has precedence over =, we have that

$$1 \mathsf{+} 2 \mathsf{=} 3 \in (\alpha \mathsf{=} \beta) \rightsquigarrow [\alpha \mapsto (1 \mathsf{+} 2), \beta \mapsto 3]$$

where we use parentheses to indicate a generic box schema. □

Abstraction is a function $\mathcal{A}(\cdot)$ defined as follows:

$$\mathcal{A}(t) = P_i^{\mathsf{c}}\sigma' \text{ where } t \in P_i^{\mathsf{p}} \rightsquigarrow \sigma \text{ and } \sigma'(\alpha) = \begin{cases} \mathcal{A}(\sigma(\alpha)) & \text{if } \alpha \in \mathsf{dom}(\sigma) \\ \text{undefined otherwise} \end{cases}$$

The function $\mathcal{A}(t)$ is well-defined as long as the terms in the image of σ are all proper subterms of $\hat{t}$.

2.2 Rendering

Rendering creates a presentation term from a content term. Like abstraction, we can think of this as a two-step transformation: during the first phase the structure of the content term t is inspected for finding those parts of the term matching the right-hand side of a notation $P_i^{\mathsf{p}} \iff P_i^{\mathsf{c}}$. Then, the left-hand side is instantiated accordingly. Unlike abstraction annotations and hyperlinks must be propagated to the presentation term and this is what makes rendering tricky. Table 2 shows the pattern matching of a content term t against a content pattern P as a system of inference rules. We use the notation

$$t \in P \rightsquigarrow_A \sigma, A', H$$

meaning that given an initial set of cross references A, the matching of the term t against a pattern P yields a substitution σ, a final set of cross references A', and a set H of hyperlinks harvested from the symbols in t.

We define the rendering function $\mathcal{R}(\cdot)$ as

$$\mathcal{R}(t) = A\{I_\sigma^H(P_i^{\mathsf{p}})\} \text{ where } t \in P_i^{\mathsf{c}} \rightsquigarrow_\emptyset \sigma, A, H$$

and the instantiation function $I_\sigma^H(\cdot)$ as

$$\begin{aligned} I_\sigma^H(l) &= l@H \\ I_\sigma^H(L[P_1, \dots, P_n]) &= L[I_\sigma^H(P_1), \dots, I_\sigma^H(P_n)] \\ I_\sigma^H(B[P_1, \dots, P_n]) &= B[I_\sigma^H(P_1), \dots, I_\sigma^H(P_n)] \\ I_\sigma^H(\alpha) &= \mathcal{R}(\sigma(\alpha)) \end{aligned}$$

Table 2. Pattern matching of content terms.

(Symbol)
$s@H \in s \rightsquigarrow_A \varepsilon, A, H$

(Variable)
$t \in \alpha \rightsquigarrow_A [\alpha \mapsto A\{t\}], \emptyset, \emptyset$

(Annotation)
$$\frac{t \in P \rightsquigarrow_{A \cup A'} \sigma, A'', H}{A\{t\} \in P \rightsquigarrow_{A'} \sigma, A'', H}$$

(Constructor)
$$\frac{(t_i \in P_i \rightsquigarrow_\emptyset \sigma_i, A'_i, H_i)^{i \in 1..n}}{C[t_1, \dots, t_n] \in C[P_1, \dots, P_n] \rightsquigarrow_A \sigma_1 \cdots \sigma_n, A, H_1 \cup \cdots \cup H_n}$$

In the process rendering a content term t annotations of subterms of t are preserved only in two occasions: either when they are found at the top level of t, in which case they become annotations for the resulting presentation term, or when they wrap proper subterms of t that have been bound by variables, in which case they will wrap the rendered subterms. As there is no obvious way of relating the other annotations, they are simply discarded (see the (Constructor) rule in Table 2). Hyperlinks, on the other hand, are handled pattern-wise. All the hyperlinks found in the part of a term matched by a content pattern are gathered together and sprinkled over the literals of the corresponding presentation pattern. That is to say, any visible part of the term is considered the concrete rendering of its symbols and should thus be linked to their definitions.

The definition of $\mathcal{R}(\cdot)$ omits two secondary details: (1) the function $\mathcal{R}(\cdot)$ must provide appropriate rendering for all the built-in notation defined in $\mathcal{G}_0$; (2) precedence and associativity of the productions are used to spot the subterms that must be protected by fences, in order to guarantee a presentation term that is consistent with the structure of the content term.

Example 3. Consider the notational equation

$$\alpha \texttt{≠} \beta \iff \texttt{apply}[\texttt{not}, \alpha \texttt{=} \beta]$$

where we assume that the notation for the equality `=` has been given as in Example 1. The content term

$$t = i_1\{\texttt{apply}[i_2\{\texttt{not}@h_1\}, i_3\{\texttt{apply}[i_4\{\texttt{eq}@h_2\}, i_5\{\texttt{1}@h_3\}, i_6\{\texttt{2}@h_4\}]\}]\}$$

represents the inequality $1 \neq 2$ where the two constants 1 and 2 are located at h_3 and h_4 and are identified by i_5 and i_6 respectively. The whole term has reference i_1, the symbol `not` has reference i_2 and is located at h_1, while the symbol `eq` has reference i_4 and is located at h_2. The term t would be rendered as

$$i_1\{i_5\{\texttt{1}@h_3\} \texttt{≠}@\{h_1, h_2\}\ i_6\{\texttt{2}@h_4\}\}$$

where we note that the reference of the whole term is preserved, whereas the references of the `not` and `eq` symbols have been lost (there is no natural rendered subterm corresponding to them). There are two links associated with the `≠` literal

corresponding to the locations of the `not` and `eq` symbols. Finally, the symbols `1` and `2` have been rendered with all the information preserved (in the rendering we have omitted explicit box schemata for simplicity). □

3 Handling ambiguity in MATITA

Since disambiguation and ambiguation (the transformations from content to semantics and back) inherit the quality of being application-specific from the semantic level, we cannot give a fully general recipe for handling them. We will therefore present their instantiation in the context of MATITA,[1] a document-centric proof assistant being developed at the University of Bologna. Nevertheless, as we will see shortly, we only require the semantic language to be compositional, as most structured languages are.

In MATITA the semantic language is the Calculus of (Co)Inductive Constructions [15] (CIC for short), a typed λ-calculus enriched with inductive data types. In this setting, an *interpretation* is a pair

$$s\ \alpha_1 \cdots \alpha_n \iff t[\alpha_1, \dots, \alpha_n]$$

where s is a content symbol of arity $n \geq 0$ and $t[\alpha_1, \dots, \alpha_n]$ is a CIC term with n holes labelled $\alpha_1, \dots, \alpha_n$. The intention is to give *one of* the possible meanings for the symbol s when applied to n content terms $t_1, \dots, t_n$, in terms of the CIC term t in which the hole α_i has been replaced by the meaning of t_i. The "one of" is to remark that there can be multiple interpretations for the same symbol s, not necessarily having the same arity.

3.1 Disambiguation

Of the two transformations dealing with the semantic level, disambiguation is the most challenging, since it has to resolve the ambiguity of content terms with respect to semantic terms.

When the semantic level is CIC, the ambiguity is induced by the one-to-many mapping of symbols to CIC term, which in turn is induced by overloading of operators and missing information at the notational level.[2] Consider the content level expression obtained after the abstraction of Example 2. Its ambiguity with respect to CIC derives from the overloading of `+` (two different plus do exists in the standard library of MATITA), and from the missing type argument of `=`, which is needed by the CIC encoding of Leibniz's equality.

Example 4. The following interpretations taken from the MATITA standard library show this ambiguity:

[1] `http://matita.cs.unibo.it/`

[2] Numbers and unbound identifiers also induce ambiguity. For the sake of brevity in this paper we treat them as 0-ary symbols for which the appropriate interpretations have been given.

```
interpretation "natural plus" 'plus x y =
 (cic:/matita/nat/plus/plus.con x y).
interpretation "integer plus" 'plus x y =
 (cic:/matita/Z/plus/Zplus.con x y).
interpretation "Leibniz's equality" 'eq x y =
 (cic:/matita/logic/equality/eq.ind#xpointer(1/1) _ x y).
```

The first two provide for overloading of `+`, the last uses an implicit CIC term (`_`) to represent the missing argument. □

Intuitively, *disambiguation* is a two phase process. In the first phase all possible CIC terms corresponding to a content term, according to the current set of interpretations, are built. In the second phase they are filtered by means of an oracle able to decide whether a term is valid or not. Such an oracle for CIC is the *refiner* described in [12]. The actual disambiguation algorithm implemented in MATITA exploits the type inference capabilities of the refiner and is far more efficient than the naive algorithm entailed by this intuition. The interested reader can find a detailed description of the disambiguation algorithm, as well as a discussion on its computational complexity, in [13].

3.2 Ambiguation

We call *ambiguation* the reverse transformation that creates a content term from a CIC term. It is simpler than disambiguation since the mappings from CIC to content are not ambiguous (they may be non-injective though). This step resembles rendering in many ways: ambiguation works by pattern matching on CIC terms, and it instantiates content terms according to the matching interpretations. As usual, the system provides a finite set of built-in mappings for transforming uninterpreted CIC terms to the corresponding content terms. Propagation of cross references and hyperlinks can be implemented in exactly the same way as described in Section 2.2, the URIs appearing in interpretations are the original sources of hyperlinks.

4 A full-scale example

In this section we provide a complete example of notation in use in the MATITA proof assistant: existential quantification. The purpose of the example is twofold. On one hand it presents all together the aspects of notation from presentation to semantics. On the other hand, it allows us to glance at some features of the notational framework offered to the user for describing notational equations and interpretations that we had to omit from Sections 2 and 3 due to lack of space.

The existential quantifier is not built-in in CIC, but it is defined as an inductive data type in the `logic/connectives` module of the MATITA standard library. Its notation is given thus:

```
notation "hvbox(\exists ident i opt (: ty) break . p)"
 right associative with precedence 20
for @{ 'exists ${ default
  @{ \lambda ${ident i} : $ty. $p }
  @{ \lambda ${ident i} . $p }
 }}.
```

The presentation pattern is enclosed in double quotes. It consists of variables (`i`, `p`, and `ty`) that stand for arbitrary CIC sub-terms, and literals (`\exists`, `:`, and `.`) assembled together in a box schema. The special keyword `break` indicates the breaking point and the box schema `hvbox` indicates a horizontal or vertical layout, according to the available space for the rendering. The `opt` indicates a meta-operator that surrounds an optional part in the presentation pattern. Given this presentation pattern, MATITA's input syntax is extended so that, for example, `\exists x:nat. x \le y` is a valid presentation term. Because of the `opt` meta-operator, the type annotation `:nat` can be omitted, the resulting term still being syntactically valid.

The line beginning with `right associative`... is self explicative: it specifies associativity and precedence of the notation, thus determining the binding strength of the existential quantifier during parsing and giving the renderer appropriate information for inserting parentheses when needed.

The content pattern begins right after the `for` keyword and extends to the end of the declaration. Parts of the pattern surrounded by `@{...}` denote verbatim content fragments, those surrounded by `${...}` denote meta-operators and meta-variables (for example `$ty`) referring to the meta-variables occurring in the presentation pattern. The content pattern of the example defines the application of the content symbol `exists` to a λ-abstraction. In this case there are two possibilities according to the presence or absence of the type annotation in the presentation term that matched the pattern. For this reason there is a corresponding meta-operator at the content level, named `default`, that has two branches which are chosen depending on the matching of the optional subexpression. In the example this is used to account for the optionality of type annotation on the quantified name, since its type can be inferred during disambiguation. Thus, if the type is given, the content term created after parsing has the form `'exists (\lambda x:nat.(x \le y))`. Otherwise, the resulting content term has the form `'exists (\lambda x.(x \le y))`.

Our notational language supports additional meta-operators for dealing with variable-size terms having a regular structure: the `list` operator, which can be used for describing sequences of presentation terms and literals, has a corresponding `fold` operator, which describes trees at the content level. Like for `opt` and `default`, `list` and `fold` together express a bi-directional relationship between the presentation and the content level.

In MATITA, the interpretation of the `exists` symbol is as follows:

```
interpretation "exists" 'exists \eta.x =
 (cic:/matita/logic/connectives/ex.ind#xpointer(1/1) _ x).
```

where the word `"exists"` enclosed in double quotes is an informal comment that can be used for keyword-based searching. In this interpretation the `exists` symbol has arity 1 and its only argument is required to be a function. This is expressed by the variable `x` being annotated with `\eta.`. Indeed, the content pattern shown previously regarding the `exists` symbol matches only when the symbol's argument is a function. Since this is not guaranteed at the CIC level, an η-abstraction is performed when necessary: if the CIC term matching `x` is not a λ-abstraction, a content term will be created for $\lambda \mathit{fresh}.($`x` $\mathit{fresh})$ instead.

The following statement can now be used to start a new proof

```
theorem increasing_to_le:
 \forall f:nat \to nat. increasing f \to
 \forall m:nat. \exists i. m \le f i.
```

and the initial sequent of the proof is rendered as shown on the left of Figure 2. Notice that when entering a formula in the system the user is allowed to use a TEX-like concrete syntax, and the system can render the formula both on a textual terminal in the same concrete syntax, or in a graphical canvas like that of Figure 2 where the layout schemata of the formula have been properly encoded using MathML Presentation markup. This second view offers a more familiar rendering and it also enables point-and-click functionalities, like those for remote control. In this particular example the system figures that `i` must have type `nat`. In case more than one interpretation for the entered formula is feasible, the system lists them in a dialog box and asks the user to pick the desired one.

Remote control is exploited in MATITA in two ways: the first is *hypertextual browsing* of objects in the library. As can be seen on the left of Figure 2, the URI of the `"exists"` interpretation flowed through the levels reaching the literal $\exists$ as an hyperlink, which can be recognized at the bottom of the figure, in the status bar of the application. By clicking on the literal, the corresponding object from the library is shown to the user. If multiple hyperlinks are associated with the same symbol, a pop-up window appears and the user decides which one to follow. Incidentally, this gives the user some information about how a mathematical construct is encoded at the CIC level.

The second form of remote control, *semantic selection*, exploits cross references to constraint the selection on the presentation markup to CIC subterms. On the right of Figure 2 for instance, the GUI inhibits the selection of $\forall m : nat$ despite it corresponds to a proper subterm at the presentation level, since it has no corresponding subterm at the semantic level. Contextual semantic actions can then be safely offered to the user: the pop up menu in the figure shows actions for type-checking, reducing, and using the term as a parameter for the next reasoning step. A classical copy operation to copy the subterm into the clipboard is also available.

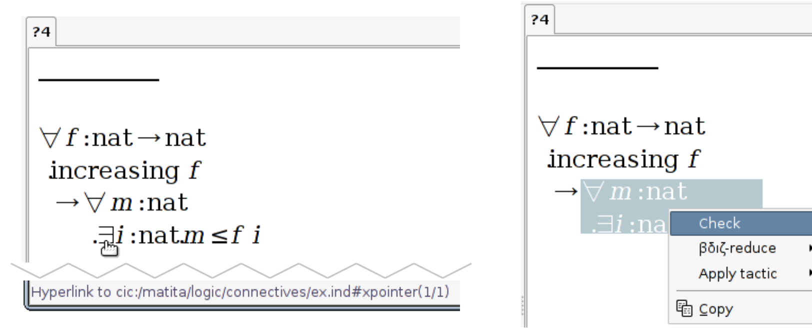

Fig. 2. Remote control in action: hyperlinks on the left hand side, semantic selection and contextual actions on the right hand side.

5 Related work

The layered architecture that we have proposed is similar in structure to that of previous projects in which notation played a major role. In [2,9] ambiguation and rendering are implemented by XSLT stylesheets [16] and they can only be extended by adding XSLT templates. Support for further notation is thus limited to the system designers. From the point of view of maintenance of the transformations, an improvement is the introduction of meta-stylesheet [7] that generate XSLT templates starting from a slightly higher-level specification. A somehow similar approach is proposed by Naylor and Watt [10] for supporting alternative notations. In any case, all the solutions mentioned are one-way only and cannot be inverted, both because XSLT is a very general transformation language, and also because the reverse path must reconstruct information that is not always available.

Our transformation language is not as general as XSLT but has been carefully designed so as to guarantee invertibility (the meta-operators mentioned in Section 4 are all invertible). Furthermore, it has a purely declarative style and is thus more appropriate for users who do not have any programming experience. The notational level consists of a finite set of layout schemata, basically those that are found in MathML Presentation [8] and TEX, box schemata for line-breaking inspired by previous work on pretty-printers [5], and a few meta-operators (like `opt` and `list`) inspired to the constructs of BNF grammars. The content level is an internal version of MathML Content and OpenMath [14], with the addition of meta-operators corresponding to those of the notational level.

The Coq proof assistant [4] provides a similar language for extending notation, with two main differences: it does not supply a content level and it does not deal directly with remote control. Our language represents a more open and interoperable solution, and the implementation shows that remote control can be achieved effectively even when notation is extensible, limiting built-in transformations to a bare minimum.

6 Conclusions and future work

In this paper we have characterized meaningful mathematical notation as a tool that necessarily mixes presentational as well as semantic aspects. We have identified a set of requirements that any MKM application supporting meaningful notation should fulfill and we have proposed an adequate architecture that builds upon the three well-known levels of formulae encoding: notation, content, and semantics. As an assessment of the generality of the architecture, we have given a formal dressing to the concept of notation which makes a minimum set of assumptions, and we have described an instantiation of its application-specific parts to the MATITA proof assistant.

The described architecture has been fully implemented in MATITA. The actual code has been written in OCaml[3] reusing components of the MATITA code base, most notably the code for disambiguation [13] and transformation from CIC to content and from content to MathML Presentation. Ambiguation and rendering have been implemented efficiently using a variant of the pattern matching algorithm in functional languages [3,6], which has been enriched with more expressive backtracking capabilities for dealing with meta-operators. Abstraction has been implemented using Camlp4, an extensible top-down parser with limited support for ambiguous grammars. This choice does not allow us to deal with structural ambiguity, that is with presentation terms admitting more than one corresponding content term. We plan to relax this constraint by implementing one of the several extensible parser generators that can be found in the literature (see [11] for an example).

Remarkably the proposed architecture does not deal with numbers in a practically useful way, since it assumes that there exists an infinite set of interpretations for them. In the Coq proof assistant, which basically shares the same semantic language used in MATITA, support for numbers is hard-coded in the application and thus it cannot be easily extended. We are currently investigating a declarative, finite interpretation scheme for numbers in MATITA, exploiting the regularity of their encoding in CIC, but it is still not clear whether this scheme is sufficiently general to make sense in different settings as well.

A major extension that we are considering is support for *local notation*, that is notation associated with content level binders that is in effect only in their scope. Local notation is a frequently asked feature in the formalization of algebraic theories, where quantification over notational symbols (as in "let $\odot$ be a binary operation over…") is a common mathematical practice. Since local notation requires an even tighter cooperation between the notational and the content levels, this could be a challenging test bench for verifying the scalability of our framework.

References

1. A. A. Adams. Digitisation, representation and formalisation: Digital libraries of mathematics. In J.H. Davenport A. Asperti, B. Buchberger, editor, *Proceedings*

[3] `http://caml.inria.fr/`

of Mathematical Knowledge Management 2003, volume LNCS, 2594, pages 1–16. Springer-Verlag, 2003.
2. Andrea Asperti, Ferruccio Guidi, Luca Padovani, Claudio Sacerdoti Coen, and Irene Schena. Mathematical knowledge management in HELM. *Annals of Mathematics and Artificial Intelligence*, 38(1-3):27–46, May 2003.
3. Lennart Augustsson. Compiling pattern matching. In Jean-Pierre Jouannaud, editor, *FPCA 1985: Functional Programming Languages and Computer Architecture, Proceedings*, volume 201 of *LNCS*, pages 368–381. Springer-Verlag, 1985.
4. The Coq proof-assistant. `http://coq.inria.fr`.
5. M. de Jonge. A pretty-printer for every occasion. In Ian Ferguson, Jonathan Gray, and Louise Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*, pages 68–77. University of Wollongong, Australia, June 2000.
6. Fabrice Le Fessant and Luc Maranget. Optimizing pattern-matching. In *Proceedings of the 2001 International Conference on Functional Programming*. ACM Press, 2001.
7. Pietro Di Lena. Generazione automatica di stylesheet per notazione matematica. Master's thesis, University of Bologna, 2003.
8. Mathematical Markup Language (MathML) Version 2.0. W3C Recommendation 21 February 2001, `http://www.w3.org/TR/MathML2`, 2003.
9. The MoWGLI Proposal, HTML version. `http://mowgli.cs.unibo.it/html_no_frames/project.html`.
10. Bill Naylor and Stephen Watt. Meta-stylesheets for the conversion of mathematical documents into multiple forms. *Annals of Mathematics and Artificial Intelligence*, 38(1-3):3–25, May 2003.
11. Jan Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.
12. Claudio Sacerdoti Coen. *Mathematical Knowledge Management and Interactive Theorem Proving*. PhD thesis, University of Bologna, 2004. Technical Report UBLCS 2004-5.
13. Claudio Sacerdoti Coen and Stefano Zacchiroli. Efficient ambiguous parsing of mathematical formulae. In Andrzej Trybulec Andrea Asperti, Grzegorz Bancerek, editor, *Proceedings of Mathematical Knowledge Management 2004*, volume 3119 of *LNCS*, pages 347–362. Springer-Verlag, 2004.
14. The OpenMath Society. The OpenMath Standard 2.0. `http://www.openmath.org/standard/om20/omstd20html-0.xml`, June 2004.
15. Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris VII, May 1994.
16. XSL Transformations (XSLT). Version 1.0. W3C Recommendation, 16 November 1999, `http://www.w3.org/TR/xslt`.