J Autom Reasoning (2007) 39:109–139
DOI 10.1007/s10817-007-9070-5

User Interaction with the Matita Proof Assistant

**Andrea Asperti · Claudio Sacerdoti Coen ·
Enrico Tassi · Stefano Zacchiroli**

Received: 1 February 2006 / Accepted: 1 February 2007 / Published online: 21 June 2007

Abstract Matita is a new, document-centric, tactic-based interactive theorem prover. This paper focuses on some of the distinctive features of the user interaction with Matita, characterized mostly by the organization of the library as a searchable knowledge base, the emphasis on a high-quality notational rendering, and the complex interplay between syntax, presentation, and semantics.

Keywords Proof assistant · Interactive theorem proving · Digital libraries · XML · Mathematical knowledge management · Authoring

1 Introduction

Matita is the interactive theorem prover under development by the HELM team [3] at the University of Bologna, under the direction of Prof. Asperti. Matita (which means *pencil* in Italian) is free software and implemented in OCaml. The source code is available for download at http://matita.cs.unibo.it.

"We are nearly bug-free" – *CSC, Oct. 2005.*

A. Asperti (✉) · C. Sacerdoti Coen · E. Tassi · S. Zacchiroli
Department of Computer Science, University of Bologna,
Mura Anteo Zamboni, 7-40127 Bologna, Italy
e-mail: asperti@cs.unibo.it

C. Sacerdoti Coen
e-mail: sacerdot@cs.unibo.it

E. Tassi
e-mail: tassi@cs.unibo.it

S. Zacchiroli
e-mail: zacchiro@cs.unibo.it

The main concern of this paper is presenting the user interaction with Matita. Before entering into this, we give in this section a preliminary idea of the system considering its foundations, proof language, and interaction paradigm.

Foundations Matita is based on the Calculus of Inductive Constructions (CIC) [36]. Proof terms are represented as λ-terms of the calculus. Proof checking is implemented by the system kernel that is a CIC type-checker. Metavariables can occur in terms to represent incomplete proofs and missing subterms.

Terms are not conceived as proof records kept by the system for efficiency reasons but become part of a distributed library of mathematical concepts (encoded in a XML dialect). Hence terms are meant as the primary data type for long-term storage and communication. The relevance we give to the library influences most aspects of Matita.

The competing Coq system is an alternative implementation of CIC, allowing for a direct and for obvious reasons privileged comparison with Matita. The two systems are compatible at the proof term level: mathematical concepts can be exported from Coq in XML to become part of the library of Matita.

Proof language Matita adopts a procedural proof language, following the approach dating back to the LCF theorem prover [17] and characteristic of many other successful tools such as Coq, NuPRL, PVS, and Isabelle (the last also supporting a declarative proof language).

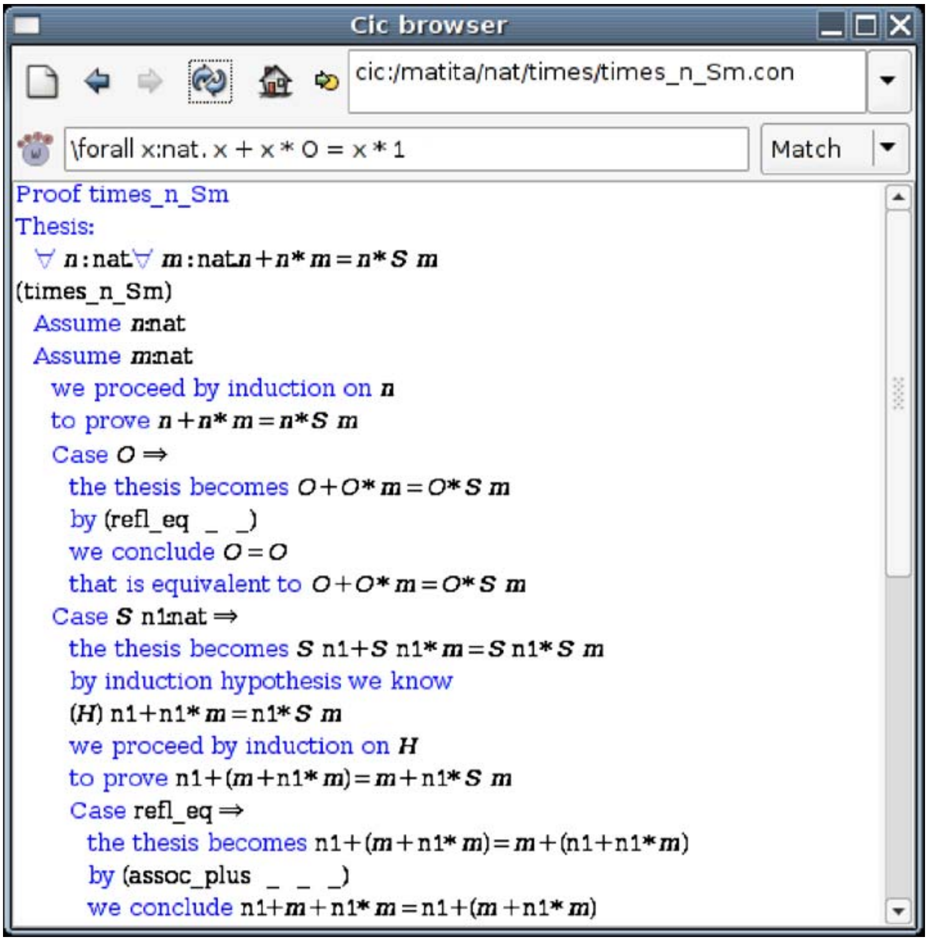
The statements of the language are called *tactics* and are collected in textual *scripts* that are interpreted one statement at a time. Structured scripts can be obtained forming new tactics from existing tactics and *tacticals*. In Matita structured scripts do not force atomic execution of such new tactics, resulting in a very valuable support for structured editing of proof scripts (see Section 3.4).

Proof terms generated by tactics can be rendered in pseudo-natural language (see Fig. 1). The natural language is precise enough to be currently used as an executable declarative language for Matita.

Interaction paradigm All the user interfaces currently adopted by proof assistants have been influenced by the CtCoq pioneering system [11] and are traditionally organized in three windows: one for script editing, one for the open goals (or *sequents*) that need to be proved, and one for messages. A particularly successful incarnation of those ideas is the Proof General generic interface [8], which has set a standard interaction paradigm between the user and the system. The authoring interface of Matita (shown in Fig. 2) adopts the same interaction paradigm and essentially offers the same functionalities of Proof General.

We differ from other user interfaces in the sequents window, where we focus on a high-quality and hypertextual rendering of mathematical expressions (see Section 3.1). In Matita, the sequents window is based on a MathML-compliant GTK+ widget, originally conceived and developed by the Helm-team to be used in Matita, and it then evolved into an independent and successful component of many well-known applications (such as AbiWord).

Structure of this paper We present the user interaction with Matita from two different angles. In Section 2 we describe the philosophical approach of Matita to



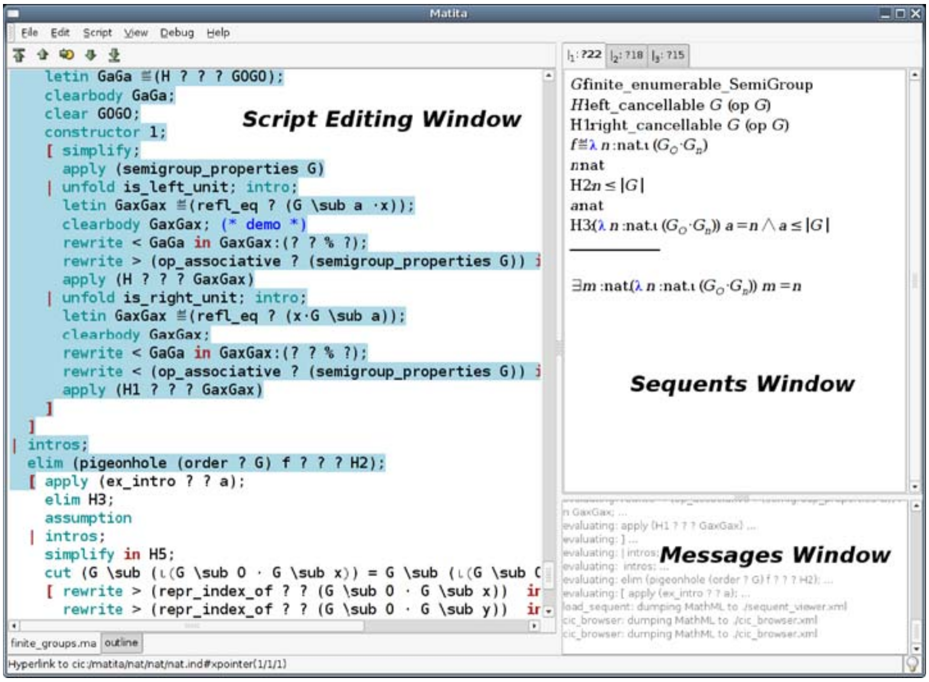
```
cic:/matita/nat/times/times_n_Sm.con
\forall x:nat. x + x * O = x * 1
Proof times_n_Sm
Thesis:
  ∀ n:nat∀ m:nat.n+n*m=n*S m
(times_n_Sm)
  Assume n:nat
  Assume m:nat
    we proceed by induction on n
    to prove n+n*m=n*S m
    Case O ⇒
      the thesis becomes O+O*m=O*S m
      by (refl_eq _ _)
      we conclude O=O
      that is equivalent to O+O*m=O*S m
    Case S n1:nat ⇒
      the thesis becomes S n1+S n1*m=S n1*S m
      by induction hypothesis we know
      (H) n1+n1*m=n1*S m
      we proceed by induction on H
      to prove n1+(m+n1*m)=m+n1*S m
      Case refl_eq ⇒
        the thesis becomes n1+(m+n1*m)=m+(n1+n1*m)
        by (assoc_plus _ _ _)
        we conclude n1+m+n1*m=n1+(m+n1*m)
```

Fig. 1 Browsing a proof in pseudo-natural language

library management and its practical consequences on user interaction: searching (Section 2.1), preservation of library consistency (Section 2.2), automation (Section 2.3), and concept naming (Section 2.4). In Section 3 we move to the concrete interaction level, presenting the authoring interface of our system, emphasizing its most innovative aspects: direct manipulation of terms and its textual representation (Sections 3.1 and 3.2), disambiguation of formulas (Section 3.3), and step-by-step tacticals (Section 3.4). Section 4 concludes the paper with a historical presentation of Matita (Section 4.1), some software engineering considerations based on our development experience (Sections 4.2 and 4.3), and our present plans (Section 4.4).

2 Library Management

The Matita system is meant to be first of all an interface between the user and the mathematical library; this makes a clear methodological difference between Matita



```
   letin GaGa ≝(H ? ? ? GOGO);
   clearbody GaGa;
   clear GOGO;
   constructor 1;
   [ simplify;
     apply (semigroup_properties G)
   | unfold is_left_unit; intro;
     letin GaxGax ≝(refl_eq ? (G \sub a ·x));
     clearbody GaxGax; (* demo *)
     rewrite < GaGa in GaxGax:(? ? % ?);
     rewrite > (op_associative ? (semigroup_properties G)) i
     apply (H ? ? ? GaxGax)
   | unfold is_right_unit; intro;
     letin GaxGax ≝(refl_eq ? (x·G \sub a));
     clearbody GaxGax;
     rewrite < GaGa in GaxGax:(? ? % ?);
     rewrite < (op_associative ? (semigroup_properties G)) i
     apply (H1 ? ? ? GaxGax)
   ]
  ]
| intros;
  elim (pigeonhole (order ? G) f ? ? ? H2);
  [ apply (ex_intro ? ? a);
    elim H3;
    assumption
  | intros;
    simplify in H5;
    cut (G \sub (ι(G \sub O · G \sub x)) = G \sub (ι(G \sub C
    [ rewrite > (repr_index_of ? ? (G \sub O · G \sub x))  ir
      rewrite > (repr_index_of ? ? (G \sub O · G \sub y))  ir
```

Hyperlink to cic:/matita/nat/nat/nat.ind#xpointer(1/1/1)

h_1: ?22 h_2: ?18 h_3: ?15

Gfinite_enumerable_SemiGroup
Hleft_cancellable G (op G)
H1right_cancellable G (op G)
$f ≝ \lambda\, n$:nat.ι $(G_O \cdot G_n)$
$n$nat
H2$n \leq |G|$
$a$nat
H3$(\lambda\, n$:nat.ι $(G_O \cdot G_n))\ a = n \wedge a \leq |G|$

$\exists m$:nat$(\lambda\, n$:nat.ι $(G_O \cdot G_n))\ m = n$

```
n GaxGax; ...
evaluating: apply (H1 ? ? ? GaxGax) ...
evaluating: ] ...
evaluating: | intros; ...
evaluating:  intros; ...
evaluating: elim (pigeonhole (order ? G) f ? ? ? H2); ...
evaluating: [ apply (ex_intro ? ? a); ...
load_sequent: dumping MathML to ./sequent_viewer.xml
cic_browser: dumping MathML to ./cic_browser.xml
cic_browser: dumping MathML to ./cic_browser.xml
```

Fig. 2 Authoring interface of Matita

and most of the current tools for the interactive support to formal reasoning, whose emphasis is traditionally on different aspects, such as authoring or proof checking.

The library of Matita comprises mathematical concepts (theorems, axioms, definitions) and notation. The concepts are authored sequentially using scripts in a procedural style. Once concepts are produced, we store them independently in the library; as soon as a new concept is defined, it becomes visible and will remain so upon re-entering the system, with no need of explicitly requiring or including portions of the library. The only relation implicitly kept between concepts are the logical, acyclic dependencies among them. In this way the library forms a global (potentially distributed) hypertext.

2.1 Indexing and Searching

To support efficient search and retrieving of mathematical notions, Matita uses a sophisticated indexing mechanism for mathematical concepts, based on a rich metadata set that has been tuned along the European project MoWGLI. The metadata set, and the searching facilities built on top of them – collected in the so-called Whelp search engine, have been extensively described in [4]. We recall here that the Whelp metadata model is essentially based on a single ternary relation $Ref_p(s, t)$ stating that a concept s refers a concept t at a given position p, where the position specifies the place of the occurrence of t inside s (we currently work with a fixed set of positions, discriminating the hypothesis from the conclusion and outermost from innermost occurrences). This approach is extremely flexible; by extending the set of positions, we may improve the granularity and the precision of our indexing technique, with no additional architectural impact.

Every time a new mathematical concept is created and saved by the user, it gets indexed and becomes immediately visible in the library. Several interesting and innovative features of Matita described in the following sections rely in a direct or indirect way on the global management of the library, its metadata system, and the search functionalities: most notably the disambiguation technique (Section 3.3), made even more compelling by the "global" visibility policy of Matita, and the support for automatic theorem proving (Section 2.3).

Related Work

To our best knowledge these facilities are peculiarities of Matita; other systems offer to the user less expressive or less integrated querying facilities. Both Isabelle and Coq, for instance, offer searching facilities only on the explicitly loaded parts of the library and offer more coarse-grained queries, such as finding theorems whose statement refer to some constant (with no distinction on where the constant occurs). These techniques, unlike ours, are unlikely to scale if applied to large, distributed libraries.

The Mizar Mathematical Library (MML) can be searched with a search engine that indexes the whole library [10]. When a declarative script is executed by Mizar, metadata is collected in special files used by the search engine. This solution is in principle more inefficient when compared with ours, which stores metadata in a relational database. So far, however, performances are good. As for the previous systems, no distinction can be done in Mizar's queries on the positions where constants occur. Nevertheless, queries can be quite precise because of the great flexibility provided by chaining of queries to refine search results.

Automation in Matita is heavily based on the search engine. On the contrary, in Mizar the search engine is an external tool, and automation cannot rely on it. The main justification for the different choices is to be found in the role of automation in the two systems. In Mizar automation is used to fill the gaps in the user-provided declarative proofs, and its main role is to justify proof steps considered trivial by the human being. In Matita automation can also be used for larger proofs, making the system closer to an automatic theorem prover.

2.2 Invalidation and Regeneration

In this section we focus on how Matita ensures the library consistency during the formalization of a mathematical theory, giving the user the freedom of adding, removing, and modifying mathematical concepts without losing the feeling of an always visible and browsable library.

The two mechanisms employed are *invalidation* and *regeneration*. A mathematical concept and those depending on it are invalidated when the concept is changed or removed and need to be regenerated to verify if they are still valid.

Invalidation is implemented in two phases. The former computes all the concepts that recursively depend on those we are invalidating. It can be performed by using the metadata stored in the relational database. The latter phase removes all the results of the generation, metadata included. To regenerate an invalidated part of the library, Matita re-executes the scripts that produced the invalidated concepts.

Related Work

Every system that remembers already checked concepts to avoid duplicated work must implement some form of invalidation and regeneration. The peculiarity of Matita's having an always visible and browsable library imposes the need of hiding concepts as soon as they become invalid. In other systems invalidation can be postponed: in Coq, for instance, a checksum is computed on scripts when they are executed, and an error is reported only when the user tries to include a script with an unexpected checksum.

In [18] and subsequent works, Hutter describes a framework for maintaining logical consistency in a distributed library. The framework is based on invalidation and regeneration but also adds the orthogonal notion of modularization via hierarchically structured developments. Moreover, a notion of refinement between developments (specified by morphisms) is provided that allows parts of large developments to be decoupled. For instance, once the group axioms are fixed, the user can develop the theory of groups and independently prove that some structure forms a group, automatically obtaining the instantiated theory. Modularization in different forms can be found in other systems, too: Isabelle has locales, while Coq has modules and functors.

In Matita we do not currently have any modularization mechanism, even if dependent records [15] partially provide the functionality. When modularization is considered, the dependencies used for invalidation and regeneration inherit the hierarchical structure of developments (as happens in [18]). Dependencies in Matita are simply flat.

2.3 Automation

In the long run, one would expect to work with a proof assistant such as Matita using only a small set of basic tactics: `intro` (∀-introduction), `apply` (modus ponens), `elim` (induction), `cut` (forward reasoning), and a powerful tactic `auto` for automated reasoning. Incidentally, these are also the only primitives available in declarative languages. The current state of Matita is still far from this goal, but this is one of the main development directions of the system.

Even in this field, the underlying philosophy of Matita is to free the user from any burden relative to the overall management of the library. In Coq, for instance, the user is responsible for defining small collections of theorems to be used as parameters for the `auto` tactic; in Isabelle the same situation happens with lemmas used by the simplifier. In Matita, the system itself automatically retrieves from the whole library a subset of theorems worth consideration, according to the signature of the current goal and context.

At present, our basic automation tactic (`auto`) merely iterates the use of the `apply` tactic. The search tree may be pruned according to two main parameters: the *depth* (with the obvious meaning) and the *width*, that is, the maximum number of (new) open goals allowed in the proof at any instant.

Recently we have extended automation with paramodulation-based techniques. The extension works reasonably well with equational rewriting, where the notion of equality is parametric and can be specified by the user: the system requires only a proof of *reflexivity* and *paramodulation* (or rewriting, to use the name employed in the proof assistant community).

Given an equational goal, Matita recovers all known equational facts from the library (and the local context), applying a variant of the so-called given-clause algorithm [23], the procedure currently used by the majority of modern automatic theorem provers.

We have recently run the `paramodulation` tactic of Matita on the unit-equality set of problems of the 2006 CASC competition [32], obtaining a score comparable with (actually, slightly better than) Otter [22].

In our view, the existence of a large, growing library of known results is the main and distinctive feature of automated tactics for interactive theorem provers with respect to automatic theorem provers. An interesting challenge is the automatic generation of new results, for example by means of saturation-based techniques. This goal requires the ability of the system to rate and possibly automatically select "interesting" results among the set of valid but trivial facts; such a feature is *per se* an interesting open problem (see [21] for preliminary discussions and [13, 14] for more advanced material on the larger topic of theory exploration).

2.4 Naming Convention

A minor but not entirely negligible aspect of Matita is that of adopting a (semi)-rigid naming convention for concept names, derived by our studies about metadata for statements. The convention is applied only to theorems (not definitions) and relates theorem names to their statements.

The basic rule is that each name should be composed by an underscore separated list of identifiers, occurring in a left-to-right traversal of the statement. Additionally, identifiers occurring in different hypotheses or in an hypothesis and in the conclusion should be separated by the string `_to_`. Moreover, the theorem name may be following by a *decorator*: a numerical suffix or a sequence of apostrophes.

Example 1 Consider for instance the statement:

```
\forall n: nat. n = n + O
```

Possible legal names for it are `plus_n_O`, `plus_O`, `eq_n_plus_n_O`, and so on. Similarly in the following statement `lt_to_le` is a legal name, while `lt_le` is not:

```
\forall n,m: nat. n \lt m \to n \leq m
```

But what about, say, the symmetric law of equality? Probably one would like to name such a theorem with something explicitly recalling symmetry. The correct approach in this case is the following.

Example 2 You should start with defining the symmetric property for relations:

```
definition symmetric :=
 \lambda A: Type. \lambda R. \forall x,y: A. R x y \to R y x.
```

Then, you may state the symmetry of equality as

```
\forall A: Type. symmetric A (eq A)
```

and `symmetric_eq` is a legal name for such a theorem.

So, somehow unexpectedly, the introduction of a semi-rigid naming convention has an important beneficial effect on the global organization of the library, forcing the user to define abstract concepts and properties before using them (and formalizing such use).

3 The Authoring Interface

The basic mechanisms underpinning the usage of the Matita authoring interface should be familiar to every user coming from Proof General and is, in our experience with master's students, easy for newcomers to learn (at the minimum, as easy as learning how to use Proof General or CoqIde, a Proof General-like Coq GUI). In spite of that fact, we deliberately chose not to develop Matita targeting Proof General as our user interface of choice. The first reason for not doing that is the ambition to integrate in the user interface our high-quality rendering technologies, mainly GtkMathView, to render sequents exploiting the bidimensional mathematical layouts of MathML-Presentation. At the time of writing Proof General supports only text-based rendering.[1] The second reason is that we wanted to build the Matita user interface on a state-of-the-art and widespread graphical toolkit as GTK+ is.

On top of the basic mechanisms of script-based theorem proving, Matita sports innovative features not found in competing systems. In the remaining part of this section we will discuss them in turn.

3.1 Direct Manipulation of Terms

The windows that show formulas and concepts to the user are based on Gtk-MathView, which is used to render *notational-level* representations of terms. Those representations are encoded in a mixed markup built on top of two XML dialects: MathML-Presentation and BoxML. The former language is used to encode the visual aspects of mathematical formulas by using the vocabulary of mathematical notation, which comprises atomic entities such as identifiers, numbers, and operators together with a set of layouts such as subscripts and superscripts, fractions, and radicals. The latter language is used to describe the placement of formulas with respect to each other and where to break formulas if the actual window is too small to fit them on a single physical line.

[1]This may change with the new Proof General based on the Eclipse platform.

3.1.1 Contextual Actions and Semantic Selection

Once rendered in a window, notational-level terms still play a role and permit hypertextual browsing of referenced concepts and also limited forms of direct manipulation [31] by using the mouse. *Hypertextual browsing* is shown in Fig. 3.

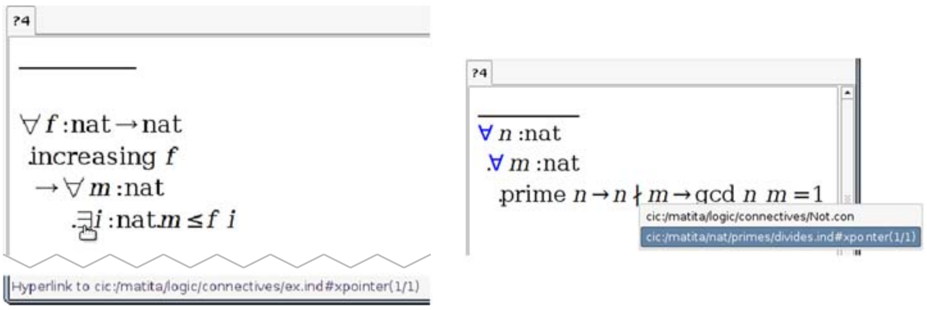
Markup elements that visually represent concepts from the library (identifiers or glyphs coming from user defined notations) act as anchors of hyperlinks. Targets of the hyperlinks are the concepts themselves, referenced by using their URIs. On the left of Fig. 3 the mouse is over an ∃ symbol, which is part of a user-defined notation for the existential quantifier, available in the Matita standard library as a concept whose URI is shown in the status bar. Clicking on an anchor will show the target concept.

Since user-defined notations are often used to hide details of complex CIC terms, markup elements may reference more than one concept from the library; in a sense our hyperlinks are one-to-many. For instance, on the right of Fig. 3 the symbol $\nmid$ is a user-defined notation that uses two concepts (logical negation and the divisibility operator); following that hyperlink will pop up a window asking the user to choose the browsing destination.

Limited forms of direct manipulation are possible on (sub)terms. Figure 4 shows the contextual menu that will pop up (clicking with the right button) when part of the markup is visually selected.

Menu items of the pop-up menu permits one to perform semantic *contextual actions* on the CIC term corresponding to the selected markup. Examples of such actions are type inquiries, application of tactics having the selected term as argument, various kinds of reduction, and *semantic copy and paste*. The last is called "semantic" to distinguish it from ordinary textual copy and paste, where the text itself is subject of the copy and paste actions. In our case the subject is rather the underlying CIC term. This permits us to perform semantic transformations on the copied term, such as renaming variables to avoid captures or λ-lifting free variables – transformations that can hardly be performed at the notational level where not even a notion of scope is available.

Contextual actions can also be performed on several terms selected at once (GtkMathView supports multiple selections). The typical use case of multiple selection is simplification in multiple subterms at once.



$\forall f:\mathrm{nat}\to\mathrm{nat}$
$\mathrm{increasing}\ f$
$\to\forall m:\mathrm{nat}$
$\exists i:\mathrm{nat}.m\le f\ i$

Hyperlink to cic:/matita/logic/connectives/ex.ind#xpointer(1/1)

$\forall n:\mathrm{nat}$
$\forall m:\mathrm{nat}$
$\mathrm{prime}\ n\to n\nmid m\to \mathrm{gcd}\ n\ m=1$

cic:/matita/logic/connectives/Not.con
cic:/matita/nat/primes/divides.ind#xpointer(1/1)

Fig. 3 Hypertextual browsing

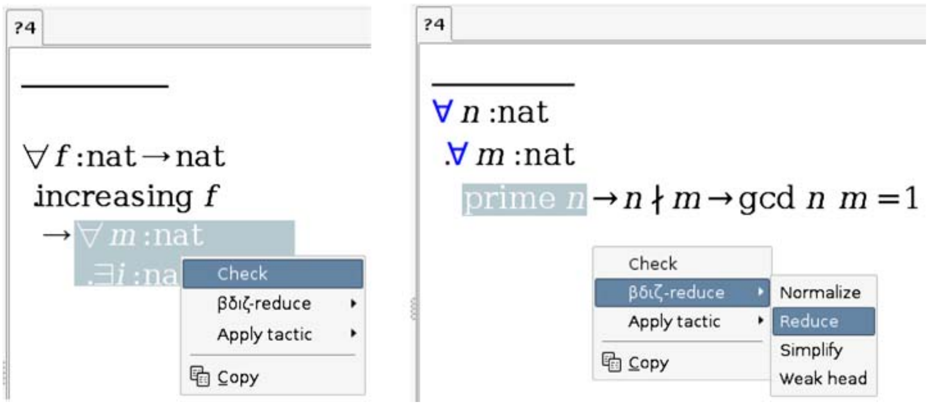


Fig. 4 Contextual actions and semantic selection

A requirement for semantic contextual actions is that the markup visually selected in a window have a corresponding CIC term: we call this *semantic selection*. This requirement is nontrivial to achieve because selection in GtkMathView (and more generally in rendering engines for XML-based markup languages) is constrained to the structure of the presentational markup, which is not necessarily related to the structure of the underlying CIC term. On the left of Fig. 4, for instance, the formula "$\forall m$: nat" is a well-formed markup snippet: a horizontal box containing two symbols and two identifiers. Nonetheless, no well-formed CIC term corresponds to it; intuitively a binder would, but a binder requires a body (something after ".") to be a well-formed term in CIC.

3.1.2 Implementation

Both hypertextual browsing and semantic selection are implemented by enriching the presentational markup with semantic attributes. The notational framework of Matita [37] is in charge of adding them. Figure 5 shows the architecture of the notational framework, including the different encoding of terms and the transformations among them. The intermediate content level – between the CIC, or semantic, level and the notational level – is an internal representation isomorphic to MathML-Content, useful for interoperability with other mathematical systems not sharing the same mathematical foundation of Matita.

The `hrefs` attribute is used to implement hypertextual browsing. An element annotated with such an attribute represents an hyperlink whose anchor is the rendered form of the element itself. Targets of the hyperlink are the URIs listed as value of the `hrefs` attribute. Hyperlinks can be present only on atomic markup elements (identifiers, symbols, and numbers). URIs are collected on nodes of the content syntax tree during *ambiguation* (the transformation from semantic to content level) and then spread on atomic markup elements pertaining to the notation chosen for a given content element during *rendering* (the transformation from content to notational level).

The `xref` attribute (for "cross reference") is used to implement semantic selection. Each CIC subterm is annotated with a unique identifier; the set of those

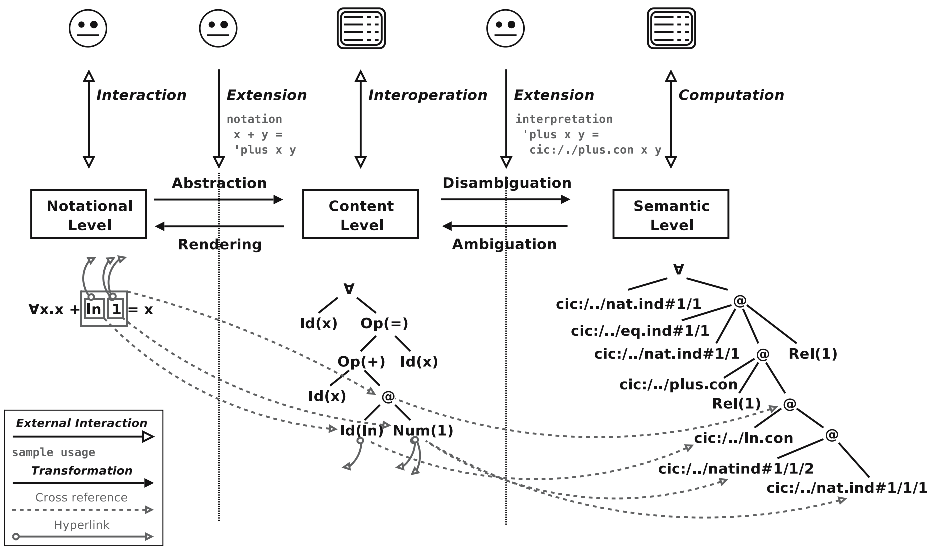


Fig. 5 Overview of the notational framework: term encodings and their usage

identifiers is the domain of the `xref` attribute. During ambiguation, identifiers are collected on nodes of the context syntax tree, cross-referencing nodes of the CIC syntax tree. During rendering, identifiers are collected on the structures available in the presentational markup (e.g., atomic elements for concepts or numbers, but also layouts for applications and more complex CIC terms). Since each node of the CIC syntax tree denotes a well-formed CIC term, one can now go "back" to a well-formed CIC term starting from an element of the presentation markup that ends up having an `xref` attribute.

During interactive visual selection, the user is permitted to select some markup only when the mouse is located on an element having an `xref` attribute. When this is not the case, the selection is automatically extended to the first element, in a visit toward the markup root, having such an attribute (the markup root is granted to have it). The requirement of always having a correspondence between the selected markup and a well-formed CIC term is hence fulfilled.

3.2 Patterns

In several situations working with direct manipulation of terms is simpler and faster than typing the corresponding textual commands [12]. Nonetheless we need to record such visual actions in scripts. In Matita *patterns* are textual representations of selections: users can select a pattern by using the GUI and then ask the system to paste the corresponding pattern in the script. This process is often transparent to the user: once an action is performed on a selection, the corresponding textual command is computed and inserted in the script.

3.2.1 Pattern Syntax and Semantics

Patterns are composed of two parts: ⟨*sequent_path*⟩ and ⟨*wanted*⟩; their concrete syntax is reported in Table 1. The term ⟨*sequent_path*⟩ mocks up a sequent, discharging

Table 1 Concrete syntax of patterns

Two-part patterns		
⟨*pattern*⟩	::=	[`in` ⟨*sequent_path*⟩] [`match` ⟨*wanted*⟩]
⟨*sequent_path*⟩	::=	{ ⟨*ident*⟩ [`:` ⟨*multipath*⟩] } [`\vdash` ⟨*multipath*⟩]
⟨*multipath*⟩	::=	⟨*term_with_placeholders*⟩
⟨*wanted*⟩	::=	⟨*term*⟩

unwanted subterms with "`?`" and selecting the interesting parts with the placeholder "`%`". The term ⟨*wanted*⟩ is a term living in the context of the placeholders.

Textual patterns produced from a graphical selection are made of the ⟨*sequent_path*⟩ only. Such patterns can represent every selection but can get quite verbose. The ⟨*wanted*⟩ part of the syntax is meant to help the users in writing concise and elegant patterns by hand.

Patterns are evaluated in two phases. The former selects roots (subterms) of the sequent, using the ⟨*sequent_path*⟩, while the latter searches the ⟨*wanted*⟩ term starting from that roots. *Phase 1* concerns only the ⟨*sequent_path*⟩ part. Here ⟨*ident*⟩ is an hypothesis name and selects the assumption where the following optional ⟨*multipath*⟩ will operate; `\vdash` does the same for the conclusion of the sequent. If the whole pattern is omitted, only the conclusion will be selected; if the conclusion part of the pattern is omitted but hypotheses are not, selection will occur only in them as specified by the corresponding ⟨*multipath*⟩. Remember that the user can be mostly unaware of pattern concrete syntax because Matita is able to write a ⟨*sequent_path*⟩ from a graphical selection.

A ⟨*multipath*⟩ is a CIC term in which two special constants "`%`" and "`?`" are allowed. The roots of discharged and selected subterms are marked respectively with "`?`" and "`%`". The default ⟨*multipath*⟩, the one that selects the whole term, is simply "`%`". Valid ⟨*multipath*⟩ terms are, for example, "`(? % ?)`" or "`% \to (% ?)`" that respectively select the first argument of an application or the source of an arrow and the head of the application that is found in the arrow target.

Phase 1 not only selects terms (roots of subterms) but also determines their context that may be used in the next phase.

Phase 2 plays a role only if the ⟨*wanted*⟩ part is specified. From phase 1 we have some terms, which we will use as roots, and their context. For each of these contexts the ⟨*wanted*⟩ term is disambiguated in it, and the corresponding root is searched for a subterm that can be unified to ⟨*wanted*⟩. The result of this search is the selection the pattern represents.

3.2.2 Patterns in Action

Consider the following sequent.

$$\begin{array}{l} n : nat \\ m : nat \\ \underline{H : m + n = n} \\ m = O \end{array}$$

Example 3 To change the right part of the equality of the H hypothesis with $O+n$, the user selects and pastes it as the pattern in the following statement.

```
change in H:(? ? ? %) with (O + n).
```

The pattern mocks up the applicative skeleton of the term, ignoring the notation that hides, behind $m+n=n$, the less familiar eq nat $(m+n)$ n.

Supporting notation in patterns is not a urgent necessity since patterns are not meant to be written by hand: the user can select with the mouse a subterm in the sequents window, where notation is used, and ask the system to automatically generate the pattern. Moreover this choice simplifies the implementation also improving efficiency. The main drawback of our choice is that patterns do not benefit from the high readability that infix notation grants. Again, we believe that a script based on a procedural language is not readable if not re-executed step-by-step; by re-executing the script the user sees exactly where the tactic is applied, without even trying to understand the pattern.

Example 4 The experienced user is not forbidden to write by hand a concise pattern to change all the occurrences of n in the hypothesis H at once.

```
change in H match n with (O + n).
```

In this case the ⟨*sequent_path*⟩ selects the whole H, while the second phase locates all n occurrences, thus resulting equivalent to the following pattern, which the system would have automatically generated from the selection.

```
change in H:(? ? (? ? %) %) with (O + n).
```

3.2.3 Comparison with Other Systems

Patterns were studied and implemented to represent in text format the user's visual selection. The majority of the systems have no support for acting on visually selected subformulas in the sequent. Notable exceptions are CtCoq and Lego in combination with Proof General, both supporting proof by pointing [12] and no longer maintained. When subformulas cannot be acted on, all the performed operations can be represented in the proof script by means of a textual command without patterns. For instance, even if rewritings are extensively used in Isabelle proof scripts, there are no comfortable facilities to restrict them to subformulas, probably because the deep integration with automated tactics makes this need less compelling.

All these considerations make a comparison of the pattern facility harder from the pragmatic point of view. A deeper and more technical comparison can be done with how Coq users restrict (by means of textual expressions, not mouse movements) the application of a tactic to subformulas occurring in a sequent.

While in Matita all the tactics that act on subformulas of the current sequent accept pattern arguments, in Coq the user has two different ways of restricting the application of tactics to subformulas, both relying on the following syntax to identify an occurrence:

```
change n at 2 in H with (O + n).
```

The idea is that to identify a subformulas of the sequent, we can write it down and say that we want, for example, its third and fifth occurrences (in a left to right textual visit of the formula). In the previous example, only the second occurrence of `n` in the hypothesis `H` would be changed. Some tactics support directly this syntax, while others need the sequent to be prepared in advance using the `pattern` tactic. Note that the choice is not left to the user and that there is no way to achieve the same result by visually selecting formulas.

The tactic `pattern` computes a β-expansion of a part of the sequent with respect to some occurrences of the given formula. In the previous example the following command

```
pattern n at 2 in H.
```

would have resulted in the sequent

$$\begin{array}{l} n : nat \\ m : nat \\ \underline{H : (fun\ n0 : nat} => m + n = n0)\ n \\ m = 0 \end{array}$$

where `H` has been β-expanded over the second occurrence of `n`. At this point, since the Coq unification algorithm is essentially first-order, the application of an elimination principle (a term of type $\forall P.\forall x.(H\ x) \to (P\ x)$) will unify `x` with `n` and `P` with `(fun n0: nat => m + n = n0)`. Since `rewrite`, `replace`, and several other tactics boil down to the application of the equality elimination principle, this trick implements the expected behavior.

The idea behind this way of identifying subformulas is similar to the patterns idea but fails in extending to complex notation because it relies on a monodimensional sequent representation. Real mathematical notation places arguments on top of each other (like in indexed sums or integrations) or even puts them inside a bidimensional matrix. In these cases using the mouse to select the wanted formula is probably the more effective way to tell the system where to act. One of the commitments of Matita is to use modern publishing techniques, so we prefer that our method not discourage the use of complex layouts.

3.3 Disambiguation

One of the most frequent activities in the interaction with any tool for mathematics is input of formulas. Since Matita uses textual typing as its input mechanism, formulas are linearized in a TEX-like encoding (a widespread choice among mathematicians) and are rendered to the user via MathML. Unicode can also be freely exploited for input of mathematical glyphs.

For the purpose of input, the main problem posed by the wish of sticking to the standard mathematical notation is its ambiguity, induced by various factors: conflicting parsing rules, hidden information to be recovered from the context, overloading, and subtyping. In the setting of Matita subtyping is implemented by means of coercive subtyping [20]: a function of type $A \to B$ declared as a *coercion* is automatically inserted by the system whenever a term of type A is used with expected type B. Since coercions are more general than subtyping, in the remainder of this section we will discuss only them.

In programming languages these challenges are usually solved by limiting the language that, being imposed to the user, can be freely modeled by its designers. The restrictions imposed are guided by performance reasons and by the need for the system, usually not interactive, to produce at the end exactly one interpretation. For instance, overloading in C++ does not allow the user to declare two functions that take the same input but returns output with different types.

Since we do not want to change mathematical notation too much, we need to drop the standard techniques. The user is no longer forced to adapt to the system, and the grand challenge becomes *automatic detection of the intended interpretation among the ones that make sense*.

The challenge, which we call *disambiguation*, can be roughly described as follows: Starting from the concrete syntax of a formula, build an internal *representation* of the formula among the ones that "make sense" for the system. We interpret the latter notion as being well typed in some weak type system (e.g., weak type theory [19] or the type system used in Mizar [9]). In the case of Matita, CIC terms are used as internal representations of formulas, and CIC plays the role of the weak type system.

Disambiguation can be split into two tasks that need not to be implemented sequentially: the first one is the resolution of ambiguities in order to map the formula to the set of possible internal representations that are well typed; the second one is to rate the obtained representations to recognize those that are more likely to have the meaning intended by the user. User interaction is requested in the case where multiple representations receive the highest rating.

In [30] we proposed an efficient algorithm to implement the first task considering only the difficulties posed by overloading. The *sources of ambiguities* the algorithm deals with are unbound identifiers (they can refer to multiple concepts available in the knowledge base), numbers, and symbolic operators (they are usually overloaded). The algorithm can easily be adapted to most type systems; its extension to also handle coercions does not pose any additional problem, and it has been implemented in Matita.

The second task (representations rating) is the topic of the remainder of this section. Provided an implementation of the first is available, the proposed solution is completely independent of the type system.

3.3.1 Representations Rating

The problem we address is finding some reasonable criteria that allow us to rate representations in order to guess most of the time the one the user meant. When the guess is wrong, the rating can still be used to present choices to the user in order of likelihood of being the wanted representation. Since the guessed choices is chosen among the maximally rated ones, rating several choices in the same way is also preferred to arbitrarily guessing an order.

Locality of reference appears to be a good criterion to solve overloading: a source of ambiguity is likely to be interpreted the same way it was interpreted last time. Consecutive lemmas, for example, are likely to be about the same concepts and thus are likely to require the same overloading solving choices. In practice this criterion allows exceptions very frequently. For instance, in the case of analysis it is common practice to mix in the same statement order relations over real numbers and order relations over natural numbers that are used to index sequences.

For this reason, rather than the criterion of locality of reference we prefer *user preferences* that can be explicitly given by the user or implicitly set by the system. At any time a stack of preferred *interpretations* is associated to any source of ambiguity; other interpretations can be retrieved from the library. An interpretation builds a representation from the representations of the subformulas of the source of ambiguity. For instance, the interpretation

```
interpretation 'plus x y =
  (cic:/matita/nat/plus/plus.con x y).
```

interprets the content-level `'plus` node as addition over the natural numbers (see Fig. 5 for an example of how notation can be associated to the content-level node [26]).

The topmost interpretation on the stack is intuitively the *current* (hence most likely) one. For instance, the user who intends to prove some lemmas over rational numbers should set preferences for interpreting the “less than” relation over natural, integer, and rational numbers, the last being the current preference. Representations of a formula that do not respect the user preferences for some symbol are not rejected but simply get a low rating. This approach allows exceptions such as stating a lemma that requires testing inequality over real numbers; however, the exceptional interpretation for “less than” is not taken into account as a new user preference (as happens with the locality of reference criterion).

The criterion based on user preferences allows the user to rate the interpretations for a single occurrence of an ambiguity source. As for overloading, a reasonable local criterion for coercions is at hand: an interpretation that does not insert a coercion in one particular position is to be preferred to an interpretation that does. The real difficulty is now the extension of rating to representations in order to take in account the rating for each ambiguity source and the insertion of coercions to make the interpretation well typed.

No reasonable criterion for the global rating of a representation is evident a priori. The one we are now using is motivated by several concrete examples found in the standard library of Matita. On the basis of the examples collected so far, we achieved results deemed satisfactory by our (small) user community rating representations as follows.

1. Representations where ambiguity sources are interpreted outside the domain of user preferences are rated $+\infty$ (worst rating) if the domain is nonempty; the representation is also rated $+\infty$ if every ambiguity source has an empty domain.

Table 2 User preferences and other interpretations in effect for the disambiguation examples. Current preference underlined

Ambiguity source	Preferences	Other interpretations
$<$, $+$, $*$	$\underline{\mathbb{R}^{\mathbb{R}\times\mathbb{R}}}, \mathbb{Z}^{\mathbb{Z}\times\mathbb{Z}}, \mathbb{N}^{\mathbb{N}\times\mathbb{N}}$	$\mathbb{C}^{\mathbb{C}\times\mathbb{C}}$
$\lvert \cdot \rvert$	$\underline{\mathbb{R}^{\mathbb{R}}}, \mathbb{N}^{\mathbb{Z}}$	$\mathbb{R}^{\mathbb{C}}$
$\sqrt{\cdot}$	$\underline{\mathbb{C}^{\mathbb{C}}}$	
$\mid$	$\underline{2^{\mathbb{Z}\times\mathbb{Z}}}, 2^{\mathbb{N}\times\mathbb{N}}$	
$\lfloor \cdot \rfloor$	$\underline{\mathbb{Z}^{\mathbb{R}}}$	
cos		$\mathbb{R}^{\mathbb{R}}$
π		{cic : /matita/reals/trigo/pi.con}
Numbers	$\underline{\mathbb{R}}, \mathbb{Z}, \mathbb{N}$	$\mathbb{C}$

2. Other representations are rated by the couple $\langle c, o\rangle \in \{0, 1\}^2$, where c is 0 if no coercion has been used while interpreting, 1 otherwise; o is the worst rating on the interpretations of the symbols, numbers, and unbound identifiers that occur in the formula, where an interpretation is rated 1 if it is a preference that is not the current one, 0 otherwise.
3. The couples $\langle c, o\rangle$ are ordered lexicographically, and $+\infty$ is greater than any couple; the best representations are those with minimal rating.

Note that the global rating is very approximative in weighting coercions with respect to the local one previously presented. This is mostly for simplifying the implementation (see Section 3.3.2).

The following examples show our criterion at work supposing that the user preferences are currently set as shown in Table 2. Coercions are automatically inserted to provide for the usual subtyping relation on number classes.

Example 5 (Preferences are respected).

```
theorem Rlt_x_Rplus_x_1: \forall x. x < x+1.
```

The best representation is $\forall x : \mathbb{R}.x <_{\mathbb{R}} x +_{\mathbb{R}} 1_{\mathbb{R}}$, that is, the only one rated $\langle 0, 0\rangle$. The current preferences of the user are respected.

Example 6 (Forcing a different representation).

```
theorem lt_x_plus_x_1: \forall x:nat. x < x+1.
```

The user can select a different representation adding just one type annotation. The best representation is $\forall x : \mathbb{N}.x <_{\mathbb{N}} +_{\mathbb{N}} 1_{\mathbb{N}}$ that is the only one rated $\langle 0, 1\rangle$.

Example 7 (The user is asked).

```
theorem divides_nm_to_divides_times_nr_times_mr:
  \forall n,m,r. n | m \to n*r | m*r.
```

In this case the current preferences cannot be satisfied. The two best representation are $\forall n, m, r : \mathbb{N}.n|_{\mathbb{N}}m \to n *_{\mathbb{N}} r|_{\mathbb{N}}m *_{\mathbb{N}} r$ and $\forall n, m, r : \mathbb{Z}.n|_{\mathbb{Z}}m \to n *_{\mathbb{Z}} r|_{\mathbb{Z}}m *_{\mathbb{Z}} r$, which are rated $\langle 0, 1\rangle$. The user is asked for the intended meaning since our global rating cannot detect that the second representation is locally better because it respects more preferences.

Example 8 (Coercions are better avoided).

```
theorem Zdivides_to_Zdivides_abs:
  \forall n,m. n | m \to |n| | |m|.
```

This case is close to the previous one, but the representation that interprets n and m as natural numbers has the worst score because it requires a coercion. Thus the best representation is $\forall n, m : \mathbb{Z}.n|_{\mathbb{Z}}m \to |n|_{\mathbb{Z}}\ |_{\mathbb{Z}}\ |m|_{\mathbb{Z}}$ that is rated $\langle 0, 1\rangle$.

Example 9 (Forcing a representation with coercions).

```
theorem divides_to_Zdivides_abs:
  \forall n,m:nat. n | m \to |n| | |m|.
```

Adding a single type annotation (`:nat`), the representation chosen in the previous example is pruned out. The best representation is now $\forall n, m : \mathbb{N}.n|_{\mathbb{N}}m \to |n|_{\mathbb{Z}}\ |_{\mathbb{Z}}\ |m|_{\mathbb{Z}}$, which is rated $\langle 1, 0\rangle$.

Example 10 (Multiple occurrences distinguished).

```
theorem lt_to_Zlt_integral:
  \forall n,m. n < m+1 \to
    \lfloor n \rfloor < \lfloor m \rfloor.
```

In this example the two less than relations are interpreted over different domains without requiring any coercion. The best representation is $\forall n, m : \mathbb{R}.n <_{\mathbb{R}} m +_{\mathbb{R}} 1_{\mathbb{R}} \to \lfloor n\rfloor <_{\mathbb{Z}} \lfloor m\rfloor$, which is rated $\langle 0, 1\rangle$.

Example 11 (Lack of preferences do not affect rating).

```
theorem Rlt_cos_0:
  \forall x. \pi / 2 < x \to x < 3*\pi/2 \to \cos x < 0.
```

No preferences are given for π and cos. Thus both operators can be interpreted over the real numbers without affecting the rating. The user preferences for less-than are still in effect. Thus the best representation is $\forall x : \mathbb{R}.\pi/_{\mathbb{R}}2_{\mathbb{R}} <_{\mathbb{R}} x \to x <_{\mathbb{R}} 3_{\mathbb{R}} *_{\mathbb{R}} \pi/_{\mathbb{R}}2_{\mathbb{R}} \to \cos x <_{\mathbb{R}} 0_{\mathbb{R}}$ that is rated $\langle 0, 0\rangle$.

Example 12 (Preferences are not respected).

```
theorem lt_to_Clt_sqrt:
  \forall n,m. n < m \to \sqrt n < \sqrt m.
```

This case is similar to the previous one, but the preferences of the user for less than cannot be satisfied. Thus the only two representations are $\forall n, m : \mathbb{Z}.n <_{\mathbb{Z}} m \to \sqrt{n} <_{\mathbb{C}} \sqrt{m}$ and $\forall n, m : \mathbb{N}.n <_{\mathbb{N}} m \to \sqrt{n} <_{\mathbb{C}} \sqrt{m}$, both rated $+\infty$. The user is asked for the intended meaning.

3.3.2 Implementation

In having to deal with multiple representations that need to be checked for well-typedness (first task) and to be rated (second task), the issue of performance needs to be faced. Indeed, a priori the number of potential representations is exponential in the number of ambiguity sources. The algorithm we presented in [30] for the first task is linear in the number of well-typed representations that, in nonartificial cases, is almost linear in the number of ambiguity sources.

To reduce the cost of the second task, we generate the potential representations in increasing order of rating, stopping as soon as a nonempty set of well-typed representations is found. The technique is to interleave the two tasks, tuning the parameters of the disambiguation algorithm of [30] in order to bound the rating of the generated representations. The parameters are a flag that triggers the insertion of coercions and a map from sources of ambiguity to their domain of preferences. In Table 3 we show how the parameters are set in the five disambiguation attempts tried in order. Since subsequent attempts reconsider representations generated in previous attempts, memorization can be used to speed up the process.

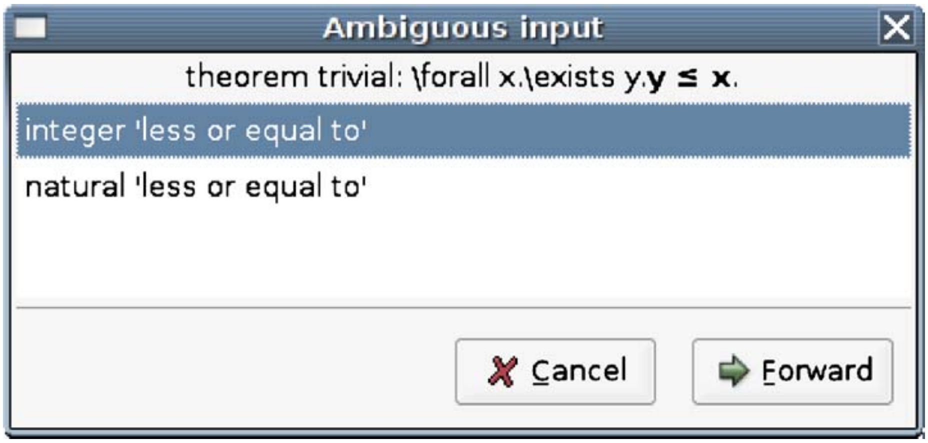
One issue not yet addressed is how the user sets the preferences. The basic mechanism consists in adding explicit commands to the script, as shown in the following example.

```
alias ident "i" = "cic:/matita/complex/i.con"
alias symbol "plus" = "addition over real numbers"
alias num = "real numbers"
```

For unbound identifiers (first line) the preference is set by giving the URI of a concept in the library. For symbols and numbers (second and third lines) the preference is set by using the label given when an interpretation was declared.

Table 3 Disambiguation attempts sequence

Attempt no.	Preferences	Coercions	Max. rating
1	Current preferences only	Disabled	$\langle 0, 0\rangle$
2	All preferences	Disabled	$\langle 0, 1\rangle$
3	Current preferences only	Enabled	$\langle 1, 0\rangle$
4	All preferences	Enabled	$\langle 1, 1\rangle$
5	No preferences	Enabled	$+\infty$



theorem trivial: \forall x.\exists y.y ≤ x.

integer 'less or equal to'

natural 'less or equal to'

Fig. 6 Ambiguity resolution

To reduce the need to give preferences explicitly, we provide two complementary mechanisms. The first one is *inclusion*: an explicit command is provided to import all the preferences that were in effect at the end of the execution of a given script. The command does not load any concept from the library, nor it is required before using a concept: it affects only preferences. The second mechanism consists in automatically setting implicit preferences for concepts and new notations just defined.

An additional issue consists in remembering the outcome of user interaction in order to avoid it the next time the script is executed. This approach also allows future batch compilation of the script. At the end of the interaction a minimal set of preferences, which would have been sufficient to avoid asking the user, is automatically added to the script as a set of explicit preferences (`alias` commands).

An orthogonal problem is that of finding a suitable user interface for letting the user select the intended representation in case of multiple representations with the same maximal rating. A list requires too much effort to be investigated and stresses the problem of providing an immediate feedback of the different choices.

Our solution, shown in Fig. 6, consists in posing to the user a sequence of simple questions: each question is about the interpretation of a single source of ambiguity that is highlighted in the formula. The questions that prune more representations are asked first. This interface is a great improvement over the one that shows all possible representations at once because the user does not have to reason globally on representations, but locally on single sources of ambiguity.

3.3.3 Related Issues

Since the internal representation of formulas has more information than what has been typed by the user, two issues arise: that of providing enough feedback to the user on the additional information in nonintrusive ways and that of pretty-printing the formulas to the user closely to what was typed.

We believe that the feedback should be provided only on demand, since permanent feedback by means of colors or subscripts inserted in the formula is both distracting for the user and insufficient to show all the information automatically inferred by the system but hidden in the standard mathematical notation. Thus the main feedback we provide is by means of hyperlinks from every source of ambiguity to the mathematical concepts used in its interpretation (see Section 3.1.1). The remaining hidden information such as the coercions inserted is shown when the user asks the system to temporarily disable mathematical notation or coercion hiding. Right now, disabling notation affects the pretty-printing of the whole formula. Doing this only for subformulas is a possible future work that right now we do not feel urged to do.

Pretty-printing of formulas in a syntax close to what the user has typed is currently not implemented in Matita, and it can sometimes be annoying when the system needs to report messages that include the formula. We plan to implement this as a future work. However, error messages are already localized in the script-editing window: the subformula the error message refers to is underlined in red in the script. This approach greatly reduces the need to show formulas in the error messages.

After disambiguation, Matita keeps only one well-typed representation for the formula typed by the user, requiring his help in case of draws in representations rating. An alternative approach that we do not fully implement in Matita consists in keeping the whole set of well-typed representations that received the maximal rating; further operations on the representations in the set will prune out those elements that produce errors when used as arguments. For instance, with this approach the statement of a theorem can be left ambiguous and will be clarified while proving.

Metavariables, which are typed placeholders for missing terms, already provide in the logic of Matita the possibility of representing in a single term a set of representations. The actual representations can be obtained by instantiating the metavariable compatibly with the typing rules. This opportunity comes for free and is already heavily exploited. For instance, if `t` is a theorem that proves `\forall x,y:A. P x \to P y \to x = y` and `H` is a proof of `(P 2)`, it is possible in Matita to pass to a tactic the term `(t ? ? ? H)` that, after disambiguation, becomes `(t ?1 2 ?2 H)`, which represents at once a whole set of terms. The tactic can use the term in a context where the term is expected to have type $n = 2$, further instantiating it to `(t n 2 ?2 H)`.

In Matita we do not keep explicitly those sets of representations that cannot be abstracted in a single term of the logic by means of metavariables. Doing so would pose additional problems on the user interface in providing feedback on the set of terms used as representations of a formula. Moreover, most of the time the performances of the system would suffer because every operation on a representation should be applied in turn to each element of the set. When the operation under discussion is the application of a long-running automation tactic, the additional time spent in trying every interpretation can really make the tactic unusable. However, this topic deserves further investigation.

3.3.4 Comparison with Other Systems

Overloading and subtyping are addressed in most interactive theorem provers. We compare with Coq and Isabelle, which have adopted alternative solutions for both problems.

Overloading in Coq, Matita, and the majority of systems is done at the syntactic level and must be resolved during the parsing (or disambiguation) phase. On the contrary, overloading in Isabelle [35] is a special form of constant definition where a constant is declared as an axiom with a certain generic type; recursive rewriting rules are associated to occurrences of the axiom specialized to certain types. The rules must satisfy some criteria that grant the logical consistency of the declaration. As explained in [24], the criterion adopted in Isabelle 2005 is not sufficient for consistency; an alternative criterion proposed in the same paper accepts most common examples but requires detecting termination in a particular term rewriting system that is associated to the overloaded definition.

We feel that overloading should be considered a user interface issue to be addressed in a logic-independent way and possibly to be implemented as a stand-alone component to be plugged in different systems. As previously discussed, our solution can be parameterized on the type system, whereas the solution of Isabelle is more tightly bound to the logic because of the need of detecting consistent definitions (that, in Matita, has already been done before declaring the overloaded notation). Moreover, in Isabelle overloading should always be combined with type classes to restrict the type of the arguments of the overloaded functions; otherwise type inference becomes too liberal, inferring well-typed representations that are not those intended by the user.

Overloading in Coq is more restricted than in Matita. Each symbol can be overloaded several times, but each overloading must have a different return type (in Coq terminology, it must belong to a different *interpretation scope*). When the application of a function to an overloaded notation is interpreted, the type expected by the function determines the only interpretation of the overloaded notation with that return type. The user is also given special syntax to explicitly change the interpretation scope for a subformula.

Coq's mechanism allows to perform disambiguation in linear time, but it suffers from several important limitations. When a function is polymorphic, for example, it has type `\forall A:Type. A \to` ldots, Coq is unable to associate an interpretation scope to the second argument of the function because the expected type will be known only after the interpretation of the first argument. Moreover interpretation scopes do not mix well with subtyping: if the current interpretation scope is that of integer numbers and if multiplication is overloaded in that scope with type `Z -> Z -> Z`, then `forall (n:nat), n * 1 = n` will be rejected because the first argument of multiplication will be also parsed in the interpretation scope of integer numbers without attempting any insertion of coercions (that is, performed only during type checking, after disambiguation). Finally, the restriction on overloading inside an interpretation scope is problematic in several examples where the type of the arguments of the overloaded notation, and not the return type, differentiates the interpretations.

Subtyping is implemented in Coq and Matita by means of coercive subtyping. The two systems allows one to declare general coercions, which are useful not only to simulate subtyping. In order to prove generic theorems over algebraic hierarchies, dependent records in the spirit of [15] are exploited. The subtyping relation for dependent records is replaced by ad hoc coercions. This approach has been successfully used in Coq for the Constructive Coq Repository of Nijmegen [16], where the standard algebraic and arithmetic hierarchies have been developed up to ordered complete fields and complex numbers, respectively.

Isabelle does not have coercions, but it implements type classes that can be exploited to represent algebraic hierarchies thanks to the subtyping relation between type classes. Type classes have been inspired by the similar concept in Haskell, which behind the scenes is implemented by using a technique similar to dependent records. Thus the approach of Isabelle can be seen as more abstract with respect to the one of Matita, and it can be more natural for the user. The price to pay for not having coercions is that quite often the user needs to explicitly insert functions to embed values of a data type into values of a "supertype" that is a completion of the first one that requires a different representation. This is, for instance, the case when the user needs to map a natural number n into the correspondent integral representative $\langle n, 0\rangle$ (integer numbers are quotiented pair of natural numbers).

3.4 Step-by-Step Tacticals

Two of the most frequent activities in interactive proving are proof step formulation and execution [1]. In the Proof General interaction paradigm, adopted by Matita, they are implemented respectively by textually typing a command in the script editing window and moving the *execution point* past the typed text. Already executed parts of the script are locked to avoid accidental editing and highlighted to make evident the position of the execution point. Feedback to the user is provided by rendering in the sequents window the information about what need to be done to conclude the proof.

In Matita, additional feedback can be given on demand by means of pseudo-natural language representation of the current proof term, to verify whether it corresponds to the intended proof plan.

3.4.1 LCF Tacticals: State of the Art

Tactics, which represent single proof steps in a script, can be combined to create complex proof strategies using higher-order constructs called *tacticals*. They first appeared in LCF [17] in 1979 and are nowadays adopted by several mainstream provers like Isabelle, Coq, PVS, and NuPRL. In this section we present the state of the art for LCF tacticals, emphasizing the user interface aspects to reveal several problems. In Matita we solve these problems by replacing LCF tacticals with *tinycals*, which will be briefly discussed in the next section.

Paradigmatic examples of LCF tacticals are *sequential composition* and *branching*. The former, usually written as "t_1 ; t_2", takes two tactics t_1 and t_2 and applies t_2 to each of the goals resulting from the application of t_1 to the current goal; sequential composition can also be repeated to obtain pipelines of tactics "t_1 ; t_2 ; t_3 ; $\cdots$". The latter, "t ; [t_1 | $\cdots$ | t_n]", takes $n + 1$ tactics, applies t to the current goal, and, requiring t to return exactly n goals, applies t_1 to the first returned goal, t_2 to the second, and so forth. The script snippet of Fig. 7 is structured with LCF tacticals, expressed by using the concrete syntax of Matita tinycals.

LCF tacticals account for two improvements over plain tactic sequences. The first improvement is their ability to support *script structuring*. Using branching, indeed the script representation of proofs can mimic the structure of the proof tree (the tree having sequents as nodes and tactic-labeled arcs). Since proof tree branches usually reflect conceptual parts of the pen-and-paper proof, the branching tactical helps in improving script readability, which is on the average very poor, if compared with declarative proof languages.

```
theorem lt_O_defact_aux:
  \forall f:nat_fact. \forall i:nat. O < defact_aux f i.
intro; elim f;
[1,2:
  simplify; unfold lt;
  rewrite > times_n_SO;
  apply le_times;
  [ change with (O < \pi _ i);
    apply lt_O_nth_prime_n
  |2,3:
    change with (O < (\pi _ i)^n);
    apply lt_O_exp;
    apply lt_O_nth_prime_n
  | change with (O < defact n1 (S i));
    apply H ] ].
qed.
```

Fig. 7 Sample Matita script with tacticals

Maintainability of proof scripts is also improved by the use of branching, for example when hypotheses are added, removed, or permuted, since a static look at the script can spot where changes are needed. The second improvement is the degree of *conciseness* that can be achieved by factoring out common parts of proof scripts using sequential composition. According to the proof as programming metaphor [2], factorization of common scripts parts is a practice as good as factorization of code in ordinary programming.

The major drawback of LCF tacticals is in how they interact with Proof General-like user interfaces. In all the proof assistants we are aware of, a tactic obtained applying a tactical is evaluated atomically, and placing the execution point in the middle of complex tacticals (for example, at occurrences of "**;**" in pipelines) is not allowed. Thus, a fully structured script as the one shown in Fig. 7 can be executed only atomically, either successfully (without any feedback on the inner proof status the system passes through) or with a failure that is difficult to understand. Several negative effects on the usability of the authoring interface of such provers are induced by such coarse-grained execution.

The first effect is the impossibility of showing interactively what is happening during the evaluation of complex and potentially large script snippets. This aspect is particularly annoying given that scripts by themselves are rarely meaningful without interactive proof replaying.

The second negative effect is in the inefficient practice of proof development induced on users willing to obtain structured scripts. Since it is not always possible to predict the outcome of complex tactics, the following is common practice among them:

1. evaluate the next tactic of the script;
2. inspect the set of returned sequents;

3. decide whether the use of "**;**" or "**[**" is appropriate;
4. if it is, retract the last statement, add the tactical, and go to step (1).

The third and last negative effect is imprecise error reporting. Consider the frequent case of a script breakage induced by changes in the knowledge base. The error message returned by the system at a given execution point may concern an inner status unknown to the user, since the whole tactical is evaluated at once. Moreover, the error message will probably concern terms that do not appear verbatim in the script. Finding the statement that need to be fixed is usually done by replacing tactics with the identity tactic (a tactic that simply returns unchanged the goal it was applied to) proceeding outside-in, until the single failing tactic is found. This technique not only is error prone but is not reliable even in the presence of *side effects* (tactics closing sequents other than that on which they are applied), since the identity tactic has no side effects and proof branches may be affected by their absence.

3.4.2 Matita Tinycals

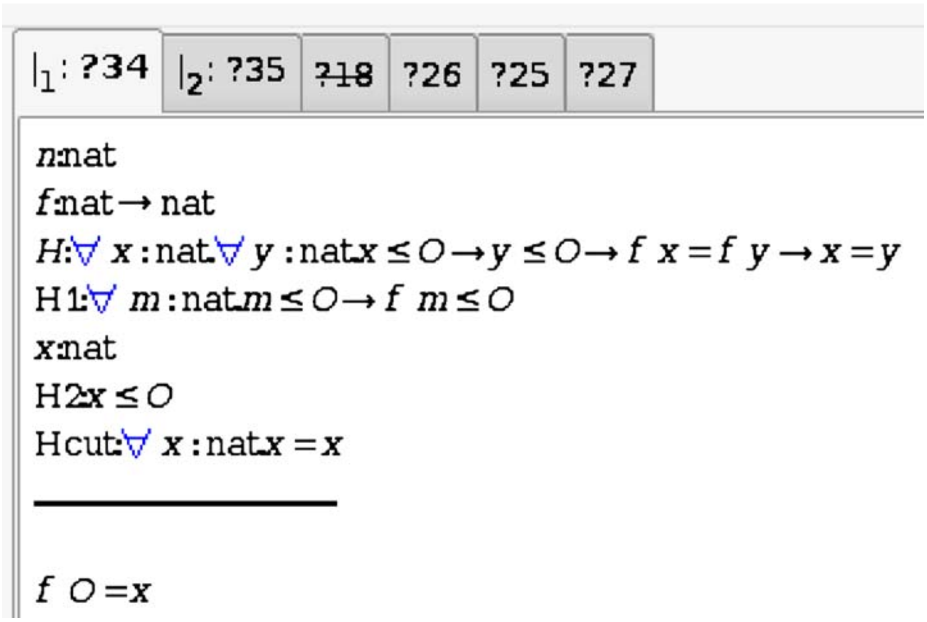
In Matita we break the tension between LCF tacticals and Proof General-like interfaces by developing a new language of tacticals called *tinycals* [29]. Tinycals can be evaluated in small steps, enabling the execution point to be placed inside complex structures such as pipelines or branching constructs. The syntax of tacticals is destructured and the semantics modified accordingly, enabling parsing and immediate execution of fine-grained constituents of LCF tacticals. In Fig. 2, for instance, the occurrence of "**[**" just before the execution point has been parsed and executed without executing the whole "**[** ... **]**" block.

The semantics of tinycals is stated as a transition system over *evaluation status*, structures richer than the proof status tactics act on. The key component of an evaluation status is the context stack, a representation of the proof history up to the execution point. The context stack is ruled by a last-in-first-out policy: levels get pushed on top of it when tinycals that behave like "**[**" are executed and get popped out of it when tinycals that behave like "**]**" are.

In this way users can incrementally author script blocks encouraging both proof structuring and conciseness. Tinycals also provide additional improvements not found in other tactical languages: nonstructured sequences of tactics are locally permitted inside single branches of structured snippets; addressing of several branches at once is possible (using, for example, the "`1,2:`" syntax that can be found in Fig. 7); special support for focusing on a group of related goals is provided.

3.4.3 Visual Representation of the Context Stack

The user interface of Proof General already presents part of the proof status to the user in the sequents window. Our implementation in Matita is based on a tabbed window, with one tab per sequent (see Fig. 8, where a zoomed sequents window is shown). Since the context stack is peculiar of tinycals, we decided from scratch how to present it to the user. The stack being a run-time representation of (part of) the proof tree, the idea of representing it visually as a tree is tempting. However, we failed to see any real advantage in such representation, and past empirical studies on the topic of proof status representation failed to see evidence of the utility of similar representations [2].



n:nat
f:nat → nat
H:$\forall\, x$: nat.$\forall\, y$: nat.$x \le O \to y \le O \to f\ x = f\ y \to x = y$
H1:$\forall\, m$: nat.$m \le O \to f\ m \le O$
x:nat
H2:$x \le O$
Hcut:$\forall\, x$: nat.$x = x$

$f\ O = x$

Fig. 8 Visual representation of the context stack in the sequents window

Our representation choice can be seen in the upper part of Fig. 8. The uppermost level of the context stack (the *current branching context*) is represented as tab label annotations. All goals on which the next tactic in a pipeline will be applied have their labels typeset in boldface (only **?34** in the example), goals of the current branching context have labels presented by $|_n$ (where n is their positional index, $|_1$: ?**34** and $|_2$: ?35 in the example), and goals already closed by side effects have strike-through labels (~~?18~~ in the example). All remaining goals are simply shown by using their unique identifiers (e.g. ?25, ?26, ?27).

This choice makes the user aware of which goals will be affected by a tactic evaluated at the execution point and of all the indexing information that might be needed there. Yet, this user interface choice minimizes the drift from the usual way of working in the Proof General interaction paradigm, welcoming users used to other systems.

3.4.4 Related Work

Two other approaches for authoring structured HOL scripts have been proposed in [33] and [34]. The first approach, implemented in Syme's TkHOL, is similar to tinycals but was lacking a formal description that we provide in [29]. Moreover, unlike HOL, we consider a logic with metavariables that can be closed by side effects. Therefore the order in which branches are closed by tactics is relevant and must be made explicit in the script. For this reason we support focusing and positional addressing of branches which were missing in TkHOL.

The second approach, by Takahashi et al., implements syntax directed editing by automatically claiming lemmata for each goal opened by the last executed tactic. This technique breaks down with metavariables, however, because they are not allowed in the statements of lemmata.

4 Conclusions

In this paper we presented the interactive theorem prover Matita, focusing on user interface aspects. We reported the main motivations behind our design choices, but we silently omitted the historical perspective. The overall choice to build the system aggregating many available components and technologies (like XML, MathML, GTK+, and relational databases) is a clear consequence of the way the system was born.

In the remainder of this section we discuss how the origins of Matita influenced its design and our plans for system development.

4.1 Historical Perspective

The origins of Matita go back to 1999. At the time we were interested mostly in developing tools and techniques to enhance the accessibility, via a Web of libraries, of formalized mathematics. Because of its dimension, the library of the Coq proof assistant (of the order of 35,000 theorems) was chosen as a privileged test bench for our work, although experiments have been also conducted with other systems, and notably with NuPRL. The work, performed mostly in the framework of the recently concluded European project MoWGLI [7], consisted mainly in the following steps.

1. Exporting the information from the internal representation of Coq to a system- and platform-independent format. Since XML was at the time an emerging standard, we naturally adopted that technology, fostering a content-centric architecture [5] where the documents of the library were the main components around which everything else has to be built.
2. Developing indexing and searching techniques supporting semantic queries to the library.
3. Developing languages and tools for a high-quality notational rendering of mathematical information.

According to our content-centric commitment, the library exported from Coq was conceived as being distributed, and most of the tools were developed as Web services. The user interacts with the library and the tools by means of a Web interface that orchestrates the Web services.

Web services and other tools have been implemented as front-ends to a set of software components, collectively called the HELM components. At the end of the MoWGLI project we already had the following tools and software components at our disposal:

- XML specifications for the Calculus of Inductive Constructions, with components for parsing and saving mathematical concepts in such a format [27];
- metadata specifications with components for indexing and querying the XML knowledge base;
- a proof checker (i.e., the *kernel* of a proof assistant), implemented to check that we exported from the Coq library all the logically relevant content;

– a sophisticated term parser (used by the search engine), able to deal with potentially ambiguous and incomplete information, typical of the mathematical notation [30];
– a *refiner* component (i.e., a type inference system), based on partially specified terms, used by the disambiguating parser [28];
– algorithms for proof rendering in natural language [6];
– an innovative, MathML-compliant rendering widget [25] for the GTK+ graphical environment, supporting high-quality bidimensional rendering and semantic selection.

Starting from all these, developing our own proof assistant was not out of reach: we "just" had to add an authoring interface, a set of functionalities for the overall management of the library and integrate everything into a single system. Matita is the result of this effort.

4.2 On the Exploitation of Standard Technologies

The software components Matita has been built on are heavily based on a multitude of libraries for standard tasks such as XML parsing, construction of DOM trees, and syntax highlighting. These libraries are often written in low-level languages (C or C++) and have independent wrappers for the OCaml language. The neat result is increased robustness of the code and more advanced functionalities, but also a heavy set of dependencies needed to compile and run Matita. Similarly, the dependency on a relational database provides scalability and good performance but introduces additional configuration complexity.

Most of the other systems have adopted the opposite approach of providing self-contained code, at the price of reimplementing standard functionalities for the benefit of simplified installation procedures. The same can be said of generic graphical interfaces such as Proof General, which so far has been entirely based on Emacs.

We think that an interesting goal for the research community is to split as much as possible the logic-dependent code from the code that deals with interface issues and to further separate the parts related to the user interfaces. The aim is to provide not only generic user interfaces but also generic components for more advanced tasks. The logic-dependent code could be self-contained for easy installation, while the generic components, to be installed only once and to be developed collaboratively by the whole community, should rely on external, state-of-the-art libraries. We will now clarify the proposed splitting as components layering.

4.3 Components Layering in Interactive Theorem Proving

Even if Matita runs as a single process, its code can be roughly layered into three layers. The *inner layer* (about 40,000 lines of OCaml code, 59% of the total), heavily logic dependent, is responsible for proof verification and tactic execution. It relies on a library for parsing XML files, used for storing and sharing the proof objects.

The *outer layer* (about 8,000 lines of OCaml code, 12% of the total) is the graphical user interface. It is responsible for final rendering of formulas and error messages, syntax highlighting of scripts, representations of graphs, semantic selection, and so on. It is currently based on high-quality libraries constantly improved by the lively community of free software developers. Among them we use GTK+ (a graphical toolkit), Glade (rapid application development of user interfaces), GtkMathView (MathML+BoxML rendering), GtkSourceView (syntax highlighting), GraphViz (graphs rendering), and Gdome (model-view-controller for MathML documents).

The *middle layer* (about 20,000 lines of OCaml code, 29% of the total), the most innovative one, is largely logic independent. It is responsible for indexing and searching facilities (based on a logic-independent metadata set and on the use of a relational database), transformation from notational-level terms to content-level terms (parsing and rendering, preserving cross references), and disambiguation of content-level formulas (can be abstracted on a few logic-dependent functionalities). We regard all these functionalities as related to user interaction but independent from the user interface of the system. We claim that the middle layer of generic mechanisms we presented can conveniently fit between the kernel of other existing interactive theorem provers and their user interfaces, exposing nice features to the user without requiring major modification to the systems.

4.4 Future Work

In the near future we plan to continue the development focusing on and enhancing the peculiarities of Matita, starting from its document-centric philosophy. In particular, we do not plan to maintain the library in a centralized way, as most of the other systems do. On the contrary, we are currently developing Wiki-technologies to support collaborative development of the library, encouraging people to expand, modify, and elaborate previous contributions. As a first step in this direction, we will integrate Matita with a revision control system, building on the invalidation and regeneration mechanisms to grant logical consistency.

Thanks to an increasing dissemination activity, we hope in the medium term to attract users to form a critical mass and enter in direct competition with the (still too few) major actors in the field. To this aim, we are also progressing in the development of the standard library, mainly to identify possible unnoticed problems and to give evidence of the usability of the system.

Acknowledgements We thank all the people that during the past seven years collaborated in the HELM project and contributed to the development of Matita, and in particular P. Di Lena, M. Galatà, A. Griggio, F. Guidi, L. Padovani, I. Schena, M. Selmi, and V. Tamburrelli.

References

1. Aitken, S.: Problem solving in interactive proof: a knowledge-modelling approach. In: European Conference on Artificial Intelligence (ECAI), pp. 335–339 (1996)
2. Aitken, S., Gray, P., Melham, T., Thomas, M.: Interactive theorem proving: an empirical study of user activity. J. Symb. Comput. **25**(2), 263–284 (1998)
3. Asperti, A., Guidi, F., Padovani, L., Sacerdoti Coen, C., Schena, I.: Mathematical knowledge management in HELM. Ann. Math. Artif. Intell. **38**(1–3), 27–46 (2003)
4. Asperti, A., Guidi, F., Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: A content based mathematical search engine: Whelp. In: Post-proceedings of the Types 2004 International Conference. LNCS, vol. 3839, pp. 17–32 (2004)
5. Asperti, A., Padovani, L., Sacerdoti Coen, C., Schena, I.: Content-centric logical environments. Short presentation at the Fifteenth IEEE Symposium on Logic in Computer Science, 2000

6. Asperti, A., Padovani, L., Sacerdoti Coen, C., Schena, I.: XML, stylesheets and the re-mathematization of formal content. In: Proceedings of EXTREME Markup Languages, 2001
7. Asperti, A., Wegner, B.: An approach to machine-understandable representation of the mathematical information in digital documents. In: Electronic Information and Communication in Mathematics. LNCS, vol. 2730, pp. 14–23 (2003)
8. Aspinall, D.: Proof general: A generic tool for proof development. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000. LNCS, vol. 1785 (2000)
9. Bancerek, G.: On the structure of Mizar types. Electron. Notes Theor. Comput. Sci. **85**(7), (2003)
10. Bancerek, G., Rudnicki, P.: Information retrieval in MML. In: Proceedings of the Mathematical Knowledge 2003. LNCS, vol. 2594 (2003)
11. Bertot, Y.: The CtCoq System: design and architecture. Form. Asp. Comput. **11**, 225–243 (1999)
12. Bertot, Y., Kahn, G., Théry, L.: Proof by pointing. In: Symposium on Theoretical Aspects Computer Software (STACS). LNCS, vol. 789 (1994)
13. Buchberger, B., Craciun, A., Jebelean, T., Kovacs, L., Kutsia, T., Nakagawa, K., Piroi, F., Popov, N., Robu, J., Rosenkranz, M., Windsteiger, W.: Theorema: towards computer-aided mathematical theory exploration. Journal of Applied Logic. **4**(4), 470–504 (December 2006)
14. Colton, S.: Automated Theory Formation in Pure Mathematics. Springer, Berlin Heidelberg New York (2002)
15. Coquand, T., Pollack, R., Takeyama, M.: A logical framework with dependently typed records. Fundam. Inform. **65**(1-2), 113–134 (2005)
16. Cruz-Filipe, L., Geuvers, H., Wiedijk, F.: C-CoRN, the constructive coq repository at Nijmegen. In: MKM, pp. 88–103 (2004)
17. Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF: a mechanised logic of computation. In: LNCS, vol. 78 (1979)
18. Hutter, D.: Towards a generic management of change. In: Workshop on Computer-supported Mathematical Theory Development, IJCAR (2004)
19. Kamareddine, F., Nederpelt, R.: A Refinement of de Bruijns formal language of mathematics. J. Logic, Lang. Inf. **13**(3), 287–340 (2004)
20. Luo, Z.: Coercive subtyping. J. Log. Comput. **9**(1), 105–130 (1999)
21. McCasland, R.L., Bundy, A., Smith, P.F.: Ascertaining mathematical theorems. Electron. Notes Theor. Comput. Sci. **151**(1), 21–38 (2006)
22. McCune, W., Wos, L.: Otter-The CADE-13 competition incarnations. J. Autom. Reason. **18**(2), 211–220 (1997)
23. Nieuwenhuis, R., Rubio, A.: Paramodulation-based Theorem Proving. vol. 1, pp. 371–443. Elsevier and MIT Press. ISBN-0-262-18223-8 (2001)
24. Obua, S.: Conservative overloading in higher-order logic. In: Rewriting Techniques and Applications. LNCS, vol. 4098, pp. 212–226 (July 2006)
25. Padovani, L.: MathML formatting. PhD thesis, University of Bologna (2003)
26. Padovani, L., Zacchiroli, S.: From notation to semantics: there and back again. In: Proceedings of Mathematical Knowledge Management 2006. Lectures Notes in Artificial Intelligence, vol. 3119, pp. 194–207 (2006)
27. Sacerdoti Coen, C.: From proof-assistans to distributed libraries of mathematics: tips and pitfalls. In: Proceedings of the Mathematical Knowledge Management 2003. LNCS, vol. 2594, pp. 30–44 (2003)
28. Sacerdoti Coen, C.: Mathematical knowledge management and interactive theorem proving. PhD thesis, University of Bologna (2004)
29. Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: Tinycals: step by step tacticals. In: Proceedings of User Interface for Theorem Provers. ENTCS **174**(2), pp. 125–142 ISSN: 1571–0661 (May 2007)
30. Sacerdoti Coen, C., Zacchiroli, S.: Efficient ambiguous parsing of mathematical formulae. In: Proceedings of Mathematical Knowledge Management 2004. LNCS, vol. 3119, pp. 347–362 (2004)
31. Shneiderman, B.: Direct manipulation for comprehensible, predictable and controllable user interfaces. In: Proceedings of the 2nd International Conference on Intelligent User Interfaces. New York, NY, pp. 33–39 (1997)
32. Sutcliffe, G.: The CADE-20 automated theorem proving competition. AI Commun. **19**(2), 173–181 (2006)
33. Syme, D.: A new interface for HOL – ideas, issues and implementation. In: Proceedings of Higher-order Logic Theorem Proving and its Applications. 8th International Workshop, TPHOLs 1995. LNCS, vol. 971, pp. 324–339 (1995)

34. Takahashi, K., Hagiya, M.: Proving as editing HOL tactics. Form. Asp. Comput. **11**(3), 343–357 (1999)
35. Wenzel, M.: Type classes and overloading in higher-order logic.. In: TPHOLs, pp. 307–322 (1997)
36. Werner, B.: Une théorie des constructions inductives. PhD thesis, Université Paris VII (1994)
37. Zacchiroli, S.: User interaction widgets for interactive theorem proving. PhD thesis, University of Bologna (2007)