User Interaction with the Matita Proof Assistant

Andrea Asperti (asperti@cs.unibo.it), Claudio Sacerdoti Coen (sacerdot@cs.unibo.it), Enrico Tassi (tassi@cs.unibo.it) and Stefano Zacchiroli (zacchiro@cs.unibo.it)
Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7 — 40127 Bologna, ITALY

Abstract. Matita is a new, document-centric, tactic-based interactive theorem prover under development at the University of Bologna. This paper focuses on some of the distinctive features of the user interaction with Matita, mostly characterized by the organization of the library as a searchable knowledge base, the emphasis on a high-quality (MathML-based) notational rendering, and the complex interplay between syntax, presentation, and semantics (e.g. for disambiguation of formulae input by the user, and direct manipulation of terms).

Keywords: Proof Assistant, Interactive Theorem Proving, Digital Libraries, XML, Mathematical Knowledge Management, Authoring

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

Table of Contents

1. Introduction

Matita (that means *pencil* in Italian) is the interactive theorem prover (or proof assistant) under development by the HELM team [3] at the University of Bologna, under the direction of Prof. Asperti. Matita is based on the same foundational framework of Coq, namely the Calculus of Inductive Constructions (CIC) [39], allowing a direct (and for obvious reasons privileged) comparison with the latter system. Matita however is not a re-implementation of Coq: it is an alternative implementation of the Calculus of Inductive Constructions, with many distinctive architectural and methodological features, from the organization and management of the library as a suitably indexed, searchable and browsable repository, to the emphasis on usability aspects and high-quality notational rendering.

The proof language of Matita is procedural, following a tradition going back to the LCF theorem prover [19], and adopted by many other successful tools like Coq, NuPRL, PVS, and Isabelle (the latter also supporting an alternative declarative proof language). Traditionally, in a procedural system, the user only interacts via a *script* of commands, while proof terms are internal records kept by the system for efficiency reasons. In Matita, which has been conceived for a potentially distributed library, proof terms (exported to an XML dialect) are also meant as the primary data type for long-term storage and communication.

All the user interfaces currently adopted by proof assistants based on a procedural proof language 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], that has set a sort of standard interaction paradigm between the user and the system. The authoring interface of Matita (shown in Figure 1) adopts the same interaction paradigm and essentially offers the same functionalities of Proof General (and CoqIde, a Coq-specific user interface) but, in our case, the granularity of the execution steps of scripts is more fine grained, resulting in a very valuable support towards structured editing of proof scripts (see Section 3.4).

On the other side we sensibly 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). That was also our main reason for not adopting Proof General. In Matita, the sequents window is based on a MathML-compliant GTK+ widget, originally conceived and developed within the HELM-team for this use in the

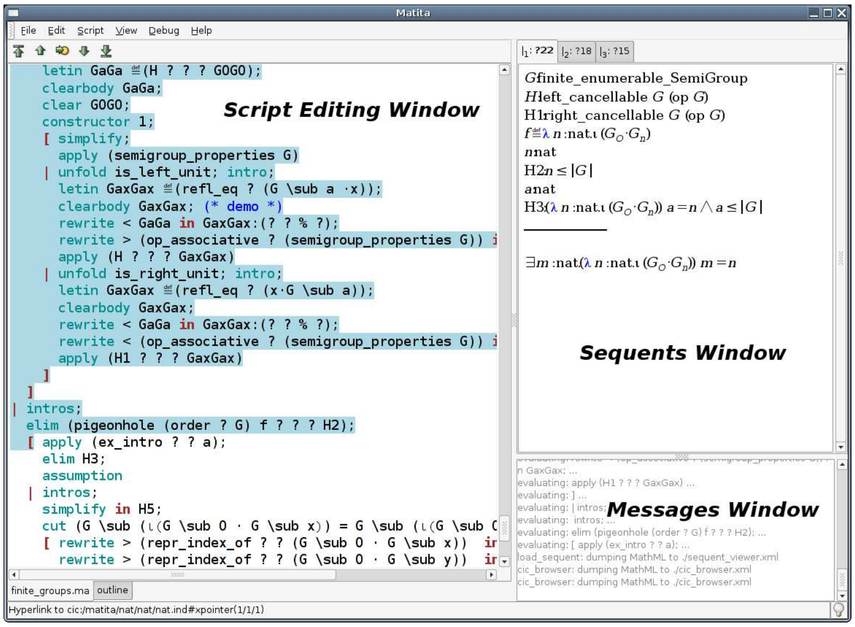

Figure 1. Authoring interface of Matita.

proof assistant and then evolved into an independent and successful component of many well known applications (such as AbiWord).

We also offer an additional utility window (the *CIC browser*, see Figures 2, 3, 4, and 5), for browsing and searching the knowledge base of Matita, directly inspired by our past work on Web interfaces for digital libraries of formalized mathematics. Proofs, even non terminated ones, can be rendered in pseudo-natural language, directly generated from CIC proof terms. The natural language is precise enough to be reused as a declarative language for Matita, and we are currently exploring this research direction.

Matita is free software and implemented in OCaml.[1] The source code is available for download at `http://matita.cs.unibo.it`.

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 and most of the current tool for the inter-

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

Figure 2. Browsing the library.

active 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. This way the library forms a global (potentially distributed) hypertext. Additionally, the dependencies can be visually inspected as a graph directly within the CIC browser (see Figure 5).

2.1. Indexing and searching

In order 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]. Let us just 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 form innermost occurrences). This approach is extremely flexible, since extending the set

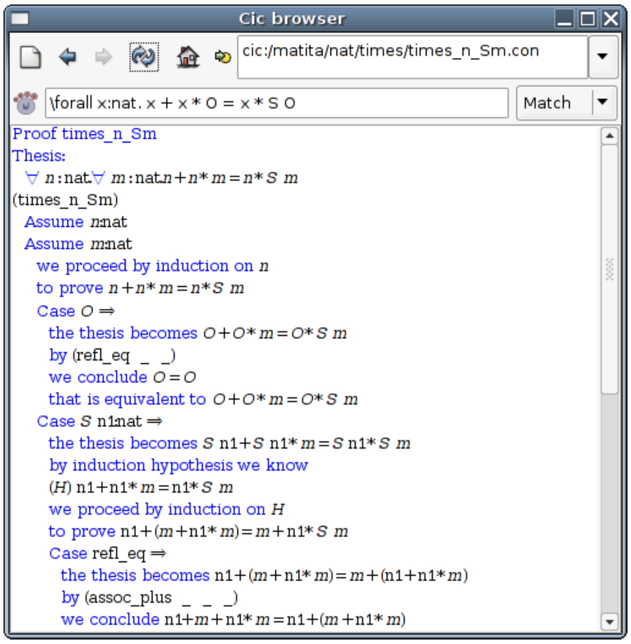

Figure 3. Browsing a proof in pseudo natural language.

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 3.2), made even more impelling by the "global" visibility policy of Matita, and the support for automatic theorem proving (Section 2.3), that is the current main research direction of Matita. Here, we shall just recall some of its most direct applications.

A first, very simple but not negligible feature is *duplicate check*. As soon as a theorem is stated, just before starting its proof, the library is searched to check that no other equivalent statement has been already proved (based on the pattern matching functionality of Whelp, see Figure 4); if this is the case, a warning is raised to the user to encourage reuse of past results. At present, the notion of equivalence adopted by Matita is convertibility, but we may imagine to weaken it in the future, covering for instance isomorphisms [18], or even trivial forms of

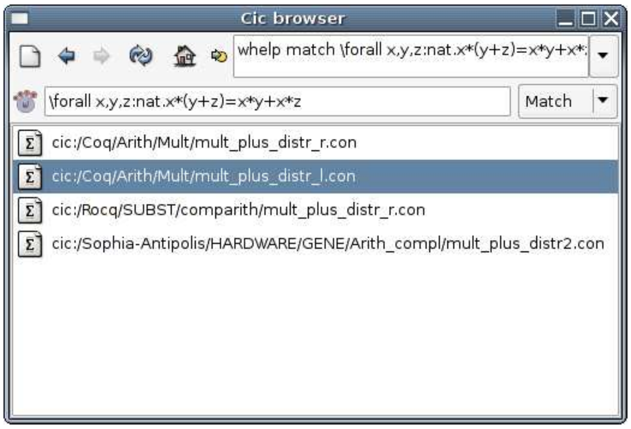

Figure 4. Pattern matching query.

rewriting (checking e.g. if the new statement can be demodulated to a know one, that is a very efficient operation).

Another useful Whelp operation is the Hint query; the user may resort to it at any moment during proof authoring, resulting in the list of all theorems of the library which can be applied to the current goal. In practice, this is mostly used not really to discover what theorems can be applied to a given goal, but to actually retrieve a theorem that the user wishes to apply, but whose name was forgotten. In fact, even if Matita adopts a semi-rigid naming convention for statements (see Section 2.4) that greatly simplifies the effort of recalling names; retrieving facts from the library remains one of the most annoying aspects of formal developments, and Hint provides a very friendly solution.

In the near future, we expect to extend the Hint query to a RewriteHint, resulting in all equational statements that can be applied to rewrite the current goal.

2.1.1. *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, like 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.

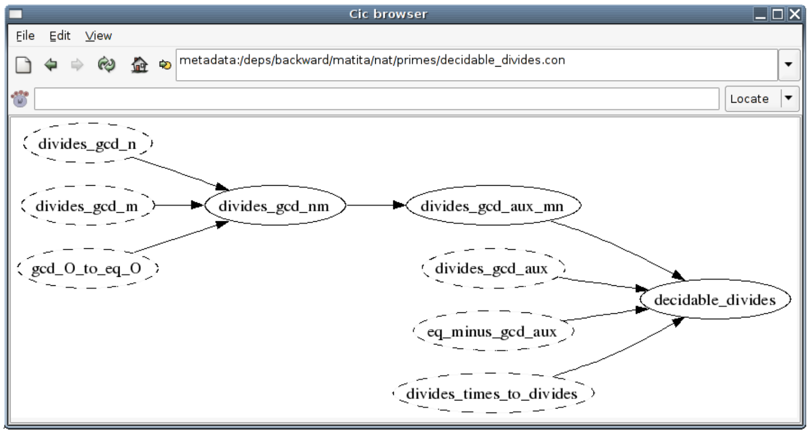

Figure 5. Dependencies analyzer.

The Mizar Mathematical Library (MML)[2] can be searched with a search engine which indexes the whole library [10]. When a declarative script is executed by Mizar, metadata are collected in special files used by the search engine. This solution is in principle more inefficient when compared with ours that stores metadata in a relational database; however so far performances are good. As for the previous systems, no distinction can be done in Mizar's queries on the positions where constants occur. In spite of that, queries can be quite precise because of the great flexibility provided in composing pipelines of queries to refine searches.

Automation in Matita is heavily based on the search engine. On the contrary, in Mizar the search engine is an external tool and automation can not 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 will focus on how Matita ensures the library consistency during the formalization of a mathematical theory, giving the user the freedom of adding, removing, modifying mathematical concepts without loosing the feeling of an always visible and browsable library.

[2] `http://mizar.org/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.

2.2.1. *Invalidation*

Invalidation is implemented in two phases.

The first one is the computation of all the concepts that recursively depend on those we are invalidating. It can be performed using the metadata stored in the relational database. This technique is the same used by the *Dependencies analyzer*; it is described in [40] and shown in Figure 5 (the reverse dependencies of dashed nodes are hidden to improve readability; the system is able to compute and show hidden nodes/edges on demand).

The second phase is the removal of all the results of the generation, metadata included.

2.2.2. *Regeneration*

To regenerate an invalidated part of the library Matita re-executes the scripts that produced the invalidated concepts. The main problem is to find a suitable order of execution of the scripts.

For this purpose we provide a tool called `matitadep` that takes in input the list of scripts that compose a development and outputs their dependencies in a format suitable for the GNU `make` tool.[3] The user is not required to run `matitadep` by hand, but simply to inform Matita about the root directory of his development (where all script files can be found) and Matita will handle all the tasks related to generation, including dependencies calculation.

The re-execution of a script to regenerate part of the library requires the preliminary invalidation of the concepts generated by the script and those depending on them.

2.2.3. *Batch vs interactive*

Matita includes an interactive authoring interface and a batch "compiler" (`matitac`). Only the former is intended to be used directly by the user, the latter is automatically invoked by Matita to regenerate invalidated parts of the library.

While they share the same engine for regeneration and invalidation, they provide different granularity. `matitac` is only able to re-execute a whole script and similarly to invalidate all the concepts generated by a script (together with all the other scripts that rely on a concept

[3] `http://www.gnu.org/software/make/`

defined in it). On the contrary, using the authoring interface the user can invalidate and regenerate parts of the library on a concept-per-concept basis.

2.2.4. *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, 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 [20] 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 also provided. This allows to decouple parts of large developments. 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. Modularizations in different forms can be found in other systems too. For instance Coq has modules and functors while Isabelle has locales.

In Matita we do not currently have any modularization mechanism, even if dependent records [16] partially provide the functionality. When modularization is considered, the dependencies used for invalidation and regeneration inherits the hierarchical structure of developments (as happens in [20]). Dependencies in Matita are simply flat.

2.3. Automation

In the long run, one would expect to work with a proof assistant like 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 to define small collections of theorems to be used as parameters for the `auto` tactic; in Isabelle the same happens with lemmas used by the simplifier. In Matita, the system itself automatically retrieves from the whole library a subset of theorems worth to be considered, 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 only requires a proof of *reflexivity* and *paramodulation* (or rewriting, as it is usually called 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* [25], that is the procedure currently used by the majority of modern automatic theorem provers.

The given-clause algorithm is essentially composed by an alternation of a *saturation* phase and a *demodulation* phase. The former derives new facts by a set of active facts and a new *given* clause suitably selected from a set of passive equations. The latter tries to simplify the equations orienting them according to a suitable weight associated to terms. Matita currently supports several different weighting functions comprising Knuth-Bendix ordering and recursive path ordering, that integrates particularly well with normalization.

We have recently run the `paramodulation` tactic of Matita on the unit-equality set of problems of the 2006 CASC competition [34], obtaining a score comparable (actually, slightly better) with Otter [24].

Disposing of a large library of results, demodulation alone is already a quite powerful technique, and it has been turned into a tactic by itself: the `demodulate` tactic, which can be seen as a kind of generalization of `simplify`. The following script snippets describe two interesting cases of application of this tactic (both of them relying on elementary arithmetic equations).

```
theorem example1:
  \forall x,y: nat. (x+y)*(x+y) = x*x + 2*x*y + y*y.
intros; demodulate; reflexivity.
qed.
```

```
theorem example2:
  \forall x: nat. (x+1)*(x-1) = x*x - 1.
intro;
apply (nat_case x);
  [ simplify; reflexivity
  | intro; demodulate; reflexivity ]
qed.
```

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, that is *per se* an interesting open problem (see [23] for some preliminary discussions and [14, 13] for more advanced material on the larger topic of mathematical theory exploration). This "completion" procedure for the library requires no user interaction and could be performed in batch mode, for example during the night. The user could be asked for a final validation before storing the new, selected results in the library.

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 only applied to theorems (not definitions), and relates theorem names to their statements.

The basic rules are the following:

- each name is composed by an ordered list of (short) identifiers occurring in a left to right traversal of the statement;
- all identifiers should be (but this is not strictly compulsory) separated by an underscore;
- identifiers occurring in two different hypotheses, or in an hypothesis and in the conclusion must be separated by the string `_to_`;
- the name may be followed by a *decorator*, being a numerical suffix, or a single or double apostrophe.

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, consider the statement:

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

In this case `lt_to_le` is a legal name, while `lt_le` is not.

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

Two cases have a special treatment. The first one concerns theorems whose conclusion is a (universally quantified) predicate variable, i.e. theorems of the shape $\forall P, \ldots, .P(t)$. In this case you may replace the conclusion with the string `elim` (for "elimination", a common name for inductive reasoning in interactive theorem proving) or `case` (for "case analysis"). For instance the name `nat_elim2` is a legal name for the double induction principle.

The other special case is that of statements whose conclusion is a `match` statement. A typical example is the following:

```
\forall n,m: nat.
  match (eqb n m) with
  [ true  \Rightarrow n = m
  | false \Rightarrow n \neq m ]
```

where `eqb` is boolean equality. In this cases, the name can be build starting from the matched expression and the suffix `_to_Prop`. In the above example, `eqb_to_Prop` is accepted.

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 students, quite easy to learn for newcomers (at the very minimum as easy as learning how to use Proof General or CoqIde). In spite of that, we deliberately choose not to develop Matita targeting Proof General as our user interface of choice. The first reason for not doing that, mentioned in Section 1, 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.[4] The second reason is that we wanted to build the Matita user interface on top of 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 some innovative peculiarities not found in competing systems. In the remaining part of this section we will discuss them in turn, starting from those related to the sequents window (Section 3.1) and then moving to those related to the script editing window (Sections 3.2, 3.3, and 3.4).

3.1. Direct manipulation of terms

The sequents window and the CIC browser are used in Matita to present formulae and concepts to the user. Both windows are based on GtkMathView 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 formulae using the vocabulary of mathematical notation, which comprises atomic entities like identifiers, numbers, and operators together with a set of layouts like sub/superscripts, fractions, and radicals. The latter language is used to describe the placement of formulae with respect to each other and where to break formulae in case the actual window is too small to fit them on a single physical line.

[4] This may change with future releases of Proof General based on the Eclipse platform (`http://www.eclipse.org/`).

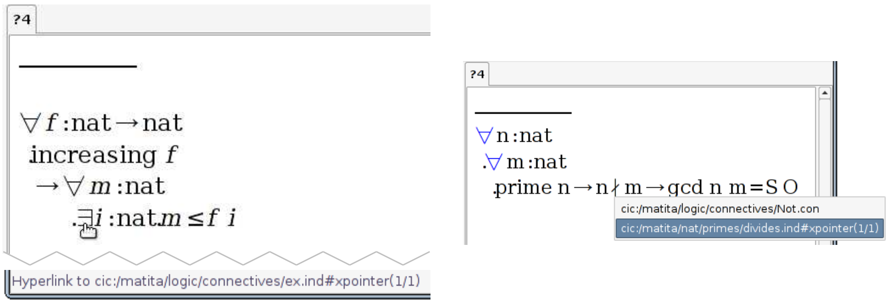

Figure 6. Hypertextual browsing.

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 [33] using the mouse. *Hypertextual browsing* is shown in Figure 6.

Markup elements which 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 using their URIs. On the left of Figure 6 the mouse is over an $\exists$ 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 start a CIC browser instance (possibly reusing the current one) to show the target concept.

Since user defined notations are often used to hide details of complex CIC terms, it can happen that markup elements reference more than one concepts from the library; in a sense our hyperlinks are one-to-many. For instance, on the right of Figure 6 the symbol $\nmid$ is a user defined notation which 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 rendered in the CIC browser on in the sequents window. Figure 7 shows the contextual menu which will pop-up (clicking with the right button) when part of the markup is visually selected.

Menu items of the pop-up menu permits 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 & paste*. The latter is called "semantic" to distinguish

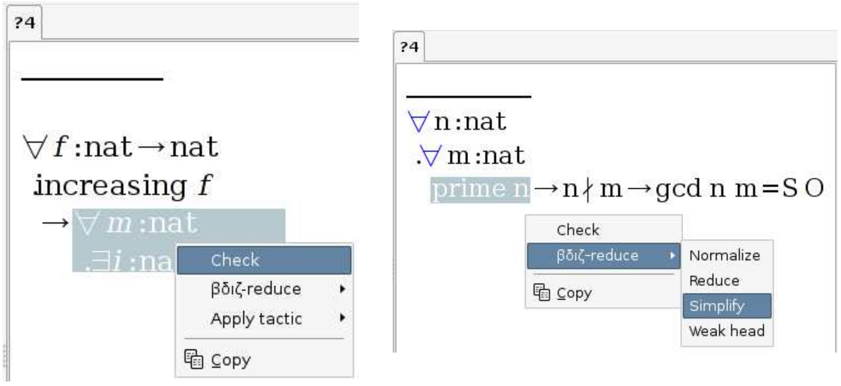

Figure 7. Contextual actions and semantic selection.

it from ordinary textual copy & 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 to perform semantic transformations on the copied term like 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 does support multiple selections). The typical use case of multiple selection is simplification in multiple sub-terms at once.

A requirement for semantic contextual actions is that the markup visually selected in a window has a corresponding CIC term. This requirement is non trivial to achieve since 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 Figure 7 for instance, the formula "$\forall m$: nat" is a well formed markup snippet: an 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.

In Matita this issues is solved by the mean of *semantic selection*, which constrains the user to visually select only markup snippets which have a corresponding CIC term. A user trying to select "$\forall m$: nat" will obtain a selection of the whole binder rooted at it, shown shaded on the left of Figure 7.

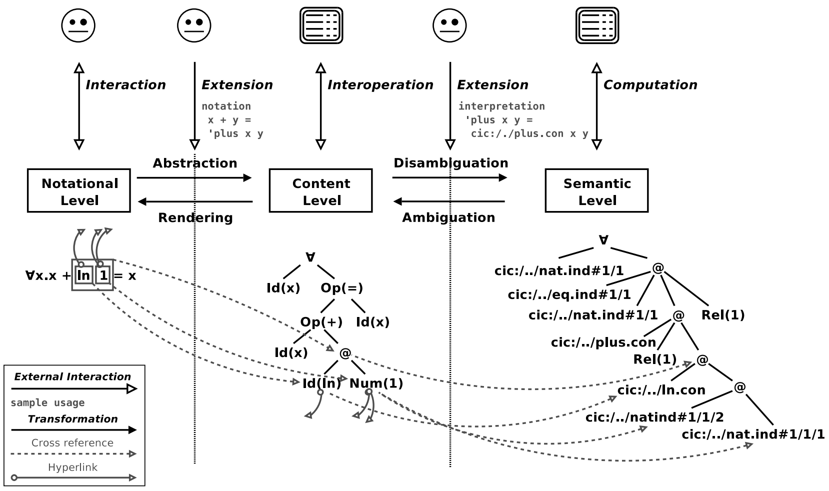

Figure 8. Overview of the notational framework: term encodings and their usage.

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 [28] is in charge of adding them during the transformations terms go through for rendering purposes. Figure 8 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 sub-term is annotated with a unique identifier; the set of those 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 it is now possible to go "back" to a well-formed CIC term starting from an element of the presentation markup who 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.

Additional details of the notational framework of Matita has been given in [28] and will not be discussed here.

3.2. Disambiguation

One of the most frequent activities in the interaction with any tool for mathematics is input of formulae. Since Matita uses textual typing as its input mechanism, formulae 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 [22]: 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 only discuss them.

In programming languages these challenges are usually solved by limiting the language that, being imposed to the user, can be freely modelled by its designers. The restrictions imposed are partially guided by performance reasons and partially 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 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, that we call *disambiguation*, can be roughly described as: 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. Kamareddine and Nederpelt's Weak Type Theory [21] or the type system used in Mizar [9]). In the case of Matita, CIC terms are used as internal representations of formulae, 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 [32] 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 handle also coercions does not pose any additional problem, and it has been implemented in Matita. The algorithm will not be discussed any further in this paper.

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

3.2.1. *Representations rating*

The problem we address is finding some reasonable criteria that allow 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 alternatives to the user in order of likelihood of being the wanted representation. Since the guessed alternative is chosen among the maximally rated ones, rating several alternatives 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 criteria 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 to 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 sub-formulae 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 Figure 8 for an example of how notation can be associated to the content level node [28]).

The topmost interpretation on the stack is intuitively the *current* (hence most likely) one. For instance, the user that intends to prove some lemmas over rational numbers should set preferences for interpreting the "less than" relation over natural, integer and rational numbers, the latter 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 allow exceptions such as stating a lemma that requires testing inequality over real numbers; however, the exceptional interpretation for "less than" is not took in account as a new user preference (as it happens with the locality of reference criterion).

The criterion based on user preferences allows 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

Table I. User preferences and other interpretations in effect for the disambiguation examples. The underlined preference is the current one.

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	$\underline{\mathbb{R}}, \mathbb{Z}, \mathbb{N}$	$\mathbb{C}$

of the examples collected so far, we achieved results deemed satisfactory by our (small) user community rating representations as follow:

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

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 3.2.2).

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

EXAMPLE 1. (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 2. (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 3. (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 $\forall n, m, r : \mathbb{N}.n|_{\mathbb{N}} m \rightarrow n *_{\mathbb{N}} r|_{\mathbb{N}} m *_{\mathbb{N}} r$ *and* $\forall n, m, r : \mathbb{Z}.n|_{\mathbb{Z}} m \rightarrow n *_{\mathbb{Z}} r|_{\mathbb{Z}} m *_{\mathbb{Z}} r$ *that 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 since it respects more preferences.*

EXAMPLE 4. (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 worst score since it requires a coercion. Thus the best representation is* $\forall n, m : \mathbb{Z}.n|_{\mathbb{Z}} m \rightarrow |n|_{\mathbb{Z}} \ |_{\mathbb{Z}} \ |m|_{\mathbb{Z}}$ *that is rated* $\langle 0, 1\rangle$*.*

EXAMPLE 5. (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* $\forall n, m : \mathbb{N}.n|_{\mathbb{N}} m \to |n|_{\mathbb{Z}} \;|_{\mathbb{Z}}\; |m|_{\mathbb{Z}}$, *that is rated* $\langle 1, 0\rangle$.

EXAMPLE 6. (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 $\forall n, m : \mathbb{R}.n <_{\mathbb{R}} m +_{\mathbb{R}} 1_{\mathbb{R}} \to \lfloor n\rfloor <_{\mathbb{Z}} \lfloor m\rfloor$ *that is rated* $\langle 0, 1\rangle$.

EXAMPLE 7. (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 8. (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.*

3.2.2. *Implementation*

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 performances need to be faced. Indeed, a priori the number of potential representations is exponential in the number of ambiguity sources. The algorithm we presented in [32] for the first task is linear in the number of well typed representations that, in non artificial 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 non empty set of well-typed representations is found. The technique is to interleave the two tasks, tuning the parameters of the disambiguation algorithm of [32] 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 II we show how the parameters are set in the five disambiguation attempts tried in order. Since subsequent attempts reconsider representations generated in previous attempts, memoization can be used to speed up the process.

Table II. 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	$+\infty$

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 using the label given when an interpretation was declared.

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:

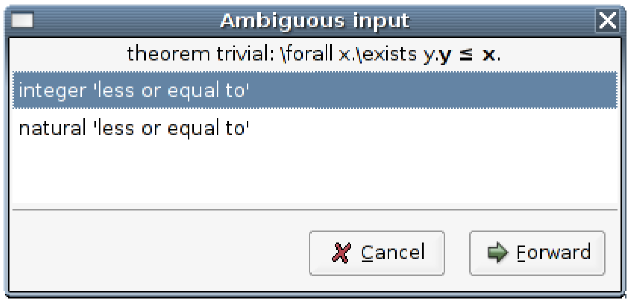

Figure 9. Ambiguity resolution.

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 also allows future batch compilation of the script. At the end of the interaction a minimal set of preferences, that 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 Figure 9, 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 greatly improves over the one that shows all possible representations at once since the user does not have to reason globally on representations, but locally on single sources of ambiguity.

3.2.3. *Related issues*

Since the internal representation of formulae 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 non-intrusive ways, and that of pretty-printing the formulae 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 sub-formulae is a possible future work which right now we do not feel urged to do.

Pretty-printing of formulae 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 sub-formula the error message refers to is underlined in red in the script. This greatly reduces the need to show formulae 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)` that 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 set of representations that can not be abstracted in a single term of the logic by means of metavariables. Doing this 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 since 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 should be further investigated in the future.

3.2.4. *Comparison with other systems*

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

Overloading in Coq, Matita, and in the majority of systems is done at the syntactic level and it must be resolved during the parsing (or disambiguation) phase. On the contrary, overloading in Isabelle [38] 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 [26], 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 an 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. When this is not done 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 sub-formula, forcing a different interpretation.

Coq's mechanism allows to perform disambiguation in linear time, but it suffers from several important limitations. When a function is polymorphic, e.g. it has type `\forall A:Type. A \to` ..., Coq is unable to associate an interpretation scope to the second argument of the function since 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`, than `forall (n:nat), n * 1 = n` will be rejected since 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, differentiate the interpretations.

Subtyping is implemented in Coq and Matita by means of coercive subtyping. The two systems allows to declare general coercions, that are useful not only to simulate subtyping. To prove generic theorems over algebraic hierarchies dependent records in the spirit of [16] 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 [17] 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, that behind the scenes is implemented 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).

Table III. Concrete syntax of patterns.

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

3.3. 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 actions and selections in scripts.

In Matita *patterns* are textual representations of selections. Users can select using the GUI and then ask the system to paste the corresponding pattern in the script. This process is transparent to the user: once an action is performed on a selection, the corresponding textual command is computed and inserted in the script.

3.3.1. *Pattern syntax*

Patterns are composed of two parts: ⟨*sequent_path*⟩ and ⟨*wanted*⟩; their concrete syntax is reported in Table III.

⟨*sequent_path*⟩ mocks-up a sequent, discharging unwanted sub-terms with "`?`" and selecting the interesting parts with the placeholder "`%`". ⟨*wanted*⟩ is a term that lives 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 are quite verbose. The ⟨*wanted*⟩ part of the syntax is meant to help the users in writing concise and elegant patterns by hand.

3.3.2. *Pattern evaluation*

Patterns are evaluated in two phases. The first selects roots (sub-terms) of the sequent, using the ⟨*sequent_path*⟩, while the second searches the ⟨*wanted*⟩ term starting from that roots.

Phase 1 concerns only the ⟨*sequent_path*⟩ part. ⟨*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 patterns concrete syntax, since the system is able to write down a ⟨*sequent_path*⟩ starting from a graphical selection.

A ⟨*multipath*⟩ is a CIC term in which two special constants "`%`" and "`?`" are allowed. The roots of discharged and selected sub-terms are marked respectively with "`?`" and "`%`". The default ⟨*multipath*⟩, the one that selects the whole term, is simply "`%`". Valid ⟨*multipath*⟩ 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.

This phase not only selects terms (roots of sub-terms) but determines also their context that will be possibly used in the next phase.

Phase 2 plays a role only if the ⟨*wanted*⟩ part is specified. From the first phase we have some terms, that 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 sub-term that can be unified to ⟨*wanted*⟩. The result of this search is the selection the pattern represents.

3.3.3. *Patterns in action*
Consider the following sequent:

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

EXAMPLE 1.
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 pattern is not a urgent necessity since patterns are not meant to be written by hand: the user can select with the mouse a sub-term 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, and re-executing the script the users sees exactly where the tactic is applied, without even trying to understand the pattern.

EXAMPLE 2.

The experienced user is not forbid 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, that the system would have automatically generated from the selection.*

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

3.3.4. *Comparison with other systems*

Patterns were studied and implemented to represent in text format the user visual selection. The majority of the systems have no support for acting on visually selected sub-formulae 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 sub-formulae 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 sub-formulae, probably because the deep integration with automated tactics makes this need less impelling.

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 sub-formulae occurring in a sequent.

While in Matita all the tactics that act on sub-formulae of the current sequent accept pattern arguments, in Coq the user has two different ways of restricting the application of tactics to sub-formulae, 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 sub-formulae 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 formulae.

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:

$n : nat$
$m : nat$
$H : (fun\ n0 : nat => m + n = n0)\ n$

$m = 0$

where `H` has been β-expanded over the second occurrence of `n`. At this point, since Coq unification algorithm is essentially first-order, the application of an elimination principle (a term of type $\forall P.\forall x.(H\ x) \rightarrow (P\ x)$) 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 sub-formulae is similar to the patterns idea, but fails in extending to complex notation, since it relies on a mono-dimensional 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 our method that does not discourage the use of complex layouts.

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 is given on demand in the CIC browser where a pseudo-natural language representation of the incomplete proof term can be inspected to verify if the proof corresponds so far 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 [19] 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, putting emphasis on the user interface aspects to reveal several problems. In Matita we solve these problems replacing LCF tacticals with *tinycals* that 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 apply 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 Figure 10 is structured with LCF tacticals, expressed using the concrete syntax of Matita tinycals.

LCF tacticals account for two improvements over plain tactic sequences. The first improvement its their ability in supporting *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 scripts readability, which is on the average very poor, if compared with declarative proof languages. Maintainability of

```
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.
```

Figure 10. Sample Matita script with tacticals.

proof scripts is also improved by the use of branching, for example when hypothesis 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 could 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 part 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 tactic pipelines) is not allowed. Thus a fully structured script as the one shown in Figure 10 can only be executed 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, 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 replacing tactics with the identity tactic (a tactic which simply returns unchanged the goal it was applied to) proceeding outside-in, until the single failing tactic is found. This technique is not only error prone, but is even not reliable in presence of *side-effects* (tactics closing sequents other than that on which they are applied), since the identity tactic has no side-effects and branches of the proof 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* [31]. Tinycals can be evaluated in small steps, enabling the execution point to be placed inside complex structures like pipelines or branching constructs. More precisely, the syntax of tacticals is destructured and the semantics is modified accordingly, enabling parsing and immediate execution of fine grained constituents of LCF tacticals. In Figure 1 for instance the occurrence of "[" just before the execution point has been parsed and executed without the need of executing the whole "[...]" block.

The semantics of tinycals is stated as a transition system over *evaluation status*, that are structures richer than the proof status tactics act on. The key component of an evaluation status is the context stack,

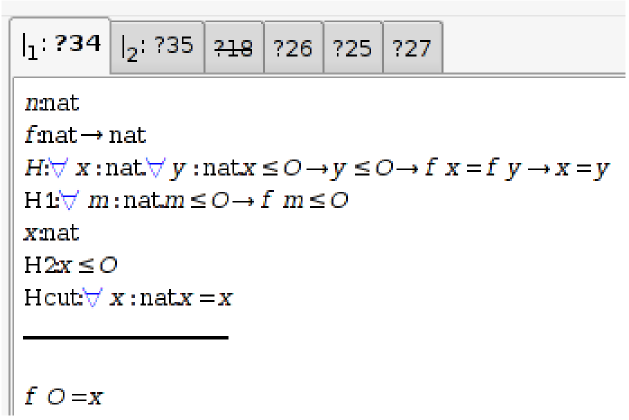

Figure 11. Zoomed sequents window: visual representation of the context stack (upper part of the figure).

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 which behave like "[" are executed and get popped out of it when tinycals which behave like "]" are.

This way users can incrementally author structured blocks and both proof structuring and conciseness are encouraged. Tinycals also provide additional minor improvements not found in other tactical languages: non structured 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 which can be found in Figure 10); special support for focusing on a group of related goals is provided.

The syntax and formal semantics of tinycals has been given in [31] and will not be discussed here.

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 Figure 11, 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. Being the stack 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].

Our representation choice can be seen in the upper part of Figure 11. 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 to have their labels typeset in boldface (only **?34** in the example), goals of the current branching context have labels prepended by $|_n$ (where n is their positional index, $|_1$: ?**34** and $|_2$: ?35 in the example), goals already closed by side-effects have strike-through labels (~~?18~~ in the example). All remaining goals are simply shown 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 alternative approaches for authoring structured HOL scripts have been proposed in [35] and [36]. The first approach, implemented in Syme's TkHOL, is similar to tinycals but was lacking a formal description that we provide in [31]. Moreover, unlike HOL, we consider a logic with metavariables which 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 because they are not allowed in the statements of lemmata.

4. Standard library

Matita is Coq-compatible, in the sense that every concept of Coq can be read (via XML exportation), checked and referenced in further developments. However, in order to test the actual usability of the system, a new library of results has been started from scratch. In this case we wrote and offer the source scripts, while in the case of concept coming from Coq, Matita may only rely on the exported proof objects.

The current library just comprises about one thousand theorems in elementary aspects of arithmetics up to the multiplicative property for Euler's totient function ϕ, Fermat's little theorem and the Fundamental Theorem of Arithmetics.

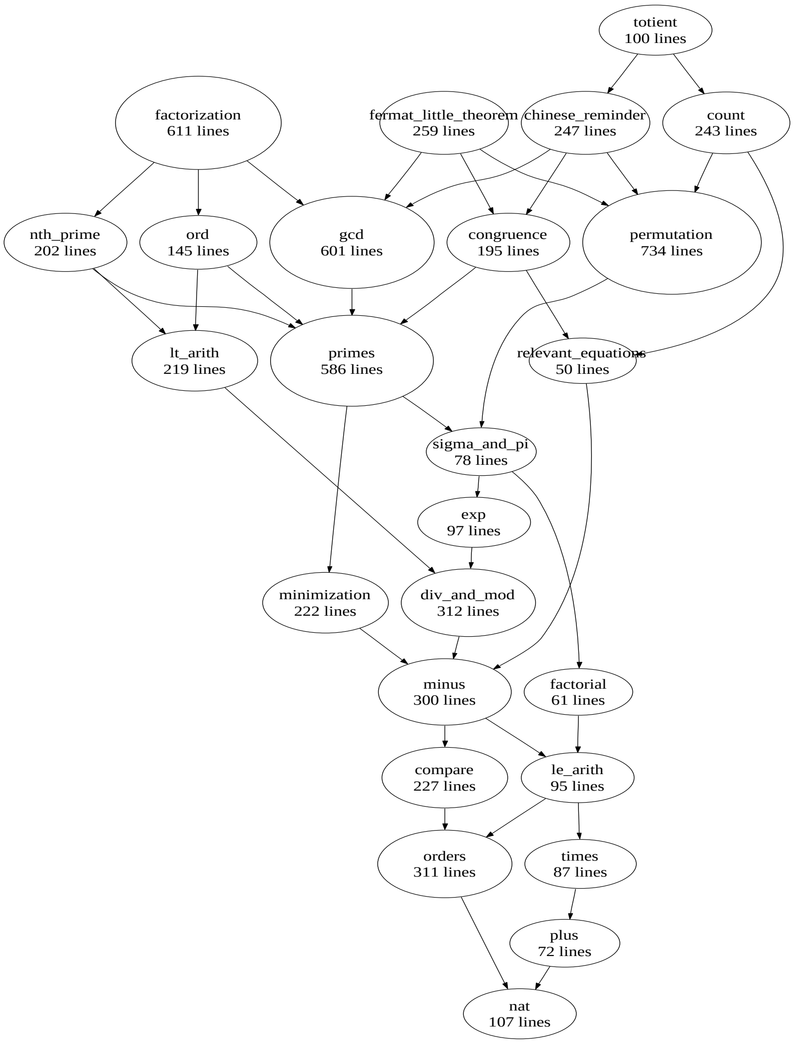

Figure 12. Standard library: scripts in the `nat` development and their dependencies. Larger shapes correspond to larger scripts.

The library is organized in five main directories: `logic` (connectives, quantifiers, equality, ...), `datatypes` (basic data types and type constructors), `nat` (natural numbers), `Z` (integers), `Q` (rationals). The most complex development is `nat`, organized in 25 scripts, shown with their dependencies in Figure 12.

5. Conclusion

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 will discuss how the origins of Matita influenced its design and how we intend to proceed in the system development.

5.1. Historical perspective

The origins of Matita go back to 1999. At the time we were mostly interested in developing tools and techniques to enhance the accessibility via Web of libraries of formalized mathematics. Due to its dimension, the library of the Coq [37] 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 [15]. The work, mostly performed in the framework of the recently concluded European project MoWGLI [7], mainly consisted 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.[5]

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.

[5] We have been active in the MathML Working group since 1999.

At the end of the MoWGLI project we already disposed of the following tools and software components:

- XML specifications for the Calculus of Inductive Constructions, with components for parsing and saving mathematical concepts in such a format [29];
- 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 [32];
- a *refiner* component, i.e. a type inference system, based on partially specified terms, used by the disambiguating parser [30];
- complex transformation algorithms for proof rendering in natural language [6];
- an innovative, MathML-compliant rendering widget [27] for the GTK+ graphical environment, supporting high-quality bidimensional rendering, and semantic selection.

Starting from all this, developing our own proof assistant was not too far away: essentially, 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.

5.2. 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, syntax highlighting, and so on. These libraries are often written in low level languages (C or C++) and they 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 performances, 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 that so far has been entirely based on Emacs.

We think that an interesting goal for the research community is splitting 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 that of providing in the future not only generic user interfaces, but also generic components for more advanced tasks. The logic dependent code could be self contained for an 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 try to clarify the proposed splitting as components layering.

5.3. 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 formulae 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), 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); disambiguation of content level formulae (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.

5.4. 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 would like to thank all the people that during the past 7 years collaborated in the Helm project and contributed to the development of Matita, and in particular M. Galatà, A. Griggio, F. Guidi, P. Di Lena, L. Padovani, I. Schena, M. Selmi, and V. Tamburrelli.

References

1. Aitken, S.: 1996, 'Problem Solving in Interactive Proof: A Knowledge-Modelling Approach'. In: *European Conference on Artificial Intelligence (ECAI)*. pp. 335–339.
2. Aitken, S., P. Gray, T. Melham, and M. Thomas: 1998, 'Interactive Theorem Proving: An Empirical Study of User Activity'. *Journal of Symbolic Computation* **25**(2), 263–284.
3. Asperti, A., F. Guidi, L. Padovani, C. Sacerdoti Coen, and I. Schena: 2003, 'Mathematical Knowledge Management in HELM'. *Annals of Mathematics and Artificial Intelligence* **38(1-3)**, 27–46.
4. Asperti, A., F. Guidi, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli: 2004, 'A content based mathematical search engine: Whelp'. In: *Post-proceedings of the*

Types 2004 International Conference, Vol. 3839 of *Lecture Notes in Computer Science*. pp. 17–32.

5. Asperti, A., L. Padovani, C. Sacerdoti Coen, and irene Schena: 2000, 'Content-centric Logical Envirnoments'. Short Presentation at the Fifteenth IEEE Symposium on Logic in Computer Science.
6. Asperti, A., L. Padovani, C. Sacerdoti Coen, and I. Schena: 2001, 'XML, Stylesheets and the re-mathematization of Formal Content'. In: *Electronic Proceedings of EXTREME Markup Languages 2001*.
7. Asperti, A. and B. Wegner: 2003, 'An Approach to Machine-Understandable Representation of the Mathematical Information in Digital Documents'. In: F. Bai and B. Wegner (eds.): *Electronic Information and Communication in Mathematics*, Vol. 2730 of *Lecture Notes in Computer Science*. pp. 14–23.
8. Aspinall, D.: 2000, 'Proof General: A Generic Tool for Proof Development'. In: *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000*, Vol. 1785 of *Lecture Notes in Computer Science*.
9. Bancerek, G.: 2003, 'On the structure of Mizar types.'. *Electronic Notes in Theoretical Computer Science* **85**(7).
10. Bancerek, G. and P. Rudnicki: 2003, 'Information Retrieval in MML'. In: A. Asperti, B. Buchberger, and J. H. Davenport (eds.): *Proceedings of the Second International Conference on Mathematical Knowledge Management, MKM 2003*, Vol. 2594 of *Lecture Notes in Computer Science*.
11. Bertot, Y.: 1999, 'The CtCoq System: Design and Architecture'. *Formal Aspects of Computing* **11**, 225–243.
12. Bertot, Y., G. Kahn, and L. Théry: 1994, 'Proof by Pointing'. In: *Symposium on Theoretical Aspects Computer Software (STACS)*, Vol. 789 of *Lecture Notes in Computer Science*.
13. Buchberger, B., A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger: 2005, 'Theorema: Towards Computer-Aided Mathematical Theory Exploration.'. *Journal of Applied Logic*.
14. Colton, S.: 2002, *Automated Theory Formation in Pure Mathematics*. Springer. ISBN-1-85233-609-9.
15. Constable, R. L., S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith: 1986, *Implementing Mathematics with the Nuprl Development System*. NJ: Prentice-Hall.
16. Coquand, T., R. Pollack, and M. Takeyama: 2005, 'A Logical Framework with Dependently Typed Records.'. *Fundamenta Informaticae* **65**(1-2), 113–134.
17. Cruz-Filipe, L., H. Geuvers, and F. Wiedijk: 2004, 'C-CoRN, the Constructive Coq Repository at Nijmegen.'. In: *MKM*. pp. 88–103.
18. Delahaye, D. and R. di Cosmo: 1999, 'Information Retrieval in a Coq Proof Library using Type Isomorphisms'. In: *Proceedings of TYPES 99*, Vol. Lecture Notes in Computer Science.
19. Gordon, M. J. C., R. Milner, and C. P. Wadsworth: 1979, 'Edinburgh LCF: a Mechanised Logic of Computation'. In: *Lecture Notes in Computer Science*, Vol. 78 of *Lecture Notes in Computer Science*.
20. Hutter, D.: 2004, 'Towards a generic management of change'. In: *Workshop on Computer-Supported Mathematical Theory Development, IJCAR 2004*.
21. Kamareddine, F. and R. Nederpelt: 2004, 'A Refinement of de Bruijn's Formal Language of Mathematics.'. *Journal of Logic, Language and Information* **13**(3), 287–340.

22. Luo, Z.: 1999, 'Coercive Subtyping.'. *Journal of Logic and Computation* **9**(1), 105–130.
23. McCasland, R. L., A. Bundy, and P. F. Smith: 2006, 'Ascertaining Mathematical Theorems.'. *Electr. Notes Theor. Comput. Sci.* **151**(1), 21–38.
24. McCune, W. and L. Wos: 1997, 'Otter - The CADE-13 Competition Incarnations.'. *Journal of Automated Reasoning* **18**(2), 211–220.
25. Nieuwenhuis, R. and A. Rubio: 2001, *Paramodulation-based thorem proving*. Elsevier and MIT Press. ISBN-0-262-18223-8.
26. Obua, S.: 2006, 'Conservative Overloading in Higher-Order Logic'. In: *Rewriting Techniques and Applications*. To appear.
27. Padovani, L.: 2003, 'MathML Formatting'. Ph.D. thesis, University of Bologna. Technical Report UBLCS 2003-03.
28. Padovani, L. and S. Zacchiroli: 2006, 'From Notation to Semantics: There and Back Again'. In: *Proceedings of Mathematical Knowledge Management 2006*, Vol. 3119 of *Lectures Notes in Artificial Intelligence*. pp. 194–207.
29. Sacerdoti Coen, C.: 2003, 'From Proof-Assistans to Distributed Libraries of Mathematics: Tips and Pitfalls'. In: A. Asperti, B. Buchberger, and J. H. Davenport (eds.): *Proceedings of the Second International Conference on Mathematical Knowledge Management, MKM 2003*, Vol. 2594 of *Lecture Notes in Computer Science*. pp. 30–44.
30. Sacerdoti Coen, C.: 2004, 'Mathematical Knowledge Management and Interactive Theorem Proving'. Ph.D. thesis, University of Bologna. Technical Report UBLCS 2004-5.
31. Sacerdoti Coen, C., E. Tassi, and S. Zacchiroli: 2006, 'Tinycals: step by step tacticals'. In: *Proceedings of User Interface for Theorem Provers 2006*. To appear.
32. Sacerdoti Coen, C. and S. Zacchiroli: 2004, 'Efficient Ambiguous Parsing of Mathematical Formulae'. In: A. T. Andrea Asperti, Grzegorz Bancerek (ed.): *Proceedings of Mathematical Knowledge Management 2004*, Vol. 3119 of *Lecture Notes in Computer Science*. pp. 347–362.
33. Shneiderman, B.: 1997, 'Direct manipulation for comprehensible, predictable and controllable user interfaces'. In: *IUI '97: Proceedings of the 2nd international conference on Intelligent user interfaces*. New York, NY, USA, pp. 33–39.
34. Sutcliffe, G.: 2006, 'The CADE-20 Automated Theorem Proving Competition'. *AI Communications* **19**(2), 173–181.
35. Syme, D.: 1995, '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*, Vol. 971 of *Lecture Notes in Computer Science*. pp. 324–339.
36. Takahashi, K. and M. Hagiya: 1999, 'Proving as Editing HOL Tactics'. *Formal Aspects of Computing* **11**(3), 343–357.
37. The Coq Development Team: 2005, 'The Coq Proof Assistant Reference Manual'. `http://coq.inria.fr/doc/main.html`.
38. Wenzel, M.: 1997, 'Type Classes and Overloading in Higher-Order Logic.'. In: *TPHOLs*. pp. 307–322.
39. Werner, B.: 1994, 'Une Théorie des Constructions Inductives'. Ph.D. thesis, Université Paris VII.
40. Zacchiroli, S.: 2003, 'Web services per il supporto alla dimostrazione interattiva'. Master's thesis, University of Bologna.