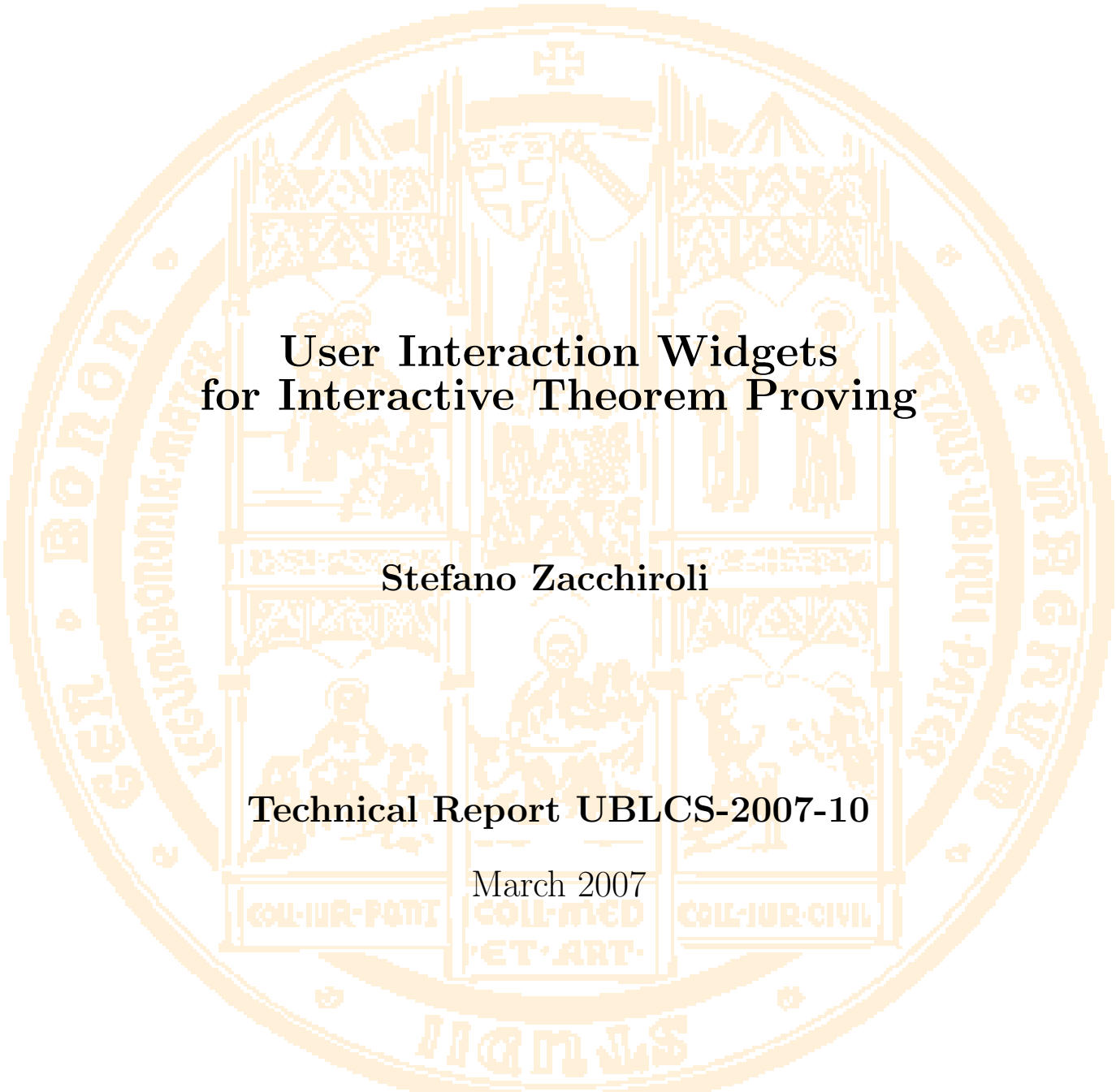

User Interaction Widgets for Interactive Theorem Proving

Stefano Zacchiroli

Technical Report UBLCS-2007-10

March 2007

Department of Computer Science

University of Bologna

Mura Anteo Zamboni 7
40127 Bologna (Italy)

The University of Bologna Department of Computer Science Research Technical Reports are available in PDF and gzipped PostScript formats via anonymous FTP from the area `ftp.cs.unibo.it:/pub/TR/UBLCS` or via WWW at URL `http://www.cs.unibo.it/`. Plain-text abstracts organized by year are available in the directory `ABSTRACTS`.

Recent Titles from the UBLCS Technical Report Series

2006-22 *Broadcasting at the Critical Threshold*, Arteconi, S., Hales, D., October 2006.

2006-23 *Emergent Social Rationality in a Peer-to-Peer System*, Marcozzi, A., Hales, D., October 2006.

2006-24 *Reconstruction of the Protein Structures from Contact Maps*, Margara, L., Vassura, M., di Lena, P., Medri, F., Fariselli, P., Casadio, R., October 2006.

2006-25 *Lambda Types on the Lambda Calculus with Abbreviations*, Guidi, F., November 2006.

2006-26 *FirmNet: The Scope of Firms and the Allocation of Task in a Knowledge-Based Economy*, Mollona, E., Marcozzi, A. November 2006.

2006-27 *Behavioral Coalition Structure Generation*, Rossi, G., November 2006.

2006-28 *On the Solution of Cooperative Games*, Rossi, G., December 2006.

2006-29 *Motifs in Evolving Cooperative Networks Look Like Protein Structure Networks*, Hales, D., Arteconi, S., December 2006.

2007-01 *Extending the Choquet Integral*, Rossi, G., January 2007.

2007-02 *Towards Cooperative, Self-Organised Replica Management*, Hales, D., Marcozzi, A., Cortese, G., February 2007.

2007-03 *A Model and an Algebra for Semi-Structured and Full-Text Queries (PhD Thesis)*, Buratti, G., March 2007.

2007-04 *Data and Behavioral Contracts for Web Services (PhD Thesis)*, Carpineti, S., March 2007.

2007-05 *Pattern-Based Segmentation of Digital Documents: Model and Implementation (PhD Thesis)*, Di Iorio, A., March 2007.

2007-06 *A Communication Infrastructure to Support Knowledge Level Agents on the Web (PhD Thesis)*, Guidi, D., March 2007.

2007-07 *Formalizing Languages for Service Oriented Computing (PhD Thesis)*, Guidi, C., March 2007.

2007-08 *Secure Gossiping Techniques and Components (PhD Thesis)*, Jesi, G., March 2007.

2007-09 *Rich Media Content Adaptation in E-Learning Systems (PhD Thesis)*, Mirri, S., March 2007.

2007-10 *User Interaction Widgets for Interactive Theorem Proving (PhD Thesis)*, Zacchiroli, S., March 2007.

2007-11 *An Ontology-based Approach to Define and Manage B2B Interoperability (PhD Thesis)*, Gessa, N., March 2007.

2007-12 *Decidable and Computational Properties of Cellular Automata (PhD Thesis)*, Di Lena, P., March 2007.

Dottorato di Ricerca in Informatica
Università di Bologna e Padova

User Interaction Widgets for Interactive Theorem Proving

Stefano Zacchiroli

March 2007

Coordinatore:	Tutore:
Ozalp Babaöglu	Andrea Asperti

Abstract

MATITA (that means *pencil* in Italian) is a new interactive theorem prover under development at the University of Bologna. When compared with state-of-the-art proof assistants, MATITA presents both traditional and innovative aspects.

The underlying calculus of the system, namely the Calculus of (Co)Inductive Constructions (*CIC* for short), is well-known and is used as the basis of another mainstream proof assistant—Coq—with which MATITA is to some extent compatible. In the same spirit of several other systems, proof authoring is conducted by the user as a *goal directed proof search*, using a *script* for storing textual commands for the system. In the tradition of LCF, the proof language of MATITA is *procedural* and relies on *tactic* and *tacticals* to proceed toward proof completion. The interaction paradigm offered to the user is based on the *script management* technique at the basis of the popularity of the Proof General generic interface for interactive theorem provers: while editing a script the user can move forth the *execution point* to deliver commands to the system, or back to retract (or "undo") past commands.

MATITA has been developed from scratch in the past 8 years by several members of the HELM research group, this thesis author is one of such members. MATITA is now a full-fledged proof assistant with a library of about 1.000 concepts. Several innovative solutions spun-off from this development effort. This thesis is about the design and implementation of some of those solutions, in particular those relevant for the topic of user interaction with theorem provers, and of which this thesis author was a major contributor. Joint work with other members of the research group

is pointed out where needed. The main topics discussed in this thesis are briefly summarized below.

Disambiguation. Most activities connected with interactive proving require the user to input mathematical formulae. Being mathematical notation ambiguous, parsing formulae typeset as mathematicians like to write down on paper is a challenging task; a challenge neglected by several theorem provers which usually prefer to fix an unambiguous input syntax. Exploiting features of the underlying calculus, MATITA offers an efficient disambiguation engine which permit to type formulae in the familiar mathematical notation.

Step-by-step tacticals. Tacticals are higher-order constructs used in proof scripts to combine tactics together. With tacticals scripts can be made shorter, readable, and more resilient to changes. Unfortunately they are de facto incompatible with state-of-the-art user interfaces based on script management. Such interfaces indeed do not permit to position the execution point inside complex tacticals, thus introducing a trade-off between the usefulness of structuring scripts and a tedious big step execution behavior during script replaying. In MATITA we break this trade-off with *tinycals*: an alternative to a subset of LCF tacticals which can be evaluated in a more fine-grained manner.

Extensible yet meaningful notation. Proof assistant users often face the need of creating new mathematical notation in order to ease the use of new concepts. The framework used in MATITA for dealing with extensible notation both accounts for high quality bidimensional rendering of formulae (with the expressivity of MathML-Presentation) and provides *meaningful notation*, where presentational fragments are kept synchronized with semantic representation of terms. Using our approach interoperability with other systems can be achieved at the content level, and direct manipulation of formulae acting on their rendered forms is possible too.

Publish/subscribe hints. Automation plays an important role in interactive proving as users like to delegate tedious proving sub-tasks to decision procedures or external reasoners. Exploiting the Web-friendliness of MATITA we experimented with a broker and a network of web services (called *tutors*) which can try independently to complete open sub-goals of a proof, currently being authored in MATITA. The user receives *hints* from the tutors on how to complete sub-goals and can interactively or automatically apply them to the current proof.

Another innovative aspect of MATITA, only marginally touched by this thesis, is the embedded *content-based search engine* WHELP which is exploited to various ends, from automatic theorem proving to avoiding duplicate work for the user.

We also discuss the (potential) reusability in other systems of the widgets presented in this thesis and how we envisage the evolution of user interfaces for interactive theorem provers in the Web 2.0 era.

Acknowledgements

The "acknowledgments" part of a manuscript is a tough one to be written, especially at the end of a Ph.D. track, when you are full of doubts and living in uncertainty. My choice to address this is then to acknowledge two classes of people, those who were my track-mate and those who shared (sometimes surviving, sometimes failing the daunting task) the burden of my rants and my behaviors during my graduate studies.

Since there is nothing better than an exception to confirm a rule, I start expressing my gratitude to a person not belonging to any of the above two categories: Fabio Menaglio. He will probably never read any of this (though the powers of Google have no limits, I know ...) since he is a craftsman and he behaves more in RL than in the scientific or hacker communities. Still, without him, I wouldn't have been here with an almost completed Ph.D. thesis in computer science on my laptop, and I wouldn't have spent a huge share of my spare time in the last 9 years contributing with my tiny bits to the free software movement. He taught me what it means to *rule* a computer as opposed to *use* one, he is a true hacker and he does not know that. A pity we lost each other, but life is funny, we will meet again. Thank you Fabio, I owe you so much.

Several people have been my track-mates, with different roles. My advisor, Prof. Andrea Asperti, deserves the first mention. I am sure I am not among its best graduate student, nonetheless he has always believed in me supporting my 3 years of Ph.D. work, both financially and with his constant presence. For all the

time I have been working in the HELM team (the research group he leads), starting from a couple of years before my master thesis ranging to these days, he has always been with me and the other HELM-ers, most of the time even sitting in the graduate student lab at one of our desks. I can hardly imagine an advisor so present and passionate for his research. I also like to remember the ability of Andrea to astonish us, with his Egg of Columbus solutions to problems able to heat team discussions for hours, time better spent by him finding a working solution. I regret having spent too few of my Ph.D. working on topics which were of direct interest for him, since working side-by-side with Andrea is enlightening. I hope the future will give me back some of that time.

I'm of course in debt with the other members of the HELM team as well, most of them has became in the years more friends than simply colleagues or paper co-authors. In particular I want to thank here Claudio Sacerdoti Coen, Luca Padovani, and Enrico Tassi.

Claudio (AKA "CSC") has been my mentor in the HELM team from the very beginning. Working side-by-side with him is as much fun as is rewarding being in touch with someone as talented as he is. I hope our office-mates will forgive the noise of our laughters in years of extreme programming (more for the huge pile of hacks, than the SWE philosophy ...), sitting in front of the same monitor with turn-over at the keyboard, spotting bugs in compilers or in the most "trustworthy" tools we had at hand.

Luca shared with me more than an office. For instance, he shared with me a ugly house near the computer science department for a couple of years. But more than that, during those years we shared moods—more bad than good—induced by our work. From time to time I still type "`vi dati/misc/luca_s_quotings.txt`" on a console of my laptop and spend tens of minutes rereading some of his memorable quotations. They still have the power of improving my mood when needed.

Enrico is a young addition to the HELM team, but we recognize each other

from the first day. We are probably both more passionate about free software than everything else related to computers. I would have hardly survived the beginning of several office days without gossiping with him about Debian while sipping the first coffee of the day.

Time to address the second part of the acknowledgments: friends, more than friends, and all who were and still are the important people of my life. Don't worry, I'll be brief with them, because they all already know how much they mean to me, and it is something that hardly can be written down here.

Standing me in the past years has proven to be a challenging task. Surprisingly, a lot of people took up the task and are still here at my side. I frankly don't understand—given that at their place I would have sent myself to hell more than once—but I'm immensely grateful to all of them. The first I want to mention here are Gaia and Marco, they are the two I look for when I need help, wiseness, or just targets for my bad mood. Thank you guys, I'm alone without you.

Then several other people deserve to be mentioned; here they come, in no particular order: Gaspa, Monty, and (la) Fra (my friends for a whole life), their relatives, Sara (who stole my heart long time ago), Arianna (who had the chance of understanding and sharing a side of mine I've always believed to be the "true" me). Thank you all, for having turned my life in a pleasant experience.

Contents

List of Tables

List of Figures

Part I

Background

Chapter 1

Introduction

This thesis is about the design and development of a set of reusable widgets—currently part of the Matita proof assistant—that have been born working at the intersection of two topics: interactive theorem proving and user interaction.

Interactive theorem proving is a relatively new field amidst computer science and mathematical logic concerned about formalizing proofs with the help of man-machine collaboration, so that the proofs developed can be mechanically checked for correctness. Interactive theorem provers—or *proof assistants*—are the software applications meant to help the user in the formalizing process (note that from our point of view the "formal" adjective refers to the quality of a proof of being encoded in a rigorous setting that enables mechanical checking of correctness, as opposed to *rigorous*, the quality of pen-and-paper mathematics).

The degree of help obtained from man-machine collaboration varies a lot. At one extreme ("no help from the system") sit *proof verifiers*, who are kind of boolean oracles that consumes as input proofs encoded in some language understandable to them and returns a boolean value stating whether the proof is correct or not, possible with additional information motivating the answer. The pioneering Automath system [33, 76] developed by De Bruijn in 1967 was an example of such a system. It was able to verify proofs written in the Automath proof language.

At the opposite extreme ("maximum possible help from the system"), and no longer part of the "interactive" theorem provers category, we find *automatic theorem*

provers. Such systems (like Otter [68], Vampire [91], and the other participants to the annual CASC competition [103]) act in a batch fashion consuming as input statements of theorems expressed in some language and either affirm of confutes their validity. Besides being more fascinating than their interactive counterparts, automatic theorem provers are still not enough powerful to be used alone in the two major applicative fields of theorem proving: formalization of mathematics and program verification.

In fact, fully automatic *decision procedures* are frequently bundled with proof assistants to solve proof sub-problems pertaining to particular domains, but the thrust of the proofs in program verification or formalization of mathematics is mostly user-driven.

Of the two mentioned applicative fields of theorem proving, the latter, *program verification*, is the activity of formally proving the correctness, with respect to a given specification, of a computer program. Similarly, theorem provers supporting *program extraction* [87] can also be used to generate a certified computer program from a proof of the satisfiability of a program specification.

Formalization of mathematics is about redoing, using a theorem proving system, pen-and-paper proofs of theorems from the knowledge base of mathematics. Though "redoing" may sound negative, several good reasons can be found to do that. From the point of view of mathematical knowledge management for example, formalizing proofs is a net-win for two of its topics, namely the digitization and preservation of mathematical knowledge and the encoding of information in machine readable format: a formal proof is both easier to preserve and communicate than a pen-and-paper proof, and is encoded in a format of which the machine has the deepest possible understanding.

Other possible reason for formalizing pre-existent mathematics include:

- verifying the correctness of pen and paper proofs for the purpose of validating work done by mathematicians in a rigorous (as opposed to formal) setting;
- verifying the correctness of programmatic parts of mathematical proofs (as

in the famous cases of the proof of the Kepler conjecture by Hales—whose computational part is being formalized collaboratively by the members of the Flyspeck project [45]—and of the proof of the four colors theorem in Coq [26]);

- developing a knowledge base of formalized mathematics, to be used as a basis for large program verification or mathematic formalization tasks.

Mathematical knowledge management [23, 70] (or MKM for short) is a new research field of interest for our dissertation, it sits at the intersection of mathematics and computer science and is articulated around a set of topics. Two of them, briefly mentioned before, are *preservation of mathematical knowledge* (with the thrust on both digitization of knowledge available in printed format only and on exporting into application-independent electronic libraries knowledge which is at present application-specific) and *encoding of information in machine readable format* (as a basis to enable meaningful processing of the information by the machine). Other topics that will be touched by this work are: *standardization of mathematical knowledge*, *indexing and retrieval*, *rendering and publishing*, and *enhancement of interoperability between systems dealing with digitized mathematics*.

The link between interactive theorem proving and mathematical knowledge management has been proven synergistic to the two worlds [94] and is at the basis of Matita, the proof assistant whose development spun-off the widgets discussed in this thesis. Matita is a new proof assistant, being developed since 1999 by the Helm team—led by Prof. Asperti—at the University of Bologna.

Started as an attempt to solve some of the technological problems hindering the adoption of proof assistants by mathematicians, exploiting advances in the fields of web publishing and digital libraries, the motivations for developing and maintaining Matita are still valid:

- The applicative area of proof assistants, after 40 years since Automath, still presents too few players.

  A recent book by Wiedijk [117] reviews the proof assistants an interested user can nowadays choose from and it mentions 17 different provers (just to mention

the ones who are probably the major players in the field: Mizar [69], Coq [26], PVS [88], Isabelle [53], HOL Light [51], and ΩMega [17]). Given that most of them do not share any logical foundation (i.e. the formal system in which proofs are encoded in order to be mechanically checkable for correctness) and that most of them are stuck to de facto standards for other aspects of the system (e.g. generic user interfaces), a systematic comparison of features is rather hard to perform, hindering the development of a positive competitive feedback loop in the proof assistant developer community.

MATITA has been developed entering in direct competition with a preexisting player—the Coq proof assistant—striving to introduce innovation in various areas of the system development, yet inheriting already successful features. Some of the attempts failed, while other succeeded bringing benefits that have been later on adopted by the competing system (for example content-based queries implemented by the WHELP search engine [7], now exploitable directly from Coq).

- As anticipated, MATITA has also been developed to give evidence of the synergy between interactive theorem proving and some of the topics of mathematical knowledge management. Nowadays the synergy manifest itself mostly in the bias of MATITA towards mathematician-friendly input and output of mathematical formulae in its authoring interface. This includes both high-quality MathML-based rendering of formulae and TEX-like input of them with support for disambiguation of information which are usually left implicit in rigorous mathematics.

  The *global visibility philosophy* of MATITA, where the system itself is thought to be an intelligent browsing tool for a library of formalized mathematical knowledge as well as the ability to easily publish formalized concepts as part of the HELM library [48] are part of the heritage of its MKM background as well.

- Last but not least, MATITA has been developed as an experimental proof as-

sistant to be used as a basis for the evaluation of new solutions, both logical and technological. The problem we are trying to solve is the immobility of several other players of the field, which are often developed by small research teams (as the HELM team is) but nonetheless tied to chains of backward compatibility with the non-negligible amount of knowledge already formalized in their libraries.

For this reason the code base of MATITA attempts to remain small (less than a half of that of Coq roughly implementing the same features [9]) and its software complexity is meant to be mastered by graduate or even master students (a relevant source of man power for academic development projects).

The mentioned motivations have affected the development of MATITA. In fact, the largest part of the code which currently forms MATITA is a set of software components which have been developed far before the idea of creating a stand-alone proof assistant came into play. All those components was originally meaningful per se as stand-alone software libraries. Many of them were developed for implementing added value services on top of the HELM (Hypertextual Electronic Library of Mathematics) library [6] before and during the European project MoWGLI [72]. The most noteworthy components are:

- an XML dialect encoding terms of the Calculus of Inductive Constructions, with libraries for parsing and marshalling mathematical concepts in such a format and an exportation module for Coq to export to this format concepts of its library [93];

- metadata specifications [43] for indexing and a search engine [7] exploiting them to query a distributed library of concepts encoded in the XML dialect above;

- a proof checker (that is the *kernel* of a proof assistant) for the Calculus of Inductive Constructions [92] offering a Web Service interface for checking objects of the HELM library [125];

- a sophisticated term parser (used by the search engine), able to deal with potentially ambiguous and incomplete information, typical of the mathematical notation [97];

- a *refiner* component, that is a type inference system able to validate and reconstruct implicit information in CIC terms, used by the disambiguating parser [94];

- complex transformation algorithms for proof rendering in natural language [8];

- a GTK+ MathML-Presentation-compliant rendering widget [83], supporting high-quality bidimensional rendering and semantic visual selection.

Starting from the above components, the development of MATITA as a standalone proof assistant "just" involved designing a coherent authoring interface for stating and proving theorems interactively, and connecting with it the components above. The result is a proof assistant which resembles its historical predecessor in many ways, still presenting several innovations from the point of view of user interaction and internal software engineering.

The modularity of the code of MATITA is thus a heritage of the way the system was born, and the internal architecture of the system (which will be discussed in Chapter 2) consists in a set of components depending on each other through small APIs. It is not a coincidence that the improvements over the state-of-the-art in the user interaction with the system correspond to reusable system components that can easily be ported to other competing systems, either as a code or as architectural choices.

This dissertation describes a set of those reusable components which mainly deal with the user interaction with MATITA, called for this reason *user interaction widgets*. For each of them we describe their design, their role, and their implementation in MATITA. Where needed we also present their formal specifications, more as a precise description of the algorithmic rules governing their behavior than as a theoretical basis to prove theoretical properties of the widgets.

The actual widgets described by this thesis are those on which this thesis author contributed the most. They are briefly summarized here:

Disambiguation Formulae in MATITA can be input by the user using a syntax close to the usual mathematical notation expressed with a TEX-like markup. The disambiguation component implements a disambiguation algorithm [97] (mainly developed in the work of Sacerdoti Coen [94]) which exploits the refiner to validate partial CIC terms and infer missing information, adding on top of that a notion of spatial and temporal locality easily found in theory development;

Step-by-step tacticals Proof assistants sharing with MATITA the use of a procedural tactics-based proof language rely on LCF-like tacticals [41] as the primary tool for the composition of tactics. MATITA developed *tinycals* [95] as a proof language replacing a subset of LCF tacticals, which improves the latter offering a small step execution semantics. This aspect of tinycals improves the integration with authoring interface based on *script management*—like the de facto standard generic interface for theorem provers Proof General [12]—solving several user interaction problems.

Extensible yet meaningful notation User-extensible notation is a need commonly found in interactive theorem provers, it is exploited by authors to assign mathematical notation to newly developed concepts. MATITA has such a feature and the framework used to implemented it offers improvements over the state-of-the-art supporting *meaningful notation* [85], that is notation which blends together notational and presentational aspect, enabling direct manipulation of presentation-level terms in a semantic meaningful way, without renouncing to interoperability at the content level.

Out-of-band hints Exploiting the web friendliness of its various software components, MATITA can be integrated with a publish-subscribe network of proof helpers which notify the user when (part of) the current proof goal can be

solved applying particular tactics. The proof helpers are called *tutors* and can be easily deployed being just MATITA components compiled and run separately from the proof assistant itself.

Structure of the Thesis

This thesis is structured in 7 chapters and 1 appendix. Chapter 2 gives an overview of the MATITA proof assistant, both from the point of view of the user of the system (comparing its features with that of competitors, keeping a particular eye on user interaction aspects of the authoring interface) and from the point of view of the system developer. A detailed description of the internal software architecture of the system is provided in term of macro components, discussing their sizes and their reusability in other systems.

Chapter 3 is the first chapter presenting widgets; it present the disambiguation algorithm and how it has been extended to be used in a theory development context using MATITA. Chapter 4 discuss tinycals, showing how they improve the user experience when both LCF-like tacticals and script management are used in an authoring interface like that of MATITA. Chapter 5 present the novel concept of meaningful notation, an architecture implementing it, and its instantiation in MATITA. Chapter 6 present the last widget, namely HBUGS, the publish-subscribe architecture to which MATITA can be connected supporting the user with out-of-band hints.

Chapter 7 is a first investigation in a field that will probably see the interest of the theorem proving community in the near feature: wiki-like Web sites. It introduces both an open challenge for the scientific community doing research on wikis—"how can wikis be used to edit content on which constraints have to be enforced?"—and propose the novel concept of *light constraints* [52] to model the interaction with such wikis. In our opinion wikis are the platform of Web 2.0 which will inherit the burden of collaborative development of formal mathematics which has failed to be carried by former web technologies.

Appendix A lists and briefly describes all the stand-alone software components

that have been developed during the Ph.D. work of this thesis author. The list does not include the widgets described in the previous chapters, nor other components of Matita, but rather the external libraries which are loosely coupled with the proof assistant itself and which have grown into mature software libraries used by applications other than and not related to Matita.

Chapter 2

The Matita Proof Assistant

The widgets discussed in this thesis are reusable software architectures and implementations, currently part of the MATITA proof assistant. This system is rather new: at the time of writing no release with an official version number has been distributed yet. Nonetheless the code is free (as in "free speech") and snapshots of the latest development versions can be downloaded from `http://matita.cs.unibo.it` and tested by anyone.

The dissemination of MATITA started presenting the system at several international conferences and workshops during the past two years, but since the user base is still small we present in this chapter the peculiarities of the system, with a particular emphasis on the user interaction angle.

We also present the internal software architecture of the system which will be used in further chapters to better understand the role of the various widgets in the design and implementation of MATITA.

2.1 Matita in a Nutshell

MATITA is a new document-centric proof assistant being developed by the HELM team, led by Prof. Asperti, at the University of Bologna. The word "matita" is rather evocative in Italian: it means "pencil", the primary authoring tool used by mathematicians to produce mathematical knowledge.

The logical foundation of MATITA is the Calculus of Inductive Constructions (*CIC* for short) [113], the same of the Coq [26] proof assistant which inspired the development of our system and with which MATITA is at some extent compatible.

Still, the implementation of the calculus is not shared by the two systems which can be distinguished by several other characteristics, the most peculiar being library management. In MATITA the knowledge base of the system is conceived as a suitably indexed, searchable and browsable repository of mathematical concepts (definitions and theorems) which is always fully visible to the user. MATITA is meant both as a tool to author new concepts of such library and as a browsing tool for the concepts contained therein, with added value services inherited from the HELM [6] library.

Definitions, theorem statements and proofs are given by the user textually in a procedural proof language, following the tradition who goes back to the LCF theorem prover [41]. Such a tradition is still followed nowadays by several other competing and successful systems like Coq, NuPRL [78], PVS [88], and Isabelle [53], whereas sometime coupled with an alternative declarative proof language [112] (path MATITA is following these days as well).

Systems descending from LCF also share the use of textual *scripts* recording commands to be fed into the prover, using them as the primary interaction mechanism to chat with the system. Proof terms are generated as side-effects of command execution and are stored internally by the system only for efficiency reasons. MATITA is no exception. Its proof language—called *grafite* ("graphite" in English)—is used to write `.ma` script files; the execution of such scripts generates proof terms which are recorded in an XML dialect encoding the calculus of inductive constructions.

The thrust of MATITA toward its library—also meaning that we consider the library relevant per se, even without the system who produced it—gives to the XML-encoded proof terms an additional light other than efficiency: they are also meant as the primary data type for long-term storage of formalized mathematical concepts and communication with other systems. In fact, they can also be published on the Web building a potentially distributed library of mathematical knowledge which can be used by other users running MATITA elsewhere.

All proof assistants based on procedural proof languages share also the choice of a user interaction paradigm—namely *script management*—which has been pioneered by the CtCoq system [20]. The idea of script management is to rely on a textual buffer containing the commands to be executed and an execution point on it which can be moved forward to execute more commands or backward to retract past commands. User interfaces (UIs) based on such idea are traditionally organized in three windows: one for giving feedback to the user of the current proof state (in several systems this reduces to showing the open conjectures, or *sequents*, in the current proof), one for editing the proof script, and one for delivering messages like errors or notices to the user.

The Proof General generic interface for theorem provers [12] is a successful incarnation of those ideas. Proof General is the de facto standard implementation of script management. It is based on a standardized protocol for communicating with an underlying proof system and for this reason it is widespread and compatible with several of the proof assistants previously mentioned. Other more modern interfaces like CoqIDE (a Coq-specific integrated development environment) have been implemented from scratch, but are really similar from the final user point of view.

The authoring interface of Matita—shown with window label annotations in Figure 2.1—shares the same interaction paradigm and essentially offer the same functionalities. A peculiar added benefit of our script editing window (which will be discussed at length in Chapter 4) is the granularity of the execution step on scripts which is more fine grained, resulting in a valuable support for structured editing of proof scripts.

On the other hand Matita sensibly differ from other script-management-based user interface in the sequents window. There we offer high-quality bidimensional rendering based on MathML, obtained exploiting GtkMathView[1] [83], a GTK+ widget for rendering MathML Presentation markup. GtkMathView was originally conceived as a component of the, at that time forthcoming, proof assistant

[1] `http://helm.cs.unibo.it/mml-widget/`

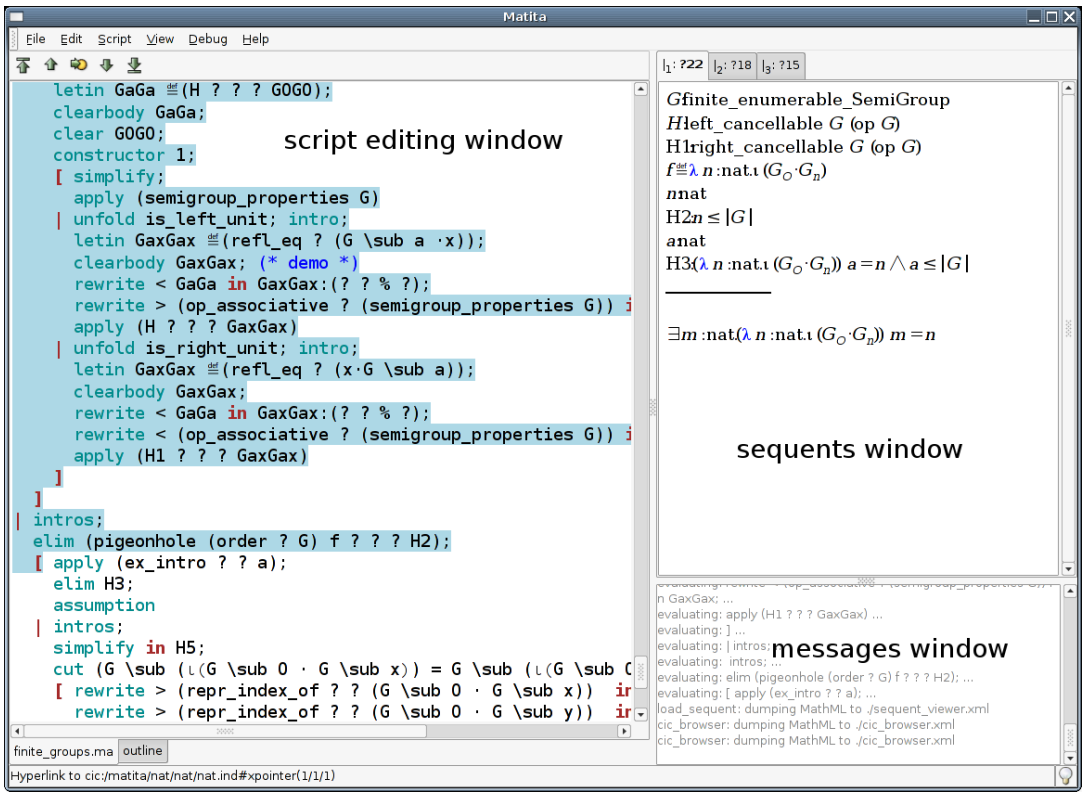

Figure 2.1: Authoring interface of MATITA

and has evolved in an independent component used in several mainstream applications not related to theorem proving (for example in the AbiWord[2] word processing program).

On top of GTKMATHVIEW we implemented added value features like MATITA hypertextual capabilities (with links pointing to concepts available in the library) and semantic selection which constrains the visual selection on the widget to logically meaningful terms. Such features will be discussed in Chapter 5.

The browsing role of the MATITA user interface is fulfilled by an additional utility window known as the *CIC browser*. It is used for showing several different kinds of information to the user, in a way similar to how a Web browser is used to surf the Web. Plain browsing of the MATITA library (see Figure 2.2) is implemented as a

[2] `http://www.abisource.com`

directory service in the Gopher tradition. The namespace used to address objects in the library is structured by URIs (Uniform Resource Identifiers) [110] which can be navigated as folders containing other folders and mathematical concepts.

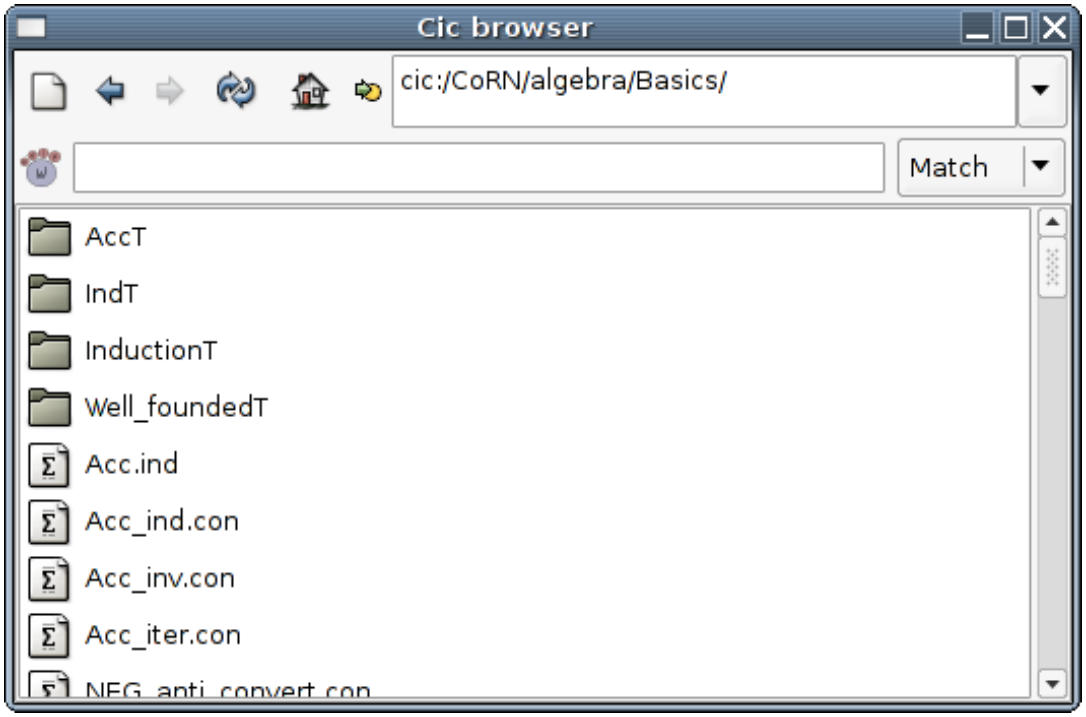

Figure 2.2: Browsing the library

Proofs (even incomplete ones) available in the library can be inspected within the CIC browser rendered in a pseudo-natural language generated on the fly from CIC terms, on the line of [27, 28] (see Figure 2.3). The pseudo-natural language rendering is precise enough to be reused as the declarative version of the grafite proof language.

Finally, the CIC browser can be used to query the metadata used to index the concepts available in the library. In particular, the dependencies among concepts can be visually browsed as lazily expandable directed graphs (see Figure 2.4), and a Google-like search engine bar is available to perform content-based queries over the whole knowledge base of the system (see Figure 2.5).

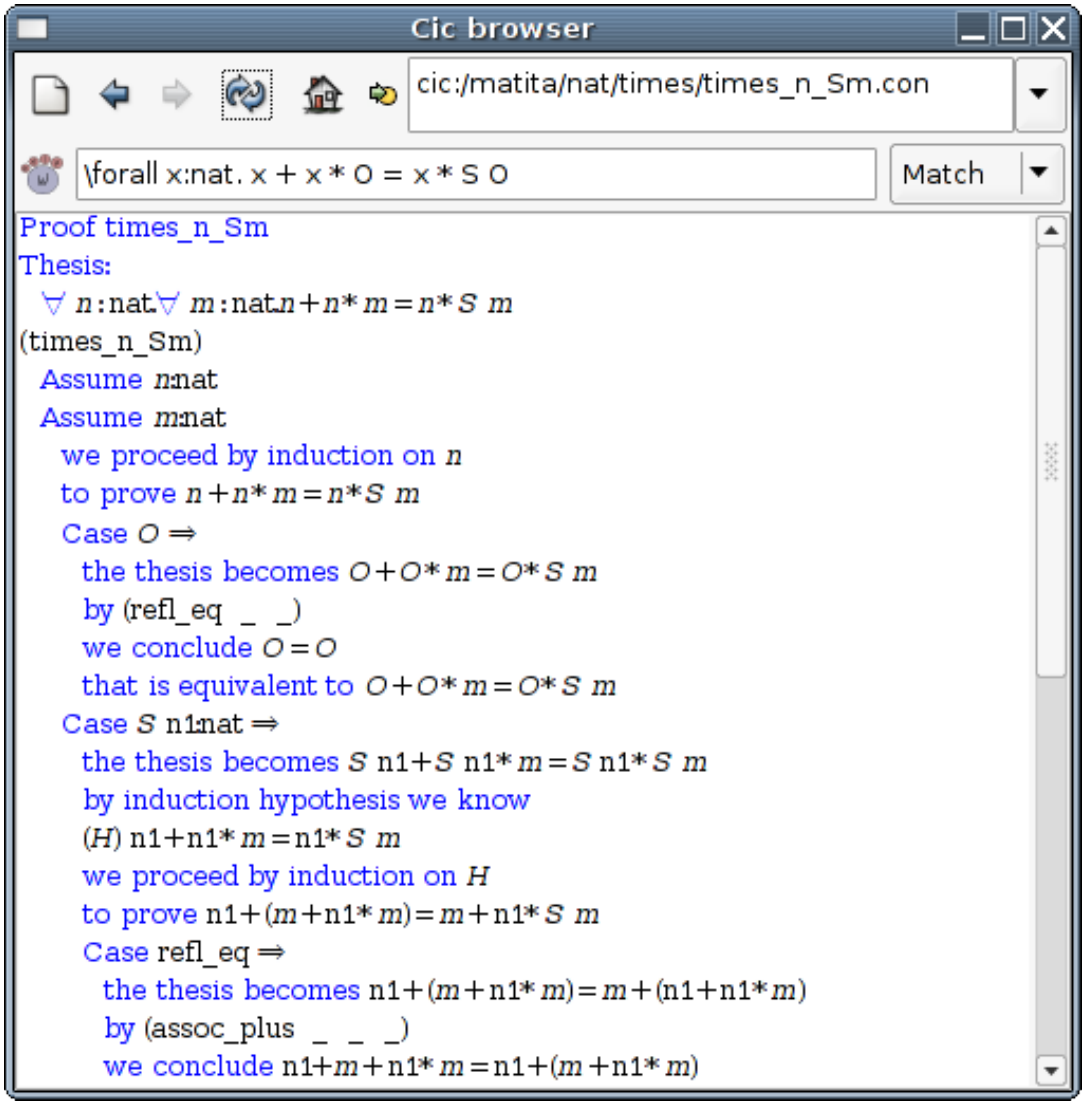

Figure 2.3: Browsing a proof in pseudo natural language

MATITA is entirely free software (licensed under the terms of the GNU General Public License[3]). Its code is mostly written in the Objective CAML[4] (OCaml) programming language. The source code is available for download from the MATITA Web site[5].

[3] `http://www.gnu.org/copyleft/gpl.html`
[4] `http://caml.inria.fr`
[5] `http://matita.cs.unibo.it`

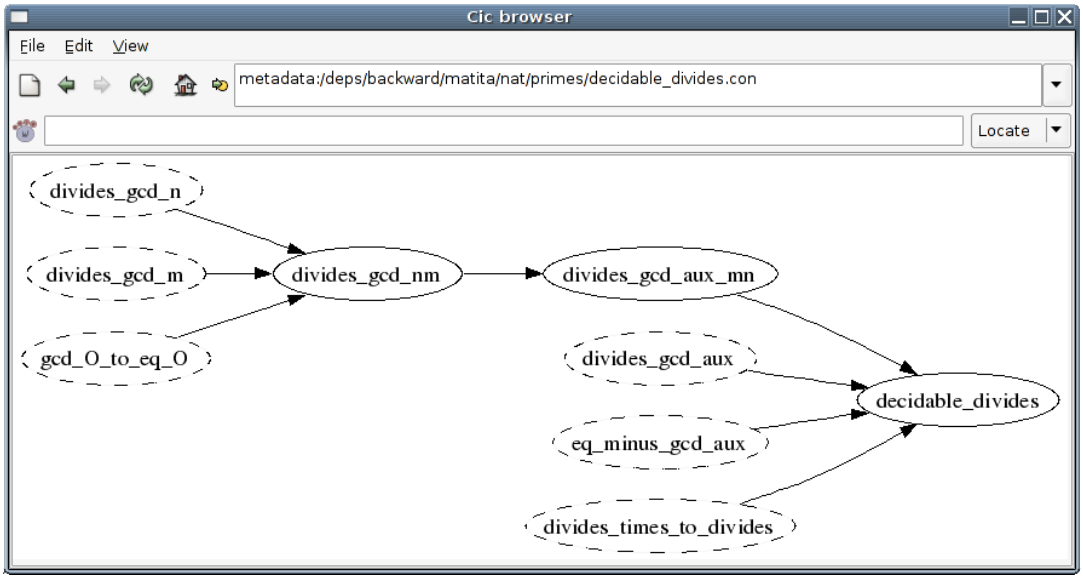

Figure 2.4: Dependencies analyzer

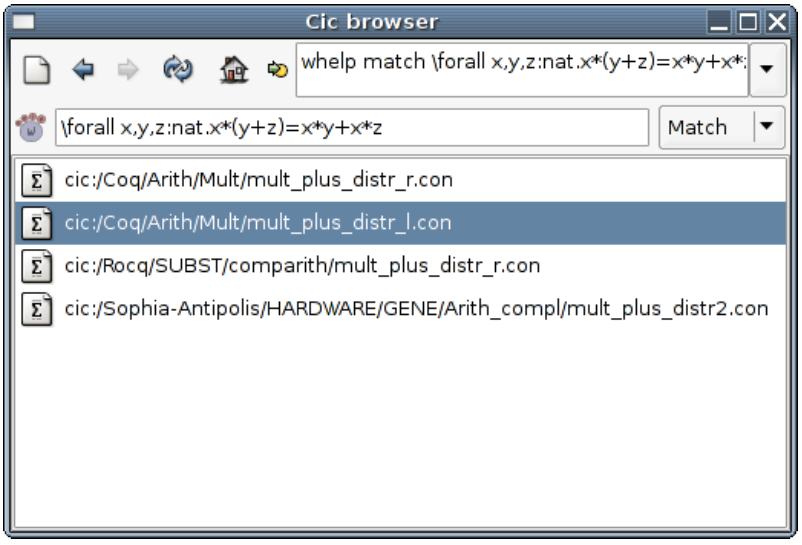

Figure 2.5: Pattern matching query

In the remainder of this section we give some highlight on the peculiarities of MATITA, and in particular of its library management and authoring interface.

2.1.1 Library Management

Despite the similarities in the user interface, working with MATITA is (supposedly) radically different from a methodological point of view with respect to other current tools for formal reasoning, which traditionally put emphasis on other aspects (like proof checking or authoring on a per-proof basis).

Components of the knowledge base of MATITA are mathematical concepts (inductive definitions, theorems, and axioms) and user defined notational definitions (interpretations and notational equations). Both concepts and notational definitions are authored sequentially by typing textual commands in the script editing window and then fed them into the system by moving the execution point past them. Once evaluated, the corresponding persistent versions are stored in the library—or better in the fragment of the library the user can edit—and become visible; they will remain so upon re-entering the system until the user asks for their deletion. There is no need to explicitly require or include portion of the library to see previously authored concepts.

Indexing and Searching

As a side-effect of storing concepts in the library just after authoring, each concept gets indexed, extracting from it a rich metadata set which can be exploited later on to various ends.

The metadata set has been tuned in previous works of the HELM team [11, 42, 43] and during the MoWGLI European project. It is based on a single ternary relation $Ref_p(s, t)$ stating that a (source) concept s contains a reference to a (target) concept t and that that reference occurs in position p of the source concept. The set of supported positions is currently fixed and is able to discriminate concepts occurring in the hypotheses or in the conclusion of statements, taking into account the depth (i.e. the number of binders walked through in a visit of the term syntax tree) of the occurrence.

The points of the $Ref_{\square}(\square, \square)$ relation extracted indexing authored concepts provide an approximation of such concepts and are stored in a relational database for

efficiency reason. The precision of the approximation is governed by the granularity of the positions, which we have tuned during years of experiments with different kinds of queries as use cases.

Several features of MATITA exploits directly or indirectly the indexing mechanism. The most noteworthy feature implemented on top of them is a set of content-based queries, which has been factored out from the WHELP search engine [7] and can be submitted directly using a CIC browser window. The available queries are briefly described below, a detailed description of them can be found in [7]:

Match given a CIC term as input, this query indexes the input term and looks in the library for all concepts whose statements share the same metadata set obtained indexing the input term. Intuitively this query returns all concepts that prove a given statement, and is implicitly able to deal with changes in the form the statement is expressed, like α-conversion or permutation of operands.

Hint given a possibly incomplete CIC term as input, this query returns all concepts which can be applied (in the backward reasoning sense) to prove a conclusion matching the given input term. This can be expressed in our metadata model by looking for all concepts containing, in appropriate positions, the constants occurring in the input term (a detailed description of the solution currently implemented has been given in [11]).

Elim given a CIC term corresponding to an inductive data type, this query returns the list of all concepts which are induction principles or recursors on the given data type; that is all concepts which can be used either to prove by induction theorems on a given data type or to write recursive programs on it.

Locate not really exploiting the metadata set, this query simply implement a lookup by short name for concepts in the library. Given an identifier as input locate returns the list of all the concepts which share the very same non qualified name.

Instance given a CIC term mocking up a mathematical notion, this query returns all the concepts which are instances of the given notion. For example, the notion of "being commutative" can be expressed giving as input the term $\lambda A : \texttt{Set}.\lambda f : A \to A \to A.\forall x, y : A.(f\ x\ y) = (f\ y\ x)$.

Examples of additional features which internally use the indexing and querying mechanisms are:

- the *duplicate check*, which performs behind the scene a match query as soon as the users states she want to prove a new theorem. If a theorem matching it is found in the library and if such theorem is convertible (a logical and stronger than matching notion as expressed by the metadata set) with the one the user is willing to prove, then a warning is raised to the user to encourage the reuse of already proven results;

- the *disambiguation*, which will be discussed later on, uses the locate query to resolve short names, helping the user in not requiring her to enter fully qualified names when those can be automatically resolved;

- the *semi-rigid naming convention* for theorem names, which have been derived from our studies about statement metadata. The idea behind the convention is to require the user to enter names composed by particles separated by underscores ("_"), where each particle is either the short name of a concept occurring in the statement, or a "`to`" denoting logical implication.

  For instance a valid name for the statement "`\forall n,m:nat. n \lt m \to n \leq m`" would be "`lt_to_leq`". In our experience such a convention hints better structuring of groups of related theorems and helps the user in guessing theorem names. More information on the MATITA naming convention are available in [10];

- the *automation* fragment of MATITA, currently implemented by the "`auto`" tactic can exploit the indexing of the library if asked to do so. Some of the

different flavours of the tactic indeed iterate the hint query to find applicable theorems and then applies them recurring the application of the tactic. Other flavours do not make direct use of the hint query, but rather perform more low-level queries, but still on the same metadata set on which match and the other queries are based on.

This way the user is freed by the burden of the overall management of the library for the purpose of automation: there is no need to explicitly state which part of the library has to be used for automatic proof search. The theorems which are more likely to be useful for the task are filtered out by extracting a signature of the goal the user is willing to prove.

Invalidation and Regeneration

The workflow of authoring concepts of a library may require to go back to previously authored concept and change them to better fit the changing needs of the formalization task [2]. This aspect interacts non trivially with the philosophy of Matita of an always visible library, in particular the question "what happen to the previous version of the concepts which was stored in the library and to all the previously defined concepts which depend on it?" needs to be answered.

In Matita this aspect is addressed by two mechanisms: invalidation and regeneration. Used together they give the user the freedom of adding, removing, and modifying mathematical concepts without loosing the feeling of an always visible and browsable library.

When the user attempts to redefine a part of the library which is already defined (i.e. starting to store concepts under a URI which already contains concepts), Matita asks the user if its intention is to redefine that part of the library. If this is the case all concepts belonging to the given URI fragment (the list can be easily obtained from the abstraction mechanism of Matita over its library, the Getter; see Section A.4) are invalidated.

Invalidation is a two phase process. The first one collects all the concepts that recursively depend on the concept being invalidated (query that can be easily an-

swered exploiting the metadata stored in the database); the second phase remove all the collected concepts from the library. The same technique, described at length in [125] is used by the dependencies analyzer shown in Figure 2.4.

Regeneration is the inverse mechanism of invalidation. Since each script in MATITA starts with a statement of which part of the library is defined by that script, having knowledge of the scripts who compose a development it is possible to automatically generate parts of the library when there is the need to do so (for example when a script explicitly states that it requires the availability of a particular part of the library to be built properly).

Two auxiliary tools shipped with MATITA are used to implement regeneration.

`matitadep` takes as input a set of scripts and return a suitable compilation order for them;

`matitac` is a batch compiler for scripts which executes in turn every command contained in a script, storing in the library all the resulting concepts and notational definitions.

Note that the user is not required to invoke by hand neither of these tools, they are transparently invoked by the authoring interface when needed.

We now move to the peculiarities of the MATITA authoring interface with respect to other interfaces commonly found in competing systems.

2.1.2 Authoring Interface

Direct Manipulation of Terms

Both the sequents window and the CIC browser can show formulae to the user, possibly embedded in the pseudo-natural language rendering of concepts. Both windows use the GTKMATHVIEW technology for rendering a mixed MathML Presentation/BoxML (a language for expressing line breaking hints) markup, enriched with MATITA specific annotations. The annotations (hyperlinks and cross-references) are

used as pointers to concepts in the library and to CIC sub-terms which were used to generate the markup rendered to the user.

Matita exploits the annotations offering *hypertextual browsing* by the means of hyperlinks whose anchors are glyphs occurring in the markup (identifiers, literal numbers, and operators) and whose targets are concepts contained in the library. Clicking on the anchor launches an instance of the CIC browser showing the target concept.

Another feature implemented on top of annotations is *direct manipulation* of terms. The user can indeed visually select part of the rendered markup with the mouse, with the invariant that the selected part always corresponds to a well formed CIC sub-term, and in particular to the sub-term which generated, during the rendering phase, the selected markup (this invariant is said to implement *semantic selection*). Once selected, semantic meaningful transformation can be performed on the corresponding CIC sub-term like semantic copy&paste (where the pasted text is granted to be parsed back as the selected sub-term) and application of tactics which consume the selected sub-term as one (or more, given that multiple selections are supported as well) of their term arguments.

Direct manipulation of terms is described in more detail in Chapter 5.

Disambiguation

During theory development with a proof assistant, input of terms is a frequent need: theorems are stated giving terms of a particular sort, tactics might need term arguments, queries require terms as input, and so forth. The choice of the input syntax for terms is thus rather important and has the power of drive away or attract particular kinds of users. In Matita we chose to stick to the standard mathematical notation and permit our users to input terms via textual typing of formulae linearized in a TEX-like encoding (a well-known markup language among mathematicians). Unicode can also be freely exploited for input of mathematical glyphs.

For the purpose of input, the main problem posed by this choice is its ambiguity, which is induced by various factors: conflicting parsing rules, hidden information to be recovered from a vague notion of context, overloading, and subtyping (implemented in Matita and other provers by means of coercive subtyping [63]).

In most programming languages these challenges are usually solved by fixing an unambiguous input language. Since we do not want our input syntax to drift too much from mathematical notation we decided to not take this path. This way 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.

The *disambiguation* mechanism of Matita accepts the challenge obtaining more than satisfying results. The mechanism can be roughly split in two tasks: a local one (with respect to a formula) able to resolve the ambiguities mapping a given formula to the set of possible internal representations which are well-typed; and a non-local one in charge of rate all the resulting possible interpretations in order to guess the meaning intended by the user.

The first task is solved in Matita implementing the disambiguation algorithm originally developed by Sacerdoti Coen in [94] and presented in [97]. The algorithm exploits the refiner component of Matita to both filter out formula interpretations which are not well-typed and to automatically fill information missing in the input formula. The second task is solved implementing a notion of spatial and temporal locality during theory development with Matita [10], which prefers interpretations containing concepts "near" the formula being disambiguated, possibly permitting to the user to explicitly state her preferences on the matter.

Chapter 3 contains a brief overview of the disambiguation algorithm and a detailed discussion of the disambiguation preferences mechanism of Matita.

Patterns

We saw how direct manipulation of terms in the sequent window can be used to apply semantic actions (like application of tactics with term patterns, reductions, and so on) quickly using the mouse. We also discussed the role of scripts with respect to the long term storage encoding of concepts: in the proof as programming metaphor [3] we can think scripts as source code and proof terms as object code (though we recognize to proof terms much more relevance and reusability in our document centric view).

Hence we should find a way to store direct manipulation actions in scripts and we should do it textually, since scripts are plain text documents. In doing so we should also strive for a syntax which is directly writable by the user instead of being a "blob" non-intelligible to the human user. *Patterns* are the silver bullet which in MATITA addresses all these requirements.

Patterns are textual representations of selections. They not only can be directly typed by users in proof scripts, but are also automatically generated by the system and added to the script in response to a semantic action performed with the mouse in a direct manipulation fashion. In addition patterns can also be generated on demand by copying (as in "copy&paste") part of the markup and then pasting it as pattern in the script editing window.

The syntax of pattern ("[`in` ⟨*sequent_path*⟩] [`match` ⟨*wanted*⟩]") is composed by two parts. Their evaluation is performed in two phases following closely the syntax.

The former part (⟨*sequent_path*⟩) mocks-up a sequent discharging unwanted sub-terms with "`?`" and selecting the interesting parts with the "%" placeholder. This choice is very flexible enabling the user who is manually typing a pattern to perform coarse-grained selections of sub-terms, while permitting to be very precise in the automatic generation of patterns. It also accounts for selecting multiple sub-terms of a sequent (feature exploited for example in reduction and rewriting tactics which are often used to act on multiple sub-terms at once).

The latter part ($\langle wanted \rangle$) is meant to help the user in manually writing concise and elegant patterns. It is used to provide a term which is looked for in the subterm(s) matched by the sequent path. Only terms matching that term (actually terms which are convertible to the provided one) are finally returned as the output of the whole pattern matching process.

An example will clarify the use of patterns in MATITA.

Example 2.1 *Pattern usage in* MATITA

Consider the following sequent:

n : nat
m : nat
$H : m + n = n$
———————
$m = O$

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".

□

Step-by-Step Tacticals

We discussed how the vast majority of user interfaces for proof assistants with procedural proof languages are based on script management. In all those UIs this choice interacts badly with LCF-like tacticals.

Tacticals are the primary tool inherited from the LCF proof assistant for combining tactics together both to structure and make proof scripts more resilient to changes. Peculiar examples of tacticals are sequential composition (which chains together a sequence of tactics, applying the next tactic in the sequence to all new

goals generated by the application of the previous tactic) and branching (which permits to "branch" the evaluation flow of the script specifying which tactic has to be applied to the first new goal generated by a previous tactics application, which on the second goal, and so on).

The bad interaction among tacticals and script management manifest itself in the coarseness of tactical evaluation. For a script fully structured with nested application of the branching tactical for instance, the evaluation is "all or nothing": the execution point in the script editing window can either be placed before the beginning of the outermost branching tactical or after its end. This behavior introduces a trade-off among the benefits of using LCF-like tacticals and the nuisance inherited by the very same choice.

This trade-off is artificial and in Matita we have shown how to get rid of it for a relevant subset of LCF tacticals, replacing them with a new tactical language called *tinycals*. Using tinycals the syntax and the evaluation of branching, sequential composition, and some new constructs is sliced in small units which can be executed piecewise. For example the open square bracket ("["), often used to denote the beginning of the branching tactical) can be executed by itself, placing the execution point just after it, even without the need of having written the full body of the branching tactical.

Tinycals have been implemented by defining a new language and giving for it a small-step semantics. The semantics builds upon an abstract notion of proof status enriched with the novel notion of evaluation status, which mock-up the proof structure of an ongoing proof attempt. We believe that tinycals are generic enough to be instantiated to other provers.

Chapter 4 discusses the syntax and semantics of tinycals in detail and also presents our choices for presenting to the user all the information contained in the proof and evaluation status.

2.1.3 Standard Library

In spite of its compatibility with Coq (each Coq concept can be exported to an XML format and then referenced from native MATITA concepts), MATITA has its own standard library. That library has been developed, mostly by Prof. Andrea Asperti, to test the usability of the system and the innovative features (for example during such tests we discovered the usefulness of our naming convention in structuring a new library of formalized mathematical knowledge).

In spite of that the standard library has now become a non negligible amount of formalized mathematics. Its figures are as follows:

- about 50 grafite proof scripts, for about 10,000 lines of (grafite) code, and 400 Kbytes;
- about 1,000 theorems (mostly in elementary aspects of arithmetics up to the multiplicative property of Euler's totient Φ function, the little theorem of Fermat, and the fundamental theorem of arithmetics);
- structured in five main directories (both on disk and in the `cic:/` URI namespace): `logic` (logical connectives and quantifiers, equality), `datatypes` (basic datatypes like booleans and datatype constructors like pairs), `nat/Z/Q` (natural, integer, and complex numbers).

Efforts to port to MATITA non-trivial development previously formalized in Coq—like C-CoRN [30]—starting from the scripts rather than from the proof terms are underway in order to see the role played in proof development of our choices in the authoring interface of the system.

2.2 Internals

Several of the internal software components of theorem provers (dropping for a moment the distinction between automatic theorem provers and their interactive

counterparts) are well-documented in the scientific literature. Just to mention some of them:

- the kernels—according to the De Bruijn principle the, supposedly small, part of a theorem prover whose correctness is the only requisite for the correctness of the generated proofs—are usually direct implementations of the type checking rules of some calculus, thus they are directly documented by the huge amount of literature on the calculi themselves;

- decision procedures have a history of subject-specific workshops,[6] and the software architecture of the major players in the automatic theorem provers field has been thoroughly documented (see for example the thesis on the internals of Vampire [91]);

- similarly, user interfaces for theorem provers have seen in the last 10 years subject-specific workshops and conferences [14].

Nonetheless, the amount of code above, compared with the entirety of the code base composing a full-fledged theorem prover, is definitely not the largest slice. For example, at the time of writing MATITA amounts to about 64.9 klocs (thousands of lines of code). Of that entirety the kernel is 10.3 klocs, the graphical user interface (GUI for short) is as few as 4.3 klocs, automation is 7 klocs, and decision procedures 2.1 klocs[7] (see Figure 2.6). The remaining part—41.2 klocs—is by a huge margin the largest part of the system. We found similar proportions in the Coq code base [9].

The lack of literature on the largest part of the code required to implement a proof assistant can be partially explained by the computational logics background (as opposed to an engineering one) of the development communities behind the majority of theorem provers. Nonetheless the result is that the interactive part of

[6] for example: `http://dit.unitn.it/~rseba/pdpar06/`

[7] just for the records: this figure is highly overestimated, since at present automation is an active research direction for MATITA and that amount of code contains several different implementation of the same automation philosophy, tuned differently to experimentally test their effectiveness

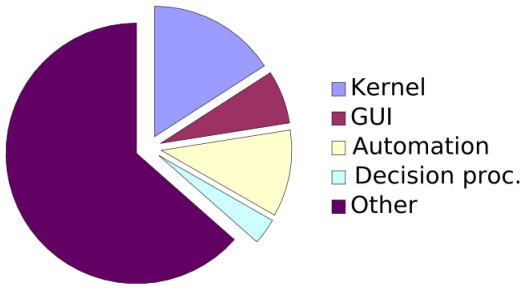

Figure 2.6: Sizes of the macro-parts of MATITA internals, with respect to what is documented in the literature

the theorem proving community lack systematic descriptions of how those systems are developed.

We think that an effort of documenting design and implementative choices is needed in order to counter the lack of an extended number of comparable players in the field. Such an effort would benefit research groups and company interested in developing new systems and, in case of systems pairwise comparable, can push toward a fruitful competition among different systems. Competition that has been absent in the field for a long time now. A similar pattern has already shown to be effective in the automatic theorem proving field, it is worth a try for the interactive counterpart.

For this reason in [9] we gave a systematic description of the internal software architecture of MATITA, also attempting a first comparison with the competing system we consider the most similar to ours and the code base of which we have the deepest understanding of: Coq. Both the description and the comparison have been made only at the design level, without entering too much in implementative details. We indeed feel that for digging more the level of details joint work of people of the development teams of different systems is needed, and we are looking forward for such collaborations.

In the remainder part of this chapter we recall the most relevant parts of the soft-

ware architecture of Matita from different angles (macro-components, their roles, and their level of coupling with the underlying logics of the system) since we will need them in the design description of our interaction widgets. A more in depth description of the implementation of the widgets—which were lacking in [9]—will be given in the chapters devoted to each widget.

2.2.1 Macro Components

As we discussed in Section 1, due to its origin, Matita is more a bundle of cooperating software components than a monolithic proof assistant developed at once from scratch. Its architecture is therefore best described as a connected graph of software components, with edges denoting relationships among them. Such a graph is shown in Figure 2.7.

Each node denotes a separate software component of Matita reporting both its name and its size in klocs.[8] The visual size of each node is linear in the corresponding component size in klocs, in order to give a rough visual feedback of the effort needed to develop the various components. The components are classified in clusters, according to the kind of term representation they act on.

Two kind of relationships among components are shown in the graph as arrows. Solid arrows denote functional dependencies among components; the intended meaning is that if a component A depends on a component B, then the code implementing B have to be compilable (in the compiler sense) before that implementing component A, since it is a prerequisite to properly compile the latter. Dashed arrows denote abstractions, meaning that if a component A (source of the dashed arrow) is abstracted over a component B (destination of the arrow), then the functionalities of A's implementation can be used by any other component implementing the API of B. In term of the actual OCaml code implementing the components this is traduced in either set of values and types which are polymorphic parametric on other

[8]The attentive reader may notice that Figure 2.7 does not explicitly contains the nodes for automation and decision procedures that have been mentioned before. Both functionalities are considered in Matita part of the "tactics" component and are therefore counted in that node

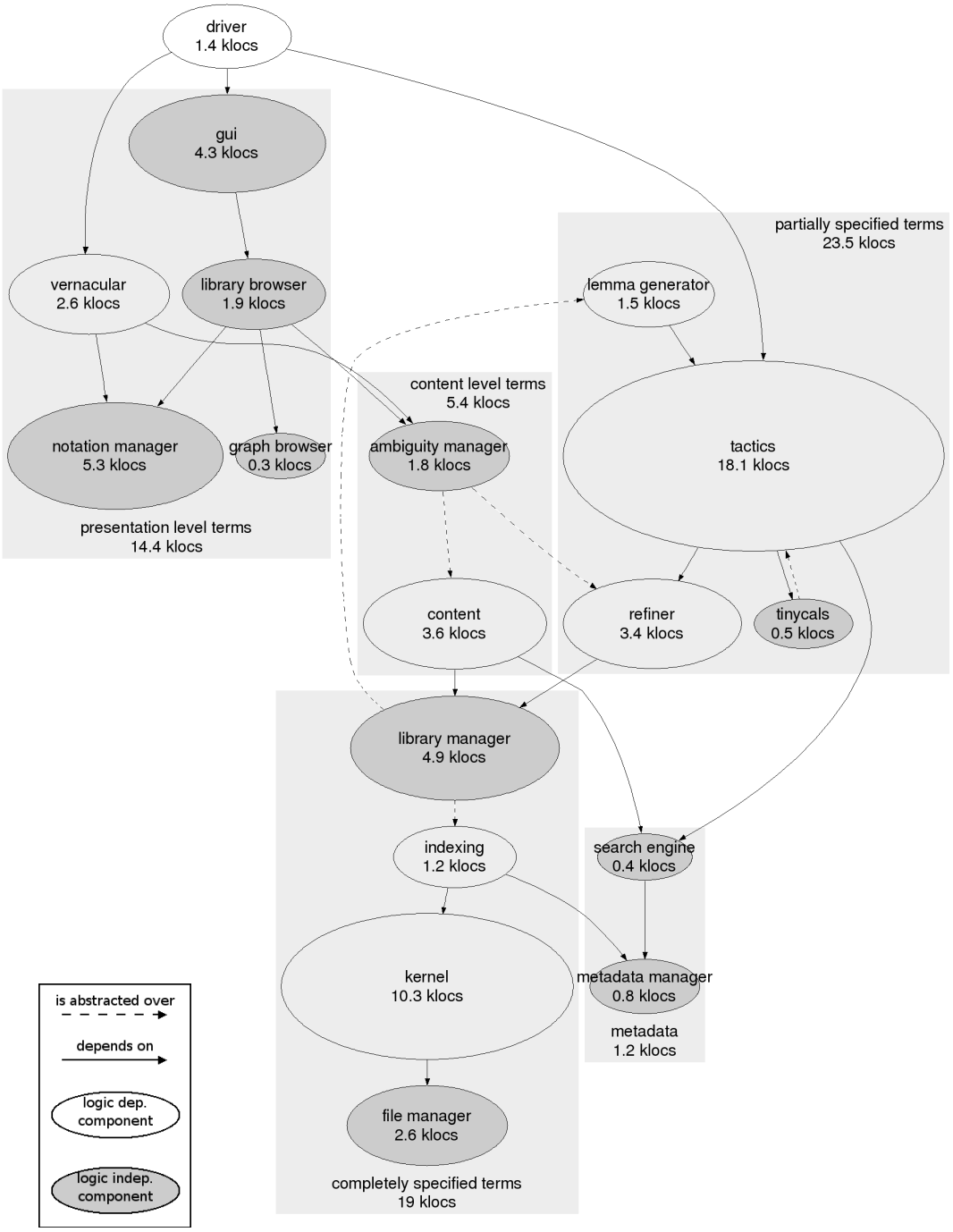

Figure 2.7: MATITA architecture: software components

values and types, or in functors whose input are modules implementing agreed upon module interfaces.

Each component is also classified according to whether it is a logic dependent or independent component: dark gray nodes denote logic independent components, while other nodes denote logic dependent components. Note how abstraction arrows are always from logic independent component to logic dependent ones. Arrows going in the opposite directions are meant to represent dependencies to logic independent components *after* they have been instantiated to a given logic.

Implementing a theorem proving system requires dealing with different representations of terms. In Matita we distinguish five such representations, discussed below, together with an high-level description of each of the components shown in Figure 2.7.

Completely specified terms In Matita, CIC is used as the calculus in which terms, and in particular proofs, are encoded for long term storage purposes. Completely specified terms are thus terms encoded in CIC. The requirement for such representation, fulfilled by CIC, is that all the information needed to formally check for correctness a given proof are represented in the encoding: no logically relevant information is left implicit.

The *kernel* is the part of a theorem prover in charge of checking for correctness proofs, in the case of Matita is a type-checker for CIC. Its size is justified by the complexity of the calculus itself. It is worth noticing that, sharing the same calculus with Coq, our kernel has been used to independently check the correctness of (a past version of) the standard library of Coq, hence providing additional assessment of the correctness of proofs formalized with such system. The converse can of course be done too.

Any system aiming at formalizing a non-trivial body of mathematical knowledge will need sooner or later persistent storage of previously formalized concept so that they can be reused across different usage sessions of the system. Also, persistent storage can be shared to foster collaborative development of libraries of formalized

mathematical knowledge. The *file manager* is the component abstracting the kernel over the persistent storage used to store objects and over the ways used to access it.

In the particular case of MATITA, whose library is meant to be a potential distributed library, such an abstraction is even more relevant than in other systems since it permits to act transparently on concepts contained in the library, no matter where they are actually located. The "`cic:/`" URI namespace we briefly mentioned above is used to reference all concepts, and the file manager take care of both resolving from URIs to URLs (Uniform Resource Locators) and to implement the needed access methods (file system access for the "`file://`" URL scheme, HTTP download for "`http://`", and so forth).

In MATITA the file manager has a name of its own—the HTTP Getter (see [125] and Section A.4)—and it is a logic independent component (this is also testified by the HTTP Getter usage in handling NuPRL documents in the HELM library).

The *indexing* component is a functional component acting on CIC terms and extracting from them the set of metadata described in Section 2.1.1. It is logical dependent since it needs knowledge of the actual logics in order to tune which information are worth to be collected as metadata. Note however that the collected metadata are logic independent and queries have been implemented acting solely on them.

The last component dealing with completely specified terms is the *library manager*. Its role is to maintain the coherence among related concepts in the library and between concepts stored in the library and their approximations as metadata. For example, every time an inductive definition is given by the user, MATITA automatically generates induction and recursion principles for the defined concept, and from these it can in turn prove automatically higher level properties like injectivity of the constructors. The library manager provides the mechanism that grants the invalidation of all those generated concepts upon invalidation of the inductive definition.

Besides, in case of invalidation of a concept every other manually generated concepts that (transitively) depend on it must also be invalidated to avoid dangling

links and logical incoherence in the library. This functionality is also provided by the library manager.

Metadata In a sense, the metadata we discussed in Section 2.1.1 are terms representations. In particular they are approximations of completely specified terms generated as output of the indexing component.

The *metadata manager* component is an abstraction over the actual storage used for metadata (a MySQL[9] relational database management system) and offers methods for storing and querying the metadata set. It is used by the indexing component to save the metadata for permanent storage and by the search engine for answering to queries.

The *search engine* implements the high-level content-based queries of the WHELP search-engine (see Section 2.1.1) mapping high-level queries to low-level queries for the metadata manager, possibly performing pre/post-processing where needed. Both the metadata manager and the search engine are logic independent components which only need a starting metadata set to be used properly.

Partially specified terms Completely specified terms are an highly verbose representation of terms given that, per definition, all information is represented explicitly. During some inner working on terms however, some information are either able to be inferred or simply not yet available. For example ongoing proofs contains "holes" which are transformed and rewritten into terms with further holes until the proof is completed, when they become entitled to be represented as fully specified terms; tactics need to be able to act on such pierced terms. Similarly, terms coming from a textual input of the user might be missing type or other kind of annotations which in some cases can be automatically inferred by the system.

Partially specified terms is a term representation which allows the omission of sub-terms by the means of untyped linear placeholders or with typed metavariables in the spirit of [40, 73]. Recasting conversion to unification in the typing rules

[9] `http://www.mysql.com`

the missing information can often be reconstructed automatically [101], obtaining a type inference system. The *refiner* component [94] implements a type inference procedure, also able to insert implicit coercions [16, 63] to fix, when possible, local type-checking errors. The dependency from the refiner to the library manager is motivated by the fact that the latter component is also in charge of the management of coercions (namely, to keep track of which concepts available in the library are flagged to be used as coercions).

The *tactics* component contains the implementation of all the tactics currently available in Matita, including automation and domain-specific decision procedures. The dependency on the search engine is induced by the automation tactics which query the library to discover applicable theorems to solve a given goal (using the hint query) or to collect a corpus of concepts of a particular kind (e.g.: equalities) needed as input of some proof search technique (e.g.: paramodulation [77], currently being worked on in Matita).

The *tinycals* engine is the heart of our small-step language of tacticals, and is a generic component abstracted over a proof status representation (contained in the tactics component). The reverse dependency from tactics to tinycals is due to the reuse of the tinycals engine for implementing the LCF-like tacticals which have counterparts in tinycals (see Section 4.4.2).

The *lemma generator* component is in charge for the automatic generation of lemmata, which is triggered by the insertion of new concepts in the library (like induction principles, as we have seen previously). The lemmata are generated automatically computing their statements and then proving them by means of tactics or by direct construction of the proof terms.

As completely specified terms, partially specified terms are not well suited for user consumption since their syntax is not extensible and it is not possible to adopt the usual mathematical notation. However they are already an improvement over completely specified terms since they allow to omit redundant information that can be inferred by the refiner. On the contrary, they are not suitable for interoperability with other applications either, unless for applications sharing the same logical

foundation of Matita.

Content level terms Content level terms are a step forward towards a term representation suited for user consumption and the representation best suited for communication with other applications. They are conceptually the content level of [1] and Section 5.1.2. Terms represented this way encode the syntactic structure (or the abstract syntax tree if you prefer) of the human-oriented representation of formulae and proofs. Those abstract syntax trees are more compact than previously presented representations and share characteristics like overloading and missing information with the standard mathematical notation.

The *content* component defines the concrete data types used to represent formulae and proof at the content level and also contains the translations from partially specified terms to content level and vice versa. The former translation is lossy and must discriminate between terms used to represent proofs and terms used to represent formulae. Using techniques inspired by [27, 28], proofs are translated to a content level representation of proof steps that can in turn easily be rendered in natural language.

The specific representation we adopted in Matita for proofs has greatly influenced the OMDoc [80] proof format which is now isomorphic to it. On the other hand the representation used for formulae has been inspired by MathML Content [66]. The content component inherit its logic dependence from the fact it has to deals, as input of translations, with partially specified terms.

The reverse translation—from content level terms to partially specified terms—for terms representing formulae is called disambiguation and requires discriminating between the several possible partially specified terms which instantiate information missing at the content level and resolve overloading. Disambiguation is implemented by the *ambiguity manager* component and will be the subject of Chapter 3.

The corresponding translation from proofs at the content level to partially specified terms is being implemented as a set of tactics and reduces to the task of implementing a declarative proof language on top of a procedural one. It is a work

in progress at the time of writing being conducted on the steps of previous work in this direction [46, 116].

The translations involving formulae are described in more details in Chapter 5.

Presentation level terms The last representation used in MATITA—presentation level terms—is meant for user consumption, for both the purposes of input (the user typing a formula or a proof in the script editing window) and output (the system delivering a proof or a formula to the user via a graphical widget). Conceptually term at this level are the presentation level of [1] and Section 5.1.2.

Presentation level terms are elided of any logical information and encode the visual aspect of formulae and proof. At this level the layout schemata of formulae (like fractions, radicals, subscripts, ...) are expressed, as well as line-breaking hints used to re-arrange proofs and formulae when they does not fit on a single physical line on the screen.

It is worth noticing that, on the contrary of what happens at the content level, the presentation level language is fixed (i.e. non-extensible) mimicking the notational language of usual informal mathematics. For this reasons standards do exist for encode terms at this level and we adopted one of them for the final output of formulae: MathML Presentation [66]. Similarly, for the purpose of input we adopted a markup close to TEX [58], a language very popular among mathematicians. Lacking MathML Presentation support for line-breaking hints, but being well-known in the literature the basic mechanism behind them (see for example [34]), we defined our own dialect for line-breaking hints: BoxML.

The *notation manager* component is similar to the content component we saw for partially specified term, since it defines the datatypes representing in OCaml the presentation level language and the translations from content level terms to presentation level terms and vice versa. In addition to that, and hence the name, the notation manager implements the support for extensible notation in MATITA (see Chapter 5) and defines the language used by the user to incrementally change the notation used to perform input and output of terms.

The other components of the presentation level terms cluster compose together the GTK+-based authoring interface of MATITA:

The *library browser* mainly consists of the CIC browser we discussed in Section 2.1.

The *graph browser* is a widget based on Graphviz[10] able to render dependency graphs in a GTK+ window, and implementing on top of them hypertextual capabilities like having graph nodes and edges as hyperlink anchors.

The graph browser is used in MATITA to explore the dependency graphs of concepts (see Figure 2.4), but also to browse topologically the set of scripts which compose the development of a theory (see Figure 2.8 for an example) and to inspect the DAG formed by the declared coercions;

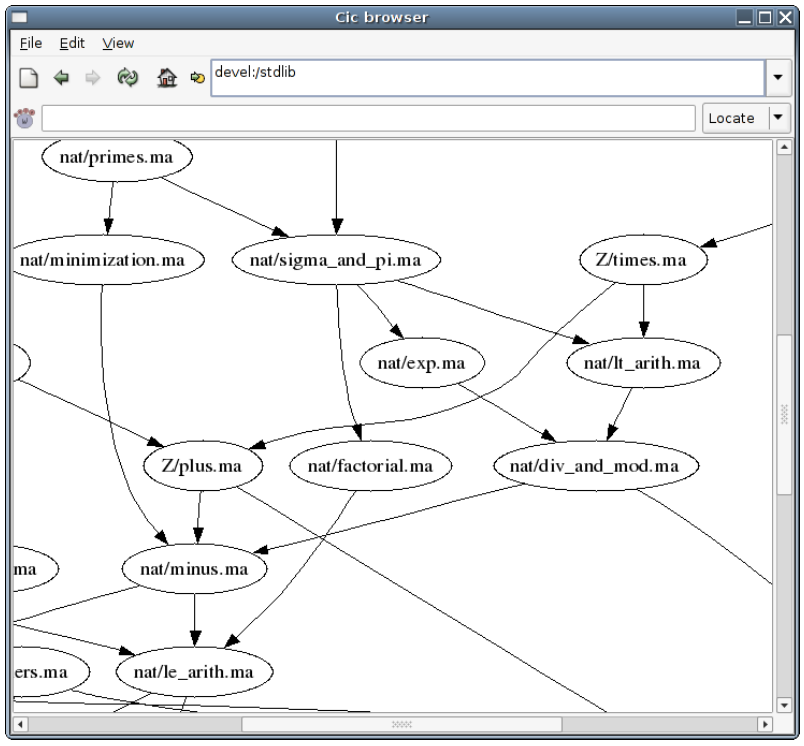

Figure 2.8: Dependencies between the scripts of a development

[10] `http://www.graphviz.org/`

The *GUI* is the graphical user interface of MATITA, which implements proof management on the lines of Proof General and orchestrates the other visual components.

The *vernacular* is the textual parser of the commands the user can directly type in the script editing window and acts as a proxy among the script and the corresponding end points in the other visual components of the authoring interface.

To conclude the description of the components of MATITA, the *driver* component, that does not act directly on terms, is responsible for pulling together the other components, for instance to parse a command (using the vernacular component) and then trigger its execution (for instance calling the tactics component if the command is a tactic).

2.3 Component Reusability

We conclude this section looking at the component of MATITA from a different angle: their potential reusability in the implementation of a different proof assistant.

One of our long-term aims with MATITA is to foster a reuse culture in the development of proof assistants, following the path of what Proof General did for their user interfaces, but looking at the more substantial parts of the system. Instantiating Proof General to a new proof system is simple: it is enough to add support for talking the Proof General Interaction Protocol[11] (PGIP)—a textual and fairly easy to implement protocol—and then the system can be used through Proof General in a running Emacs[12] instance.

The key of the success of Proof General was that the user interface of a proof assistant is essentially logic (and system) independent and as such it can be factorized and reused by others. In our vision of theorem proving development however,

[11] `http://proofgeneral.inf.ed.ac.uk/wiki/Main/PGIP`

[12] `http://www.gnu.org/software/emacs/`

there is an additional layer of "interface", a layer of logic related services. Those services are often needed in the implementation of target systems, but their need of logic knowledge can be abstracted over and defined via a set of APIs that can be implemented by different systems.

On the bright side those services are a wider slice of the code that need to be written to implement a target proof assistant than the mere user interface (the GUI is about 6% of the total code of Matita, see Figure 2.6) and thus would enable a larger degree of reusability. On the dark side though, the implemented services are much more intertwined with the target proof assistant than the GUI, so that implementing them as separate processes (a precise design choice of [107]) is not an option. This reduces the potential reusability to only those system that will be implemented in the same programming language of ours (ignoring on purpose cross-language reuse hacks).

During the development of Matita we have tried to identify the components that can be reused in other systems, no matter their logics. Those components are shown in Figure 2.7 as dark gray components.

Figure 2.9 focuses on their API, dividing the functionalities of a proof assistant in five categories:

1. visual interaction and browsing of the mathematical library (*GUI* column);
2. input/output of formulae and proofs (*I/O* column);
3. indexing and searching of library concepts (*search* column);
4. management of the concepts library (*library* column);
5. interactive development of proofs by means of proof scripts (*proof authoring* column).

For each category, the lower part of the column shows the reusable components of Matita and their mutual dependencies, while the upper part shows the logic dependent components the programmer needs to implement to instantiate the generic components to a given logics.

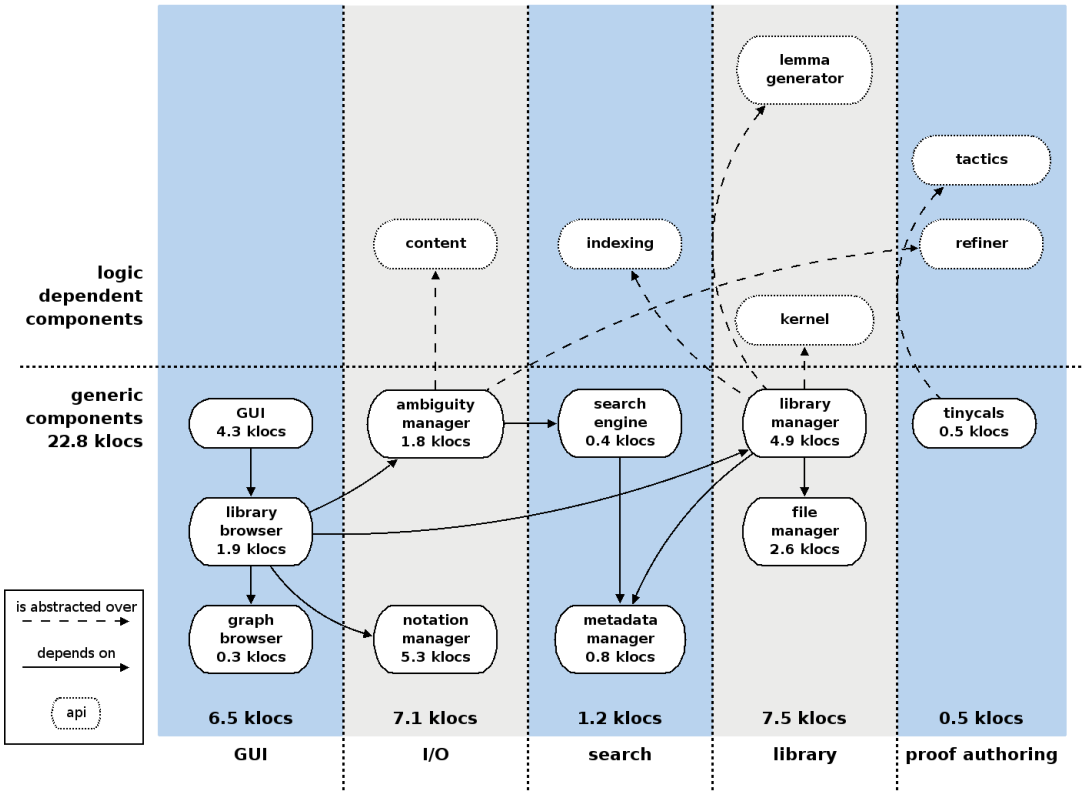

Figure 2.9: MATITA architecture: API of the reusable components

In the implementation of a different target system, not necessarily all API components need to be instantiated. If a given functionality is not desired in the target system (e.g. searching), than the API required to implement it (e.g. indexing) need not to be implemented. As another example, if the target system has a logic which does not admit metavariables, then the refinement is pointless and can be substituted for the type-checker, and unification can simply invoke the reduction implemented in the kernel (if any, or alternatively the identity function).

In Figure 2.10 we shown the actual components implemented in MATITA to fulfill the API the generic components depend on. The arrows show the dependencies among the logic dependent components and from the logic dependent components to instances of the generic components. The figure also compares the size of the logic dependent and logic independent code required to implement each functionality. To

implement Matita from scratch reusing the logic independent components 2/3 of the original code need to be rewritten.

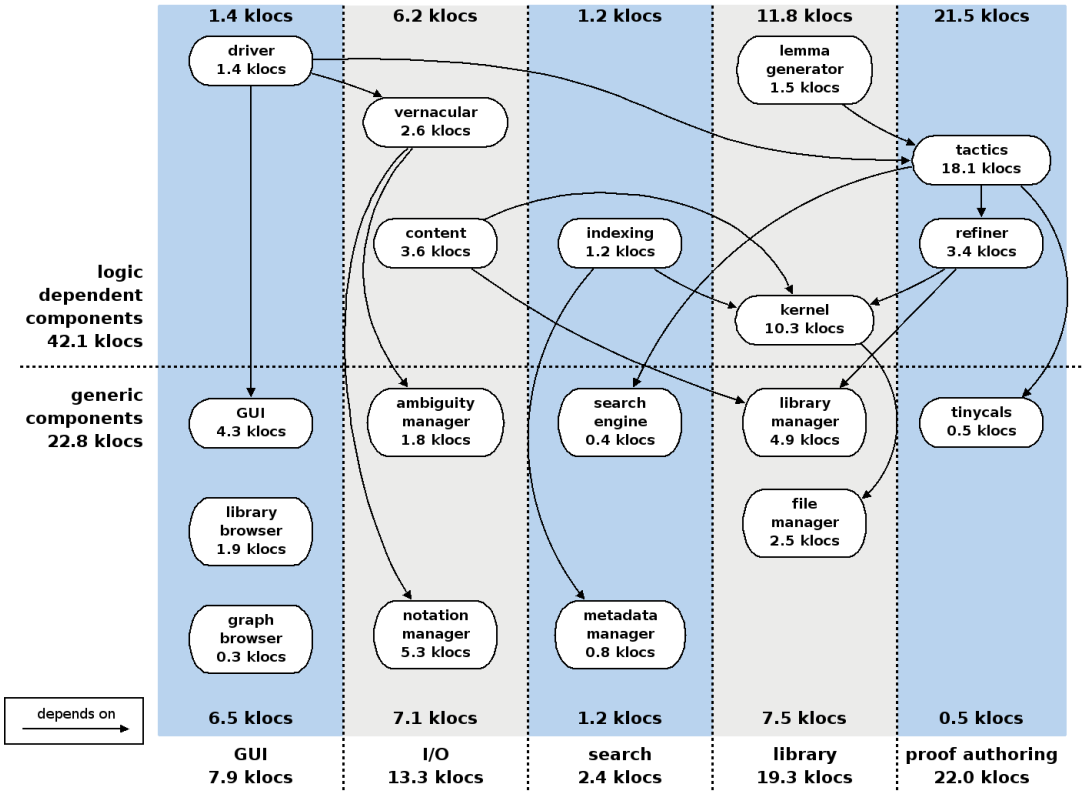

Figure 2.10: Matita architecture: API implementation for CIC

Although the amount of reusable code is quite limited (still being more than 22,000 lines of code), the advantages of decoupling the functionalities should not be neglected. On one hand the skills required to develop the reusable components that deal with extra-logical functionalities are really different from those required to implement the kernel of the system or the decision procedures. In the history of Matita this aspect really mattered, given that sub-task in the development of the various components where assigned to master students sometime coming from logic-related courses whereas some other to students completely lacking a logic background.

Moreover, we expect systems built on top of a collection of reusable components

like those presented to obtain more quickly some of the functionalities required for early testing, speeding up the development cycle in the spirit of rapid prototyping. In [9] we assessed this point presenting a Gantt diagram for the development of the system showing how inner milestones can be reached quickly.

2.4 Conclusions

In this chapter we gave an overview of the Matita proof assistant, both from the point of view of the users of the system and from that of the developers. We highlighted the peculiarities of this new interactive theorem prover with respect to the state-of-the art and laid down the terrain for a systematic discussion of its macro component. In the next chapters we will move to four of the user interaction widgets of Matita, subject of this thesis.

Original Contributions

The design and implementation of Matita is definitely not to be credited as a whole as work of this thesis author. As discussed in Section 1, most part of the work behind Matita is an inheritance of past research efforts of which this thesis author is just a contributor.

Nonetheless he is an active developer of the system and the one who started assembling a proof assistant with a script-based user interface, on top of the available components, replacing the former gTopLevel prototype (discussed in Part II of [94]). The major original contributions of this thesis author to the system are the widgets discussed in Part II of this thesis and the additional software components briefly presented in Appendix A.

Related Publications

Part of the work described in this chapter has been previously published in the following papers:

- Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. *User Interaction with the* MATITA *Proof Assistant* [10].
  To appear in Journal of Automated Reasoning[13], special issue on User Interfaces for Theorem Proving[14].

- Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. *Crafting a Proof Assistant* [9].
  Accepted for publication in Proceedings of the Types 2006 International Conference[15].

[13] http://www.springerlink.com/link.asp?id=100280
[14] http://www.informatik.uni-bremen.de/~cxl/uitp-jar/
[15] http://www.cs.nott.ac.uk/types06/

Part II

Widgets

Chapter 3

Disambiguation of Formulae in Interactive Theorem Proving

Mathematical formulae frequently occur in the authoring interface of a proof assistant like Matita. When the need for the user it to input them, a handy way to do so is to textually type them using a syntax (e.g. the usual mathematical notation) and an encoding (e.g. TEX) the user is familiar with.

Such a handy input mechanism is challenging to implement though, mainly due to the ambiguity of mathematical notation, ambiguity which mathematicians are used to solve exploiting a broad notion of context and their ability to spot incompatible interpretations of—mainly—operators.

The task of retrieving the most suitable logical meaning of a formula typed by the user is the subject of this chapter. We will discuss a decomposition of the problem in two sub-problems (efficient resolution of ambiguities and representation ratings) and how we addressed both of them in Matita, obtaining a widget that can be instantiated to other systems that need to address similar issues.

3.1 Rationale

Most of the activities involved with theory development [2] require the user to provide a term of some calculus to the system, just to mention some of them: asserting definition and axioms (since both types and bodies are terms), formulating the goal

(usually achieved by stating the type of a proposition, i.e. a term), applying tactics (which often require terms arguments), and so on.

All terms occurring in the above activities are internally used by the system to represent formulae, hence from the point of view of the interaction designer the first question is how can the user input such formulae to the system. In the interaction paradigm set forth by CtCoq [20] and Proof General [12] (namely script management), textual typing is the input mechanism of choice and as such formulae are better textually typed to avoid frequent context switches between different mechanisms. For the purpose of this chapter we will then ignore other formula input mechanisms like palette-based editing.

The next question is then which (concrete) input syntax should be chosen for letting the user type formulae inside proof scripts. The most natural choice is to stick to what mathematicians are used to. This means letting formulae to be written using the usual mathematical notation linearized in some widespread textual encoding.

With "usual mathematical notation" we mean the fixed bi-dimensional language made of a set of literals (like identifiers, numbers, and symbols) and a set of layout schemata (like superscripts, subscripts, matrices, radicals, ...) used to write formulae on paper and encoded by various markup languages in the computer era (like MathML Presentation [66]). A wide-spread textual encoding among mathematicians and more general among scientists used to write formulae is TEX [58] which implements a mono-dimensional textual syntax—linearizing the bi-dimensional notational language of mathematicians—explicitly designed for the purpose of being quick and easy to write.

While choosing a TEX-like encoding poses no particular troubles, the choice of usual mathematical notation for input of formulae is challenging. The reason for that is the habit of using ambiguous notational conventions. Let's consider a couple of examples.

Example 3.1 *Ambiguity solved using some* context

Sometimes mathematicians reading a formula solve the notational ambiguities exploiting a notion of *context*. For example, if g is known to be a scalar value, the

formula g^{-1} is likely to be interpreted as the inverse value of g: $\frac{1}{g}$. On the other hand, if g is a function, then g^{-1} is probably the inverse function of g, mapping element of g's codomain to g's domain.

□

Example 3.2 *Ambiguity solved by ... "folklore"*

More often the context is not enough and mathematicians solves notational ambiguities by folklore, sticking to the habits they learned in their studies. Without additional information a mathematician is like to interpret $\phi^2(x)$ as the composition of ϕ with itself applied to x: $(\phi \circ \phi)(x)$, while $\sin^2(x)$ is likely to be interpreted as $(\sin x)^2$, just because this is a well-established convention for sin.

□

Programming languages faced similar disambiguation issues, usually for the arithmetic fragment of formulae only. Their solution usually consists in dropping the freedom of the mathematical notation, imposing to the user a language which is modelled to achieve a smaller degree of ambiguity, provided that the ambiguity can always be solved obtaining a unique interpretation, usually in a non-interactive fashion. For instance, overloading in C++ [102] does not allow to declare two functions that take the same input types, but returns output with different types.

Since we do not want to drift too much from the usual mathematical notation we need to drop this design technique, historically adopted by programming languages.

In the applicative field of interactive theorem provers, and more generally of all mathematical knowledge management applications, sticking to the usual mathematical notation with the ability to disambiguate it also have added benefits. We briefly mention some of them:

- different mathematicians may use different notations for expressing the same concept, we do not want to give up this possibility for barely technical reasons. In the MATITA setting, where the library of the system is a possibly distributed library of formalized mathematics, the notion of context is not always obvious. For example, when the user exploits the searching capabilities of WHELP [7], the search engine should be able to parse whatever notation in the library

has been used for a single concept and then perform the search modulo the notation;

- during formalization activities, notation is user-defined and evolving. When parts of a shared library are developed independently, sooner or later it will happen that the two parts need to be used together, opening the flank to notational collisions. At the very minimum, overloading of operators should be supported permitting the merge of different interpretations for overloaded operators;

- mathematical knowledge management is also about digitization and enhancement of pre-existing mathematical knowledge. Since the notation used in such knowledge is ambiguous, ambiguity has to be addressed in both phases. Techniques aiming at merging digitization and enhancement phases with interactive theorem proving [106, 115] can only benefit from disambiguation techniques for ordinary formulae.

Not allowing ourselves to change mathematical notation too much, we must face the fact that a single textually typed formulae have several possible internal *representations* in the system the user typed them in. In the particular case of MATITA the representations are terms in the calculus of the system: the Calculus of Inductive Constructions (CIC). Roughly speaking, our objective is "automatic detection of the intended interpretation of a formula typed by the user, among the ones that make sense".

More precisely, our challenge—which we call *disambiguation*—can be stated as: starting from the concrete syntax of a formula, we want to build an internal representation of it among the ones that "make sense".

The latter notion—"making sense"—should be interpreted as being well typed in some weak type systems which have been proved suitable for expressing the constraints mathematicians use in writing formulae (for example weak type theory [55] or the Mizar type system [15]). In MATITA the role of "weak" type system is easily fulfilled by CIC itself.

The disambiguation task can be split into two separate tasks:

Ambiguity resolution given as input a formula as typed by the user, build the set of all well-typed internal representations of it.

Ambiguity resolution can be further split in the two well-known task from the compiler theory of parsing and semantic analysis, and is affected by serious efficiency concerns. In non-trivial formalization of arithmetics and calculus for example the number of potential representations that need to be type-checked can easily reach the thousands of elements, a challenging figure for a type-checker in a calculus as complex as CIC.

Representation rating rate the possible internal representations of a given formula, so that the higher is the rating, the higher is the likelihood of the corresponding representations to have the meaning intended by the user.

Representation rating can exploit the particular interactive setting in which we found ourselves, asking for user intervention in case of equal rating of different representations.

Claudio Sacerdoti Coen proposed in his Ph.D. work an efficient algorithm for addressing ambiguity resolution [94]. The algorithm was then refined and re-implemented in MATITA and presented by this thesis author and him in [97]. Representation rating is novel and has been introduced in MATITA exploiting temporal and spatial locality as a natural encoding of the notion of context sometimes exploited by mathematicians to solve ambiguities.

In the next section we will present the efficient disambiguation algorithm used in MATITA. Section 3.3 will be devoted to representation rating.

3.2 Efficient Resolution of Ambiguities

Let us start with an example which can be encountered in everyday work with a proof assistant.

Example 3.3 *Representations of a simple algebraic formula*

Consider the formula:

$$\forall x.\ x * (1 + 2) * 3^{-1} = x$$

Such a formula in MATITA (depending on the sort of the type assigned to x) can stand for a logical proposition, suitable to be used as the statement of a theorem, and can also be encountered as intermediate goal during an interactive proof.

Several possible internal (CIC) representations can be built for it, depending on the interpretation chosen for the several sources of ambiguity the formula presents. Let's consider the standard library of Coq (just because it is more developed than that of MATITA and because it is often use as the legacy library to develop new concepts in MATITA), and look at the possible meaning of each ambiguity source:

1. each literal number can be a natural, integer, rational, real, or complex number;
2. "-1" can either be the application of the unary operator "$-$" to 1, or the integer, rational, real or, complex number "-1";
3. "$=$" can represent various kinds of equality: Leibniz's equality, the (negated) apartness relation over real numbers, a decidable equality over natural numbers, an equivalence relation over natural numbers, ...
4. "$+$" and "$*$" can be binary operators defined over naturals, integers, ...;
5. implicit coercions can be inserted to inject a number of one type in a "super type" (e.g. coercion to build real numbers from natural numbers);
6. since different representations of a concept have different computational properties, several representations of, say, natural numbers may be available (in the Coq library for example both unary a binary encoding are given).

Note that the "x" identifier is not a source of ambiguity, since it is a variable bound by the "$\forall$" quantifier.

□

The number of possible representations of the formulae of Example 3.3 grows with the product of the number of choices for each ambiguity source. Note however that not all possible choices are valid: for example, in the formula above if "2" has to be chosen as the unary representation of the Peano number "S (S O)", then the choice for the plus operator should be able to accept such a number as its second argument (or alternatively an argument to which such a number is coercible).

3.2.1 Ambiguity Sources

Our disambiguation algorithm has been designed to support three *sources of ambiguities*, in our experience the most common in interactive proof development:

Literal numbers as shown in the example different encodings of numbers are used in formalization of mathematics, but a frequent user's desire is to write them down literally in decimal notation.

Unbound identifiers not shown in the example, identifiers which are not bound to some binder (e.g. "`\forall`" in the example) are frequently used to refer to concepts defined somewhere in the library. In MATITA the peculiar use of such a feature is to refer to concepts using their short names, to avoid the need of writing verbose URIs in proof scripts. For example the short name *plus_assoc* is shared in the HELM library by three theorems stating the associativity of plus operators defined on different domains.

Symbols usually used as prefix or infix operators forming the syntax of some concept, symbols are often highly overloaded and may be used to hide so called "implicit arguments" which can be automatically inferred by the system. For example, one of the meaning of the notation "$a = b$" in the MATITA standard library is defined as follows:

```
interpretation "leibnitz's equality" 'eq x y =
  (cic:/matita/logic/equality/eq.ind#xpointer(1/1) _ x y).
```

stating that the symbol `'eq` can be interpreted as the application of Leibniz's equality (which can be defined as an inductive data type with one constructor in CIC) to an implicit argument (the underscore) and the two other arguments it was applied to. The actual syntax of the notation is not defined by the script snippet above, because it is rather defined as a notational equation (see Chapter 5).

Intuitively, each source of ambiguity is an agent which brings uncertainty during the mapping from the textual representation of a formula to its encoding in CIC. As discussed in Chapter 5, this uncertainty in MATITA is currently permitted only after parsing, while mapping the (sole) abstract syntax tree (AST) returned by the parser to CIC terms. Hence, the sources of ambiguities have to be thought as nodes of the abstract syntax tree (we indeed have nodes for number literals, node for unbound identifiers, and node for symbols).

While practically this hinder the possibility of conflicting notations when they would imply conflicting parsing rules, this is not a flaw in the proposed approach, but rather in our technological choice of using Camlp4, a non-ambiguous extensible parser. Having a technological tool which enable ambiguous parsing returning more than one abstract syntax tree as output, the approach we are proposing can seamlessly be applied handling ambiguities at both the parsing and semantic analysis phases. It must also be observed that, having both this possibilities, the road of handling the ambiguity solely after a parse tree has been generated has to be preferred where possible, given that uniformity of the content level is considered by the MKM community to be the proper way of achieving interoperability between applications not sharing semantic foundations.

Each source of ambiguity is associated to a given *disambiguation codomain*, a set of functions returning CIC terms. The actual arguments of the functions depend on the kind of source of ambiguity: literal numbers have functions from the natural number interpretation of the literal to CIC terms; unbound identifiers have 0-arguments functions (i.e. they are constants); symbols have functions which takes as input the CIC representations of their arguments.

The disambiguation codomains for numbers and symbols in MATITA is (incrementally) defined by the user. The script snipped previously shown is just a statement to the system that an additional point in the codomain of the equality symbol should be added. On the other hand, the disambiguation codomain for unbound identifiers is automatically inferred by MATITA, which during indexing extracts the short names of defined concepts like concept names, but also constructor names, record projections and so on.

3.2.2 The Algorithm

A *disambiguation algorithm* takes as input an *ambiguous term*—that is a term with at least one source of ambiguity—and return as output a (possibly empty) set of well-typed CIC terms.

The *naive disambiguation algorithm* (NDA for short) is a first example of a disambiguation algorithm. It takes as input an ambiguous term t and proceeds as follows:

1. Create interpretation domains $\{D_i | i \in Sig(t)\}$, where $Sig(t)$ is the set of ambiguity sources of t. Each D_i is a set of CIC terms and can be built applying in turn the functions composing the disambiguation codomain of i.

2. Let $\Phi = \{\phi_i | i \in Sig(t), \phi_i \in D_i\}$ be an *interpretation* for t. Given t and an interpretation Φ, a CIC term is fully determined. Iterate over all possible interpretations of t and type-check them, keep only well-typed interpretations (i.e. interpretations that determine well-typed terms).

3. Let n be the number of interpretations who survived step 2.

   - If $n = 0$ signal a type error.
   - If $n > 0$ returns all the CIC terms determined by the survived interpretations as output of the disambiguation algorithm.

The naive disambiguation algorithm is highly inefficient since the number of possible interpretations Φ grows with the product of the codomain sizes of ambiguity sources occurring in the formula to be disambiguated. The efficient algorithm used in MATITA (and also in the WHELP search engine, for what is worth) is far more efficient being, in the average case, linear in the number of ambiguity sources. It has been presented extensively in [97] and for this reason will not be discussed here, we will just recall the intuition behind its efficiency.

The efficient algorithm can be applied if the logic can be extended with metavariables and a refiner can be implemented. This is the case for CIC and several other logics. *Metavariables* [73] are typed, non linear placeholders that can occur in terms; the "$?_i$" notation usually denotes the i-th metavariable, while "?" denotes a freshly created metavariable. A *refiner* [67] is a function whose input is a term with placeholders and whose output is either a new term obtained instantiating some placeholder or ϵ, meaning that no well typed instantiation could be found for the placeholders occurring in the term (type error). The refiner currently used in MATITA is the refiner described in [94], it is the same type inference component used by tactics.

The efficient algorithm starts with an interpretation $\Phi_0 = \{\phi_i | \phi_i =?, i \in Sig(t)\}$, which associates a fresh metavariable to each source of ambiguity. Then it iterates refining the current CIC term (i.e. the term obtained interpreting t with Φ_i). If the refinement succeeds the next interpretation Φ_{i+1} will be created *making a choice*, that is replacing a placeholder with one of the possible choice from the corresponding disambiguation domain.

The placeholder to be replaced is chosen following a pre-order visit of the ambiguous term. If the refinement fails the current set of choices cannot lead to a well-typed term and backtracking is attempted. Once an unambiguous correct interpretation is found (i.e. Φ_i does no longer contain any placeholder), backtracking is attempted anyway to find the other correct interpretations.

The intuition which explains why this algorithm is more efficient is that as soon as the refinement of a term containing placeholders fails, no further instantiation of its placeholders could lead to a term whose refinement suceeds. For example, during

the disambiguation of user input `\forall x. x*0 = 0`, an interpretation Φ_i is encountered which associates ? to the instance of `0` on the right, the real number 0 to the instance of `0` on the left, and the multiplication over natural numbers to `*`. The refiner will fail, since the multiplication requires a natural argument, and no further instantiation of the placeholder will be tried.

The presentation of the disambiguation algorithm given in [97] is alternative to the above one, not requiring backtracking.

3.2.3 Efficiency

The efficiency of the proposed disambiguation algorithm has been assessed performing batch disambiguation over the fragment of the Coq standard library which deals with real numbers. The choice have been motivated by the non-trivial degree of overloading which is obtained in the formalization of real analysis and calculus.

The strategy of pre-order visiting the abstract syntax tree affects performances. The strategy is motivated by the observation that the type of a function constraints the type and the number of its arguments, enabling to prune at each step of the algorithm several partial interpretations at once. In spite of the possibility of not being always optimal, it performs particularly well in our benchmarks exhibiting on the average performances close to the optimal case of the algorithm.

Figure 3.1 compares the number of refinements performed by the naive disambiguation algorithms with those performed by the efficient algorithm we proposed. The latter performs more refinements than NDA when the number of source of ambiguities is small (and hence the number of total refinements is), but quickly get worse than the proposed algorithm, with peeks of 50,000 refinement attempts to be compared with less than 70 attempts of the proposed algorithm.

3.3 Representation Rating

We now tackle the second task of disambiguation: representation rating. In the big picture of the disambiguation process, it will be used in a pipeline which first invokes

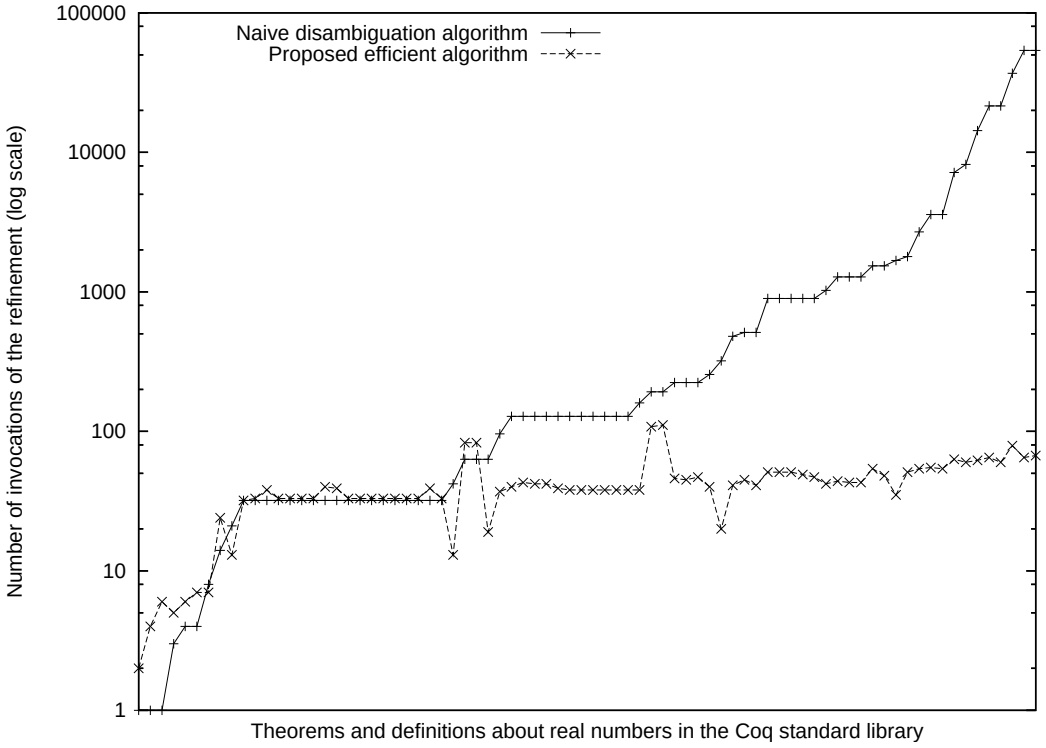

Figure 3.1: Disambiguation algorithms comparison

the efficient disambiguation algorithm presented in the previous section and then rates all the returned representations (CIC terms) using the approach presented in this section.

In order to characterize our approach for representation rating we will need to be a bit more formal about the concepts we are discussing.

Definition 3.1 (Interpretation domain) *Given an ambiguity source s (a literal number, an unbound identifier, or a symbol as per Section 3.2.1), its* interpretation domain $\mathcal{D}_s$ *is a set of CIC terms obtained applying the functions composing the disambiguation codomain of s.*

Intuitively an interpretation domain is a set of choices for an ambiguity source that a possible meaning for a given ambiguous formula can choose from.

Definition 3.2 (Signature of an ambiguous term) *Given an ambiguous term t (i.e. an abstract syntax tree in which a positive number of sources of ambiguity do occur), its* signature Σ_t *is the set of sources of ambiguity occurring in it.*

Signatures are meant to represent the set of ambiguity "issues" that need to be resolved before a single, fully-determined CIC term can be obtained from it.

Definition 3.3 (Interpretation) *An* interpretation *for an ambiguity source occurring in an ambiguous term t is a mapping σ associating to each $s \in \Sigma_t$ a choice $\sigma(s) \in \mathcal{D}_s$.*

Similarly, an interpretation for an ambiguous term t is a CIC term obtained substituting $\forall s \in \Sigma_t$ the interpretation of s, that is $\sigma(s)$.

In other words an interpretation is a set of choices which solves all the ambiguity issues for a given ambiguous term. Intuitively it can be applied to an ambiguous term obtaining a resulting CIC term.

We will give various classification of interpretations, the first one is about its validity with respect to the typing judgement of the CIC term it induces. The validity of an interpretation is meant to encode the "make sense" notion we have formerly used in this chapter.

Definition 3.4 (Valid interpretation) *An interpretation σ for an ambiguous term t is* valid *if and only if $\sigma(t)$ is a well-typed CIC term.*

It is worth noticing that other notion of validity can be explored, like for example imposing constraints on the type of the term which is being disambiguated if, for example, it is known that it should be a theorem statement. This choice of ours however is generic enough to be re-casted to different type systems when the disambiguation mechanism is ported to different logical foundations.

The rating problem we are addressing can now be described as follows:

> Let $\mathcal{V}_t$ be the set of valid interpretations for an ambiguous term t. Find a total ordering $\leq$ of all elements of $\mathcal{V}_t$, so that the lower an interpretation is in the ordering, the higher is the likelihood of the CIC term induced by it to be the term "intended" by the user.

Intuitively the total ordering we are looking for is a ranking, and the minimum element in the ordering is the interpretation who "won" over the others, representing the interpretation guessed by the system. Ties are admitted and solved asking interactively the user the meaning she intended. When the guess of the system is wrong, the ranking is still useful and can be exploited to present an ordered list of interpretation to the user from which she can choose from.

We will first see some criteria which can be used to rate the interpretations of sources of ambiguity and then how they can be composed together to rate interpretation of ambiguous terms.

Locality of reference appears to be a good first attempt to rate interpretations of a source of ambiguity: 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 real analysis it is common practice to mix in the same theorem 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. The preferences implicitly added by the system implement the criterion of locality of reference, but they can be overridden by the user when there is the need to do so.

Definition 3.5 (Disambiguation preferences) *A set of* disambiguation preferences *for an ambiguity source s is a pair $\langle c_s, \mathcal{P}_s \rangle$ where is a subset $\mathcal{P}_s \subseteq \mathcal{D}_s$ and c_s is an optional element $c_s \in \mathcal{P}_s$ called* current preference.

We will write `None` *for* c_s *when no current preference has been chosen on a set of disambiguation preferences as opposed as* `Some` f *when* f *has been chosen as the current preference.*

The starting status for each ambiguity source is $\langle \texttt{None}, \emptyset \rangle$. The only operation currently allowed on a set of preferences is the statement of a preference for f for a source of ambiguity s, defined as follows:

$$set_{s,f}\langle c, \mathcal{P}_s \rangle \rightarrow \langle \texttt{Some } f, \mathcal{P}_s \cup \{f\} \rangle$$

We can now refine our classification of interpretations for sources of ambiguity.

Definition 3.6 (Current interpretation) *Let* t *be an ambiguous term,* σ *an interpretation for it and* $\langle c_s, \mathcal{P}_s \rangle,\ \forall s \in \Sigma_t$ *a family of disambiguation preferences for each ambiguity source of* t*.*

The interpretation σ *is said to be* current *if and only if* $\forall s \in \Sigma_t,\ \sigma(s) = c_s$*.*

The concept of current interpretation encodes the idea of an interpretation which satisfies the latest preferences expressed by the user on whatever ambiguity issue occurring in the ambiguous term being disambiguated. Intuitively a current interpretation is a "good" interpretation which should have an high rating. A class comprising interpretations which are worse than current is the class of recent interpretations.

Definition 3.7 (Recent interpretation) *Let* t *be an ambiguous term,* σ *an interpretation for it and* $\langle c_s, \mathcal{P}_s \rangle,\ \forall s \in \Sigma_t$ *a family of disambiguation preferences for each ambiguity source of* t*.*

The interpretation σ *is said to be* recent *if and only if it is not current and* $\forall s \in \Sigma_t,\ \sigma(s) \in \mathcal{P}_s$*.*

Finally, and to be ranked worse than both current and recent interpretations, we define the class of acceptable interpretations.

Definition 3.8 (Acceptable interpretation) *Let t be an ambiguous term, σ an interpretation for it and $\langle c_s, \mathcal{P}_s \rangle$, $\forall s \in \Sigma_t$ a family of disambiguation preferences for each ambiguity source of t.*

The interpretation σ is said to be acceptable *if it is valid but neither current, nor recent.*

An orthogonal way of classifying valid interpretations for a given ambiguous term t is by considering the number of implicit coercions required for the correct typing of a resulting CIC term. For interpretations of single sources of ambiguity the local criterion of quality is evident: an interpretation which does not insert a coercion in one particular position is to be preferred to an interpretation that does.

No evident good solution for extending this local criterion to a global is at hand, hence we will stick to a conservative solution stating that an interpretation of term which requires no coercions (a so called *limpid interpretation*) is better than those interpretations which require at least one coercion (*obfuscated interpretations*).

We now have two orthogonal global ranking criteria for interpretations of ambiguous term. No reasonable criterion for the global rating of representation is evident a priori. The one we are now using is motivated by several concrete examples found in the standard library of MATITA. On the basis of the examples collected so far, we achieved results deemed satisfactory by our (small) user community rating representation as shown in Table 3.1, current-limpid interpretations are ranked as the best ones, followed by recent-limpid and so on until acceptable-limpid and acceptable-obfuscated which are equaly ranked as the worst class of interpretations.

Table 3.1: Ranking rules for ambiguous term interpretations

	limpid	**obfuscated**
current	I	III
recent	II	IV
acceptable	V	

Table 3.2: User preferences and other interpretations in effect for the disambiguation examples. The underlined preference is the current one.

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

3.3.1 Examples

In the remainder of this section we will present a set of examples showing our criterion at work.

In all the examples we will assume that the user preferences are set as shown in Table 3.2. In the table the superscript notation is used to informally refer to functions having as domain the set in superscript position and as codomain the set in basis position. The third column is used to show other interpretations available in the library which are neither current, nor recent.

Example 3.4 *Rating: 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 current-limpid. The current preferences of the user are respected. □

Example 3.5 *Rating: 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 recent-limpid.

□

Example 3.6 *Rating: the user is asked*

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

In this case the current preferences cannot be satisfied. The two best representation are:

$$\forall n, m, r : \mathbb{N}.\ n|_{\mathbb{N}} m \to n *_{\mathbb{N}} r|_{\mathbb{N}} m *_{\mathbb{N}} r$$
$$\forall n, m, r : \mathbb{Z}.\ n|_{\mathbb{Z}} m \to n *_{\mathbb{Z}} r|_{\mathbb{Z}} m *_{\mathbb{Z}} r$$

both recent-limpid.

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 3.7 *Rating: 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 rank since it requires a coercion. Thus the best representation is:

$$\forall n, m : \mathbb{Z}. n|_{\mathbb{Z}} m \to |n|_{\mathbb{Z}}\ |_{\mathbb{Z}}\ |m|_{\mathbb{Z}}$$

that is recent-limpid.

□

Example 3.8 *Rating: 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 current-obfuscated.

□

Example 3.9 *Rating: 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 recent-limpid.

□

Example 3.10 *Rating: 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 current-limpid.

□

Example 3.11 *Rating: 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 cannot be satisfied. Thus the only two representations are:

$$\forall n, m : \mathbb{Z}.n <_{\mathbb{Z}} m \rightarrow \sqrt{n} <_{\mathbb{C}} \sqrt{m}$$
$$\forall n, m : \mathbb{N}.n <_{\mathbb{N}} m \rightarrow \sqrt{n} <_{\mathbb{C}} \sqrt{m}$$

both acceptable. The user is asked for the intended meaning. □

3.4 Implementation

3.4.1 The Code

The disambiguation mechanism composed by both ambiguity resolution and representation rating is implemented in MATITA and used everywhere the input of terms from the user is required: terms required as tactic arguments, content-based queries using the WHELP technology, theorem statements, and so on. The implementation sharply distinguishes the two tasks while being reported as a single macro-component (the `ambiguity manager`) in the architecture diagram of Figure 2.10. The part of the diagram related to disambiguation is also reported in Figure 3.2.

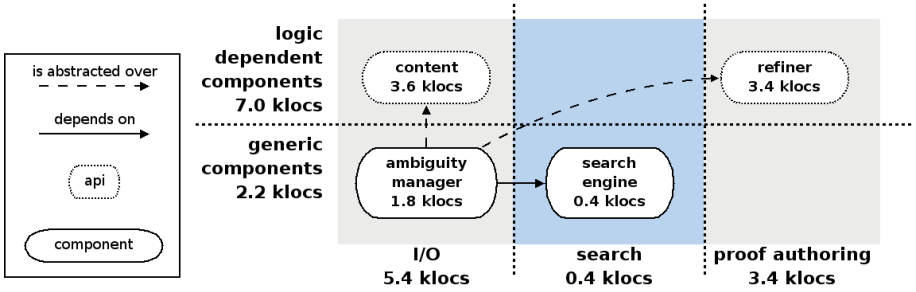

Figure 3.2: Components of the disambiguation implementation in MATITA

The `ambiguity manager` itself is logic dependent since it needs to generate CIC terms. Nonetheless it is abstracted over both the `content` component (which contains the definition of the abstract syntax tree representing ambiguous terms) and the `refiner` which in MATITA is used as the validity predicate used to check correctness of (partially) disambiguated terms. The dependency on the `search engine` is used to automatically fill the disambiguation codomains of unbound identifiers using the locate query of our content-based search engine.

The efficient disambiguation algorithm of Section 3.2 is implemented in the `Disambiguate` module, whose main part is a functor abstracted over a module which should mainly implement a callback function `interactive_interpretation_choice`. Such a function is invoked (only if requested) to interactively ask the user to choose among the various choices of CIC terms returned by the algorithm.

The functor exports two functions (`disambiguate_term` and `disambiguate_obj`) which are used respectively to disambiguate AST of formulae and AST of concepts, which can be expressed verbatim in the proof language of MATITA (think for example at definitions where both the type and the body of a concept are directly written in a script).

Internally the two functions work first constructing the disambiguation domain of the AST to be parsed (i.e. the signature Σ_t) and then applying the algorithm presented in [97]. The differences between the cases of formulae and concepts are negligible and mostly delegated to the use of two different refinement functions.

The two disambiguation functions have two parameters which are exploited to implement efficiently representation rating. The parameters are:

- a boolean flag that states whether or not the refiner can try to insert coercions to fix local type-checking errors or not, when this flag is set to disable coercion insertion the terms returned by the disambiguation algorithm are granted to be coercion-free;

- a map from sources of ambiguity to their interpretation domains; being passed externally to the disambiguation functions we can play with it to affect the

set of possibilities the algorithm has to fix ambiguity issues.

Representation rating is implemented in the module `GrafiteDisambiguator` which is a wrapper around `Disambiguate` which exports the very same interface (`Disambiguate.Disambiguator`) and as such can be used as a drop-in replacement for it. Its aim is to generate the set of possible representations and guess the one closest to what the user intended using the ranking criteria of Section 3.3.

Instead of generating all possible representations and then rate them we lazily build representations in increasing order of ranking. The technique is to interleave the tasks of resolution of ambiguities and representation rating tuning the parameters of the disambiguation algorithm to ensure that the CIC term it returns have an upper bound on rating.

In practice this means invoking multiple times the disambiguation algorithm with different tuning of the parameters. In Table 3.3 we show how the parameters are set in the five disambiguation attempts tried in order. Since subsequent attempts reconsider representations already considered in previous attempts, memoization can be used to speed up the process (but is currently not, since so far the execution speed of the current implementation has been satisfactory).

Table 3.3: Disambiguation attempts sequence

Attempt no.	**Domain map**	**Coercions**	**Max. rating**
1	current preferences	disabled	current-limpid
2	all preferences	disabled	recent-limpid
3	current preferences	enabled	current-obfuscated
4	all preferences	enabled	recent-obfuscated
5	all choices available in the library	enabled	acceptable

3.4.2 Preferences Tuning

We mentioned that preferences can be both explicitly set by the user and implicitly by the system. The basic mechanism given to the user to set preferences explicitly is with a few commands which can be used in scripts:

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

The way of setting preferences changes according to the kind of source of ambiguity. For unbound identifiers (first line of the script snippet above) 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 (see Chapter 5).

Implicit preferences are handled in MATITA by the mean of 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. For example, near the beginning of `nat/chinese_reminder.ma`[1], one can read:

```
include "nat/exp.ma".
include "nat/gcd.ma".
include "nat/permutation.ma".
include "nat/congruence.ma".
```

Inclusion might look like similar commands in other systems, but in fact is a completely different concept (which might deserve a better name . . .). It only affects preferences and does not load any concept from the library. It is just meant to provide continuity in development even for theories which are split among several scripts.

The second mechanism consists in automatically setting implicit preferences for concepts and new notations just defined. Hence, after defining a new theorem called

[1] `http://matita.cs.unibo.it/library/nat/chinese_reminder.ma`

"i", an alias like the first one we saw above will be implicitly added as the current preference for "i" and remembered by MATITA.

A related issue is that of granting the possibility of compiling in a batch fashion (with `matitac`) scripts which when executed the first time required user intervention to choose among a set of equally rated representations. Our current solution is to compute, after a disambiguation process which required user intervention, the minimal set of preferences that would have been sufficient to avoid asking the user. Once such a set is computed it gets automatically inserted as aliases by MATITA in the script just before the command that triggered disambiguation.

3.4.3 User Interface

In order to add the disambiguation machinery to a graphical interactive system like MATITA, one also need to choose a suitable user interface for interactively asking the user which representation she intended among a set of equally rated representations.

According to our experience a plain list of choices requires too much effort to be investigated when the list is long and is affected by the problem of failing to provide an immediate feedback of the different choices.

Our current solution is shown in Figure 3.3. It 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 emphasized in the formula (shown in boldface in the dialog window of the picture). 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.

Since the internal representation of formulae has more information than what has been typed by the user, two additional issues related to user interface 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.

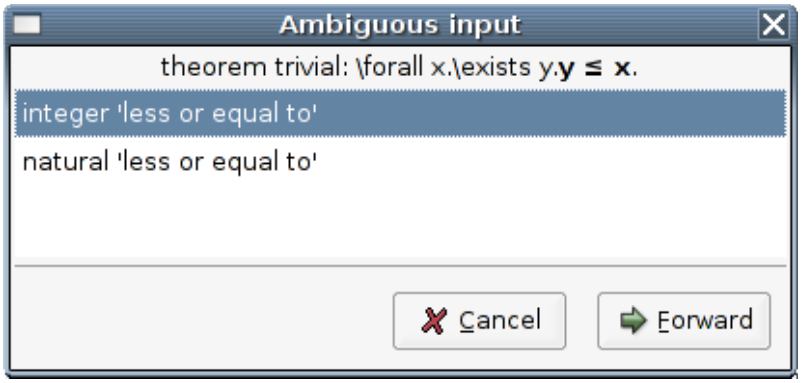

Figure 3.3: Interpretation choice dialog

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 Chapter 5).

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.

3.5 Related Work

The proposed efficient algorithm is novel and, accidentally, follows the line of type-based parsing as is done in the Grammatical Framework [89]. In spite of that in our approach we avoid imposing an additional type system just for the purpose of parsing, but rather reuse the type system (or part of it) of the underlying logical foundation of MATITA. This approach is portable to the entire applicative area of proof assistants and, as discussed in Chapter 2 the requirement of a refiner should be easily fulfillable in competing systems.

Regarding the more general issue of handling overloading in theorem provers, we should compare the approach of MATITA with those of Coq and Isabelle that implemented alternative solutions.

Overloading in Coq is more limited than in MATITA. Symbols can be overloaded, but each additional meaning of a symbol must belong to a different *interpretation scope* (in Coq's terminology). As a matter of fact this means that each interpretation must have a different return type. When an overloaded symbol is used in some context, the type expected for that context is used to determine the scope in which the symbol will be interpreted. For example the context can be the argument position of a function, whose input type is known and used as the expected type for that context. This mechanism is a shortcut for a more generic syntax that enable users to manually change interpretation scopes inside formulae.

The disambiguation time of interpretation scopes is granted to be linear, but has drawbacks. One is that it does not deal properly with polymorphic functions (i.e. functions having a type starting with `\forall A:Type. A \to ...`) since the expected interpretation scope for polymorphic arguments cannot be determined until the first argument has been interpreted.

An additional drawback is the bad mix of subtyping and interpretation scopes which can reject formulae due to scope mismatch notwithstanding the fact that they can be well-typed after the insertion of coercions. For example 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, while inserting a coercion can solve the problem.

Also the restriction of overloading inside a single interpretation scope is tight in several examples where the type of the arguments of the overloaded notation, and not the return type, differentiate the interpretations.

Overall we believe that interpretation scopes behaves as a new kind of types, in the spirit of Grammatical Framework. As such they fail to reuse the underlying type system already present in Coq duplicating efforts and opening the flank to inconsistencies among the two systems which will result in practical nuisances as those described above.

Overloading in Isabelle [111] 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 [79] however, 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.

3.6 Conclusions

In this chapter we presented the disambiguation mechanism used in MATITA to parse formulae that can be typed by the user supporting the degree of freedom and ambiguity of the usual mathematical notation.

The approach is novel with respect to those that can be found in competing system and builds upon the MATITA philosophy of an always visible library of formalized mathematics. It has been successfully applied in the past to other technologies—most notably to the WHELP search engine—as an assessment of the generality and reusability of the adopted techniques. The current representation rating is being tested and continuously tuned from real life examples we encounter in the development of the standard library of MATITA. The current result is satisfactory according to our user base.

Several future developments for the disambiguation in MATITA are being considered. From the point of view of performances memoization can be added in several places, both in the refinement itself and in the invocation of the disambiguation algorithm as we previously discussed.

From the point of view of user interaction a currently open challenge is how to present disambiguation errors effectively to the user. The choice of multiple disambiguation attempts indeed raise the question of which is the "real" error of the user since several typing errors are available. We are currently investigating presenting them on different axis (like all errors in the same attempts, or all equal errors in different attempts).

Original Contributions

The efficient disambiguation algorithm of Section 3.2 has been designed by Claudio Sacerdoti Coen in its Ph.D. thesis [94] who also wrote the first implementation in an old proof assistant prototype of the HELM team. Later on it has been better described and re-implemented from scratch by this thesis author and Andrea Asperti taking into account the content level (see Chapter 5) used internally by MATITA

to represent term after parsing. It has since then being benchmarked and compared with alternative algorithms by this thesis author as a joint work with Claudio Sacerdoti Coen.

The representation rating mechanism has been developed by this thesis author as a joint work with Claudio Sacerdoti Coen, and has been implemented in MATITA and currently maintained by this thesis author.

Related Publications

Part of the work described in this chapter has been previously published in the following papers:

- Claudio Sacerdoti Coen and Stefano Zacchiroli.
  Efficient Ambiguous Parsing of Mathematical Formulae [97].
  In Proceedings of MKM 2004: The 3rd International Conference on Mathematical Knowledge Management 2006[2], Lecture Notes in Computer Science, Vol. 3119, pages 347–362. Springer-Verlag, 2004.

- Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli.
  User Interaction with the MATITA *Proof Assistant* [10].
  To appear in Journal of Automated Reasoning[3], special issue on User Interfaces for Theorem Proving[4].

[2] http://mizar.org/MKM2004/

[3] http://www.springerlink.com/link.asp?id=100280

[4] http://www.informatik.uni-bremen.de/~cxl/uitp-jar/

Chapter 4

Tinycals: Step by Step Tacticals

Most of the state-of-the-art proof assistants share a common recipe on how proofs are written. Ingredients of the recipe are: a procedural proof language based on tactics, *scripts* as storage for proof language statements, LCF-like *tacticals* as the primary tool for composing tactics.

In this chapter we discuss how these ingredients badly interact with user interfaces based on the *de facto* standard interaction paradigm for proof authoring: *script management*, an idea pioneered by CtCoq and then adopted by the popular Proof General generic interface for theorem provers.

We identify the key problem in the coarseness of tactical evaluation, and propose *Tinycals* as an alternative to a subset of LCF tacticals, showing that the user does not experience the same problem if tacticals are evaluated in a more fine-grained manner.

We present the syntax and the small step formal operational semantics of tinycals, as well as their implementation in the Matita proof assistant.

4.1 Some Best Practices of Interactive Theorem Proving

Tradition is an important thrust of modern interactive theorem proving, no matter how that can sound as an oxymoron. Best practices whose origins date back to

elder systems like LCF [41]—to name the one who probably influenced most modern provers—are still with us.

Goal-directed proof search, pioneered by Edinburgh LCF, is still *the* way of working of most mainstream theorem provers like Coq [26], Mizar [69],[1] HOL Light [51], PVS [88], and Isabelle [53]. In such a setting the user states a theorem, usually about previously defined concepts, and then enters an interactive proof mode. While in proof mode the user works backward from the statement to be proved (a *goal*) to previously proved theorems or axioms that imply it.

A single proof step consist in reducing a given goal to simpler (sub-)goals. Once a goal is so simple to match a logic axiom or a previously defined theorem it gets elided. The proof is completed once all goals have been elided. How proof steps are performed is one of the many possible characterization of the proof language of a system [47].

"Procedural"[2] proof languages rely on *tactics* which are a tool to let the user explicitly tell the system how a goal (subject of the tactic application) has to be decomposed into simpler sub-goals. Usually, tactics either mimic (composition of) inference steps of the logical foundation of the prover, or implement decision procedures able to automatically (dis-)prove a given goal. "Declarative" proof languages on the other hand let the user give a more abstract description of a given proof, which usually resembles more than the procedural version the pen and paper proof. Declarative proofs are usually easier to read but take away from the user fine-grained control on how the proof is developed.

No clear winner among the two style stands [47], as also shown by various attempts of mixing the two styles [46, 116]. As a matter of fact though, large bodies of proofs written in a procedural style do exist (for example the 40.000 theorems con-

[1] while the "Mizar mode" of working is arguably different from that of the other systems mentioned here, from the user point of view progressive article refinement is very similar to goal directed proof search.

[2] double quotes are a must, since no sharp distinction between the two style of proof languages does exist, see [47] for a deeper discussion of the topic

tained in the Coq standard library[3]), in this chapter we are mainly concerned about such proofs. The issue we are facing though can similarly affect (and hereby can be similarly solved in) declarative proof languages which have support for embedding of procedural proof snippets.

We now present two best practices of interactive theorem proving which can be observed in the behaviour of users of such systems: the usage of LCF tacticals and script management.

4.1.1 LCF Tacticals

A tactic is the smallest command that can be used to proceed in a proof, which is meaningful for the system. In abstract terms—see Section 4.3 for the actual formalization tinycals rely on—a tactic is a partial function of two arguments, a *proof status* and a *goal*. A proof status is the logical status of an ongoing proof, intuitively containing a set of conjectures that need to be proven before a proof is completed. A goal is a pointer to one of such conjectures. Being partial a tactic can either return a value or fail. In the former case its return value is a new proof status (containing a priori a different set of conjectures from the input proof status), and two goal lists: one referencing the new conjectures created by the tactic, and one referencing the conjectures which have been closed by the tactic.

Tacticals are higher-order tactics, used to combine tactics together. They were first introduced by LCF [41] and are still part, though extended with new constructs, of the procedural proof languages of modern proof assistants like Coq and PVS. The original 5 LCF tacticals, together with their types and informal semantics are reported below. A formal semantic for them, or better for their modern counterpart in Coq and PVS, has been given in [57].

`IDTAC` (type:[4] `tactic`; synopsis: `idtac`)

[3] `http://coq.inria.fr/library-eng.html`

[4] we use an ML-like notation for types, assuming that `tactic` is the type of tactics; the synopsis uses the concrete syntax used in Coq and Matita for their counterparts of LCF tacticals

identity tactical: simply returns the proof status unchanged, a singleton list containing the same goal it was applied to, and an empty list of new conjectures

`ORELSE` (type: `tactic` $\times$ `tactic` $\rightarrow$ `tactic`; synopsis: t_1 || t_2)

try-recover tactical: behaves as t_1 unless t_1 fails, in which case behaves as t_2;

`THEN` (type: `tactic` $\times$ `tactic` $\rightarrow$ `tactic`; synopsis: t_1 ; t_2)

sequential composition tacticals: applies t_1 to the input. If t_1 returns a value apply t_2 to each of the new conjectures create by t_1, "folding" the final proof status and goal sets while iterating. Fails otherwise.

Repeated application of "**;**" can be used to form *tactic pipelines* of the form t_1 ; t_2 ; $\cdots$; t_n, very common in procedural proof scripts;

`THENL` (type: `tactic` $\times$ `tactic list` $\rightarrow$ `tactic`; synopsis: $t; [t_1 \mid \cdots \mid t_n]$)

branching tactical: applies t to the input. If t returns a value and its application generated n new conjectures applies t_1 to the first conjecture returned, t_2 to the second and so on, "folding" as above. Fails when t or whatever application of t_i fails.

`REPEAT` (type: `tactic` $\rightarrow$ `tactic`; synopsis: `repeat` t)

looping tactical: apply t to the input and repeat the application of t to all generated new conjectures until t fails, then return the value collected thus far. `repeat` itself never fails.

In addition to these five core tacticals, modern systems implemented their own variations. One of those which is commonly found in other provers (NuPRL and Coq for example) is `TRY`:

`TRY` (type `tactic` $\rightarrow$ `tactic`; synopsis: `try` t)

try tactical: behaves as t unless t fails, in which case behaves as `idtac`. `try` itself never fails, and can be implemented on top of `ORELSE` as follows:

$$\texttt{try}\ t \stackrel{\text{def}}{=} t \ ||\ \texttt{idtac}$$

LCF tacticals and their variants implemented in modern systems improved procedural proof languages providing concrete advantages, that we illustrate with the help of Figure 4.1. The same script snippet proving a lemma about natural numbers factorization is shown in two different versions: one using LCF-like tacticals (on the right) one not using them.

The concrete syntax used in the snippets is that of the proof language of MATITA and the full original script is available in the standard library of MATITA.

The following sections discuss in more detail the advantages of LCF-like tacticals usage.

Proof Structuring

Using tacticals proof scripts can be structured as nested blocks, as is common practice in traditional programming, instead of being a flat list of statements one after another.

Using branching for instance, the script representation of proofs can mimic the structure of the proof tree (the tree having conjectures 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* (on the average very poor, if compared with declarative proof languages).

Script *maintainability* is improved as well by the use of branching, since tactic sequences related to a particular conjecture can be statically spotted looking at the script. This way script fixes needed by additions or removals of hypotheses or by conjectures reordering can be performed without the need of interactive replay, and are delimited in space. A concrete experience on the maintenance of the standard library of MATITA, after changing the implementation of a tactic so that the returned hypotheses were swapped taught us how much time can be saved by thoroughly using the branching tactical!

For instance, in the right hand side of Figure 4.1 it is now clear that `elim f` splits the proof in two branches; both of them (selected by "`[1,2:`") are attacked

```
theorem lt_O_defactorize_aux:
 \forall f:nat_fact.
 \forall i:nat.
 O < defactorize_aux f i.
intro.
elim f.
simplify.
unfold lt.
rewrite > times_n_SO.
apply le_times.
change with (O < \pi _ i).
apply lt_O_nth_prime_n.
change with (O < (\pi _ i)^n).
apply lt_O_exp.
apply lt_O_nth_prime_n.
simplify.
unfold lt.
rewrite > times_n_SO.
apply le_times.
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.
```

```
theorem lt_O_defactorize_aux:
 \forall f:nat_fact.
 \forall i:nat.
 O < defactorize_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 ] ].
```

Figure 4.1: The same proof script with (on the right) and without (on the left) tacticals.

beginning with the same tactic sequence until each branch is split again by the application of the `le_times` lemma. Of the four resulting branches, the second and third one (selected by "`|2,3:`") are proven by the same tactic sequence, being proofs of the same fact. All the tactics that are not followed by branching do not introduce

ramifications in the proof.

In practice, the proof on the left hand side of Figure 4.1 would have been written using indentation and blank lines to understand where branches start and end. This way readability would have been improved, but a lesser effect would have been achieved for proof maintenance. Moreover, that way the system can not verify the layout of the proof script and thus does not guarantee consistency when the script is changed. We expect that users will abandon this behaviour as soon as an alternative without drawbacks—not the case of plain LCF tacticals—will surface.

Conciseness

As code factorization is a good practice in programming, proof factorization is in theorem proving.

The use of tacticals like sequential composition reduce the need of copy & paste in proof scripts helping in factorizing common cases in proofs (so frequent in formal proofs pertaining to the computer science field). Conciseness is evident in Figure 4.1. Conciseness of course also help reducing the De Bruijn factor [114].

Note that in long, real-life procedural proof scripts the conciseness gain of using tacticals is usually larger than that shown in Figure 4.1, due to the use of indentation, blank lines, and comments as section delimiter markers.

Robustness

A final advantage of using tacticals in procedural proof scripts, not shown in Figure 4.1 is the possibility to increase the robustness of scripts to changes (in the logical system, in the behaviour of widely used tactics, ...).

Using conditional tacticals like try-recover and looping proof scripts authors can deal with (potential) failures of tactics which do not fail at the moment in which a scripts is written, but that might fail in the future.

We now move to another best practice of interactive theorem proving: script management.

4.1.2 Script Management

Historically, theorem provers descended from LCF used a simple read-compile-eval loop as their primary user interface. Directly using such an interface was rather painful. The common way of working with such systems was to keep a textual scratch pad with theorem statements and tactics, and then copy & paste text from there to the top-level, using appropriate undo commands to restore past system states.

In [21] Bertot and Théry discussed *script management* and proposed a style of user interaction which has then become the *de facto* standard interaction paradigm for working with interactive theorem provers. The purpose of script management is to record a clean script of commands sent to the proof system during a working session, meaning with "clean" that neither undo commands should be present in the script nor previously undone command not part of the final proof. Replaying a clean script in a new working session should of course bring the system in the same status it was at the end of the proof when the script was generated.

The proposed style of user interaction was initially implemented in the CtCoq [20] system and is nowadays implemented in the Proof General generic interface for theorem provers [12]. More recent, prover-specific user interfaces—like CoqIDE[5] for Coq and the Matita authoring interface itself—implement that interaction style as well. Figure 4.2 is a screenshot of Coq's flavour of Proof General, we will refer to it to describe the ingredients of script management.

The *state window* (at the top-right) is used to give feedback to the user of the current proof status. In the figure the current proof has three conjectures yet to be proved, one of which is shown in full details. The *command window* (on the left) is used to receive input by the user, is the place where textual statements of the proof language can be edited (as is normally done using vanilla Emacs buffers).

The added value of the command window over a plain textual scratch pad, is its being split in four regions used to different aims, two of these regions are shown

[5] `http://coq.inria.fr/coqide/index.html`

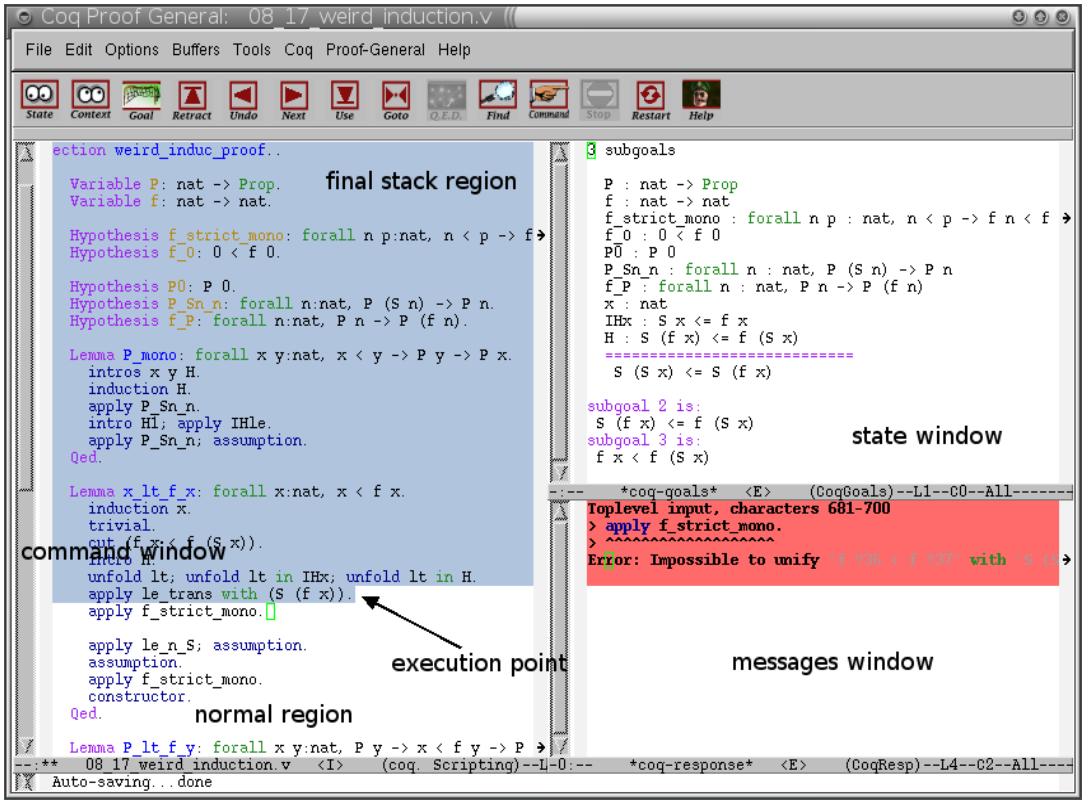

Figure 4.2: Annotated screenshot of Coq's flavour of the Proof General generic interface for theorem provers

in the figure.[6] The topmost one—the *final stack region*—is the depot of all (not undone) commands sent to the proof system. It is locked for editing: statements laying there cannot be changed. On the contrary, text in the *normal region* can be freely modified and it is used as a scratch pad. The *execution point* is the boundary separating the two regions, and is used as an handle to extend or shrink the final stack region.

New commands can be sent to the system moving the execution point forward (i.e. toward the bottom of the script) and retracted moving it backward, with no

[6] the remaining two regions, namely the *buffer* and *queue* regions, are relevant only from the point of view of communication delays between the user interface and the system; we are ignoring those aspects here, being mainly concerned about the resulting interaction paradigm

need of explicitly writing undo commands in the script. This is possible due to the generality of Proof General, whose authors standardized PGIP (Proof General Interaction Protocol), a protocol abstracting over the commands offered by the different provers to deal with the proof history. This way Proof General knows how to retract a statement and has no need to ask the user to type a verbatim command for that purpose in the proof script, thus easing the task of producing a clean proof script.

Auxiliary windows are used, depending on the prover, for various tasks. The *messages window* shown at the bottom-right of Figure 4.2 is for instance used to deliver (error) messages from Coq to the user. Another example of auxiliary window is the *context window*, used in CtCoq for out-of-band notification of query results (e.g. the list of theorems which can be used to rewrite the current goal).

We will now highlight the tension between LCF tacticals and script management in state-of-the art user interface for theorem provers.

4.2 An Unwanted Trade-Off

The major drawback of procedural proof scripts with respect to declarative proof scripts is probably the almost impossibility to understand them statically. They are meaningful only by step-by-step execution (or *replaying*), following the evolution of the proof status in the state window.

The refresh ratio of the state window is thus important for the final user, the higher: the finer understanding of proof evolution she can have. At the very minimum though, tactics should be executed atomically since the proof status may be inconsistent during tactic execution. The tactic granularity is thus an upper limit to the refresh ratio.

Unfortunately, state-of-the-art user interface are far from this limit. In all the proof assistants we are aware of indeed, tacticals are always evaluated atomically and placing the execution point in the middle of tacticals (for example at occurrences

of "**;**" in tactic pipelines) is not allowed. In the script snippet on the right of Figure 4.1, this means that having the execution point at the beginning of the proof (just before `intro;`) and asking the system to move it forward (i.e. to execute the next statement), the user will result—probably after a non negligible computation time—in a "proof completed" status, without having any feedback of the inner proof statuses the system passed through.

The only way for the user to inspect intermediate proof statuses—a frequent need, for instance for script maintenance or proof presentation—is to manually de-structure the tacticals. Where for "de-structure" we mean textually editing the script and change complex tacticals in tacticals-free tactics sequence. In a sense we are going back to the flat script on the left of Figure 4.1, loosing the proof structuring and conciseness advantages of tacticals.

The big step evaluation of tacticals has also drawbacks on how proof authors develop their proofs. Since it is not always possible to predict the outcome of complex tactics, the following is a common (not best though) practice developed by proof authors:

1. evaluate the next tactic of the script;
2. inspect the set of returned conjectures;
3. decide whether the use of "**;**" or "[" was appropriate;
4. if it is:
   (a) retract the last statement;
   (b) edit the script snippet to add the tactical;
   (c) go back to step (1).

The last drawback of the bad mix of LCF tacticals and script management, but not less important, is the imprecise error reporting induced by big step evaluation of tacticals. Consider the case of a script breaking, meaning that the execution of a proof snippet who used to reach proof completion fails to evaluate a (possibly

complex) tactical. This case is pretty frequent in practice due to breakages induced by the user changing definitions or theorems used by other proofs, or by system developers changing tactics implementation.

The error message returned by the system 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, making impossible error localization in it.

The current best practice to find the statement that need to be fixed is usually to progressively replace tactics with the identity tactical in the proof script, until the single failing statement is found. This technique is not only error prone, but is even not reliable in presence of *side-effects* (tactics closing conjectures 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.

Due to all these drawbacks, we observe that there is an unwanted trade-off between the advantages of using LCF tacticals in proof scripts, and both the degraded quality of proof script maintenance and the inability to interactively replay proofs.

4.2.1 Our Solution: Tinycals

We claim that the above trade-off is artificial for a significant subset of LCF tacticals, and that better intermixing of tacticals and script management can be achieved using the tiny language of tacticals we are going to propose: the so called *tinycals*.

Tinycals can be evaluated in small steps, enabling the execution point to be placed inside complex structures like pipelines or branching constructs. This goal is achieved by de-structuring the syntax of tacticals and stating the semantics as a transition system over *evaluation status*, that are structures richer than the proof status tactics act on.

Note that de-structuring does not necessarily mean changing the concrete syntax of tacticals, but rather enabling parsing and immediate evaluation of tactical fragments like "[" alone. Actually, the concrete syntax of tinycals has been chosen

to be as close as possible to the modern implementation of LCF tacticals in other provers, most notably Coq.

4.3 Syntax and Semantics of Tinycals

4.3.1 Tinycals Syntax

The grammar of tinycals is reported in Table 4.1, where $\langle L \rangle$ is the top-level nonterminal generating the proof language meant to be used in scripts for writing proofs. $\langle L \rangle$ is a sequence of statements $\langle S \rangle$. Each statement is either an atomic tactical $\langle B \rangle$ (marked with "`tactic`") or a tinycal.

Note that the part of the grammar related to the tinycals themselves is completely de-structured. The need for embedding the structured syntax of LCF tacticals (nonterminal $\langle B \rangle$) in the syntax of tinycals will be discussed in Section 4.5. For the time being, the reader can suppose the syntax is restricted to the case $\langle B \rangle ::= \langle T \rangle$.

4.3.2 Tinycals Semantics

Semantics Parameters

Tinycals are not specific to Matita. Both from a theoretical point of view and from an implementative one, tinycals can be reused in provers other than Matita. To this end, their semantics is parametric in a few items, reported in Table 4.2; in the table parameters are reported together with their type, using an ML-like notation. Every system able to provide an instantiation of those parameters can benefit of tinycals. Intuitively the semantics of tinycals is parametric only in the proof status tactics act on and in their semantics.

A *proof status* is the logical status of an ongoing proof. It can be seen as the current proof tree, but there is no need for it to actually be a tree. Matita for instance just keeps the set of conjectures to prove, together with a proof term where meta-variables occur in place of missing components. From a semantic point of view

Table 4.1: Abstract syntax of tinycals and core LCF tacticals

$\langle L \rangle$	::=	(**proof language**)
	$\langle S \rangle$	(statement)
|	$\langle S \rangle\ \langle L \rangle$	(sequence)
$\langle S \rangle$	::=	(**statements**)
	"`tactic`" $\langle B \rangle$	(tactic)
|	"`.`"	(dot)
|	"`;`"	(semicolon)
|	"`[`"	(branch)
|	"`|`"	(shift)
|	$i_1, \ldots, i_n$ "`:`"	(projection)
|	"`*:`"	(wild card)
|	"`accept`"	(acknowledgement)
|	"`]`"	(merge)
|	"`focus`" $[g_1; \cdots; g_n]$	(selection)
|	"`done`"	(de-selection)
$\langle B \rangle$	::=	(**tacticals**)
	$\langle T \rangle$	(tactic)
|	"`idtac`"	(identity)
|	"`try`" $\langle B \rangle$	(recovery)
|	"`repeat`" $\langle B \rangle$	(looping)
|	$\langle B \rangle$ ";" $\langle B \rangle$	(composition)
|	$\langle B \rangle$ ";[" $\langle B \rangle$ "|" ... "|" $\langle B \rangle$ "]"	(branching)
$\langle T \rangle$	::= ...	(**tactics**)

the proof status is an abstract data type. Intuitively, it must describe at least the set of conjectures yet to be proved. A *goal* is just another abstract data type used

Table 4.2: Semantics parameters

proof status:	ξ
proof goal:	*goal*
tactic application:	$apply\_tac : T \to \xi \to goal \to \xi \times goal\ \texttt{list} \times goal\ \texttt{list}$

to index conjectures occurring in a proof status.

The function *apply_tac* implements tactic application. It consumes as input a tactic, a proof status, and a goal (the conjecture the tactic should act on), and returns as output a proof status and two lists of goals: the set of newly opened goals and the set of goals which have been closed.

This choice—slightly different from the choice made in LCF, where the `tactic` type does not return the set of closed goals—enables our semantics to account for *side-effects*, that is: tactics can close goals other than that to which they have been applied, a feature implemented in several proof assistants via existential or meta-variables [40, 73]. The proof status was not directly manipulated by tactics in LCF because of the lack of meta-variables and side-effects.

It is also worth noticing that, assuming goals are always freshly generated by tactics (i.e. never reused), a wrapper having the type of *apply_tac* in Table 4.2 can be created on top of an LCF-like tactic application function (e.g. $lcf\_apply\_tac : T \to \xi \to \xi \times goal\ \texttt{list}$) by simply recording the set of goals available before tactic application and comparing it with the one available after tactic application. When tinycals where first introduced in Matita, we actually migrated our old tactics implementing an LCF-like interface to the interface of *apply_tac* by implementing such a wrapper. We will come back to this in Section 4.4.2.

Evaluation Status

We will define the semantics of tinycals as a transition (denoted by " $\longrightarrow$ ") on evaluation status. Intuitively, an evaluation status carries three pieces of information:

1. the list of statements still to be executed (the idea being that evaluation will be iterated consuming such a list);

2. the proof status of the ongoing proof;

3. a data structure mocking up the logical structure of the statements executed so far in the ongoing proof.

Evaluation status are formally defined in Table 4.3. Each evaluation status is a triple, having one component for each piece of the above information.

Table 4.3: Evaluation status

status	=	$code \times \xi \times ctxt\_stack$	(evaluation status)
code	=	$\langle S \rangle$ `list`	(statements)
ctxt_stack	=	$(\Gamma \times \tau \times \kappa \times ctxt\_tag)$ `list`	(context stack)
Γ	=	*task* `list`	(context)
τ	=	*task* `list`	("to do" list)
κ	=	*task* `list`	(dot's continuations)
ctxt_tag	=	`B` | `F`	(stack level tag)
task	=	`int` × (`Open` *goal* | `Closed` *goal*)	(task)

The first component of the status (*code*) is a list of statements of the tinycals grammar. The list is consumed, one statement at a time, by each transition. This choice has been guided by the non structured form of our grammar and is crucial for fine-grained execution of tinycals.

The second component is the proof status, which we enrich with a *context stack* (the third component). The context stack, a representation of the proof history so far, is governed by a last-in first-out policy: levels get pushed on top of it either when the branching tinycal "[" is evaluated, or when "`focus`" is; levels get popped out of it when the matching closing tinycals are ("]" for "[" and "`done`" for "`focus`").

Since the syntax is non-structured, we can not ensure statically proper nesting of tinycals, therefore each stack level is equipped with a *tag* which annotates it with the kind of tinycal who added it on top of the stack (`B` for "[" and `F` for "`focus`").

In addition to the tag, each stack level has three components Γ, τ and κ, respectively for representing active tasks, tasks postponed to the end of branching, and tasks postponed by "**.**". The role of these components will be explained in the description of the tinycals acting on them. Each component is a sequence of numbered tasks.

A *task* is a handler to either a conjecture yet to be proven, or one which has been closed by a side-effect. In the latter case the user will have to confirm the instantiation with "`accept`".

Each evaluation status is meaningful to the user and can be presented to her by slightly modifying already existent user interfaces. Our presentation choice is described in Section 4.4.3. The impatient reader can take a sneak preview of Figure 4.6, where the interesting part of the proof status is presented as a notebook of conjectures to prove, and the conjecture labels represent the relevant information from the context stack by means of:

1. bold text (for conjectures in the currently selected branches, targets of the next tactic application; they are kept in the Γ component of the top of the stack);

2. subscripts (for not yet selected conjectures in sibling branches; they are kept in the Γ component of the level below the top of the stack).

The rest of the information hold in the stack does not need to be shown to the user since it does not affect immediate user actions.

Before moving to the actual semantics, we will define the utility functions we will need later on.

Utility Functions

The informal notion of *active task* refers to a single task the user is working on. In addition to their other duties, "[" and "|" branching tinycals also automatically set an active task for the user, to avoid the need of an explicit selection. The user is of course free to change the active task (possibly preferring a *list* of tasks) using appropriate tinycals.

Active tasks automatically selected by "[" or "|" are called *unhandled* until a tactic is applied to them, or a side-effect closes them. Unhandled tasks are just postponed (not moved into the to do list τ) by $i_1,\ldots,i_n$ ":".

Given that: `Closed` tasks can't be unhandled (since the only way to close a task is either direct tactic application or side-effect, hence per definition they can't be unhandled) and that the integer components of tasks are initialized to positive integers; then a reliable way to test if a task is unhandled or not is the *unhandled* function, defined as follows:

$$unhandled(l) = \begin{cases} true & \text{if } l = \langle n, \texttt{Open } g\rangle \wedge n > 0 \\ false & \text{otherwise} \end{cases}$$

The actual function in charge of naming branches (i.e. initializing the integer components of tasks to increasing positive integers) is *renumber_branches*:

$$renumber\_branches([\langle i_1, s_1\rangle; \cdots; \langle i_n, s_n\rangle]) = [\langle 1, s_1\rangle; \cdots; \langle n, s_n\rangle]$$

Goals opened by a tactic are marked with *mark_as_handled* to distinguishing them from unhandled goals. *mark_as_handled* is defined as follows:

$$mark\_as\_handled([g_1; \cdots; g_n]) = [\langle 0, \texttt{Open } g_1\rangle; \cdots; \langle 0, \texttt{Open } g_n\rangle]$$

The next three functions returns open goals or tasks in the status or parts of it. The name *open goal* is used to refer to goals pointing to conjectures still to be proved.

$$get\_open\_tasks(l) = \begin{cases} [\,] & \text{if } l = [\,] \\ \langle i, \texttt{Open } g\rangle :: get\_open\_tasks(tl) & \text{if } l = \langle i, \texttt{Open } g\rangle :: tl \\ get\_open\_tasks(tl) & \text{if } l = hd :: tl \end{cases}$$

$$get\_open\_goals\_in\_tasks\_list(l) = \begin{cases} [\,] & \text{if } l = [\,] \\ g :: get\_open\_goals\_in\_tasks\_list(tl) & \text{if } l = \langle \_, \texttt{Open } g\rangle :: tl \\ get\_open\_goals\_in\_tasks\_list(tl) & \text{if } l = \langle \_, \texttt{Closed } g\rangle :: tl \end{cases}$$

$$get\_open\_goals\_in\_status(S) = \begin{cases} [\,] & \text{if } S = [\,] \\ get\_open\_goals\_in\_tasks\_list(\Gamma @ \tau @ \kappa) & \\ \quad @ get\_open\_goals\_in\_status(tl) & \text{if } S = \langle \Gamma, \tau, \kappa, \_\rangle :: tl \end{cases}$$

To keep the correspondence between branches in the script and ramifications in the proof, the semantics enforce that conjectures closed by side-effects correspond to tasks marked as `Closed` if they are in Γ (that keeps track of open branches). If they are elsewhere (in to do list τ or dot continuation κ) they are silently removed. `Closed` branches have to be accepted by the user with "`accept`".

close_tasks is the function used to mark tasks as closed, or remove them, as appropriate for the place where a task lives. It is defined in term of an auxiliary function, and of *remove_tasks* which remove tasks matching a given list of goals. All those three functions are defined as follows:

$$close\_tasks(G, S) = \begin{cases} [\,] & \text{if } S = [\,] \\ \langle close_{aux}(G, \Gamma), \tau', \kappa', t\rangle :: close\_tasks(G, tl) & \text{if } S = \langle \Gamma, \tau, \kappa, t\rangle :: tl \\ \qquad \text{where } \tau' = remove\_tasks(G, \tau) & \\ \qquad \text{and } \kappa' = remove\_tasks(G, \kappa) & \end{cases}$$

$$close_{aux}(G, l) = \begin{cases} [\,] & \text{if } l = [\,] \\ \langle i, \texttt{Closed } g\rangle :: close_{aux}(G, tl) & \text{if } l = \langle i, \texttt{Open } g\rangle :: tl \wedge g \in G \\ hd :: close_{aux}(G, tl) & \text{if } l = hd :: tl \end{cases}$$

$$remove\_tasks(G, l) = \begin{cases} [\,] & \text{if } l = [\,] \\ remove\_tasks(G, tl) & \text{if } l = \langle i, \texttt{Open } g\rangle :: tl \wedge g \in G \\ hd :: remove\_tasks(G, tl) & \text{if } l = hd :: tl \end{cases}$$

Stack-Free Tinycals Semantics

We will first describe the semantics of the tinycals that do not require neither pushing nor popping of stack levels. The semantics is shown in Table 4.4.

Tactic application Consider the first case of the tinycals semantics of Table 4.4. It makes use of the first component (denoted Γ) of a stack level, which represent the "current" goals, that is the set of goals to which the next tactic evaluated will be applied.

When a tactic is evaluated, the set Γ of current goals is inspected (expecting to find at least one of them), and the tactic is applied in turn to each of them in order to obtain the final proof status. At each step i the two sets C_i^o and G_i^c of goals opened and closed so far are updated. This process is atomic to the user (i.e. no feedback is given while the tactic is being applied to each of the current goals in turn), but she is free to cast off atomicity using branching.

Table 4.4: Basic tinycals semantics

$$\langle \text{“}\texttt{tactic}\text{”}\ \langle T\rangle :: c, \xi, \langle \Gamma, \tau, \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi_n, S'\rangle \qquad n \geq 1$$

where $[g_1;\cdots;g_n] = \mathit{get\_open\_goals\_in\_tasks\_list}(\Gamma)$

and $\begin{cases} \langle \xi_0, G^o_0, G^c_0\rangle = \langle \xi, [\,], [\,]\rangle & \\ \langle \xi_{i+1}, G^o_{i+1}, G^c_{i+1}\rangle = \langle \xi_i, G^o_i, G^c_i\rangle & g_{i+1} \in G^c_i \\ \langle \xi_{i+1}, G^o_{i+1}, G^c_{i+1}\rangle = \langle \xi', (G^o_i \setminus G^c) \cup G^o, G^c_i \cup G^c\rangle & g_{i+1} \notin G^c_i \\ \quad \text{where } \langle \xi', G^o, G^c\rangle = \mathit{apply\_tac}(T, \xi_i, g_{i+1}) & \end{cases}$

and $S' = \langle \Gamma', \tau', \kappa', t\rangle :: \mathit{close\_tasks}(G^c_n, S)$

and $\Gamma' = \mathit{mark\_as\_handled}(G^o_n)$

and $\tau' = \mathit{remove\_tasks}(G^c_n, \tau)$

and $\kappa' = \mathit{remove\_tasks}(G^c_n, \kappa)$

$$\langle \text{“}\texttt{;}\text{”} :: c, \xi, S\rangle \longrightarrow \langle c, \xi, S\rangle$$

$$\langle \text{“}\texttt{accept}\text{”} :: c, \xi, \langle \Gamma, \tau, \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$$

where $\Gamma = [\langle j_1, \texttt{Closed}\ g_1\rangle;\cdots;\langle j_n, \texttt{Closed}\ g_n\rangle] \qquad n \geq 1$

and $G^c = [g_1;\cdots;g_n]$

and $S' = \langle [\,], \mathit{remove\_tasks}(G^c, \tau), \mathit{remove\_tasks}(G^c, \kappa), t\rangle :: \mathit{close\_tasks}(G^c, S)$

$$\langle \text{“}\texttt{.}\text{”} :: c, \xi, \langle \Gamma, \tau, \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi, \langle [l_1], \tau, [l_2;\cdots;l_n] \cup \kappa, t\rangle :: S\rangle \qquad n \geq 1$$

where $\mathit{get\_open\_tasks}(\Gamma) = [l_1;\cdots;l_n]$

$$\langle \text{“}\texttt{.}\text{”} :: c, \xi, \langle \Gamma, \tau, l :: \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi, \langle [l], \tau, \kappa, t\rangle :: S\rangle$$

where $\mathit{get\_open\_tasks}(\Gamma) = [\,]$

After the tactic has been applied to all goals, the new set of current goals is created containing all the goals which have been opened during the applications, but not already closed. They are marked (using the *mark_as_handled* utility) so

that they do not satisfy the *unhandled* predicate, indicating that some tactic has been applied to them. Goals closed by side-effects are removed from τ and κ and marked as `Closed` in S.

Sequential Composition Since sequencing is implicitly handled by the semantics of "`tactic`" $\langle T \rangle$ and Γ, the semantics of "**;**" is simply the identity function. We have an explicit entry for sequential composition in the syntax of tinycal in order to preserve the parallelism with LCF tacticals.

Side-Effects Handling "`accept`" (third case in Table 4.4) is a tinycal used to deal with side-effects.

Consider for instance the case in which there are two current goals on which the user branches. It can happen that applying a tactic to the first one closes the second, removing the need of the second branch in the script. Using tinycals the user will never see branches she was aware of disappear without notice.

Cases like the above one are thus handled marking the branch as `Closed` (using the *close_tasks* utility) on the stack and requiring the user to manually acknowledge what happened on it using the "`accept`" tinycal, preserving the correspondence among script structure and proof tree.

Example 4.1 *Acknowledgement of Side-Effects*
Consider the following script snippet:

```
apply trans_eq;
 [ apply H
 | apply H1
 | accept ]
```

where the application of the transitivity property of equality to the conjecture $L = R$ opens the three conjectures $?_1 : L =?_3$, $?_2 : ?_3 = R$ and $?_3 :$ `nat`. Applying the hypothesis H instantiates $?_3$, implicitly closing the third conjecture, that thus has to be acknowledged.

□

Local De-Structuring Structuring proof scripts enhances their readability as long as the script structure mimics the structure of the intuition behind the proof. For this reason, authors do not always want to structure proof scripts down to the most deep leaf of the proof tree.

Example 4.2 *Flat script snippets in structured scripts*
Consider the following (template of) script snippet:

```
tactic1;
[ tactic2.
  tactic3.
| tactic4;
  [ tactic5
  | tactic6 ] ]
```

here the author is trying to mock-up the structure of the proof (two main branches, with two more branches in the second one), without caring about the structure of the first branch (where indeed a flat script snippet is used).

□

LCF tacticals do not allow non-structured script snippets to be nested inside branches. In the example above, they would only allow to replace the first branch with the identity tactic, continuing the un-structured snippet "tactic2. tactic3." at the end of the snippet, but this way the correspondence among script structure and proof tree would be completely lost.

The semantics of the tinycal "**.**" (last two cases of Table 4.4) accounts for local use of non-structured script snippets. When "**.**" is applied to a non-empty set of current goals, the first one is selected and become the new singleton current goals set Γ. The remaining goals are remembered in the third component of the current stack level (*dot's continuations*, denoted κ), so that when the "**.**" is applied again on an empty set of goals they can be recalled in turn. The locality of "**.**" is inherited by the locality of dot's continuation κ to stack levels.

Branching Tinycals Semantics

Table 4.5 describes the semantics of tinycals that require a stack discipline.

Table 4.5: Branching tinycals semantics

$\langle$ "[" $:: c, \xi, \langle [l_1;\cdots;l_n], \tau, \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$ $\quad n \geq 2$

where $renumber\_branches([l_1;\cdots;l_n]) = [l'_1;\cdots;l'_n]$

and $S' = \langle [l'_1], [\,], [\,], \mathtt{B}\rangle :: \langle [l'_2;\cdots;l'_n], \tau, \kappa, t\rangle :: S$

$\langle$ "|" $:: c, \xi, \langle \Gamma, \tau, \kappa, \mathtt{B}\rangle :: \langle [l_1;\cdots;l_n], \tau', \kappa', t'\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$ $\quad n \geq 1$

where $S' = \langle [l_1], \tau \cup get\_open\_tasks(\Gamma) \cup \kappa, [\,], \mathtt{B}\rangle :: \langle [l_2;\cdots;l_n], \tau', \kappa', t'\rangle :: S$

$\langle i_1,\ldots,i_n$ ":" $:: c, \xi, \langle [l], \tau, [\,], \mathtt{B}\rangle :: \langle \Gamma', \tau', \kappa', t'\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$

where $unhandled(l)$

and $\forall j = 1 \ldots n, \quad \exists l_j = \langle j, s_j\rangle, \quad l_j \in l :: \Gamma'$

and $S' = \langle [l_1;\cdots;l_n], \tau, [\,], \mathtt{B}\rangle :: \langle (l :: \Gamma') \setminus [l_1;\cdots;l_n], \tau', \kappa', t'\rangle :: S$

$\langle$ "$*$:" $:: c, \xi, \langle [l], \tau, [\,], \mathtt{B}\rangle :: \langle \Gamma', \tau', \kappa', t'\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$

where $unhandled(l)$

and $S' = \langle l :: \Gamma', \tau, [\,], \mathtt{B}\rangle :: \langle [\,], \tau' \cup get\_open\_tasks(\Gamma) \cup \kappa, \kappa', t'\rangle :: S$

$\langle$ "]" $:: c, \xi, \langle \Gamma, \tau, \kappa, \mathtt{B}\rangle :: \langle \Gamma', \tau', \kappa', t'\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$

where $S' = \langle \tau \cup get\_open\_tasks(\Gamma) \cup \Gamma' \cup \kappa, \tau', \kappa', t'\rangle :: S$

$\langle$ "`focus`" $[g_1;\cdots;g_n] :: c, \xi, \langle \Gamma, \tau, \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$

where $g_i \in get\_open\_goals\_in\_status(S)$

and $S' = \langle mark\_as\_handled([g_1;\cdots;g_n]), [\,], [\,], \mathtt{F}\rangle$

$:: close\_tasks(\langle \Gamma, \tau, \kappa, t\rangle :: S)$

$\langle$ "`done`" $:: c, \xi, \langle [\,], [\,], [\,], \mathtt{F}\rangle :: S\rangle \longrightarrow \langle c, \xi, S\rangle$

Branching Support for branching is implemented by "[", which creates a new level on the stack for the first of the current goals. Remaining goals (the current *branching context*) are stored in the level just below the freshly created one. There are three different ways of selecting them:

1. repeated uses of "|" consume the branching context in sequential order;
2. $i_1,\ldots,i_n$"**:**" enables multiple positional selection of goals from the branching context;
3. "$*$**:**" recalls all goals of the current branching context as the new set of current goals.

The semantics of "|", $i_1,\ldots,i_n$"**:**", and "$*$**:**" is implemented by the first five cases of Table 4.5.

Each time the user finishes working on the current goals and selects a new goal from the branching context, the result of her work (namely the current goals in Γ) needs to be saved for restoring at the end of the branching construct (the next occurrence of "]"). This is needed to implement the "re-flowing" behaviour of LCF tacticals.

Example 4.3 *Re-Flowing of Goals*

Consider the following (template of) script snippet:

```
tactic1;
[ tactic2
| tactic3 ];
tactic4
```

here, the goals resulting by the application of `tactic2` *and* `tactic3` are "re-flowed" together to create the goals set for `tactic4`.

□

The place we use to store the goals that need to be restoread later on is the second component of stack levels: the *to do list* (denoted τ). Each time a branching selection tinycal is used the current goals set (possibly empty) is appended to the to do list of the current stack level.

When "]" is used to conclude branching (fifth rule of Table 4.5), the to do list τ is used to create the new set of current goals Γ, together with the goals not handled during the branching. Note that this is a small improvement over LCF tactical semantics, where the `THENL` tactical requires as second argument an amount of branches equal to the number of goals returned by the application of the first tactic argument.

Focusing The pair of tinycals "`focus`"... "`done`" is similar in spirit to the pair "["... "]", but is not required to work on the current branching context. With "`focus`", goals located everywhere on the stack can be recalled to form a new set of current goals. On such a set the user is then free to work as she prefer, for instance branching, but is required to close all of them before invoking "`done`".

The intended use of "`focus`"... "`done`" is to deal with meta-variables and side-effects. The application of a tactic to a conjecture with meta-variables in the conclusion or hypotheses can instantiate the meta-variables making other conjectures false. In other words, in presence of meta-variables conjectures are no longer independent and it becomes crucial to consider and close a bunch or dependent conjectures together, even if in far away branches of the proof. In these cases "`focus`"... "`done`" is used to select all the related branches for immediate work on them.

Alternatively, "`focus`"... "`done`" can be used to jump on a remote branch of the tree in order to instantiate a meta-variable by side-effects before resuming proof search from the current position.

Note that using "`focus`"... "`done`", no harm is done to the proper structuring of scripts, since all goals the user is aware of, if closed, will be marked as `Closed` requiring her to manually "`accept`" them later on in the proof.

We will now give some highlights on the tinycals implementation in Matita.

4.4 Implementation

Tinycals have been implemented in the Matita proof assistant following closely the described semantics. This section describes the interesting aspects of the implementation and motivate some of the choices made in the user interface.

4.4.1 The Code

The implementation of tinycals in Matita comes in two macro-parts:

the engine which implements the semantics of tinycals and is abstracted over the parameters of our semantics (see Table 4.2);

the glue which consists in both the parser for tinycals concrete syntax and a set of graphical widgets used to present the evaluation status to the final user.

Figure 4.3 fits the parts of tinycals implementation in the software architecture of Matita (see Figure 2.9 and Figure 2.10).

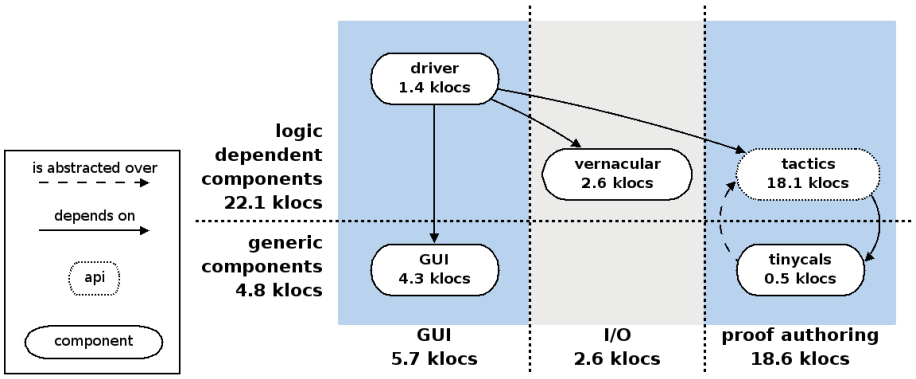

Figure 4.3: Components of the tinycals implementation in Matita

Engine According to the reusability philosophy of the implementation of MATITA, the tinycals engine (the `tinycals` component in the figure) is a fully reusable software component which follows closely the semantics described in this chapter. Indeed it mimics the generality of the semantics, being a functor (module `Tinycals.Make`) abstracted over a proof status representation with goals, and over a tactic application function. The dashed arrow on the `tactics` component represents such an abstraction, since in MATITA the proof status is contained in that component. This also means that the actual dependency on `tactics` is not on the whole 18.1 klocs, but only on less than 200 loc (module `ProofEngineTypes`); this is probably a flaw in the current implementation and should be fixed factoring out the proof status from `tactics`.

The dependency on the (instantiated) engine shown in Figure 4.3 is due to the reuse of the engine for implementing the tacticals which does not work on evaluation status, but only on proof status (see Section 4.4.2) and of course to the tactics actually using that kind of tacticals in their implementations.

The engine implementation is actually even more generic than the presented semantics, enabling the functor user to have tactics which consume as input a status of a given datatype and return as output a status of a different datatype, provided that appropriate projections on those status are provided. The module type input of the functor is shown in Figure 4.4.

Once provided such a module the functor returns a tinycals implementation consisting in algebraic datatypes representing each tinycal (values having these types should be built by the concrete syntax parser) and an `eval` function implementing the semantics we presented. The actual module type returned by the functor is shown in Figure 4.5.

Thanks to a careful design of the semantics before the actual implementation, the code of the engine is rather small, being only 500 lines of code. The reader interested in the engine implementation should start looking at the `eval` function of the functor, which follows closely the semantics rules presented in this chapter.

```
module type Status =
sig
  type input_status
  type output_status
  type tactic

  val id_tactic: tactic
  val mk_tactic: (input_status -> output_status) -> tactic
  val apply_tactic: tactic -> input_status -> output_status

  val get_status: input_status -> ProofEngineTypes.status
  val get_proof: output_status -> ProofEngineTypes.proof
  val goals:
    output_status -> goal list * goal list (** opened, closed *)
  val set_goals:
    goal list * goal list -> output_status -> output_status
  val get_stack: input_status -> Stack.t
  val set_stack: Stack.t -> output_status -> output_status

  val inject: input_status -> output_status
  val focus: goal -> output_status -> input_status
end
```

Figure 4.4: Tinycals functor: input module type

Glue The glue code is spread in the `vernacular`, `driver`, and `GUI` components. The two former components are used respectively to parse the concrete syntax used in Matita scripts and to map each syntax fragment to invocations of the (instantiated) tinycals engine.

The `GUI` component implements the script management authoring interface of Matita, only a minimal part of it has been changed to present the current evaluation status to the user. It will be discussed in Section 4.4.3.

```
module type C =
sig
  type input_status
  type output_status
  type tactic

  type tactical = Tactic of tactic | Accept

  type t =
    | Dot | Semicolon
    | Branch | Shift | Pos of int list | Wildcard | Merge
    | Focus of goal list | Unfocus
    | Tactical of tactical

  val eval: t -> input_status -> output_status
end
```

Figure 4.5: Tinycals functor: output module type

4.4.2 Implementing Tacticals with Tinycals

Tacticals play two different roles in a proof assistant. They can be used both in scripts by final users and in tactic implementations by tactic implementors. As a matter of fact, in proof assistants like Coq and Matita at least one tactical among sequential composition and branching is usually used in the implementation of each derived tactic.

Tacticals operate on proof status, while tinycals operate on evaluation status. This is welcome when tinycals are used in scripts, since the additional information kept in the evaluation status is the rich intermediate state we want to present to the user.

On the contrary, this datatype change does not allow a painless replacement of tacticals with tinycals in the implementation of derived tactics. Indeed, tactics

are usually implemented on top of the proof status only and additional extra-logic information (like the evaluation status) are arguably not available in tactics code. Thus we are immediately led to consider if it is possible to express tacticals in terms of tinycals, in order to avoid an independent re-implementation of similar operations.

The answer is positive under additional assumptions on the abstract data type of proof status. Intuitively, we need to define two "inverse" functions to embed a proof status, a goal, and a code in an evaluation status (let it be *embed*) and to project an evaluation status to a proof status and two lists of opened and closed goals (let it be *proj*). Our goal is thus to define two functions having types:

$$\begin{aligned} \mathit{embed} &: \mathit{code} \times \xi \times \mathit{goal} \to \mathit{status} \\ \mathit{proj} &: \mathit{status} \to \xi \times \mathit{goal}\ \texttt{list} \times \mathit{goal}\ \texttt{list} \end{aligned}$$

Once the two functions are implemented, we can express sequential composition and branching as follows:

$$(t_1; t_2)(\xi, g) = \mathit{proj}(\mathit{eval}(\mathit{embed}([t_1; \text{“;”}; t_2], \xi, g))) \quad (4.1)$$

$$(t; [t_1 | \ldots | t_n])(\xi, g) = \mathit{proj}(\mathit{eval}(\mathit{embed}([t; \text{“[”}; t_1; \text{“|”}; \ldots; \text{“|”}; t_n; \text{“]”}], \xi, g))) \quad (4.2)$$

where *eval* is the transitive closure of $\longrightarrow$. For each status S the code of the status $\mathit{eval}(S)$ is empty.

The idea behind the *embed* function is to create an evaluation status with a single level stack, in which only the goal input of the function is active. *embed* can thus be easily defined as:

$$\mathit{embed}(c, \xi, g) = \langle c, \xi, [\langle g, [\,], [\,], \texttt{F}\rangle]\rangle$$

To define the *proj* function, however, we need to be able to compute the set of goals opened and closed by $\mathit{eval}(\mathit{embed}(c, \xi, g))$ for any given code c, proof status ξ, and selected goal g. While open goals can be easily computed using the *get_open_goals_in_status* utility of Section 4.3.2, to compute the latter the information stored in an evaluation context is not enough.

We say that tactics *do not reuse goals* whenever closed goals cannot be re-opened (remember that a goal is just an handle to a conjecture, not the conjecture itself). Concretely, it is possible to respect this property in the implementation by keeping a global counter that represents the highest goal index already used. When a tactic opens a new goal it picks the successor of the counter, that is also incremented.

When tactics do not reuse goals it is possible to determine the goals closed by a sequence of evaluation steps by comparing the set of open goals at the two extremes of the sequence. To make this comparison it is possible to add to the proof status abstract data type a method that returns the set of opened goals.

Let *diff* be the function that given two proof status ξ and ξ' returns the set of goals that were open in ξ and are closed in ξ'. For each proof status ξ the $proj_\xi$ function is defined as:

$$proj_\xi([\,], \xi', S) = (\xi', get\_open\_goals\_in\_status(S), diff(\xi, \xi'))$$

The function $proj_\xi$ has now to be used in Equation (4.1) and Equation (4.2) in place of $proj$.

4.4.3 User Interface

Tinycals would be worthless without a way to present evaluation status to the user. The current solution implemented in the MATITA user interface is shown in Figure 4.6.

When tinycals where added to MATITA, we already had a Proof General like user interface with script management (on the left of Figure 4.6) and a tabbed representation of the set of open conjectures (on the right) as sequents, using metavariable indexes as labels. What the user was missing to work with tinycals was a visual representation of the stack.

Our choice has been to represent the current branching context as *tab label annotations*: all goals in the current goals set have their labels typeset in boldface, goals of the current branching context have labels prepended by $|_n$ (where n is their

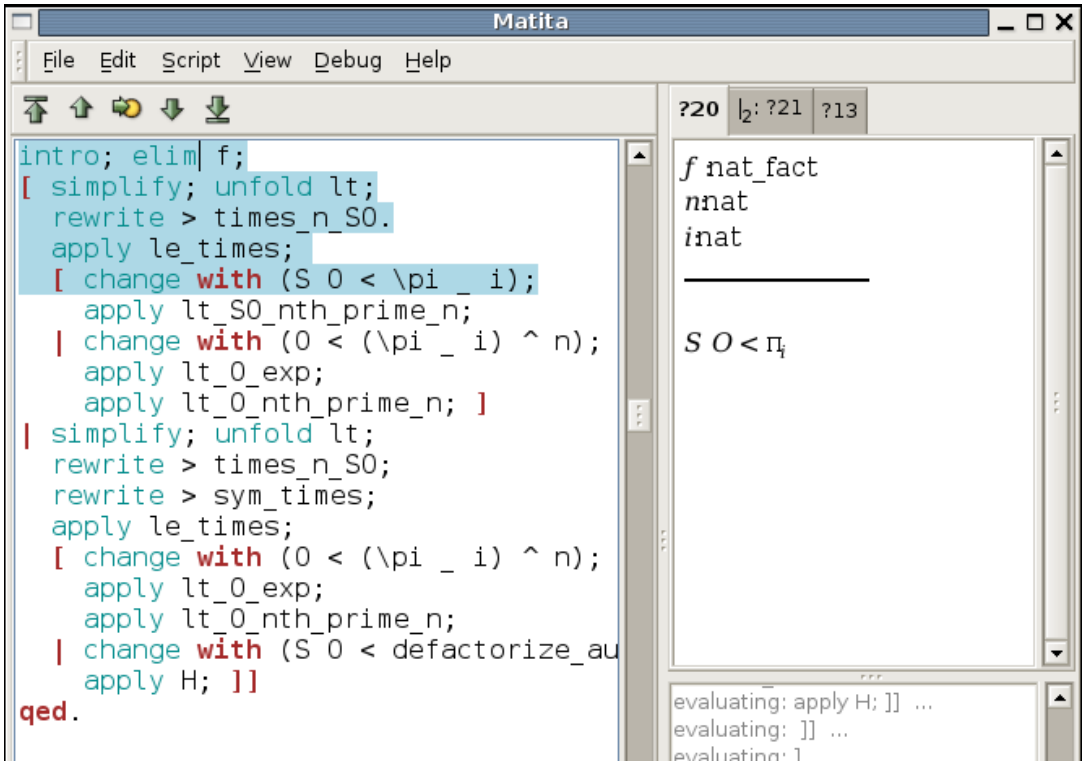

Figure 4.6: Evaluation status presentation in the MATITA user interface

positional index), and goals already closed by side-effects have strike-through labels like: ~~?*n*~~.

For instance in Figure 4.6, the only goal (in bold-face) the next tactic will be applied to is ?20 (i.e. $\Gamma = [\langle 1, \texttt{Open}\ 20\rangle]$), while goal ?21 will be selected by the next "|" tinycal.

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 she might need there. She indeed can see all meta-variable indexes (in case she wants to "`focus`") and all the positional indexes of goals in the current branching context (for $i_1,\ldots,i_n$ "**:**"and "**∗:**"). Yet, this user interface choice minimizes the drift from the usual way of working with Proof General like interfaces.

4.5 LCF Tacticals Not Accounted For

Of LCF tacticals, we have considered so far only sequential composition and branching. With the exception of `IDTAC`—which has been omitted just for its triviality—it is worth discussing the remaining ones, in particular "`try`", "||" (try-recover), and "`repeat`".

The "`try`" and "||" tacticals usually occur in scripts for two different purposes. The most common one is after sequential composition, as for example in: "t_1 ; `try` t_2" or "t_1 ; t_2 || t_3" (the latter should be parsed as "t_1 ; (t_2 || t_3)"). Here the idea is that the user knows that t_2 can be applied to some of the goals generated by t_1 (and t_3 to the others in the second case). So she is faced with two possibilities: either use branching and repeat t_2 (or t_3) in every branch, or use sequential composition and backtracking (encapsulated in the two tacticals).

Tinycals offer a better solution to either choice by means of the projection and wild card tinycals: "t_1 ; [$i_1, \ldots, i_n : t_2$ | $* :$ t_3]". The latter expression is not only more informative to the reader, but it also computationally more efficient since it avoids the (potentially costly) application of t_2 to several goals.

The second common usage of "`try`" and "||" is inside a "`repeat`", as in: `try` t, t_1 || t_2. Is it possible to provide a non structured version of both in the spirit of tinycals in order to allow the user to write and execute t step by step inspecting the intermediate evaluation status?

The answer is negative as we can easily see in the following example of the simplest case, that of `try` t.

Example 4.4 *Fine-Grained Execution of "`try`"*

Consider the statement t ; `try` $(t_1; t_2)$ where sequential composition is supposed to be provided by the corresponding tinycal.

Let t open two goals and suppose that "`try`" is executed atomically (just the keyword, not its argument) so that the execution point is then placed just before t_1. When the user asks for the evaluation of t_1, t_1 can be applied as expected to both goals in sequence. Let ξ be the proof status after the application of t and let ξ_1 and

ξ_2 be those after the application of t_1 to the first and second goal respectively. Let now the user execute the identity tinycal "**;**" followed by t_2 and let t_2 fail over the first goal.

To respect the intended semantics of the tactical, the status ξ_2 should be *partially* backtracked to undo the changes from ξ to ξ_1, preserving those from ξ_1 to ξ_2.

If the system has side-effects the latter operation is undefined, since t_1 applied to ξ could have instantiated meta-variables that controlled the behavior of t_1 applied to ξ_1. Thus undoing the application of t_1 to the first goal also invalidates the previous application of t_1 to the second goal.

Even if the system has no side-effects, the requirement that proof status can be partially backtracked is quite restrictive on the possible implementations of a proof status. For instance, a proof status cannot be a simple proof term with occurrences of meta-variables in place of conjectures, since backtracking a tactic would require the replacement of a precise sub-term with a meta-variable, but there would be no information to detect which sub-term.

The naive solution of implementing partial backtracking by means of a full backtrack to ξ followed by an application of t_1 to the second goal only, does not conform to the spirit of tinycals. With this implementation, the application of t_1 to the second goal would be performed twice, sweeping the waste of computational resources under the rug.

□

The only plausible solution consists of keeping all tacticals not accounted for by tinycals fully structured as they are now. The user that wants to inspect the behavior of t ; `try` t_1 before that of t ; `try` $(t_1$; $t_2)$ is obliged to do so by executing atomically `try` t_1, backtracking by hand and executing `try` $(t_1$; $t_2)$ from scratch.

Similar conclusions can be reached for the other remaining tacticals. For this reason in the syntax given in Table 4.1 the production $\langle B \rangle$ lists all the traditional tacticals that are not subsumed by tinycals. Notice that atomic sequential composition and atomic branching (as implemented in the previous section) are also listed since tinycals cannot occur as arguments of a tactical.

4.6 Related Work

Various presentations of the semantics of tacticals has been given in the past. The first presentation has been given in [41] by Gordon et al. Although a larger set of tacticals than that considered in this chapter was described in their work, the problem of inspecting inner proof status during tacticals execution was not considered. Script management based user interfaces were not available at the time, as well as meta-variables and hence tactics with side-effects.

In [57], Kirchner described a small step semantics of Coq tacticals and PVS strategies. Despite the minor expressive advantages offered by tinycals over the corresponding Coq tacticals (like "`focus`", "`*:`", $i_1,\ldots,i_n$"`:`", the less constrained use of "[", and the structuring facilities implemented by "`.`" and "`accept`"), the formalization of tinycals is more general and we believe that it can be applied to a large class of proof assistants. In particular our semantics only assumes an abstract proof status and a very general type for tactic applications, while in [57] a very detailed API for proof trees was assumed. This does not come as a surprise, since the work of Kirchner pursued different aims than ours. His aim was indeed providing a rigorous documentation of Coq's tactics and tacticals in an unifying semantic framework with PVS strategies. Our hope is instead to show how to concretely improve the user experience with a traditional tool like tacticals in modern provers.

Delahaye in [35] described $\mathcal{L}_{tac}$, a powerful meta-language which can be used both by final users and tactics implementors to write small automations at the proof language level. $\mathcal{L}_{tac}$ is way more powerful than tinycals, featuring constructs typical of high-level programming and defining their reduction semantics. However, since again its aim was different, $\mathcal{L}_{tac}$ fails to address the interaction problem that tinycals do address.

Two alternative approaches for authoring structured HOL scripts have been proposed in [104] and [105]. The first approach, implemented in Syme's TkHOL, is similar to the one presented in this paper but lacks a formal description. Moreover, unlike HOL, we consider a logic with meta-variables 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 tinycals like "`focus`" and $i_1,\ldots,i_n$"`:`" which were not needed 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 meta-variables because they are not allowed in the statements of lemmata.

4.7 Conclusions

In this chapter we presented a user interaction widget part of the authoring interface of the Matita proof assistant: the tactical language called tinycals. The widget it composed by the formal language (its syntax and semantics) and by its implementation, currently part of Matita.

Tinycals mimic some of the LCF tacticals so widespread in state-of-the-art proof assistants. Tinycals advantages over LCF tacticals is that their syntax is non-structured and their evaluation proceeds step by step, enabling the user to start execution of a structured script before it is completely written in the prover. Intermediate proof status can be inspected and tactics with side-effects are supported as well. The neat result is better integration with user interfaces based on script management, the paradigm incarnated by the famous Proof General user interface. Unfortunately, not all LCF tacticals can be "converted" to corresponding tinycals, if proper support for side-effects is a requirement.

The implementation consists of an evaluator implementing the semantics of tinycals and of a user interface for presenting evaluation status to the final users. The evaluator is generic and we believe it can be reused as is in the implementation of other provers. Its generality is reflected by its functorial nature: a system able to instantiate the parameters of the presented semantics with its proof status representation will probably be able to instantiate the functor obtaining a working implementation.

Tinycals are used in the MATITA proof assistant for the ongoing development of its standard library. Users experienced with other proof assistants, in particular Coq, consider them a serious improvement in the proof authoring interface. This is not a big figure (our users at the time of writing are just the members of our research team), but is enough to motivate our work on them, hoping to see them adopted soon in other systems.

Original Contributions

Claudio Sacerdoti Coen raised the discussion on the limitation of tacticals and suggested the idea of a non-structured syntax for them.[7] Developing that idea, the syntax and formal semantics of tinycals was mainly developed by this thesis author as a joint work with Enrico Tassi, then reviewed by the three of us for [95]. The first implementation for MATITA was written for scratch by this thesis author and is still maintained by him.

Related Publications

Part of the work described in this chapter has been previously published in the following papers:

- Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli.
  Tinycals: step by step tacticals. [95]
  In Proceedings of UITP'06: Seventh Workshop on User Interface for Theorem Provers 2006[8], Electronic Notes in Theoretical Computer Science. Elsevier Science, 2006. To appear.

[7] Private communication with this thesis author

[8] `http://www.ags.uni-sb.de/~omega/workshops/UITP06/`

Chapter 5

Extensible yet Meaningful Mathematical Notation

According to the Webster dictionary, a *notational system* (or *notation* for short) is:

> any particular system of characters, symbols, or abbreviated expressions used in art or science, to express briefly technical facts, quantities, etc. Esp., the system of figures, letters, and signs used in arithmetic and algebra to express number, quantity, or operations.

The mathematic vernacular, or *mathematical notation*, is the most peculiar example of notational system and plays the role of a lingua franca among mathematicians. In spite of the meaning of symbols, the set of symbols itself is quite limited and visually well-known by all mathematicians.

In the field of interactive theorem proving, "notation" is used to refer to a set of mappings from the terms of the logical foundation a system is based on, to some markup consumable by the final user for input and output of formulae. *User-extensible notation* is indeed a feature easily found in theorem provers, which let the user extend the notation mappings of the system in order to improve the input and rendering of newly defined concepts. Such a feature helps the user abstracting over details that may hinder proof discovery on non-trivial proof status and, in the web era, it is a requirement for high-quality publishing of formalized concepts on the web.

Posing the additional requirement that the definable notation should be as close as possible to the mathematical notation, turns designing and implementing such a feature in a task far from being trivial. Mathematical notation indeed is a structured, open, and ambiguous language. In order to properly support it one must necessarily take into account presentational as well as semantic aspects. The former are required to create a familiar, comfortable, and usable interface to interact with. The latter are necessary in order to process the information meaningfully.

In this chapter we investigate a framework for dealing with mathematical notation in a meaningful, extensible way. The framework blends together semantic and presentation aspects introducing the novel concept of *meaningful notation*. The architecture of the framework is generic and builds upon well-known concepts and widely-used technologies. We believe it can be easily adopted by Mathematical Knowledge Management software applications other than proof assistants.

As an assessment of the effectiveness of the framework we present an instantiation of its architecture to the field of interactive theorem proving, in particular to the MATITA proof assistant. We also discuss some of the additional features that have been implemented on top of it in the MATITA authoring interface, most notably: *hypertextual browsing* and *semantic selection*

5.1 Preliminaries

5.1.1 On the Relevance of Notation in Theory Development

Mathematical notation plays a fundamental role in mathematical practice: it helps expressing in a concise and symbolic fashion formulae of arbitrary complexity. Its availability in proof assistants like MATITA is no exception. Formal mathematical knowledge management indeed requires to encode mathematical formulae as a term of a calculus having only a restricted toolbox of syntactic constructions.

Rather informally, we call *notational support* the feature of a (formal) mathematical knowledge management application of supporting the user in reading and

writing terms of the logical foundation of the application, using a concrete syntax close to the usual mathematical notation. Similar features have historically been found in several theorem provers starting from LCF who used the ability of the Edinburgh ML top-level to define infix operators, ranging to modern systems like Mizar or Coq.

Notational support is not of interest from the point of view of the logician interested in the calculus itself since, for example, it does not really matter how the application of a binary sum over natural number is *concretely* written with respect to its two arguments (though it is interesting to observe that conventions as the infix use of operators are widespread among logicians as well). The expressiveness and other aspects of the calculus are not affected by such a choice. However, it affects in several ways the work of the final user of a theorem prover implemented on top of a given calculus.

We illustrate some of this aspects with the help of Figure 5.1, where the same conjecture—encountered during the proof of the distributivity of times over minus on natural numbers using Matita—is shown both with (lower part of the figure) and without (upper part) notational support in effect.

At a first glance, it can be observed that snippets of the sequent with notational support in effect are faster to input to the system, being more compact in term of the amount of characters. This advantage of using notational support stands independently on the actual way used by the system to input formulae (e.g. textual typing, palette based editor, copy & paste, ...).

Next, in spite of looking probably nice to Lisp fans, the terseness of the sequent rendered with notational support helps in focusing on the current proof goal, instead of getting lost in the details of the formalization of algebraic operators (though those details were important when they have been formalized in the first place). The advantages of this aspect tend to become even more evident when it comes to numbers in systems with no built-in numbers in their logical foundation. In such systems indeed numbers are often formalized as inductive types in k-ary bases and hence their built-in representation has a length proportional to the represented

x : nat
y : nat
z : nat
H : le z y
Hcut : eq nat
(plus (times x (minus y z)) (times z))
(plus (minus (times x y) (times x z)) (times x z))
---
eq nat (times x (minus y z)) (minus (times x y) (times x z))

x : nat
y : nat
z : nat
H : $z \leq y$
Hcut : $x * (y - z) + x * z = x * y - x * z + x * z$

$x * (y - z) = x * y - x * z$

Figure 5.1:
Textual renderings of the same conjecture, rendered using (above) and not using (below) notational support

number!

Finally, the De Bruijn factor [114] of scripts written with notational support is smaller than that of scripts written without it. For those interested in the "marketing" of formal mathematical knowledge management applications, good notational support able to make terms look like formulae consumed by other mathematical mainstream applications (like Mathematica or Maple, but also TEX) is a plus too.

For all these reasons, *notation development* can nowadays be considered a new

activity part of the *tool development* phase presented in [2]. It is a part of theory development: the act of formalizing in a theorem prover a set of related definitions and theorems. Tool development consists in creating the "tools" which will help in the formalization work. Example of such tools are domain specific tactics and decision procedures, proving ancillary theorems (like recursors or induction principles), and so on.

Looking at even small sized developments of systems supporting extensible notation it is evident how creating notation for just defined concepts is becoming a best practice of interactive theorem proving. Usually, before starting proving theorems about new concepts, notation for those concepts is defined in order to ease writing and reading about them in some concrete syntax.

Before moving to how notational framework are actually implemented in interactive theorem provers we need to better understand which kind of transformations they are in charge of. In the next section we therefore start presenting the different encoding of mathematical formulae such a system have to deal with. Three such encodings will be exploited by our notational framework.

5.1.2 Encoding of Mathematical Formulae

Any software application dealing with mathematical objects (formulae, proofs, scientific articles, ...) works on some internal encoding of them. Several possible encodings do exist, with advantages and disadvantages. An agreed upon classification of such encodings is on the axis of machine understandability [1, 74], an informal figure trying to grasp how much of the "meaning" of a given mathematical object is encoded in the machine, and can thus be exploited to perform "meaningful" operations. A pictorial representation of such a classification is reported in Figure 5.2.

Each block in the figure represents an *encoding level*; levels of which the machine has higher understandability are reported toward the top of the figure. Arrows

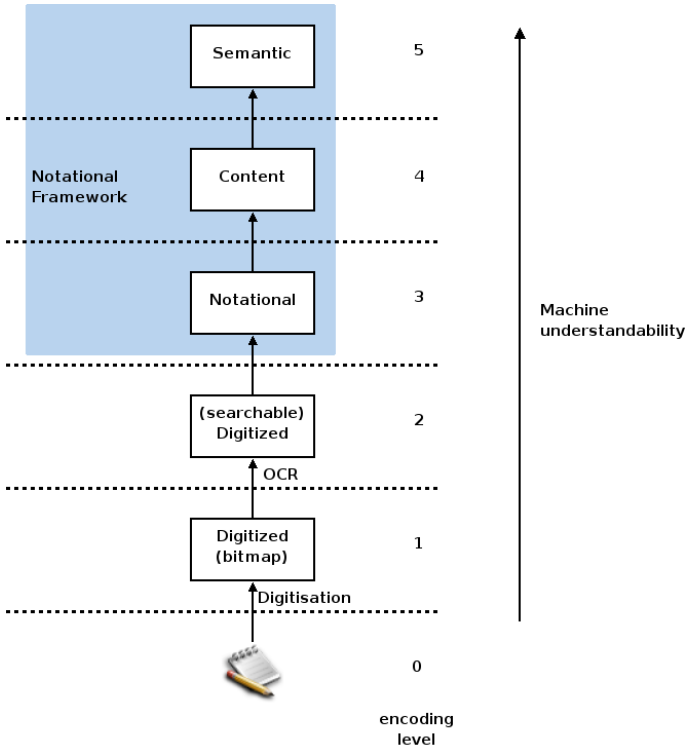

Figure 5.2: Possible encodings of mathematical objects

among blocks represent techniques currently used to bump the machine understanding of some object.

pen and paper at the bottom there is the non-encoded pen and paper version of some mathematical objects (level 0).

digitized bitmap next to it there is the digitized bitmap level (level 1), which can be obtained for example by scanning a printed or written document. The operation which can be done directly at this level are usually not related to mathematics, like printing.

searchable digitized level 2, the searchable digitized level, is the common encoding level used for exchange of scientific literature; example of formats encoding objects at this level are PostScript and PDF. It might improve over level 1 by

enabling full text searches over the digitized content. It is obtained from level 1 by Optical Character Recognition (OCR) techniques, which may be adapted to the specific field of mathematics in order to obtain better results (see for example [74], one of several works on this subject in the MKM community).

notational the next level is the *notational level*, often also referred to as *presentation level*. Objects, and in particular formulae, at this level are encoded on the basis of their visual rendering. The most prominent examples of markup languages encoding at this level are TEX [58] and MathML Presentation [66]. First class citizen of such languages are atomic literals like numbers, operators, and identifiers, together with constructors of layout schemata commonly found in mathematical notation like radicals, fractions, subscripts, superscripts, ...

Interestingly enough, the notational level is thought to be built on top of a finite set of constructors, capturing the habit of mathematicians to reuse a fixed vocabulary to provide notation for new concepts. For this reason we claim it is appropriate to offer as the language for describing presentation of formulae a language with a fixed set of constructors.

The functional interesting aspect of the notational level is that it is meant to be used for interaction with the user. User input formulae in a system at this level using some familiar notation (exploiting as a vehicle some concrete syntax, not necessarily textual). The system renders the formulae to the user in some notation, encoded at this level.

content level 4, or *content level* is used to encode the structure and, to a limited extend, the semantics of mathematical objects and in particular of formulae. MathML Content and OpenMath [100] are examples of markup languages that encode formulae at this level. Intuitively, formulae at this level can be though at abstract syntax trees (ASTs) having operators as inner nodes with children for each of their arguments.

No attempt is made to explain directly at this level the meaning of operators though. Such a task is delegated to higher level of machine understandability

via content dictionaries which map operators of this level to semantic explanations understandable by particular applications. For this reason the content level is the most effective vehicle of interoperability across mathematical knowledge management applications not sharing semantic foundations.

semantic finally, objects at the *semantic level* are those which the application has the deepest understanding of and on which it can better perform *computations*. No standardized languages do exist for encoding the semantic level, since the actual meaning of a mathematical object is intrinsically application specific: a theorem prover may consider a term of some calculus to be the meaning of a formulae, while a computer algebra system may be set with an abstract syntax tree containing symbols he understands.

In the particular case of MATITA (and of Coq which shares with it the same logical foundation) the semantic level is the Calculus of (Co)Inductive Constructions [113] (CIC), and formulae encoded at this level are terms of the calculus. A standardized—meaning that it is system agnostic and can be understood by both systems—language for this particular case does actually exists and is an XML encoding of CIC terms.

Examples of computations performed on objects encoded at this level in the fields of computer algebra systems and theorem provers are evaluation, simplification, automatic (dis-)proving, and type-checking.

The notational, content, and semantic levels are the levels of interest for our notational framework. In the next section we will present the requirements we set forth before designing it. From now on we will only discuss notation for formulae, as it is the main area of application of our notational framework.

5.2 Meaningful Notation

We can now recast the "notation" folklore of interactive theorem proving of Page 117 in a setting where the various possible encoding of a formulae in such a system are taken into account. Doing so lead us to define notation as a set of bi-directional mappings from formulae encoded at the notational level to formulae encoded at the semantic level.

Our aim is to design and implement a reusable framework for dealing with such kind of notation. The most natural architecture for such a framework is a layered one, where the same mathematical formulae is encoded at different encoding levels, according to the activities it is subjected to. The layers are connected with each other by pairs of transformations among neighborhood levels, and the encodings must be kept synchronized accordingly. In this sense we distinguish notation, which is a purely presentational tool, from *meaningful notation* that blends together both presentational and semantic aspects.

From the perspective of the framework designer, the fact that notation is extensible is a source of considerable additional complexity. It means that the layers cannot be fully described *a priori*, and that it should be possible to dynamically update their connections as the system is enriched with new notation and new mathematical concepts. It should be noted that a system supporting extensible notation in an exclusively presentational fashion is much simpler but also of limited use.

5.2.1 Requirements

We will now present a set of features we consider as characterizing a framework for meaningful notation. We claim a framework implementing them is suitable to be used for supporting notation in interactive theorem provers and, more generally, in software application willing to deal with meaningful notation.

Requirement 5.1 (extensibility) *The framework must permit its users to define their own notation in an incremental way, using a basic set of primitive constructs along with all the notation which has been defined earlier.*

Extensibility reflects common practice of notation development in mathematics and, as a special case, in theory development. Starting from a core set of primitives, mathematicians develops new concepts and new mappings for them, possibly reusing previously defined mappings.

Requirement 5.2 (remote control) *Notation should provide handles for enabling indirect manipulation of the (possibly hidden) information encoded at the content and semantic levels.*

Being abstraction over details one of the primary use of notation, the notational level often hides information to the final user. Remote control states that this is permitted, but at the same time requires that at the notational level there should be enough information (not necessarily visually presented to the user) to actually operate on the content and semantic level encodings.

Requirement 5.3 (ambiguity) *The framework must tolerate (and encourage) ambiguity, which is common practice in traditional mathematical artifacts.*

As we saw in Chapter 3 mathematical notation is usually ambiguous with respect to the detailed semantic level. The ambiguity requirement states that ambiguity handling should not be incompatible with other aspects of the notational framework.

Requirement 5.4 (interoperability) *The framework must not hinder communication with other software, notwithstanding the availability of agreements on the semantic level encoding.*

Interoperability states that the notational framework should be a social player, avoiding to add constraints on the capabilities of a system using it to communicate with other software. Note that the presence of a content level encoding kept synchronized with encodings at the notational and semantic level is enough to fulfill the interoperability requirement.

5.2.2 A Software Architecture for Meaningful Notation

In this chapter we show how a generic framework implementing the features of meaningful notation can be designed, and we demonstrate the effectiveness of our approach in the context of the MATITA proof assistant. The architecture of such a framework at work is depicted in Figure 5.3.

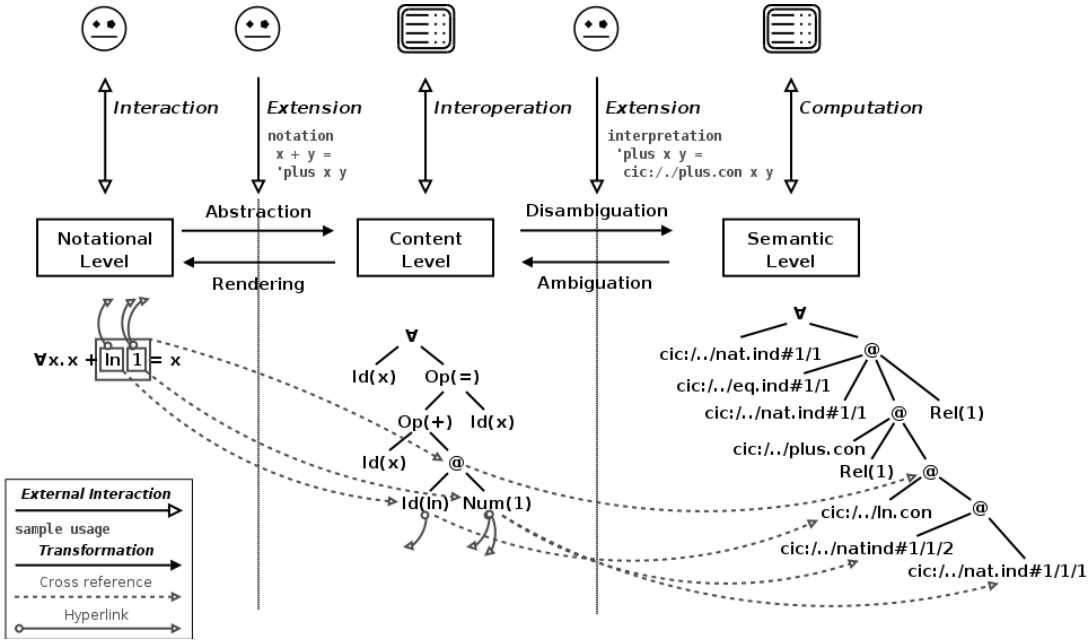

Figure 5.3: Architecture of the notational framework

The figure can be commented on several axis. The bottom part of the figure contains samples of the same formula (which in TEX would look like: "$\forall x.x + \ln 1 = x$") encoded at three different encoding levels: notational (on the left), content (in the middle), semantic (on the right). The notational level shows a hypothetical rendering, together with highlights of some of the layout schemata used, most notably rows. The content level sample is an abstract syntax tree of the structure of the formula. The semantic level is again an abstract syntax tree, but this time of a CIC term, where HELM [6] URIs are used to refer to previously defined mathematical concepts. From the explicit choice of CIC, the reader can deduce that the lower part

of the figure is already instantiated to MATITA.

Pieces of the various encodings can be associated with additional information, most notably with hyperlinks and cross references. *Hyperlinks* are one-to-many mappings, similar in other respects to web hyperlinks: they have a pieces of markup as anchors and URIs as targets. In the instantiation of the framework to MATITA we use HELM URIs to refer to mathematical concepts available in the HELM library. Typically they link objects to their definitions.

Cross references links together pieces of encodings across levels, having anchors living in a lower encoding level and targets in an upper one. Intuitively, a cross reference remember which was the source who generated a particular part of an encoding during the output of a formula.

The middle part of the figure shows the relationship among encoding levels in the notation framework and in particular defines the transformations used to traduce encodings to each other. Transformations among levels are initiated by the need of interaction between the user and the application. Left-to-right transformations are initiated by the need of input a formula, while right-to-left by the need of output one. Both input and output are implemented by sequences of two transformations.

Input first requires the formula written by the user to be converted from the notational level to the content abstract syntax tree; *abstraction* takes care of this conversion. Then, *disambiguation* is used to recover the fully semantic encoding of the formula. Conversely, during output a formula is first stripped of any application-specific semantic information by *ambiguation*, and then converted by *rendering* to a notational encoding which can be directly presented to the user.

Transformations are pairwise driven by sets of bi-directional rules: a set of *notational equations* drives abstraction and rendering, while a set of *interpretations* drives disambiguation and ambiguation.

Finally, the upper part of the figure shows the possible interactions of the various architecture components with the external (from the point of view of the framework) world. Interaction with the user is performed for input and output of formulae and

for extending the sets of notational equations and interpretations. Interoperation is performed on the content level encoding, and computation on the semantic one.

The architecture presented in Figure 5.3 fulfills all the requirements of meaningful notation (see Section 5.2.1):

extensibility is achieved permitting the user to dynamically enrich at run-time the sets of notational equations and interpretations;

remote control is accounted for by cross references, which provide handles to the content level and, transitively, to the semantic one. Hyperlinks are a plus, providing direct handles to the semantic encoding of concepts available in a given knowledge base;

ambiguity during output ambiguity is properly accounted by the framework itself: the final form of a formula shown to the user may be ambiguous with respect to the semantic encoding, but cross reference permit to work on it resolving ambiguities. During input ambiguity is delegated to the disambiguation phase, which we will discuss in Section 5.5.1;

interoperability is possible by direct importation and exportation of the content level encoding. Since transformations are fully decoupled it is possible to import content level formulae produced by other applications to further transform it as well as exporting the result of abstraction and ambiguation for consumption by other applications.

We will now present the formalization of the generic part of the framework.

5.3 Syntax and Semantics of Meaningful Notation

5.3.1 Content and Presentation Languages

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

Table 5.1 and Table 5.2 show the grammars for two streamlines languages of *presentation* and *content* expressions capturing the essence of notation.

Table 5.1: Generic syntax of presentation expressions

E^{p}	::=		(**presentation expression**)
		x	(identifier)
	|	$l@H$	(literal)
	|	$A\{E^{\mathsf{p}}\}$	(annotation)
	|	$L[E_1^{\mathsf{p}}, \ldots, E_n^{\mathsf{p}}]$	(layout)
	|	$B[E_1^{\mathsf{p}} \cdots E_n^{\mathsf{p}}]$	(box)
	|	α	(variable)

Table 5.2: Generic syntax of content (E^{c}) expressions

E^{c}	::=		(**content expression**)
		x	(identifier)
	|	$s@H$	(symbol)
	|	$A\{E^{\mathsf{c}}\}$	(references)
	|	$C[E_1^{\mathsf{c}}, \ldots, E_n^{\mathsf{c}}]$	(constructor)
	|	α	(variable)

The two grammars are parametric in the following sets:

- a set of *layout schemata* L representing basic constructs of mathematical notation such as fractions, square roots, vectors, and so on;
- a set of *box schemata* B for annotating presentation expressions with line-breaking hints;
- a set of *identifiers* x;
- a set of *literals* l representing special characters and numbers;
- a set of *symbols* s representing the basic elements in the ontology language of the content level (in MathML Content this set is predefined, in OpenMath it is completely unspecified, in either case it is open-ended and can be extended at will);
- a set of *constructors* C of the content level for building compound objects such as sets, lists, functions, relations.

Literals and symbols are annotated with sets of *hyperlinks* H. We write l and s for $l@\emptyset$ and $s@\emptyset$ respectively. Both presentation and content expressions may be annotated with sets of cross references A. We omit the annotations p and c when it is clear that we are talking about presentation and content expressions, respectively.

Layout and Box Schemata The choice of layout schemata used in our instantiation of the architecture to MATITA is shown in Table 5.3, where each supported layout schema is reported with its arity, i.e. the amount of presentation expression arguments it has to be applied to. The set of schemata is almost the same of MathML Presentation [66] with an extra additional distinction between fractions and vertical juxtaposition.

Similarly, our choice of box schemata is reported in Table 5.4. It has been inspired by previous work on pretty-printers [34]. Each box kind describes where to place on a bi-dimensional canvas (the rendered form of) presentation expressions appearing

Table 5.3: Layout schemata of presentation expressions

L ::=		(**layout schemata**)	**arity**
|	*sub*	(subscript)	2
|	*sup*	(superscript)	2
|	*below*	(underscript)	2
|	*above*	(overscript)	2
|	*frac*	(fraction)	2
|	*row*	(horizontal juxtaposition)	$n,\ n \geq 1$
|	*atop*	(vertical juxtaposition)	2
|	*sqrt*	(square root)	1
|	*root*	(indexed root)	1

as its children. Horizontal and vertical boxes always place their children in a single visual row or column respectively. Horizontal-vertical boxes try to arrange children in a single row, if there is not enough physical space on the screen to do that they fall back to the vertical box behaviour, breaking where break-point hints occur in their children. Horizontal or vertical boxes implements the rendering semantics of paragraphs, arranging children on a single row and creating new subsequent rows when the previous one has no longer space to arrange a children. The precise rendering semantics of boxes can be found in [83].

5.3.2 Notational Equations

We now want to define precisely what a notational equation is. Before doing so we need to define the notions of pattern and term.

Intuitively, a (presentation or content) pattern is an expression against which other expressions can be matched capturing sub-expressions by the mean of variables occurring in the pattern.

Table 5.4: Box schemata of presentation expressions

B ::=		(**box schemata**)	**arity**
	h	(horizontal box)	$n,\ n \geq 1$
|	v	(vertical box)	$n,\ n \geq 1$
|	hv	(horizontal-vertical box)	$n,\ n \geq 1$
|	hov	(horizontal or vertical box)	$n,\ n \geq 1$
|	$break$	(break-point hint)	0

Definition 5.1 (Well-formed pattern) *A well-formed presentation (content) pattern is a presentation (content) expression E without identifiers, hyperlinks and cross references and such that any variable in E occurs exactly once.*

Terms are final expressions, or fully instantiated patterns.

Definition 5.2 (Term) *A presentation (content) term is a presentation (content) expression without variables.*

We can now define what is a notational equation.

Definition 5.3 (Notational equation) *A* notational equation *is a pair of well-formed patterns*

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

that simultaneously defines:

1. *an* abstraction *from the notational level to the content level, and*

2. *a rendering from the content level to the notational level.*

In the following example a notational equation enabling infix use of the equality symbol in presentation expressions is shown.

Example 5.1 *Notational equation for infix equality*

The notational equation:

$$\alpha \texttt{ = } \beta \quad \Longleftrightarrow \quad \texttt{apply}[\texttt{eq}, \alpha, \beta] \tag{5.1}$$

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

□

In the following sections we will see how abstraction and rendering are related to notational equations.

5.3.3 Abstraction

Abstraction is the process of instantiating the content term corresponding to a presentation term.

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

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

$$\mathcal{G}_0 \overset{P_0^{\mathsf{p}} \Longleftrightarrow P_0^{\mathsf{c}}}{\longrightarrow} \mathcal{G}_1 \overset{P_1^{\mathsf{p}} \Longleftrightarrow P_1^{\mathsf{c}}}{\longrightarrow} \mathcal{G}_2 \overset{P_2^{\mathsf{p}} \Longleftrightarrow P_2^{\mathsf{c}}}{\longrightarrow} \cdots \overset{P_k^{\mathsf{p}} \Longleftrightarrow P_k^{\mathsf{c}}}{\longrightarrow} \mathcal{G}_k$$

where each grammar $\mathcal{G}_{i+1}$ results from $\mathcal{G}_i$ by the addition of the production for T derived from $P_i^{\mathsf{p}} \iff P_i^{\mathsf{c}}$ and P_i^{c} is a content pattern parsed with $\mathcal{G}_i$.

The fact that the content patter is parsed with $\mathcal{G}_i$ enables incremental definition of notation on top of previously defined notational equations.

In particular, the added production is $T \to \mathsf{exp}(P^{\mathsf{p}})$ where the function $\mathsf{exp}(P)$ (mnemonic for "expansion") converts a presentation pattern into a sequence of terminal and nonterminal grammar symbols. $\mathsf{exp}(P)$ is defined in Table 5.5.

Table 5.5: Expansion of presentation patterns to productions

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

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

A delicate technical problem related to grammars is ambiguity. An ambiguous grammar is one such that there may be multiple parse trees for the same term. In the most common cases ambiguity can be resolved by declaring precedence and associativity of productions. Thus, the language may provide additional constructs (see Section 5.7) so that the user can specify, for instance, that the symbol "$*$" has precedence over "$+$" and that both are left-associative.

The remaining cases of ambiguity can be treated as errors (and the notation causing the ambiguity could be rejected or ignored), or they may be admitted provided that the implementation accommodates a form of content validation that can discriminate, among the various content terms that can be built starting from the very same presentation term, which one is semantically meaningful. This validation phase usually entails a deeper understanding of content terms than it is available at the content level, thus some cooperation with the semantic level becomes fundamental for settling structural ambiguities.

Now we take care of the instantiation step. Given a presentation term t, the parser yields a parsing tree for t which we denote with $\hat{t}$. In particular, it determines a pattern P_i^{p} and a substitution σ that associates variables occurring in P_i^{p} with

subterms of $\hat{t}$ such that $P_i^{\mathsf{p}}\sigma = \hat{t}$ (equality here is considered up to cross references and hyperlinks). We abbreviate this writing $t \in P_i^{\mathsf{p}} \rightsquigarrow \sigma$.

Example 5.2 *Instantiation example*

Assuming that the "+" operator has precedence over "=", we have that:

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

where we use parentheses to indicate a generic box schema.

□

We can now formally define abstraction.

Definition 5.4 (Abstraction) *Abstraction is a function $\mathcal{A}(\cdot)$ defined as follows:*

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

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

5.3.4 Rendering

Rendering is the transformation that creates a presentation term from a content term.

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

Table 5.6 shows the pattern matching of a content term t against a content pattern P as a system of inference rules.

We use the notation:

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

Table 5.6: Pattern matching on content level terms

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

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

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

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

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

Definition 5.5 (Rendering) *Rendering is a function $\mathcal{R}(\cdot)$ defined as:*

$$\mathcal{R}(t) \;=\; A\{I^H_\sigma(P^{\mathsf{p}}_i)\} \text{ where } t \in P^{\mathsf{c}}_i \rightsquigarrow_\emptyset \sigma, A, H$$

where the instantiation function $I^H_\sigma(\cdot)$ is defined as follows:

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

In the process of rendering a content term t annotations of subterms of t are preserved only in two occasions: either when they are found at the top level of t, in which case they become annotations for the resulting presentation term, or when they wrap proper subterms of t that have been bound by variables, in which

case they will wrap the rendered subterms. As there is no obvious way of relating the other annotations, they are simply discarded (see the (CONSTRUCTOR) rule in Table 5.6.

Hyperlinks, on the other hand, are handled pattern-wise. All the hyperlinks found in the part of a term matched by a content pattern are gathered together and sprinkled over the literals of the corresponding presentation pattern. That is to say, any visible part of the term is considered the concrete rendering of its symbols and should thus be linked to their definitions.

The definition of $\mathcal{R}(\cdot)$ omits two secondary details:

1. the function $\mathcal{R}(\cdot)$ must provide appropriate rendering for all the built-in notation defined in $\mathcal{G}_0$;

2. precedence and associativity of the productions are used to spot the subterms that must be protected by fences, in order to guarantee a presentation term that is consistent with the structure of the content term.

We will come back to (2) in Section 5.7.

Example 5.3 *Rendering*

Consider the notational equation:

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

where we assume that the notation for the equality = has been given as in Example 5.1.

The content term:

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

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

The term t would be rendered as:

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

where we note that the reference of the whole term is preserved, whereas the references of the `not` and `eq` symbols have been lost (there is no natural rendered sub-term corresponding to them). There are two links associated with the $\neq$ literal corresponding to the locations of the `not` and `eq` symbols. Finally, the symbols `1` and `2` have been rendered with all the information preserved (in the rendering we have omitted explicit box schemata for simplicity).

□

5.4 Extensions

As presented thus far, notational equations would not be able to express the notation of some use cases that can be found in the standard library of MATITA.

Use case 5.1 *Notation for the existential quantifier*

In CIC, the existential quantifier is not built-in, but can be defined as an inductive data-type as follows (snippet from `logic/connectives.ma`[1]):

```
inductive ex (A:Type) (P:A \to Prop) : Prop \def
  ex_intro: \forall x:A. P x \to ex A P.
```

Without any notational equation, stating a proposition on the line of $\exists\, n \in \mathbb{N}.\ P$ (where P is a proposition over n) would look like writing "`ex nat (\lambda n.` P`)`", requiring the user to write an explicit λ-abstraction.

A first improvement can already be obtained with the notational equations presented thus far changing the required text from the user to "`\exists n: nat.` P".

However, most of the times—though not always—the inference system of MATITA can discover the type of the existential quantified variable (n in the example) without requiring the user to explicitly type it in. Thus, we want to define a notational

[1] `http://matita.cs.unibo.it/library/logic/connectives.ma`

equation that enable the user to *optionally* write the type annotation "`: nat`", only when the system is not able to figure it out by itself.

□

Use case 5.2 *Notation for lists*

Polymorphic lists (i.e. lists of elements of the same type) can be defined in CIC as follows (snippet from `list/list.ma`[2]):

```
inductive list (A:Set) : Set \def
  | nil: list A
  | cons: A -> list A -> list A.
```

We want to define the familiar ML-like notation for lists which would enable us writing then as semicolon separated sequences of other terms, delimited by brackets. For example, such a notation would let the user write "`[O; O; O]`" instead of the more cumbersome "`cons O (cons O (cons O nil))`".

□

Both use cases have been addressed in MATITA by a couple of extensions of the notational equations mechanism presented in Section 5.3. In our experience those extensions are of generic interest and useful for a wider range of use cases.

In the following we will refine the content and presentation languages adding some new operators able to deal with optional and repeated parts in patterns. In turn we will need also to refine the well-formedness rules for patterns and describe how instantiations of those new operators. This part can be safely skipped if only interested in the implementation details of Section 5.7.

Meta-Operators

For dealing with use cases 5.1 and 5.2 it will be enough to add two new meta-operators[3] to our presentation (Table 5.1) and content languages (Table 5.2). In

[2] `http://matita.cs.unibo.it/library/list/list.ma`

[3] we call them "meta" as they are used to build content-level and presentation-level expressions

fact we will add pairs of matching operators in the presentation and content languages. During abstraction the parsing rule obtained expanding an operator in the presentation language will create an *environment* that will then be consumed by its matching operator of the content language to instantiate the content level AST of the input formula. Conversely, during rendering content level ASTs will be expanded as needed and then packed in the presentation level encoding that will be the final rendering output.

The new operators we are going to add are shown in Table 5.7 and Table 5.8. `opt` pairs with `default`, while `list0` (and its variant `list1`) pairs with `fold`.

Table 5.7: Presentation expressions: meta-operators

E^{p}	::=		(**presentation expression**)
		...	(other presentation expressions, see Table 5.1)
	|	`opt` E^{p}	(optional expression)
	|	`list0` E^{p} l	(repetition)
	|	`list1` E^{p} l	(non-empty repetition)

Table 5.8: Content expressions: meta-operators

E^{p}	::=		(**content expression**)
		...	(other content expressions, see Table 5.2)
	|	`default` E^{c} E^{c}	(default value)
	|	`fold` [`l` | `r`] E^{c} `rec` x E^{c}	(left/right folding)

The expansion of presentation patterns need to be extended as well in order to be able to deal with the new meta-operators. The additional needed rules are shown in Table 5.9.

Table 5.9: Expansion of presentation patterns to productions: meta-operators

...			(other expansion rules, see Table 5.5)
$\mathsf{exp}(\texttt{opt}\ P)$	$=$	O_P	where $O_P \to \varepsilon \mid \mathsf{exp}(P)$
$\mathsf{exp}(\texttt{list0}\ P\ l)$	$=$	$I_{0,P,l}$	where $I_{0,P,l} \to \varepsilon \mid I_{1,P,l}$
$\mathsf{exp}(\texttt{list1}\ P\ l)$	$=$	$I_{1,P,l}$	where $I_{1,P,l} \to \mathsf{exp}(P) \mid \mathsf{exp}(P)\ l\ I_{1,P,l}$

Environment

Definition 5.6 (Environment) *An environment is a map from names to values:*

$$\mathcal{E} : \mathit{Name} \to V$$

where values are defined as follows:

V	$::=$		***(values)***
		$\texttt{Term}\ T$	*(term)*
	$\mid$	`None`	*(optional value)*
	$\mid$	$\texttt{Some}\ V$	*(optional value)*
	$\mid$	$[V_1, \ldots, V_n]$	*(list value)*

Intuitively a name is a reference to a variable in a pattern, and a value can either be a term, an optional value (`Some` or `None`), or a list of values.

The actual creation of the environment by the framework is delegated to the semantic actions associated to the production generated by $\mathsf{exp}(\cdot)$ and will not be formalized here. However, its informal behaviour is quite intuitive: for optional values the parser will create a `None` value if a ε terminal has been consumed, a `Some` value carrying the parsed term otherwise. Similarly, a `list*` operator will build a list value containing all parsed terms matching the $\mathsf{exp}(P)$ production. Our implementation of environment creation in Matita will be discussed in Section 5.7.

Instantiation of Content Level Terms

Once an environment has been returned by the parser, the content level term output of abstraction can be obtained recursively processing the content level pattern, right-hand side of a notational equation. The rules governing this process are shown in Table 5.10. The vars function used in the rules is a straightforward function returning all the names of the variable used in a given content pattern.

The rules dealing with `default` simply test whether the parser has actually parsed an optional part of a presentation pattern. In that case the first branch of `default` is chosen, evaluating it in an environment where the variables occuring in that branch are bound to the corresponding parsed values. Alternatively, the second branch of `default` is chosen and evaluated in an environment where the variables occurring in the P_1 branch are not bound.

The rules dealing with `fold` mimick the behaviour of left/right folding in functional programming languages, with the peculiar difference that our folding proceeds in parallel over the set of variables occurring in the recursive branch of `fold` (P_2 in the rules). The variable occurring in `fold` just after `rec` is used to collect the value of the folding so far.

Rendering

Rendering with the new meta-operators is a two-phases process. A pattern matching phase is first performed on the content term to be rendered, matching it against each content pattern (right-hand sides of the defined notational equations). Outcome of the pattern matching is an environment. Then, a second instantiation phase is performed processing the resulting environment and the presentation pattern which is left-hand side of the notational equation that matched.

The first pattern matching phase is defined by the rules of Table 5.11 and Table 5.12, while the instantiation phase is defined by the rules of Table 5.13. The details of cross references and hyperlinks propagations are not shown for the sake of conciseness.

Table 5.10: Instantiation of content level terms from presentation level

$$
\begin{array}{rcll}
[\![x]\!]^{\mathsf{c}}_{\mathcal{E}} & = & t & \mathcal{E}(x) = \texttt{Term}\ t \\
[\![C[t_1,\ldots,t_n]]\!]^{\mathsf{c}}_{\mathcal{E}} & = & C[[\![t_1]\!]^{\mathsf{c}}_{\mathcal{E}},\ldots,[\![t_n]\!]^{\mathsf{c}}_{\mathcal{E}}] & \\
[\![\texttt{default}\ P_1\ P_2]\!]^{\mathsf{c}}_{\mathcal{E}} & = & [\![P_1]\!]^{\mathsf{c}}_{\mathcal{E}[x_i \mapsto v_i]} & \mathcal{E}(x_i) = \texttt{Some}\ v_i \\
 & & & \mathsf{vars}(P_1) \setminus \mathsf{vars}(P_2) = \{x_1,\ldots,x_n\} \\
[\![\texttt{default}\ P_1\ P_2]\!]^{\mathsf{c}}_{\mathcal{E}} & = & [\![P_2]\!]^{\mathsf{c}}_{\mathcal{E}[x_i \mapsto \bot]} & \mathcal{E}(x_i) = \texttt{None} \\
 & & & \mathsf{vars}(P_1) \setminus \mathsf{vars}(P_2) = \{x_1,\ldots,x_n\} \\
[\![\texttt{fold r}\ P_1\ \texttt{rec}\ x\ P_2]\!]^{\mathsf{c}}_{\mathcal{E}} & = & [\![P_1]\!]^{\mathsf{c}}_{\mathcal{E}'} & \mathcal{E}(\mathsf{vars}(P_2) \setminus \{x\}) = \{[],\ldots,[]\} \\
 & & \mathcal{E}' = \mathcal{E}[\mathsf{vars}(P_2) \setminus \{x\} \mapsto \bot] & \\
[\![\texttt{fold r}\ P_1\ \texttt{rec}\ x\ P_2]\!]^{\mathsf{c}}_{\mathcal{E}} & = & [\![P_2]\!]^{\mathsf{c}}_{\mathcal{E}'} & \mathcal{E}(y_i) = [v_{i1},\ldots,v_{in}] \\
 & & & \mathsf{vars}(P_2) \setminus \{x\} = \{y_1,\ldots,y_m\} \\
 & & \mathcal{E}'(y) = \begin{cases} [\![\texttt{fold r}\ P_1\ \texttt{rec}\ x\ P_e]\!]^{\mathsf{c}}_{\mathcal{E}''} & y = x \\ v_{i1} & y = y_i \\ \mathcal{E}(y) & \text{otherwise} \end{cases} & \\
 & & \mathcal{E}''(y) = \begin{cases} [v_{i2};\ldots;v_{in}] & y = y_i \\ \mathcal{E}(y) & \text{otherwise} \end{cases} & \\
[\![\texttt{fold l}\ P_1\ \texttt{rec}\ x\ P_2]\!]^{\mathsf{c}}_{\mathcal{E}} & = & \mathit{eval\_fold}(x, P_2, \mathcal{E}') & \\
 & & \mathcal{E}' = \mathcal{E}[x \mapsto [\![P_1]\!]^{\mathsf{c}}_{\mathcal{E}[\mathsf{vars}(P_2) \mapsto \bot]}] & \\
\mathit{eval\_fold}(x, P, \mathcal{E}) & = & \mathcal{E}(x) & \mathcal{E}(\mathsf{vars}(P) \setminus \{x\}) = \{[],\ldots,[]\} \\
\mathit{eval\_fold}(x, P, \mathcal{E}) & = & \mathit{eval\_fold}(x, P, \mathcal{E}') & \mathcal{E}(y_i) = [v_{i1},\ldots,v_{in}] \\
 & & & \mathsf{vars}(P) \setminus x = \{y_1,\ldots,y_m\} \\
 & & \mathcal{E}' = \mathcal{E}[x \mapsto [\![P]\!]^{\mathsf{c}}_{\mathcal{E}''};\ y_i \mapsto [v_{i2};\ldots;v_{in_i}]] & \\
 & & \mathcal{E}''(y) = \begin{cases} v_1 & y \in \mathsf{vars}(P) \setminus \{x\} \\ \mathcal{E}(x) & y = x \\ \bot & \mathit{otherwise} \end{cases} &
\end{array}
$$

Examples

Equipped with the new meta-operators we are now able to address use cases 5.1 and 5.2. We will take this opportunity for showing the concrete syntax used in MATITA

Table 5.11: Pattern matching on content terms (1/2)

$$\textsc{Constr}\frac{t_i \in P_i \rightsquigarrow \mathcal{E}_i \qquad i \neq j \Rightarrow \mathsf{dom}(\mathcal{E}_i) \cap \mathsf{dom}(\mathcal{E}_j) = \emptyset}{C[t_1, \ldots, t_n] \in C[P_1, \ldots, P_n] \rightsquigarrow \mathcal{E}_1 \oplus \cdots \oplus \mathcal{E}_n}$$

$$\textsc{TermVar}\frac{}{t \in [\texttt{term}]\ x \rightsquigarrow [x \mapsto \texttt{Term}\ t]}$$

$$\textsc{DefaultT}\frac{t \in P_1 \rightsquigarrow \mathcal{E}}{t \in \texttt{default}\ P_1\ P_2 \rightsquigarrow \mathcal{E}'}$$

$$\text{where } E'(x) = \begin{cases} \mathsf{Some}\ \mathcal{E}(x) & x \in \mathsf{vars}(P_1) \setminus \mathsf{vars}(P_2) \\ \mathcal{E}(x) & \text{otherwise} \end{cases}$$

$$\textsc{DefaultF}\frac{t \notin P_1 \qquad t \in P_2 \rightsquigarrow \mathcal{E}}{t \in \texttt{default}\ P_1\ P_2 \rightsquigarrow \mathcal{E}'}$$

$$\text{where } E'(x) = \begin{cases} \texttt{None} & x \in \mathsf{vars}(P_1) \setminus \mathsf{vars}(P_2) \\ \mathcal{E}(x) & \text{otherwise} \end{cases}$$

by the user to define notational equations.

Example 5.4 *Notation for the existential quantifier in* Matita

The notation for the existential quantifier can be found in `logic/connectives.ma`[4], and is there given as follows:

[4] `http://matita.cs.unibo.it/library/logic/connectives.ma`

Table 5.12: Pattern matching on content terms (2/2)

$$\textsc{FoldRec}\frac{t \in P_2 \rightsquigarrow \mathcal{E} \quad \mathcal{E}(x) \in \texttt{fold}\ d\ P_1\ \texttt{rec}\ x\ P_2 \rightsquigarrow \mathcal{E}'}{t \in \texttt{fold}\ d\ P_1\ \texttt{rec}\ x\ P_2 \rightsquigarrow \mathcal{E}''}$$

$$\text{where } \mathcal{E}''(y) = \begin{cases} \mathcal{E}(y) :: \mathcal{E}'(y) & y \in \mathsf{vars}(P_2) \setminus \{x\} \wedge d = \texttt{right} \\ \mathcal{E}'(y)@[\mathcal{E}(y)] & y \in \mathsf{vars}(P_2) \setminus \{x\} \wedge d = \texttt{left} \\ \mathcal{E}'(y) & \text{otherwise} \end{cases}$$

$$\textsc{FoldBase}\frac{t \notin P_2 \quad t \in P_1 \rightsquigarrow \mathcal{E}}{t \in \texttt{fold}\ P_1\ \texttt{rec}\ x\ P_2 \rightsquigarrow \mathcal{E}'} \quad \mathcal{E}'(y) = \begin{cases} [] & y \in \mathsf{vars}(P_2) \setminus \{x\} \\ \mathcal{E}(y) & \text{otherwise} \end{cases}$$

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

The concrete syntax for notational equations is a bit cumbersome (we are actually looking for a cleaner syntax). Still, once used to it, the mapping with notational equations is straightforward.

The presentation pattern is enclosed in double quotes. It consists of variables ("`i`", "`p`", and "`ty`") that stand for arbitrary CIC sub-terms, and literals ("`\exists`", "`:`", and "`.`") assembled together in a box schema. The special keyword "`break`" indicates the breaking point and the box schema "`hvbox`" indicates a horizontal or vertical box schema. The "`opt`" indicates the optional meta-operator that surrounds an optional part in the presentation pattern. Given this presentation pattern, the input syntax of Matita is extended so that, for example, "`\exists x:nat. x \le y`" is a valid presentation term. Because of the "`opt`" meta-operator, the type

Table 5.13: Instantiation of presentation term from an environment

$$
\begin{array}{rcll}
[\![L[P_1,\ldots,P_n]]\!]^{\mathsf{p}}_{\mathcal{E}} & = & L[[\![(P_1)]\!]^{\mathsf{p}}_{\mathcal{E}},\ldots,[\![(P_n)]\!]^{\mathsf{p}}_{\mathcal{E}}] & \\
[\![B[P_1\cdots P_n]]\!]^{\mathsf{p}}_{\mathcal{E}} & = & B[[\![P_1]\!]^{\mathsf{p}}_{\mathcal{E}}\cdots[\![P_n]\!]^{\mathsf{p}}_{\mathcal{E}}] & \\
[\![(P)]\!]^{\mathsf{p}}_{\mathcal{E}} & = & [\![P]\!]^{\mathsf{p}}_{\mathcal{E}} & \\
[\![(P_1\cdots P_n)]\!]^{\mathsf{p}}_{\mathcal{E}} & = & h[[\![P_1]\!]^{\mathsf{p}}_{\mathcal{E}}\cdots[\![P_n]\!]^{\mathsf{p}}_{\mathcal{E}}] & \\
[\![x]\!]^{\mathsf{p}}_{\mathcal{E}} & = & t & \mathcal{E}(x) = \texttt{Term}\ t \\
[\![\texttt{opt}\ P]\!]^{\mathsf{p}}_{\mathcal{E}} & = & \varepsilon & \mathcal{E}(\mathsf{vars}(P)) = \{\texttt{None}\} \\
[\![\texttt{opt}\ P]\!]^{\mathsf{p}}_{\mathcal{E}} & = & [\![P]\!]^{\mathsf{p}}_{\mathcal{E}'} & \\
 & & \multicolumn{2}{l}{\mathcal{E}(\mathsf{vars}(P)) = \{\texttt{Some}\ v_1,\ldots,\texttt{Some}\ v_n\}} \\
 & & \multicolumn{2}{l}{\mathcal{E}'(x) = \begin{cases} v, & \mathcal{E}(x) = \texttt{Some}\ v \\ \mathcal{E}(x), & \text{otherwise} \end{cases}} \\
[\![\texttt{list}k\ P\ l?]\!]^{\mathsf{p}}_{\mathcal{E}} & = & [\![P]\!]^{\mathsf{p}}_{\mathcal{E}_1}\ l?\cdots l?\ [\![P]\!]^{\mathsf{p}}_{\mathcal{E}_n} & \\
 & & \multicolumn{2}{l}{\mathcal{E}(\mathsf{vars}(P)) = \{[v_{11},\ldots,v_{1n}],\ldots,[v_{m1},\ldots,v_{mn}]\}} \\
 & & \multicolumn{2}{l}{n \geq k} \\
 & & \multicolumn{2}{l}{\mathcal{E}_i(x) = \begin{cases} v_i, & \mathcal{E}(x) = [v_1,\ldots,v_n] \\ \mathcal{E}(x), & \text{otherwise} \end{cases}} \\
[\![l]\!]^{\mathsf{p}}_{\mathcal{E}} & = & l &
\end{array}
$$

annotation "`: nat`" can be omitted, the resulting term still being syntactically valid.

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

The content pattern begins right after the "`for`" keyword and extends to the end of the declaration. Parts of the pattern surrounded by "`@{`" ... "`}`" denote verbatim content fragments, those surrounded by "`${`" ... "`}`" denote meta-operators and term variables (for example "`$ty`") referring to the term variables occurring in the

presentation pattern.

The content pattern of the example defines the application of the content symbol "`exists`" to a λ-abstraction. In this case there are two possibilities according to the presence or absence of the type annotation in the presentation term that matched the pattern. For this reason there is a corresponding meta-operator at the content level, named "`default`", that has two branches which are chosen depending on the matching of the optional subexpression. In the example this is used to account for the optionality of type annotation on the quantified name, since its type can be inferred during disambiguation. Thus, if the type is given, the content term created after parsing has the form "`'exists (\lambda x:nat.(x \le y))`". Otherwise, the resulting content term has the form "`'exists (\lambda x.(x \le y))`". □

Example 5.5 *Notation for lists in* MATITA

The notation for polymorphic lists can be found in the script `list/list.ma`[5]. It is there given as follows:

```
notation "[ list0 x sep ; ]"
  non associative with precedence 90
  for ${fold right @'nil rec acc @{'cons $x $acc}}.
```

The given notation is straightforward: a list at the presentation level is a sequence of presentation level terms separated by a semicolon literal. Upon processing such a sequence the parser will build an environment in which the x variable is bound to a list value. (Right) folding is then used on such a value to build an applicative abstract syntax tree having a depth linear in the lenght of the list. Leaves of such a tree are the `'nil` and `'cons` symbols, the former occurring only once.

In the same script additional notational equations are given to support the infix "`::`" operator for building lists from head/tails pairs and to support the infix "`@`" operator for list concatenation. The interested reader can have a look at the script for the corresponding script snippets.

□

[5] `http://matita.cs.unibo.it/library/list/list.ma`

5.5 Handling Ambiguity in Matita

Since disambiguation and ambiguation (the transformations from content to semantics and back) inherit the quality of being application-specific from the semantic level, we cannot give a fully general recipe for handling them. We will therefore present their instantiation in the context of MATITA only. Nevertheless, as we will see shortly, the only requirement we pose on the semantic language is for it to be compositional, as most structured languages are.

In MATITA the semantic language is CIC, a typed λ-calculus enriched with inductive data types. In this setting we define the following notion of interpretation.

Definition 5.7 (Interpretation) *An* interpretation *is a pair:*

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

where s is a content symbol of arity $n \geq 0$ and $t[\alpha_1, \ldots, \alpha_n]$ is a CIC term with n holes labelled α_1, ..., α_n.

The underlying idea of an interpretation is to give *one of* the possible meanings for the symbol s when applied to n content terms t_1, ..., t_n, in terms of the CIC term t in which the hole α_i has been replaced by the meaning of t_i. The "one of" is to remark that there can be multiple interpretations for the same symbol s, not necessarily sharing the same arity.

5.5.1 Disambiguation

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

When the semantic level is CIC, the ambiguity is induced by the one-to-many mapping of symbols to CIC term, which in turn is induced by overloading of operators and missing information at the notational level.[6]

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

Consider the content level expression obtained after the abstraction of Example 5.2. Its ambiguity with respect to CIC derives from the overloading of "+" (two different plus do exists in the standard library of MATITA), and from the missing type argument of "=", which is needed by the CIC encoding of Leibniz's equality.

Example 5.6 *Interpretation in* MATITA

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

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

interpretation "integer plus" 'plus x y =
  (cic:/matita/Z/plus/Zplus.con x y).



interpretation "Leibniz's equality" 'eq x y =
  (cic:/matita/logic/equality/eq.ind#xpointer(1/1) _ x y).
```

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

□

Intuitively, *disambiguation* is a two phase process. In the first phase all possible CIC terms corresponding to a content term, according to the current set of interpretations, are built. In the second phase they are filtered by means of an oracle able to decide whether a term is valid or not. Such an oracle for CIC is the *refiner* described in [94]. The actual disambiguation algorithm implemented in MATITA exploits the type inference capabilities of the refiner and is far more efficient than the naive algorithm entailed by this intuition.

More information on the actual disambiguation algorithm used by MATITA can be found in Chapter 3 and in [97].

5.5.2 Ambiguation

We call *ambiguation* the reverse transformation that creates a content term from a CIC term. It is simpler than disambiguation since the mappings from CIC to content are not ambiguous (they might be non-injective though). This step resembles rendering in many ways: ambiguation works by pattern matching on CIC terms, and it instantiates content terms according to the matching interpretations. As usual, the system provides a finite set of built-in mappings for transforming uninterpreted CIC terms to the corresponding content terms. Propagation of cross references and hyperlinks can be implemented in exactly the same way as described in Section 5.3, the URIs appearing in interpretations are the original sources of hyperlinks.

Despite the negative connotations usually associated with its name, ambiguation is what ultimately enhances interoperability between different MKM applications. Indeed it is inevitable for two applications to represent a common concept such as equality in semantically different ways, which are often dictated by the needs of the applications themselves. If two applications can only communicate mathematical objects on the basis of a perfect match of their semantics, communication could hardly succeed but in the simplest cases. It is thus a side-effect of this point of view that turns "ambiguation" into a feature, rather than a bug, provided that unambiguous semantics can be recovered, which is exactly what our architecture does.

η-Abstraction

Use case 5.1, addressed by Example 5.4, poses an additional challenge for notation. Namely, the right hand-side of the corresponding notational equation matches, in either case, a content term only if its shape is an `'exists` symbol applied to a λ-abstraction.

Frequently however the existential quantifier obtained by rendering from the semantic is an `'exists` symbol applied to an identifier (or a constant for what is worth) which can be resolved to a CIC term which type is an arrow (implying that

the term itself should actually be convertible to a λ-abstraction). The identifier is said to appear in η-contract form.

In order for it to be properly rendered as an existential quantifier it should first be converted in an η-expanded form (e.g. in the existential quantifier case, assuming that the identifier is f, it should be converted to something like $\lambda x.\ f\ x$, where x is a fresh name free where f occurs). The actual pattern used by our notational framework on the left-hand side of interpretations accounts for this need offering the *η-abstraction* mechanism. The abstract syntax of that patterns is shown in Table 5.14.

Table 5.14: Syntax of interpretation content patterns

P	::=		**(interpretation content patterns)**
		s	(symbol)
	|	$s\ args$	(symbol application)
$args$	::=		**(arguments)**
		arg	
	|	$arg\ args$	
arg	::=		**(argument)**
		α	(term variable)
	|	$\eta.arg$	(η-abstraction)

Each pattern matches either a symbol or the application of a symbol to a number of arguments, where each argument is either a term variable or an η-abstracted argument. The number of η in front of an argument denotes the required number of λ-abstractions that should appear around the argument after the ambiguation phase. Note that those λ-abstractions are fictitious and have no meaning from a logical point of view, they will only be added to help notational equations match a given content term. The actual rules used to add λ-abstraction during ambiguation are given in Table 5.15.

Table 5.15: η-Abstraction

$$
\begin{array}{rcll}
[\![s\ a_1 \cdots a_n]\!]^a_{\mathcal{E}} & = & (s\ [\![a_1]\!]^a_{\mathcal{E},0} \cdots [\![a_n]\!]^a_{\mathcal{E},0}) & \\
[\![x]\!]^a_{\mathcal{E},0} & = & t & \mathcal{E}(x) = \texttt{Term}\ t \\
[\![x]\!]^a_{\mathcal{E},i+1} & = & \lambda y.[\![t]\!]^a_{\mathcal{E},i} & \mathcal{E}(x) = \texttt{Term}\ \lambda y.t \\
[\![x]\!]^a_{\mathcal{E},i+1} & = & \lambda y_1.\cdots.\lambda y_{i+1}.t\ y_1 \cdots y_{i+1} & \mathcal{E}(x) = \texttt{Term}\ t \wedge \forall s, t \neq \lambda s.t \\
[\![\eta.a]\!]^a_{\mathcal{E},i} & = & [\![a]\!]^a_{\mathcal{E},i+1} &
\end{array}
$$

Example 5.7 *η-abstraction*

In the standard library of MATITA we exploited η-abstraction for rendering properly the existential quantification of Example 5.4. Its default interpretation is given in `logic/connectives.ma`[7] as follows:

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

□

We will now change subject, in the next section we will discuss how we exploited the remote control annotations—hyperlinks and cross references—in the authoring interface of MATITA.

5.6 Exploiting Remote Control: Direct Manipulation of Terms

We saw in Section 2.1 that two of the windows composing the authoring interface of MATITA(the sequents window and the CIC browser) can be used to present formulae and concepts (which in turn can contain formulae) to the user.

[7] `http://matita.cs.unibo.it/library/logic/connectives.ma`

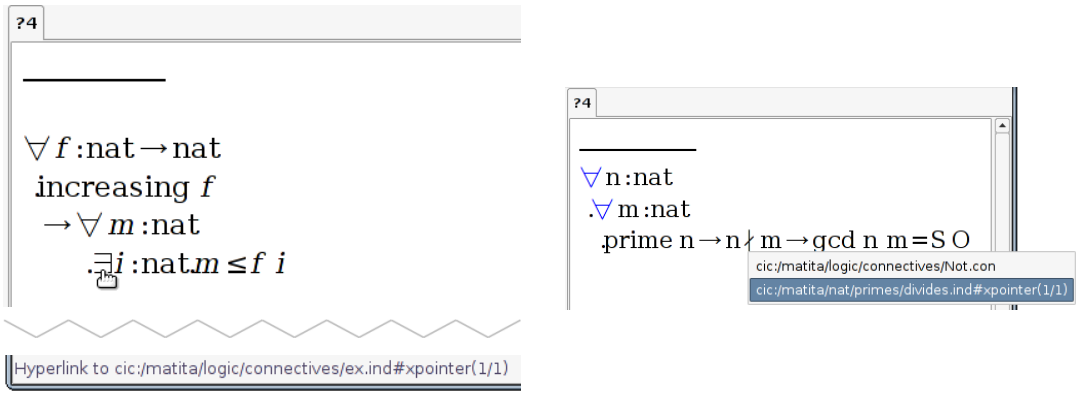

Figure 5.4: Hypertextual browsing

Both windows are based on a graphical widget able to render (on the screen or on a different media) notational level terms. The actual input of such widget is an XML dialect containing a mixture of MathML Presentation and BoxML markup. 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—BoxML—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.

5.6.1 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 [98] using the mouse. The potential of *hypertextual browsing* is shown in Figure 5.4.

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 5.4 the mouse is over an ∃ symbol which is part of a user defined

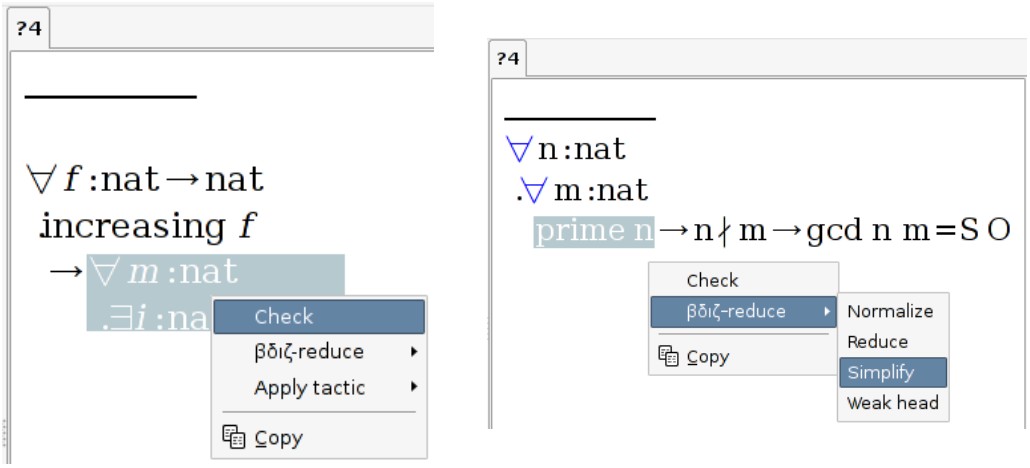

Figure 5.5: Contextual actions and semantic selection

notation for the existential quantifier, available in the MATITA standard library as a concept whose URI is shown in the status bar (bottom-left of Figure 5.4). Clicking on an anchor will start a CIC browser instance to show the target concept.

Since our hyperlinks are one-to-many, it can happen that markup elements reference more than one concepts from the library. For instance, on the right of Figure 5.4 the symbol $\nmid$ is a user defined notation which uses two concepts (logical negation and the divisibility operator over natural numbers), trying to follow 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 or in the sequents window. Figure 5.5 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 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 (the widget used for rendering 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 the rendering widget (and more generally in rendering engines for XML based markup languages) is constrained by the structure of the presentational markup, which is not necessarily related to the structure of the underlying CIC term.

On the left of Figure 5.5 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 issue 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 5.5.

Gory Details

Both hypertextual browsing and semantic selection are implemented by enriching the presentational markup with semantic attributes, which represent in the final markup the cross references and hyperlink of the notational framework.

The `hrefs` attribute is used to implement hypertextual browsing and correspond to the literal annotations of Table 5.1. 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). As we saw, 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 and correspond to the generic annotation of Table 5.1. Each CIC sub-term is annotated with a unique identifier; the set of those identifiers is the domain of the `xref` attribute. As we saw, 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 clipped to the first enclosing 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.

5.7 Implementation

We have shown the effectiveness of the proposed notational framework by instantiating its architecture to the Matita proof assistant. All the different encodings of formulae and transformations shown in Figure 5.3 has been implemented and are currently uses for input and output of formulae in Matita. Before giving an overview of the actual code we will discuss the most interesting implementation

choices.

A few technologies have been used as the basis for the instantiation of the framework to MATITA:

Camlp4 The first technology is Camlp4, an extensible top-down, non-ambiguous parser for the OCaml programming language. Camlp4 can be used like other parser generators, enabling the user to write in a concise form grammar productions (using the Camlp4 grammar language). Its engine then take care of creating the in-memory structures and code to actually parse streams of token coming from a lexer.

A distinctive feature of Camlp4 is that grammars are values in the programming language and can be manipulated by the programmer. Most notably, the productions set can be modified adding and removing productions at run-time. Adding a grammar production dynamically simply requires giving to Camlp4 a list of tokens of the grammar language, an associativity for it (left/right/no associativity), a precedence, and a function representing the semantic action to be invoked when the new production is used by the parser.

Camlp4 is a non-ambiguous parser though, requiring that each input token stream returns exactly one parsed value (or alternatively a failure). This is a shortcoming in the context of our framework since it does not allow us to deal with structural ambiguity, that is with presentation terms admitting more than one corresponding content term (while we support content terms admitting more than one corresponding CIC term). We plan to relax this constraint by implementing one of the several extensible parser generators that can be found in the literature (see [90] for an example).

Ulex In combination with Camlp4 we have used a fully compliant Unicode lexer generator, namely Ulex. Its use enables the final user of MATITA to input mathematical symbols to be used in presentation terms directly as UTF-8 characters, provided a suitable input method is available on the user worksta-

tion. If this is not the case we provide an alternative TEX-like way to input such symbols, it will be discussed below.

GtkMathView As the final rendering engine for showing presentation terms to the user we use GtkMathView, a GTK+ widget able to render an XML dialect containing a mixture of MathML Presentation [66] (which is fully implemented by the widget) and BoxML markup. The latter markup language is not a standardized language, but rather our own straightforward encoding of the boxes language of Table 5.4.

From the point of view of the programmer the widget is invoked by loading a Gdome2 [25] document[8] and reacts to events fired by user interactions, most notably mouse events like clicks on or movements over elements of the XML markup.

Since the extra XML attributes we added to the mixed markup for preserving hyperlinks and cross references are in a namespace ignored by GtkMathView, they are preserved upon loading of the Gdome2 document and looked up for by handlers of click and mouse-over events.

Disambiguator The disambiguation phase is completely delegated to the disambiguator component of Matita, separately developed and discussed in Chapter 3.

5.7.1 Precedence and Associativity of Notational Equations

The combined choice of Camlp4 and GtkMathView raises the interesting problem of how to deal with associativity and precedence of notational equations with respect to each other. While the markup rendered by GtkMathView has no precedence and associativity issues, since the scope of operators is well defined by the tree structure of the markup, the textual form of terms input by the user needs—as usual

[8] Gdome2 is the Gnome implementation of the Document Object Model Level 2 specification [38], an API for working on tree representations of XML documents

in the parsing of even trivial languages—parentheses to ensure proper recognition of operators' arguments. Our aim is to ensure that the rendered form of formulae has the minimum amount of parentheses to enable the user to "copy" it (either by retyping, or by copy & paste) and feed it back to the system.

Our current solution consists in annotating with *position annotations* the variables occurring in presentation patterns given by the user as left-hand side of notational equation. Position annotations record where in a given layout or box the term pattern occurs and it has three possible values: left, inner, or right. Additionally, core notational equations built-in in MATITA has marked patterns denoting where parentheses are never needed due to the special position of those patterns (for instance when they occur near lexer keywords). The annotation process of variables occurring in patterns input by the user is described in Table 5.16.

Table 5.16: Annotation of pattern variables with position information

$$
\begin{aligned}
pos(l)_{p,q} &= l \\
pos(x)_{1,0} &= \langle pos = L \rangle x \\
pos(x)_{0,1} &= \langle pos = R \rangle x \\
pos(x)_{p,q} &= \langle pos = I \rangle x \\
pos(B[P])_{p,q} &= B[pos(P)_{p,q}] \\
pos(B[P_1 \cdots P_n])_{p,q} &= B[pos(P_1)_{p,0} \\
&\qquad pos(P_2)_{0,0} \cdots pos(P_{n-1})_{0,0} \\
&\qquad pos(P_n)_{0,q}]
\end{aligned}
$$

During rendering, the precedence and associativity of the notational equation who actually matched producing a part of the markup are remembered in association with the generated markup. As a last pass of rendering, just before returning the generated presentation term, a *parenthesizing pass* is performed on the term in order to add parentheses where needed. That pass exploits both the precedence and associativity information extracted form the notational equations and the position

annotations. The actual rules governing the parenthesizing pass are described in Table 5.17. The underlined terms denote special positions where parentheses are never needed.

Table 5.17: Addition of parentheses where needed

$$
\begin{array}{rcll}
(\!|l|\!)^n & = & l & \\
(\!|\langle prec = m\rangle T|\!)^n & = & (\!|T|\!)^m & n < m \\
(\!|\langle prec = m\rangle T|\!)^n & = & \texttt{(}(\!|T|\!)^{\perp}\texttt{)} & n > m \\
(\!|\langle prec = n, assoc = L, pos \neq L\rangle T|\!)^n & = & \texttt{(}(\!|T|\!)^{\perp}\texttt{)} & \\
(\!|\langle prec = n, assoc = R, pos \neq R\rangle T|\!)^n & = & \texttt{(}(\!|T|\!)^{\perp}\texttt{)} & \\
(\!|\langle \cdots\rangle T|\!)^n & = & (\!|T|\!)^n & \\
(\!|L[T_1, \ldots, \underline{T_k}, \ldots, T_m]|\!)^n & = & L[(\!|T_1|\!)^n, \ldots, (\!|T_k|\!)^{\perp}, \ldots, (\!|T_m|\!)^n] & \\
(\!|B[T_1, \ldots, T_m]|\!)^n & = & B[(\!|T_1|\!)^n, \ldots, (\!|T_m|\!)^n] &
\end{array}
$$

5.7.2 The Code

The implementation of the notational framework touches several components of MATITA, though only one of them is entirely devoted to its implementation. All the involved components are shown in Figure 5.6. We will discuss the role of each component with respect to the notational framework in turn.

The `content` component contains the definition of the content level of formulae as used in MATITA. It is for most part isomorphic to MathML Content [66], with the addition of a few built-in symbols for encoding constructs typical of the Calculus of Constructions. In the need of mapping our content level to plain MathML Content those construct can be mapped to symbols whose meaning is defined in external content dictionaries.

Of interest of the notational framework, the `content` component also contains the built-in interpretations governing ambiguation and disambiguation. User defined interpretations are dynamically added to this set. For this reason the `content`

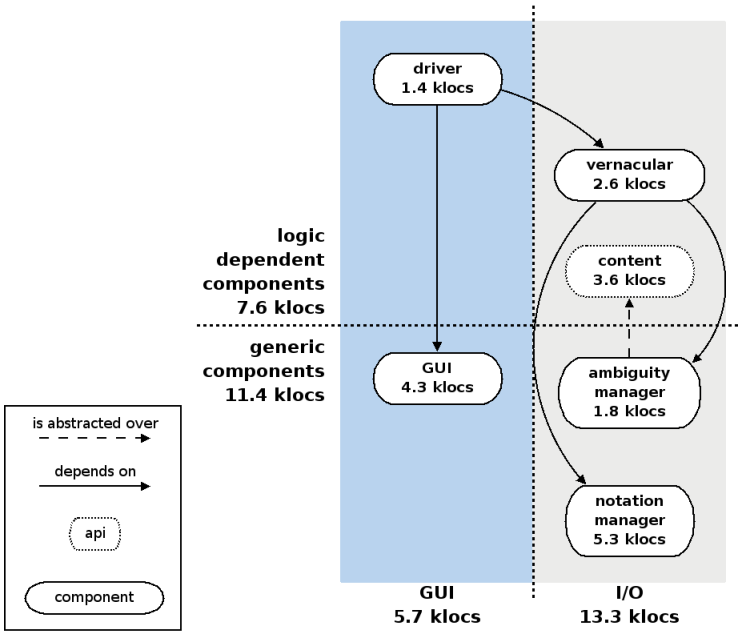

Figure 5.6: Components of the notational framework implementation in MATITA

component is logic dependent and is not considered as being reusable in the implementation of proof assistants for logics other than MATITA's.

The `driver` and `vernacular` components are marginally touched by the notational framework. They are of interest only because they implement the concrete syntax used by users to provide new notational equations and interpretations and act as proxy for invoking the appropriate entry points of the `notation manager`.

The `ambiguity manager` implements the disambiguation transformation, depends on a content level like that implemented by the `content` component, and is used directly by the `vernacular` when presentation terms appear as term arguments of commands like tactics or queries. More details on this component has been given in Chapter 3.

The `notation manager` implements the largest part of the implementation of our framework. Several more fine-grained component can be distinguished inside the `notation manager`.

A first component is `CicNotationLexer`, the lexer used in MATITA before feeding

commands typed by the user to the command parser. It is built on top of Ulex and offers a pre-lexing phases which permit to the user to type mathematical symbols both as UTF-8 byte sequences and as TEX-like macros. For instance, in use case 5.1 writing `\forall` is exactly the same as writing the Unicode character ∀ (0x2200 in the Unicode standard). The names of TEX-like macros are inherited from MathML character entities and the lexer can be queried about that. This way MATITA is able to automatically convert TEX-like macros in Unicode characters if asked to do so.

The lexer also supports *ligatures*, that are ASCII character sequences corresponding to Unicode characters. This features comes really handy to user who for some reasons are stuck with ASCII only proof scripts, but wants to write symbols in a more evocative way than TEX-like macros. Examples of ligatures are "`->`" for "`\to`" or "`=>`" for "`\Rightarrow`". The lexer can be queried about ligatures as well enabling MATITA to expand them on demand or automatically. The lexer is fully generic and can be exploited for parsing languages other than the MATITA proof language, at the moment it exports an API suitable for being used directly in conjunction with Camlp4.

Another component of the `notation manager` is a generic pattern matching engine, which implements a variant of the pattern matching algorithms typically used in the implementation of functional languages [13, 61]. Our variant has been enriched with more expressive backtracking capabilities for dealing with meta-operators. At the heart of the pattern matching engine is the `PatternMatcher` module, which contains a functor (`PatternMatcher.Matcher`) instantiating an input module with the following interface:

```
type pattern_kind = Variable | Constructor
type tag_t = int

module type PATTERN =
sig
  type pattern_t
  type term_t

  val classify : pattern_t -> pattern_kind
  val tag_of_pattern : pattern_t -> tag_t * pattern_t list
  val tag_of_term : term_t -> tag_t * term_t list
end
```

The module returned by the functor contains a single function used to compile pattern matching matrices:

```
val compiler:
  (P.pattern_t * int) list ->
  ((P.pattern_t list * int) list -> P.term_t list -> P.term_t list ->
    'a option) ->
  (unit -> 'a option) ->
    (P.term_t -> 'a option)
```

The first argument of the function is a pattern matching matrix, where each pattern is annotated with a numeric identifier. The second argument is a callback invoked in case of successful matching, passing to it the list of patterns matching the input, the list of terms bound in each of them, and the list of terms who matched constructors. If the return value of this callback is `None` backtracking is triggered. Similarly, the third argument is a failure callback, invoked in case of unsuccessful matching. The returned value is again a function, returned after that the pattern matching matrix has been internally compiled, which can be used to perform pattern matching on a given term.

The `PatternMatcher.Matcher` functor is instantiated twice in the code base of MATITA: once to implement ambiguation, and once to implement rendering. As an assessment of its generality it is also going to be used to add deep pattern matching capabilities to the concrete syntax of the `match` constructor of CIC.

Various entry points to be used by the `vernacular` are provided in the `notation manager`. In order to add notational equations two functions are provided: `add_pretty_printer` in module `TermContentPres` and `extend` in module `CicNotationParser`:

```
val add_pretty_printer:
  precedence:int ->
  associativity:Gramext.g_assoc ->
  CicNotationPt.term ->
  CicNotationPt.term ->
    pretty_printer_id

val extend:
  CicNotationPt.term ->
  precedence:int ->
  associativity:Gramext.g_assoc ->
  (CicNotationEnv.t -> CicNotationPt.location -> CicNotationPt.term) ->
    rule_id
```

The two functions are used respectively to change the behaviour of the rendering and parsing phase. `add_pretty_printer` adds a row to the pattern matching matrix used during rendering and (lazily) recompiles the pattern matching function using the instantiated pattern matcher functor. The two terms occurring in its type are respectively the left and right-hand side of the notational equation given by the user. They share the same type for factorizing out common constructors on the two types.

`extend` changes the Camlp4 grammar adding the grammar production obtained expanding the left-hand side of a notational equation, provided as a term argument

to `extend`. The latter argument of the function is a simplified form of the semantic action associated to the new production; its type gives us some hint on how enviroments are generated in our implementation. The generated grammar production has as many inner invocations of the term production as term variables occur in the presentation pattern of a given equation. According to the Camlp4 API, the semantic action that needs to be passed for extending a grammar is a function of as many arguments as inner invocations of grammar productions. Processing the presentation pattern is thus possible to incrementally built a currified function which will be used as semantic action for Camlp4. The body of such incrementally built function will simply build a representation of the environment of definition 5.6 and then pass it to the function given by the implementor as argument of `extend`. More details can be found in the `CicNotationParser` module itself.

The fact that two different functions are available for extending pretty printing and parsing rules enables the user to define asymmetric rules for input and output purposes.

The entry point for extending the interpretation list in effect is similarly split, `add_interpretation` in the `TermAcicContent` module (which acts on the ambiguation matrix as `add_pretty_printer` acts on the rendering one) is one half. The other half is delegated to disambiguation and has been discussed in Chapter 3.

All entry points in the `notation manager` return opaque unique identifiers that can be used in the future to remove the added notational rules (pretty printing, parsing, interpretation rules). Those identifiers are kept in the internal status of Matita and used to revert to a previous notational status when the user ask to retract a command affecting it (e.g. undoing of a `notation` command).

The last component shown in Figure 5.6 is `GUI`. The only part of it related to the notational framework is the `clickableMathView` widget, a class implemented in the `MatitaMathView` module. It inherits the behaviour of the vanilla GtkMathView widget, changing it in two ways:

1. it adds an event which is fired when the user clicks with the mouse on a markup element equipped with one or more hyperlinks, providing a way to

register callbacks for this event;

2. it changes the selection behaviour to implement semantic selection, that is clipping visual selection to markup elements equipped with cross references.

5.8 Related Work

The work presented in this chapter complements [97] by investigating the technical issues related to the design of a user-extensible, interactive environment for the development and the management of mathematical knowledge in a semantically driven way. In particular, it proposes an architecture that has proven effective in mixing presentational as well as semantic aspect of the processed information. This is an improvement with respect to the currently available tools related to mathematical knowledge management which typically focus on one, but not both, of these equally important aspects.

Previous work [6] describing a similar architecture to that discussed in this paper did not address the issues related to extensible input support, and it only described informally how hyperlinks and cross references were propagated from the semantic to the presentation level. In this paper we describe these important features in a more abstract, but also more formal way, hoping to provide useful guidelines for future implementations.

The layered architecture that we have proposed is similar in structure to that of previous projects in which notation played a major role. In [6, 72] ambiguation and rendering are implemented by XSLT stylesheets [124] and they can only be extended by adding XSLT templates. Support for further notation is thus limited to the system designers. From the point of view of maintenance of the transformations, an improvement is the introduction of meta-stylesheet [62] that generate XSLT templates starting from a slightly higher-level specification.

A somehow similar approach has been proposed by Naylor and Watt [75] for supporting alternative notations. In any case, all the solutions mentioned are one-way only and cannot be inverted, both because XSLT is a very general transformation

language, and also because the reverse path must reconstruct information that is not always available.

Our transformation language is not as general as XSLT but has been carefully designed so as to guarantee invertibility (the meta-operators mentioned in Section 5.4 are all invertible). Furthermore, it has a purely declarative style and is thus more appropriate for users who do not have any programming experience. The notational level consists of a finite set of layout schemata, basically those that are found in MathML Presentation [66] and TEX, box schemata for line-breaking inspired by previous work on pretty-printers [34], and a few meta-operators (like `opt` and `list`) inspired to the constructs of BNF grammars. The content level is an internal version of MathML Content and OpenMath [100], with the addition of meta-operators corresponding to those of the notational level.

The Coq proof assistant [26] provides a similar language for extending notation, with two main differences: it does not supply a content level and it does not deal directly with remote control. Our language represents a more open and interoperable solution, and the implementation shows that remote control can be achieved effectively even when notation is extensible, limiting built-in transformations to a bare minimum.

5.9 Conclusions

In this chapter we have characterized meaningful mathematical notation as a tool that necessarily mixes presentational as well as semantic aspects. We have identified a set of requirements that any mathematical knowledge management application supporting meaningful notation should fulfill and we have proposed an adequate architecture that builds upon the three well-known levels of formulae encoding: notation, content, and semantics.

As an assessment of the generality of the architecture, we have given a formal dressing to the concept of notation which makes a minimum set of assumptions, and

we have described an instantiation of its application-specific parts to the MATITA proof assistant.

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

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

Original Contributions

Rendering from a semantic encoding of mathematical formulae (and proofs) to a presentational encoding was one of the former task took by the HELM team as a whole, several works was published on the subject. Parts of these works include the ideas and the implementations of back-links and extensible notation for rendering purposes.

The design, formalization, and implementation of an unified framework for dealing with extensible notation in MATITA for both rendering *and* parsing purposes is an original contribution of this thesis author as a joint work with Luca Padovani. The notational language enabling the user to dynamically extend the set of notational equation and interpretations is also an original contribution of them. The

formalization comprises a precise specification of back-links which was missing in previous works of the HELM team.

The first implementation of the notational framework in MATITA was written from scratch by this thesis author and Luca Padovani (reusing the disambiguation component written by this thesis author and Andrea Asperti) and is now maintained by this thesis author.

Related Publications

Part of the work described in this chapter has been previously published in the following papers:

- Luca Padovani and Stefano Zacchiroli.
  From notation to semantics: There and back again [85].
  In Proceedings of MKM 2006: The 5th International Conference on Mathematical Knowledge Management 2006[9], Lecture Notes in Artificial Intelligence, Vol. 4108, pages 194–207. Springer-Verlag, 2006.

- Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli.
  User Interaction with the MATITA *Proof Assistant* [10].
  To appear in Journal of Automated Reasoning[10], special issue on User Interfaces for Theorem Proving[11].

[9] http://www.mkm-ig.org/meetings/mkm06/
[10] http://www.springerlink.com/link.asp?id=100280
[11] http://www.informatik.uni-bremen.de/~cxl/uitp-jar/

Chapter 6

HBugs: Publish/Subscribe Hints During Proof Authoring

The web-friendliness of MATITA can be exploited to various ends. In this chapter we present an experiment made with a network of brokers and web services for automatic deduction. The architecture is generic and can be adapted to different proof-assistants to deliver out-of-band "hints" to the user on how to proceed in the proof. Each web service implements one of the tactics usually available in a proof assistant. When the broker is submitted a proof status, it dispatches the proof to the web services which independently try to proceed in the proof according to the tactic they implement. The broker then collects the successful results, and send them back to the client as hints as soon as they are available.

In our experience this architecture turns out to be helpful both for experienced users (who can take benefit of distributing potentially heavy computations) and beginners (who can learn from the hints).

6.1 Introduction

The web service approach at software development seems to be a working solution for getting rid of a wide range of incompatibilities between communicating software applications. The efforts of the World Wide Web consortium[1] (W3C for short) in

[1] `http://www.w3.org`

standardizing related technologies grant longevity and implementations availability for frameworks based on web services for information exchange. As a direct consequence, the number of such frameworks is increasing and the World Wide Web is moving from a disorganized repository of human-understandable HTML documents to a disorganized repository of applications working on machine-understandable XML documents both for input and output.

The big challenge for the next future is to provide stable and reliable services over this disorganized, unreliable, and ever-evolving architecture. The standard solution is to provide a further level of stable services (called *brokers*) that behave as common gateways/addresses for client applications to access a wide variety of services and abstract over them.

Since the *Declaration of Linz*, the MONET Consortium[2] has worked on the development of a framework, based on the web services/brokers approach, aimed at providing a set of software tools for the advertisement and the discovery of mathematical web services.

Several groups have developed software bus and services[3] providing both computational and reasoning capabilities [5, 24, 127, 128]: the first ones are implemented on top of Computer Algebra Systems (or CASs); the second ones provide interfaces to well-known automatic theorem provers. Proof-planners, proof-assistants, CASs and domain-specific problem solvers are natural candidates to be clients of these services. Nevertheless, the number of examples in the literature has been insufficient to fully assess the concrete benefits of the framework.

In this chapter we present an architecture—namely HBugs—as a case study which implements a *suggestion engine* for the Matita proof assistant. We provide several web services (called *tutors*) able to suggest possible ways to proceed in a proof ongoing in the system. The tutors are orchestrated by a broker (a web service itself) that is able to dispatch a proof status from a client (the proof-assistant) to

[2] `http://monet.nag.co.uk/cocoon/monet/index.html`

[3] The most part of these systems predate the development of web services. Those systems whose development is still active are being reimplemented as web services.

the tutors; each tutor tries to make progress in the proof applying a tactic and, in case of success, notifies the client that shows an *hint* to the user. The broker is an instance of the homonymous entity of the MONET framework. The tutors are MONET services. Another web service (the HTTP Getter, see Section A.4) is used to locate and download mathematical entities; the Getter plays the role of the Mathematical Object Manager of the MONET framework.

A precursor of HBugs is the ΩMega-Ants project [18, 19], which provided similar functionalities for the ΩMega proof-planner [17]. The main architectural difference between HBugs and ΩMega-Ants is that the latter is based on a blackboard architecture and it is not implemented using web services and brokers. HBugs is not meant to be a reimplementation of ΩMega-Ants, nor to provide the same amount of functionalities. It is rather a case study meant to show the potentialities of web-friendliness for interactive proof assistants and how they can be exploited to provide added-value services and integration with external tools.

In Section 6.2 we present the architecture of HBugs, while further implementation details are delayed to Section 6.4. A sample usage session is shown in Section 6.3. Section 6.5 is an overview of the tutors that are currently implemented. Section 6.6 is devoted to future works on the HBugs experiment.

6.2 Architecture

The HBugs architecture (depicted in Figure 6.1) is based on three different kinds of actors: *clients*, *brokers*, and *tutors*. Each actor presents one or more web service interfaces to its neighbors HBugs actors.

In this section we detail the role and requirements of each kind of actors and we discuss about the correspondences between them and the MONET entities described in [71]. The study of the correspondences with MONET is motivated by the fact that the MONET framework was still under development when we first presented HBugs

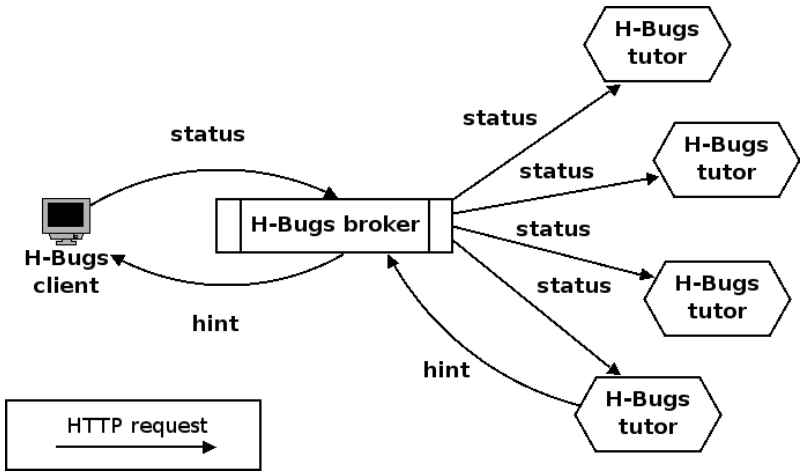

Figure 6.1: HBugs architecture

(see [96]) and that back then our implementation was one of the first experiments in web service based distributed reasoning. On the other hand, a comparison with ΩMega would be less interesting since the functionalities we provide so far are just a subset of the ΩMega-Ants ones.

6.2.1 Clients

A HBugs client is a software component able to produce proof statuses and to consume hints.

A *proof status* is a representation of an incomplete proof and is supposed to be informative enough to be used by an interactive proof assistant as a representation of an ongoing proof. No additional requirements do exist on the proof status from the point of view of HBugs, but there should be an agreement on its format between clients and tutors.

A *hint* is an encoding of a proof step that can be performed in order to proceed in an incomplete proof. Usually it represents a reference to a tactic available on some proof assistant along with an instantiation for its formal parameters. Hints can also be more structured: a hint can be as complex as a whole proof-plan.

Using W3C's terminology [122], clients act both as *web service providers* and *web service requesters*, see Figure 6.2. They act as providers receiving hints from the broker; they act as requesters submitting new status to the tutors. Clients additionally use broker services to know which tutors are available and to subscribe to one or more of them.

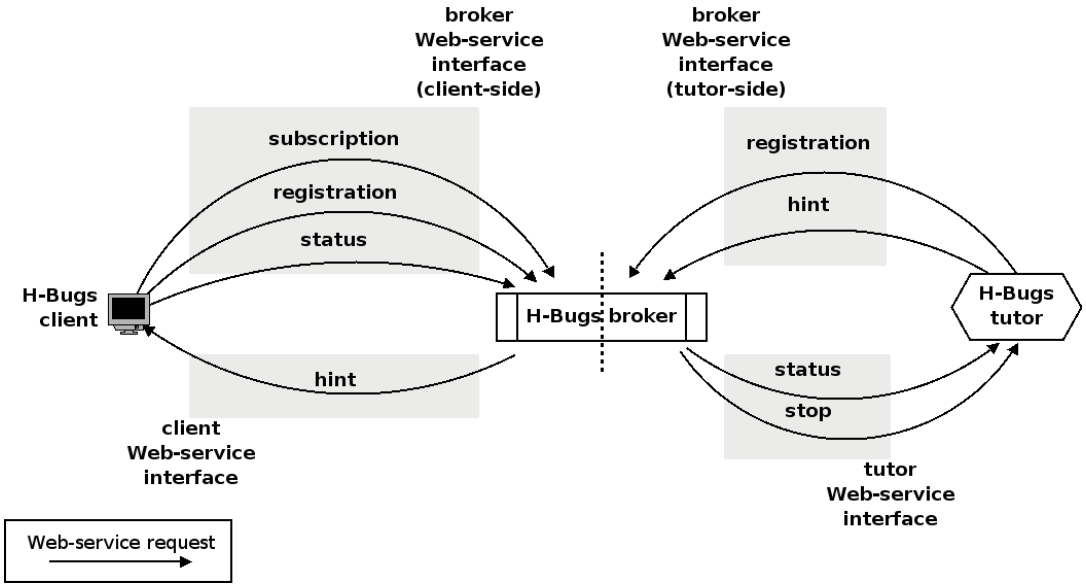

Figure 6.2: HBugs web service interfaces

Usually, when the client role is taken by an interactive proof assistant, new status are sent to the broker as soon as the proof change (e.g. when the user applies a tactic or when a new proof is started); hints are shown to the user by the means of some effects in the user interface (e.g. popping a dialog box or enlightening a tactic button).

HBugs clients act as MONET clients and ask brokers to provide access to a set of services (the tutors). HBugs has no actors corresponding to MONET's Broker Locating Service (since the client is supposed to know the URL of at least one broker). The HBugs clients and tutors contact the Getter (a MONET Mathematical Object Manager) to locate and retrieve mathematical items from the Helm library. The proof status that are exchanged by the HBugs actors, instead, are built on the

fly and are neither stored nor given an unique identifier (URI) to be managed by the Getter.

6.2.2 Brokers

Brokers are the key actors of the HBugs architecture since they act as intermediaries between clients and tutors. They behave as web services providers and requesters for *both* clients and tutors, see Figure 6.2.

With respect to the client, a broker acts as a web service provider, receiving the proof status and forwarding it to one or more tutors. It also acts as a web service requester sending hints to the client as soon as they are available from the tutors.

With respect to the tutors, the web service provider role is accomplished by receiving hints as soon as they are produced; as a requester, it is accomplished by asking for computations (*musings* in HBugs terminology) on status received by clients and by stopping already late but still ongoing musings.

Additionally, brokers keep track of available tutors and clients subscriptions.

HBugs brokers act as MONET brokers implementing the following components: Client Manager, Service Registry Manager (keeping track of available tutors), Planning Manager (choosing the available tutors among the ones to which the client is subscribed), Execution Manager. The Service Manager component is not required since the session handler, that identifies a session between a service and a broker, is provided to the service by the broker instead of being received from the service when the session is initialized. In particular, a session is identified by an unique identifier for the client (its URL) and an unique identifier for the broker (its URL).

Note that HBugs brokers have no knowledge of the domain area of proof-assistants, nor they are able to interpret the messages that they are forwarding. They are indeed only in charge of maintaining the abstraction of several reasoning blackboards—one for each client—of capacity 1: a blackboard is created when the client submits a problem; it is then "shared" by the client and all the tutors until the client submits the next problem. For instance, replacing the client with a CAS and all the tutors with agents implementing different resolution methods for differential

equations would not require any change in the broker. Still, all the tutors must expose the same interface to the broker.

The MONET architecture specification does not state explicitly whether the service and broker answers can be asynchronous. Nevertheless, the described information flow implicitly suggests a synchronous implementation. On the contrary, in HBugs every request is asynchronous: the connection used by an actor to issue a query is immediately closed; when a service produces an answer, it gives it back to the issuer by calling the appropriate method of the actor.

6.2.3 Tutors

Tutors are software components able to consume proof status producing hints. HBugs does not specify by which means hints should be produced: tutors can use any means necessary (heuristics, external automatic theorem provers or CASs, ...). The only requirement is that there exists an agreement on the formats of proof status and hints.

Tutors act both as web service providers and requesters for the broker, see Figure 6.2. As providers, they wait for commands requesting to start a new musing on a given proof status or to stop an old, out of date, musing. As requesters, they signal to the broker the end of a musing along with its outcome (a hint in case of success or a failure notification).

HBugs tutors act as MONET services.

6.3 Sample Session

In this section we describe a typical HBugs session. The goal of the session is to solve the following exercise:

Exercise 6.1 Let x be a generic real number. Using Matita prove that:

$$x = \frac{(x+1) * (x+1) - 1 - x * x}{2}$$

□

Let us suppose that the HBugs broker is already running and that the tutors have already registered themselves to the broker. When the user starts Matita, the system registers itself to the broker, that sends back the list of available tutors. By default, Matita notifies to the broker its intention of subscribing to all available tutors. The user can always fire a CIC browser instance (see Section 2.1) to inspect the available tutors and manually subscribe or unsubscribe to each of them, as shown on the left of Figure 6.3.

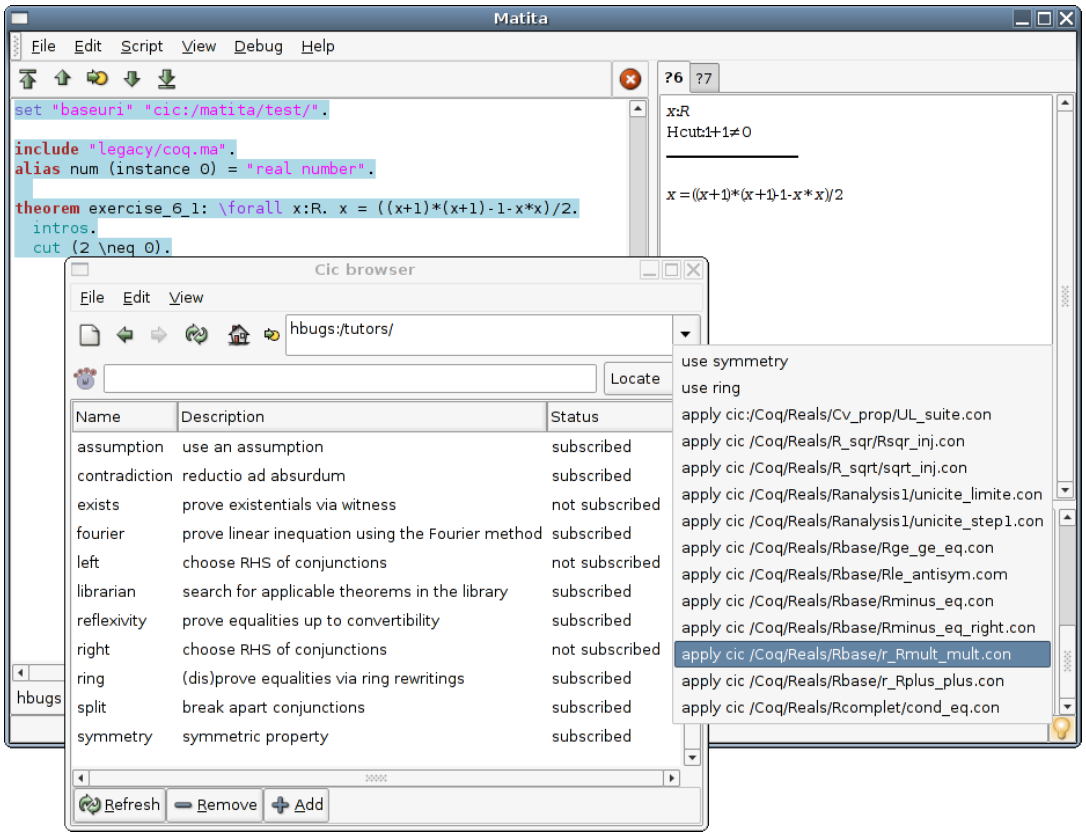

Figure 6.3: Screenshot of a HBugs session

The user can now type into the script window the statement of the theorem and start proving it. Let us suppose that the first step of the user is proving that the denominator 2 is different from 0. Once that this technical result is proven, the user must prove the goal shown in the upper right corner of the window in background

in Figure 6.3.

While the user is wondering how to proceed in the proof, the tutors are trying to progress in the proof. After a while, the hints start reaching MATITA, and as soon as at least one is available the light bulb on the bottom right of the authoring interface shown in Figure 6.3 lights up. The contextual menu available clicking on the bulb shows, in this case, 23 different hints. The first and not very useful hint suggests to proceed in the proof by exchanging the two sides of the equality. The second hint suggests to reduce both sides of the equality to their normal form by using only reductions which are justified by the ring structure of the real numbers; the two normal forms, though, are so different that the proof is not really simplified. All the residual 21 hints suggest to apply one lemma from the distributed library of HELM.

The user can now look at the list of suggestions and realize that a good one is applying the lemma `r_Rmult_mult` that allows to multiply both equality members by the same scalar[4]. Clicking on the hint automatically applies the lemma, reducing the proof to closing three new goals. The first one asks the user the scalar to use as an argument of the previous lemma; the second one states that the scalar is different from 0; the third lemma (the main one) asks to prove the equality between the two new members.

The user proceeds by instantiating the scalar with the number 2. The assumption tutor now suggests to close the second goal (that states that $2 \neq 0$) by applying the hypothesis H. No useful suggestions, instead, are generated for the main goal $2 * x = 2 * ((x + 1) * (x + 1) - 1 - x * x) * 2^{-1}$. To proceed in the proof the user needs to simplify the expression using the lemma *Rinv_r_simpl_m* that states that $\forall x, y.\ y = x*y*x^{-1}$. Since we do not provide yet any tutor suggesting simplifications, the user must find out this simplification by himself. Once she founds it, the goal is reduced to proving that $2 * x = (x + 1) * (x + 1) - 1 - x * x$. This equality is easily

[4]Even if she does not receive the hint, the user probably already knows that this is the right way to proceed. The difficult part, accomplished by the query, is guessing where the lemma is located in the library

solved by the ring tutor, that suggests[5] to the user how to complete the proof in one macro-step.

6.4 Implementation

In this section we present some of the most relevant implementation details of the HBugs architecture.

6.4.1 Proof Status

In our implementation of the HBugs architecture we used Matita as an HBugs client. We have thus implemented serialization/deserialization capabilities for its internal proof status. In order to be able to describe web services that exchange status in WSDL using the XML Schema type system, we have chosen an XML format as the target format for the serialization.

Each ongoing proof is represented by a tuple of four elements:
$\langle uri, metasenv, proof, thesis \rangle$.

uri an URI chosen by the user at the beginning of the proof process. Once (and if) proved, that URI will globally identify the term inside the Helm library, used by Matita as its knowledge base;

thesis (the type of) the ongoing proof thesis;

proof the current incomplete proof tree. It is a CIC term can contain *metavariables* (holes) that stands for the parts of the proof that are still to be completed. Each metavariable appearing in the tree references one element of the metavariables environment (*metasenv*);

metasenv the metavariables environment is a list of *goals* (unproven conjectures). In order to complete the proof, the user has to instantiate every metavariable

[5]The `Ring` suggestion is just one of the 22 hints that the user receives. It is the only hint that does not open new goals, but the user right now does not have any way to know that.

in the proof with a proof of the corresponding goal. Each goal is identified by an unique identifier and has a context and a type (the goal thesis). The context is a list of named hypotheses (declarations and definitions). Thus the context and the goal thesis form a sequent, which is the statement of the proof that will be used to instantiate the metavariable occurrences.

Each of these information is represented in XML as described in [93]. Additionally, an HBugs status carries the unique identifier of the current goal, which is the goal the user is currently focused on. Using this value it is possible to implement different client side strategies: the user could ask the tutors to work on the goal she is considering or to work on other "background" goals.

6.4.2 Hints

A hint in the HBugs architecture should carry enough information to permit the client to progress in the current proof. In our implementation each hint corresponds to either a reference to one of the tactics available to the user in Matita (together with its actual arguments) or a set of alternative suggestions (a list of hints).

For tactics that do not require any particular argument (like tactics that apply type constructors or decision procedures), the tactic name is used as a reference, and is the only information encoded in the hint. For tactics that need terms as arguments (for example the `apply` tactic, that apply in a backward reasoning fashion a given theorem) the hint includes a textual representation of them, using the same representation used by the interactive proof assistant when querying user for terms. In order to be transmitted between web services, hints are serialized in XML.

It is also possible for a tutor to return more hints at once, grouping them in a particular XML element. This feature turns out to be particularly useful for the *librarian* tutor (see Section 6.5) that queries the Helm library using the Hint query (see Section 2.1.1) and returns to the client a list of all theorems that can be applied to proceed in the proof. This particular hint is encoded as a list of `Apply` hints, each of them having one of the results as term argument.

We would like to stress that the HBugs architecture has no dependency on either the hint or the status representation: the only message parts that are fixed are those representing the administrative messages (the envelopes in web services terminology). In particular, the broker can manage at the same time several sessions working on different status/hints formats. Of course, there must be an agreement between the clients and the tutors on the format of the data exchanged.

In our implementation the client does not trust the received hints: being encoded as references to available tactics implies that an HBugs client, upon reception of a hint, simply tries to replay the work done by a tutor on the local copy of the proof. The application of the hint can even fail to type check and the client copy of the proof can be left undamaged after spotting the error. Note, however, that it is still possible to implement a complex tutor that looks for a proof doing backtracking and that send back to the client a hint whose argument is a witness (a trace) of the proof found: the client applies the hint reconstructing (and checking the correctness of) the proof from the witness, without having to re-discover the proof itself.

An alternative implementation where the tutors are trusted would simply send back to the client a new proof-status. Upon receiving the proof-status, the client would just override its current proof status with the suggested one. In the case of those clients which are implemented using proof-objects (as the Coq proof-assistant, for instance), it is still possible for the client to type-check the proof-object and reject wrong hints. Systems that are not based on proof-objects (as PVS, NuPRL, etc.), instead, must completely trust the new proof-status. In this case the HBugs architecture would need at least to be extended with client-tutor authentication.

6.4.3 Registries

Being central in the HBugs architecture, the broker is also responsible of house-keeping operations both for clients and tutors. These operations are implemented using three different data structures called *registries*: clients registry, tutors registry, and musings registry.

In order to use the suggestion engine a client should register itself to the broker and subscribe to one or more tutors. The registration phase is triggered by the client using the `Register_client` method of the broker to send him an unique identifier and its base URL as a web service. After the registration, the client can use the `List_tutors` method of the broker to get a list of available tutors. Eventually the client can subscribe to one or more of these using the `Subscribe` method of the broker. Clients can also unregister from brokers using `Unregister_client` method.

The broker keeps track of both registered clients and their subscriptions in the clients registry.

In order to be advertised to clients during the subscription phase, tutors should register to the broker using the `Register_tutor` method of the broker. This method is similar to `Register_client`: tutors are required to send an unique identifier and a base URI for their web service. Additionally tutors are required to send a human readable description of their capabilities; this information can be used by the client user to decide which tutors she wants to subscribe to. As clients do, tutors can unregister from brokers using `Unregister_broker` method.

Each time the client status changes, it get sent to the broker using its `Status` method. Using both the clients registry (to lookup the subscriptions of the client) and the tutors registry (to check if some tutors have un-subscribed), the broker is able to decide to which tutors the new status have to be forwarded.

The forwarding operation is performed using the `Start_musing` method of the tutors, that is a request to start a new musing on a given status. The return value of `Start_musing` is a musing identifier that is saved in the musings registry along with the identifier of the client that triggered the musing.

As soon as a tutor completes a musing, it informs the broker using its `Musing_completed` method; the broker can now remove the musing entry from the musings registry and, depending on its outcome, inform the client. In case of success one of the `Musing_completed` arguments is a hint to be sent to the client; otherwise there is no need to inform him and the `Musing_completed` method is called just to update the musings registry.

Consulting the musings registry, the broker is able to know, at each time, which musings are in execution on which tutor. This peculiarity is exploited by the broker on invocation of the `Status` method. Receiving a new status from the client implies indeed that the previous status no longer exists and all musings working on it should be stopped: additionally to the already described behavior (i.e. starting new musings on the received status), the broker takes also care of stopping ongoing computation invoking the `Stop_musing` method of the tutors.

6.4.4 Tutors

Each tutor exposes a web service interface and is able to work, not only for several different clients referring to a common broker, but also for several different brokers. The potential high number of concurrent clients imposes a multi-threaded or multi-process architecture.

Our current implementation is based on a multi threaded architecture exploiting the capabilities of the OCaml HTTP library (see Section A.1). Each tutor is composed by one always running thread plus an additional thread for each musing. One thread is devoted to listening for incoming web service requests; when a request is received the control is passed to a second thread, created on the fly, that handle the incoming request (usual one-thread-per-request approach in web servers design). In particular if the received request is `Start_musing`, a new thread is spawned to handle it; the thread in duty to handle the HTTP request returns an HTTP response containing the identifier of the just started `musing`, and then dies. If the received request is `Stop_musing`, instead, the spawned thread kills the thread responsible for the `musing` whose identifier is the argument of the `Stop_musing` method.

This architecture turns out to be scalable and allows the running threads to share the cache of loaded (and type-checked) theorems. As we will explain in Section 6.5, this feature turns out to be really useful for tactics that rely on a huge but fixed set of lemmas, as every reflexive tactic.

The implementation of a tutor within the described architecture is not that difficult having a language with good threading capabilities (as OCaml has) and

a pool of already implemented tactics (as Matita has). Working with threads is known to be really error prone due to concurrent programming intrinsic complexity. Moreover, there is a non-negligible part of code that needs to be duplicated in every tutor: the code to register the tutor to the broker and to handle HTTP requests; the code to manage the creation and termination of threads; and the code for parsing the requests and serializing the answers. As a consequence we have written a generic implementation of a tutor which is parameterized over the code that actually proposes the hint and over some administrative data (as the port the tutor will be listening to).

The generic tutor skeleton is really helpful in writing new tutors. Nevertheless, the code obtained by converting existing tactics into tutors is still quite repetitive: each tutor that wraps a tactic has to instantiate its own copy of the proof-engine kernel and, for each request, it has to override its status, guess the tactic arguments, apply the tactic and, in case of success, send back a hint with the tactic name and the chosen arguments. Of course, the complex part of the work is guessing the right arguments. For the simple case of tactics that do not require any argument, though, we are able to automatically generate the whole tutor code given the tactic name.

Concretely, we have written a tactic-based tutor template and a script that parses an XML specification of the tutor and generates the code of the tutor. The XML file describes the TCP port the tutor will be listening on, the code snippet to invoke the tactic, the hint that is sent back upon successful application, and the human readable explanation of the implemented tactic.

6.5 Available Tutors

To test the HBugs architecture and to assess the utility of a suggestion engine for the end user, we have implemented several tutors. In particular, we have investigated three classes of tutors:

tutors for beginners these are tutors that implement tactics which are neither computationally expensive nor difficult to understand: an expert user can

always understand if the tactic can be applied or not without having to try it. For example, the following implemented tutors belong to this class:

assumption it ends the proof if the thesis is equivalent (i.e. convertible in CIC) to one of the hypotheses[6].

contradiction it ends the proof by *reductio ad absurdum* if one hypothesis is equivalent to `False`.

symmetry if the goal thesis is an equality, it suggests to apply the commutative property.

constructor/left/right/exists/split/reflexivity the constructor tutor suggests to proceed in the proof by applying one or more constructors when the goal thesis is an inductive type or a proposition inductively defined according to the declarative style. Since disjunction, conjunction, existential quantification and Leibniz's equality are particular cases of inductive propositions in the library of MATITA, all the other tutors of this class are instantiations of the `constructor` tactic. `left` and `right` suggest to prove a disjunction by proving its left/right member; `split` reduces the proof of a conjunction to the two proof of its members; `exists` suggests to prove an existential quantification by providing a witness;[7] `reflexivity` proves an equality whenever the two sides are convertible.

Beginners, when first faced with a tactic-based proof-assistant, get lost quite soon since the set of tactics is large and their names and semantics must be remembered by heart. Tutorials are provided to guide the user step-by-step in a few proofs, suggesting the tactics that must be used. We believe that our beginners tutors can provide an auxiliary learning tool: after the tutorial, the user is not suddenly left alone with the system, but she can experiment with

[6] In some cases, especially when non-trivial computations are involved, the user is totally unable to figure out the convertibility of two terms. In these cases the tutor becomes handy also for expert users

[7] This task is left to the user.

variations of the exercises given in the tutorial as much as she like, still getting useful suggestions. Thus the user is allowed to focus on learning how to do a formal proof instead of wasting efforts trying to remember the interface to the system.

tutors for computationally expensive tactics Several tactics have an unpredictable behavior, meaning that it is unfeasible to understand whether they will succeed or fail when applied and what will be their results. Among them, there are several tactics either computationally expensive or resource consuming. In the former case, the user is not willing to try a tactic and wait for a long time just to understand its outcome: she would prefer to keep on reasoning about the proof and have the tactic applied in background and receive out-of-band notification of its success. The latter case is similar, but the tactic application must be performed on a remote machine to avoid overloading the user host with several concurrent resource consuming computations.

Finally, several complex tactics and in particular all the tactics based on reflexive techniques, usually depend on a large set of concepts available in the library. When these tactics are applied, the system needs to retrieve and load all those concepts. Pre-loading all the material needed by every tactic can quickly lead to long initialization times and to large memory footprints. A specialized tutor running on a remote machine, instead, can easily pre-load the required concepts once and for all. Of course the HBugs client will eventually have to load them as well, but it will do so with a reasonable certainty (depending on how much the hint is trusted) that it is worth the effort.

As an example of computationally expensive task, we have implemented a tutor for the `ring` tactic [22]. The tutor is able to prove an equality over a ring structure by reducing both members to a common normal form. The reduction, which may require some time in complex cases, is based on the usual commutative, associative, and neutral element properties of a ring. The tactic is implemented using a reflexive technique, which means that the reduction

trace is not stored in the proof-object itself: the type-checker is able to perform the reduction on-the-fly thanks to the conversion rules of the system. As a consequence, in the library there must be stored both the algorithm used for the reduction and the proof of correctness of the algorithm, based on the ring axioms. This big proof and all of its lemmas must be retrieved and loaded in order to apply the tactic. The ring tutor loads and caches all the required theorems the first time it is contacted.

intelligent tutors Expert users can already benefit from the previous class of tutors. Nevertheless, to achieve a significant production gain, they need more intelligent tutors implementing domain-specific automatic theorem provers or able to perform complex computations. These tutors are not just plain implementations of tactics or decision procedures, but can be more complex software agents interacting with third-parties software, such as proof-planners, CASs, or automatic theorem provers.

To test the productivity impact of intelligent tutors, we have implemented a tutor (called *librarian*) that provides an interface to the Hint query of Whelp (see Section 2.1.1) and that is thus able to look for every theorem in the Helm library that can be applied to proceed in the proof. Even if the tutor deductive power is limited, it is not unusual for the tutor to come up with precious hints that can save several minutes of work that would have been spent in proving again already proven results or figuring out where the lemmas have been stored in the library.

6.6 Concluding Remarks

HBugs is a suggestion engine architecture for proof-assistants: the client (a proof-assistant) sends the current proof status to several distributed web services (called tutors) that try to progress in the proof and, in case of success, send back an appropriate hint (a proof-plan) to the user. The user, that in the meantime was

able to reason and progress in the proof, is notified with the hints and can decide to apply or ignore them. A broker is provided to decouple the clients and the tutors and to allow the client to locate and invoke the available remote services. The whole architecture is an instance of the MONET architecture for Mathematical web services. It constitutes a reimplementation of the core features of the pioneering ΩMega-ANTS system in a web service setting.

A running prototype had been implemented in `gTopLevel`, the proof assistant prototype of the HELM project, that nowadays has become MATITA. The full porting of the HBUGS code to MATITA is still underway though. Several tutors have been implemented. Some of them are simple tutors that try to apply one or more tactics of MATITA, which is also our client. We also have a much more complex tutor that is interfaced with WHELP and looks for concepts that can be directly applied.

Future works comprise the implementation of new features and tutors, and the embedding of the system in larger test cases. One interesting case study would be interfacing a CAS as Maple to the HBUGS broker, developing at the same time a tutor that implements the `Field` tactic of Coq, which proves the equality of two expressions in an abstract field by reducing both members to the same normal form. CASs can produce several compact normal forms, which are particularly informative to the user and that may suggest how to proceed in a proof. Unfortunately, CASs do not provide any certificate about the correctness of the simplification. On the contrary, the `Field` tactic certifies the equality of two expressions, but produces normal forms that are hardly a simplification of the original formula. The benefits for the CAS would be obtained by using the Field tutor to certify the CAS simplifications, proving that the Field normal form of an expression is preserved by the simplification. More advanced tutors could exploit the CAS to reduce the goal to compact normal forms [36], making the field tutor certify the simplification according to the skeptical approach.

We have many plans for further developing both the HBUGS architecture and our prototype. Interesting results could be obtained augmenting the informative

content of each suggestion. We can for example modify the broker so that also negative results are sent back to the client. Those negative suggestions could be reflected in the user interface by inhibiting the application of some tactics available to the user. This approach could be interesting especially for novice users, but requires the clients to increase their level of trust in the other actors.

We plan to add some rating mechanism to the architecture. A first improvement in this direction could be distinguishing between hints that, when applied, are able to completely close one or more goals, and tactics that progress in the proof by reducing one or more goals to new goals: since the new goals can be false, the user can be forced later on to backtrack.

Other heuristics and or measures could be added to rate hints and show them to the user in a particular order: an interesting one could be a measure that try to minimize the size of the generated proof, privileging therefore non-overkilling solutions [22].

We are also considering to follow the ΩMega-Ants path adding "recursion" to the system so that the proof status resulting from the application of old hints are cached somewhere and could be used as a starting point for new hint searches. The approach is interesting, but it represents a big shift towards automatic theorem proving: thus we must consider if it is worth the effort given the increasing availability of automation in proof assistants tactics and the ongoing development of web services based on already existent and well developed theorem provers.

Our web services still lack a real integration in the MONET architecture, since we do not provide the different ontologies to describe our problems, solutions, queries, and services. In the short term, completing this task could provide a significant feedback to the MONET consortium and would enlarge the current set of available MONET actors on the web. In the long term, new more intelligent tutors could be developed on top of already existent MONET web services.

HBugs is a nice experiment meant to understand whether the current web services technology is mature enough to have a concrete and useful impact on the daily work of proof assistants users. So far, only the librarian tutor has effectively

increased the productivity of experts users. The usefulness of the tutors developed for beginners, instead, need further assessment.

Original Contributions

The HBugs experiment has been designed and originally implemented by this thesis author during his master's thesis. The maintenance of the architecture in Matita and the ongoing port are being worked on by him.

Related Publications

Part of the work described in this chapter has been previously published in the following papers:

- Claudio Sacerdoti Coen and Stefano Zacchiroli.
  Brokers and Web-Services for Automatic Deduction: a Case Study [96].
  In Proceedings of Calculemus 2003: 11th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning[8], Aracne Editrice S.R.L., ISBN 88-7999-545-6, pp. 43-57, 2003.

[8] `http://www-calfor.lip6.fr/~rr/Calculemus03/`

Part III

Postface

Chapter 7

A Step Toward Formal Mathematical Knowledge Management in the Web 2.0 Era

Formal mathematical knowledge management (Formal MKM) was born at the intersection of digital libraries, theorem proving, and web publishing. The pioneer research projects who followed that pedigree implementing infrastructures for publishing formal mathematics on the web (most notably HELM [6] and the European project MoWGLI [72], the projects who gave birth to MATITA) are quite old now. Since the beginning of such projects, the web started passing through major evolutionary changes that are still ongoing.

The most notable of such changes is considered by many the advent of *Web 2.0* [82] (actually considered a buzzword by as many other, for what is worth). No matter whether such a thing named "Web 2.0" does exist or not, it is a fact that the web has been—and still is—experiencing the diffusion of new technologies, new intended uses, innovative user experiences, and new business models, which taken together have the potential of radically change the way the web has been used so far.

How such changes can affect formal mathematical knowledge management is a question that, to the best of our knowledge, has not yen been posed inside the research community. Such a question is of peculiar interest for us, MATITA developers, since our system was born on top of a library whose precise intent was to exploit

web technologies to radically change the philosophy of formal reasoning tools.

One of the peculiar technology of Web 2.0 are wiki sites and a while ago we started thinking about its adoption for authoring formal mathematical knowledge on the web. Far from being an implementation of "MATITA on the web" our interest led to a more general investigation of how, and most notably *if*, wikis can deal with content on which constraints have to be enforced. Our conclusion is that wikis can be used for dealing with such kind of content.

To solve the apparent trade-off between constraints and "The Wiki Way" of working [31] we presented a novel wiki concept—*light constraints*—designed to encode community best practices and domain-specific requirements, and to assist in their application. This chapter describes a general framework to think about the interaction of wiki system with constraints, and presents a generic architecture which can be easily incorporated into existing wiki systems to exploit the capabilities enabled by light constraints.

In the next section we will make a brief digression (Section 7.1) spotting how the original goals of the HELM project can be better achieved nowadays exploiting some of the new potential of Web 2.0.

7.1 Milieu

Looking these days at the philosophy and at the objectives of the HELM project in [6] still provides for an interesting reading. The key observation was that all formal reasoning tools, whose development predates the web (1.0) era, were bound to an application-oriented architecture, where the data were meaningful only for the application rather then per se.

That architectural choice used to hinder the development of large repositories of formal mathematical knowledge, since given the challenge of the task it can better be conceived as a collaborative and distributed (i.e. the *people* working on formalization are distributed world-wide) effort. Moreover, the application-oriented

design also poses difficulties in the exploitation of the web (1.0) as a platform for exchanging information and advertising the advancements of formal reasoning to the non-academic part of the world that might be interested in it (for example, communities of developers potentially interested in certified programming).

HELM was a pioneering project with the aim of developing a culture and a technological infrastructure for the creation and maintenance of an Hypertextual Electronic Library of formalized Mathematics, hence the name. The idea since then has been adopted by the teams of several other formal reasoning tools and digital libraries of mathematics (formal or just rigorous) and has been spreading in recent years (the most well-known examples include [4, 59, 72, 109, 126]).

The choices of XML and the web were crucial for the topics of HELM: interoperability, standardization, publishing, modularity, and searching. In order to foster the diffusion of the technological infrastructure being developed by the project, a desiderata was set: *every user should be able to consult and contribute to the library with as little client-side software as possible.* Two practical requirement were derived to fulfill the desiderata (reworded from [6]):

Requirement 7.1 *An (X)HTML [123] enabled web browser should be enough to consult the* HELM *library.*

and

Requirement 7.2 *In order to contribute to the* HELM *library, some web space accessible via URL pertaining to well-known schemes (e.g.* `http` *or* `ftp`*) should be enough.*

The two requirements used to incarnate the technical essence of the web: accessible with a single, simple and (nowadays) standardized client-side application, published without a central authority in a scalable and extensible manner.

Web 2.0 is an expression coined by Tim O'Reilly [82] and then became the topic of a series of conferences about some new principles of the web. The adherence to those principles are nowadays use to qualify web sites as Web 2.0 applications. A brief list of those new principles follows:

The web as a platform meaning that web browsing is no longer an activity per se performed using an application (a web browser), but rather that end-user applications (like office automation tasks for example) can be implemented on top of the web "platform" and used via a web browser.

Architecture of participation in which the more a web application is used and is capable of attracting new users, the more it is useful and effective (think about WIKIPEDIA[1] [119], or folksonomies like del.icio.us[2].

Data-centric applications meaning that for a web application the most important resources are the data provided by users rather then the softwares written to manipulate them. The concrete consequences of this principle is that Web 2.0 application do not "steal" information from their users, they offer a place where to store them, provide services on top of that data, and offer clear, standardized and open mechanisms to access them (usually via web-friendly Soap-based [99] APIs).

Rich user experience directly in the browser, AJAX[3] technology has shown that careful use of ECMAScript[4] and XMLHttpRequest[5] can lead to a user experience as rich as we are used to with standard desktop applications.

Social networking aspects as a way to explicitly model the interaction among users of a web application.

These principles changed the way web users are exploiting the web, but did not change its democratic essence. Being the philosophy of the HELM project strongly related to such essence, it is no surprise that some of these principles can be nowadays exploited and have the potential to improve the technological achievements of the project.

[1] `http://www.wikipedia.org`

[2] `http://del.icio.us`

[3] `http://en.wikipedia.org/wiki/Ajax_(programming)`

[4] `http://www.ecma-international.org/publications/standards/Ecma-262.htm`, whose must successful implementation is JavaScript

[5] `http://www.w3.org/TR/XMLHttpRequest/`

In particular, it is now feasible to think at directly editing mathematics on the web, as opposed to what was originally though in the HELM project. Directly editing of formulae with WYSIWYG is now technically feasible without the burden of Java applets. Projects like PlanetMath [59] have also gave the evidence that there is a community of people interested in collaboratively authoring rigorous mathematics on the web. Still, a similar counterpart for formal mathematics is missing, even though interest for wiki-based authoring of mathematical knowledge is raising [60]. As a marginal work of this dissertation, we started some investigations for developing a technological infrastructure for authoring formal mathematics on the web, in a wiki-like fashion. The project—whose code name is *Miki*—is still in its embryonic stage of development.

Preliminary studies of the project, conducted at the intersection of the formal mathematics and the wiki research community, raised the general question of how (and if) the wiki workflow can deal with contents on which constraints (as well-typedness, for what concerns Miki) can be enforced.

The remainder part of the chapter is a presentation of the novel concept of light constraints has it has been proposed to the wiki research community at the 2006 International Symposium on wikis. It as an almost verbatim re-edition of [52].

7.2 Introduction

The main factor of success of wiki sites is what the inventor of the first wiki site Ward Cunningham called "The Wiki Way" [31]: an open editing philosophy that allows users to freely write and collaborate on web content, without any restriction. In a sense, a wiki site can be considered a state-of-mind, an inclination shared by the users, rather than a collection of scripts and pages. This free notion of web editing, strengthened by some careful technical choices (direct editing in the browsers, minimality, versioning, ...), has made wiki systems commonly useful tools for single users, universities, and firms.

Although authors are free to change and produce new content at will, we cannot help noticing that even the wiki editing process is often bound by some (implicit) rules. For instance, writers frequently create sets of pages (often explicitly grouped in wiki site areas) that share a predefined structure. Surprisingly, the most widely used approach to create and manage such structures is based on copy & paste with manual refinement and checking. Some solutions based on a partially constrained editing model have been investigated, mostly exploiting powerful yet flexible templating languages (see [44] for instance).

Templating control is only an example of a more general trend we observed: wiki users tend to agree on sets of non-written conventions that one or more pages must adhere to, and they then need ways for ensuring, or at least checking, that those requirements are really met. In a sense, the existence of *WikiGnomes* [118] (users who work behind the scenes to fix minor nuisances) show this need, since much of their work is based on monitoring and adjusting conformance to community best practices. Moreover, in the context of the success of grassroots information encyclopedias based on wiki technologies (like WikipediA [119] or World66 [121]), the issue of the quality of wiki site content is increasingly becoming relevant.

Apart from such spontaneous and implicit rules developed by the community, some wiki sites need to satisfy requirements that depend on the context they are being used in. Consider for instance the wiki systems that supply *in-lining*, i.e. mixing content written in varied formats within a document written in wiki syntax: SnipSnap [54] allows users to in-line a text representation of mind maps, UML and other king of diagrams; OpenWiki [81] users can enrich pages with mathematical formulas written in MathML [66], and TWiki [108] users can include LaTeX mark-up commands, if a plug-in is installed. It is very useful to require that such in-lined text is correctly parsed and rendered by the wiki system. Yet, the existence of ill-formed documents does not cause irreversible problems (actually, inconsistent statuses are accepted for intermediate savings) but it would be preferable having integrated and automatic mechanisms to validate such content.

In this chapter we analyze scenarios where non-written conventions over content

are set up by the community as well as scenarios where inherent and context-based requirements affect wiki sites. Despite apparent differences, these issues have a common denominator: the need to express and enforce rules over the wiki content, without sacrificing the wiki editing paradigm. All these scenarios can be helped by what we have called *light constraints*, to emphasize their non-mandatory nature.

All the solutions we know are essentially *ad hoc* proposals for specific domains, hard-coded within systems, rather than instantiations of a general mechanism. No wiki system we are aware of offers support for representing and exploring different classes of light constraints and no generic model has been proposed yet. This work has a double goal: on the one hand, to emphasize the strong relationship between wiki sites and content constraints, in order to foster such discussion in the wiki community; and on the other hand, to propose a solution based on a strong distinction between wiki engine and validation tools used to verify whether relevant light constraints are respected or not.

The framework we present can be instantiated whenever a form of validation on wiki content is required. The usual wiki workflow changes somewhat, since the VIEW operation is enriched with a validation report and the SAVE operation become a conditional step controlled by an intermediate validation process. It is worth remarking that validation respects the lightness of constraints, since users may still read and save invalid pages. Validation is meant to be helpful for both readers and authors, without sacrificing "The Wiki Way".

The remainder of this chapter is structured as follows. Section 7.3 discusses related works and various scenarios where light constraints need to be accounted for. Section 7.4 introduces our generic framework, describing the underlying data model and the role of validation. Section 7.5 focuses on actors (which may be either users or system components), discussing how their actions interact with the validation process. Section 7.6 presents a proof of concept implementation of our model in MoinMoin [49]. Section 7.7 concludes the paper and discusses future work.

7.3 Light Constraints

Drawing the word "wiki" close to the word "constraint" sounds as an oxymoron. We noticed however that specific forms of constraints are surprisingly compatible with wiki systems and they can be fruitful exploited by them. Generally speaking, we define as an *informal constraint on wiki content* any kind of rule that a wiki page ought to satisfy. Our focus is thus on constraints which do apply to the *content* of wiki pages, not to other entities like URLs, metadata, keywords, user profiles, ...

Two different classes of informal constraints can be distinguished:

hard constraints constraints that wiki pages must satisfy at any given time instant to be practically useful. An example is the need of having syntactically correct pages (i.e. pages which can be parsed by the wiki engine);

light constraints constraints that can be (temporarily or permanently) violated, without inhibiting the proper run-time behaviour of the wiki.

Light constraints are particularly relevant to wiki systems, since they can give fruitful help to the authors without sacrificing the wiki open editing philosophy. The lightness of such constraints plays a leading role: they can be verified providing detailed error reports, but users can ignore them. On the other hand, when verified, they improve the wiki authoring process.

In this section we present various scenarios where existing wiki systems have already dealt with light constraints (either in an implicit manner or by adopting ad hoc solutions), and new scenarios where the presence of light constraints can be observed. These scenarios taken all together emphasize that the relationship between wiki systems and light constraints is already pervasive within the wiki community. We believe that relationship deserves deeper investigation.

7.3.1 Scenarios from Existing Wiki Systems

A very basic example of light constraint is the verification of the correct spelling of words. Some wikis are supplied with an internal tool that spell checks content on

request. For instance, MoinMoin [49] integrates a Python module that validates a document against a dictionary and a list of exceptions. New words can be easily added modifying the list of exceptions using the wiki. Note that such spell checking is an optional operation that users can activate on demand. Still, the lightness of the constraint is preserved, since users can save pages without caring about correct spelling. Similarly, DokuWiki [39] implements a 2-phase editing process that allows users to edit a page, to switch in correction mode and fix misspelled word (by invoking a server-side spell checker) and then save the final document. Moreover, DokuWiki is designed to help users, teams and work groups in producing documentation: verification can be particularly interesting in such kind of wiki applications, since pages are subject to some rules about correctness and well-formedness.

The correct management of intra wiki links is another field of application for light constraints. The "broken links" do not represent a real problem, since they are practically used to create new pages. Actually, two classes of dangling links have to be distinguished: those intentionally created to add new pages, and those created (often unintentionally) when deleting fragments or whole pages. A very interesting example in such area is PurpleWiki [56]. PurpleWiki is a wiki-clone that implements a fine-grained linking mechanism, through *purple numbers*: paragraphs, heading, lists, and other text fragments are labeled with a number used to reference that elements. It is worth ensuring that any referenced purple numbers really exists, in order to avoid dangling links. Delete and move operations, as well as addition of new content, need to be carefully managed and can be once again bounded with light constraints. Moreover, it can be useful to express constraints about the non-existence of unreachable pages. Some wikis can retrieve a list (usually called *orphan pages*) of those pages, which are usually re-connected to the rest of the wiki site by a manual intervention. No wiki system we are aware of provides users a direct way for preventing the creation of orphan pages, after deleting a page or a fragment. However, their absence is a common and implicit requirement that should be fulfilled.

Light constraints can be also used to ensure minimum user capabilities. In TWiki [108] whenever a new user registers, a page with the corresponding profile

is created according to a given master page. Such master page allows users to automatically set their profile, which can be erroneously modified preventing users to modify their own page and permissions. A light constraint is implicitly defined on TWiki pages, in order to prevent the cancellation of such access control data.

In addition, a new class of light constraints can be identified considering *properties shared* among pages belonging to the same group. Many times wiki users define spontaneously conventions on pages in order to ensure uniformity on wiki subsections: they define the structure of specific pages, the type of content, the order of the elements and so on. These requirements are often non-written and manually checked or simply ignored. For instance, in [32] authors discussed the adoption of wikis within an Italian academic community, reporting examples of repeated pages, structures and patterns developed in that context.

Templating mechanisms ease the enforcement of such uniformity. WIKIPEDIA implements a powerful templating engine: to provide a page users simply instantiate a template assigning values to its variables. For instance, a summary table in the Oak page[6] is described with the following markup:

```
{{Taxobox
| color = lightgreen
| name = Oaks
| image = Quercus robur.jpg
| image_width = 240px
| image_caption = Foliage and acorns of
    ''[[Pedunculate oak|Quercus robur]]''
| regnum = [[Plant]]ae
| divisio = [[flowering plant|Magnoliophyta]]
| classis = [[dicotyledon|Magnoliopsida]]
...
}}
```

[6] http://en.wikipedia.org/wiki/Oak

In this case, expressing constraints on the final structure of a page does not make sense, but it can be interesting to enforce the instantiation of a core set of template variables. Such constraints has not to be always respected (in a page under construction or after an intermediate saving, it is fair to have an invalid state) but they can be used to notify users of the need of more information (the same role played by stub pages[7]).

WikipediA gives us the opportunity of outlining a different form of light constraints, which ensure consistency among lists of elements shown in different pages. Consider for instance the set of countries[8] described in WikipediA: many different lists of these states can be found, ordered by name or by population, grouped by continent, by timezone, and so on. All these lists are manually maintained and no check is automatically performed to ensure they contain the same sets of elements upon editing. Similarly, the correctness of the order of elements in a list is not checked. Once more, such controls do not interfere with the editing process which remain spontaneous and free.

The dilemma between unstructured wiki pages and structured ones has been investigated in [44]. The authors stressed the need of structured information, considering it a way to help users in stating their ideas and comments. They introduced the concept of wiki templates, which are pairs of edit/display templates. When a page is viewed it is formatted according to its view template; when it is edited a set of editable text area will be supplied, one for each "hole" in the edit template. Users cannot modify the whole content and structure of a page, rather only the areas identified by holes. The apparent contrast with The Wiki Way is solved using a tailoring process: users can freely modify the templates, so that no limitation is enforced on editing. Wiki templates embodied a different form of light constraints: instead of validating them after an editing session, they are enforced a priori. The possibility of freely edit templates makes such constraints light.

[7] `http://en.wikipedia.org/wiki/Wikipedia:Stub`

[8] `http://en.wikipedia.org/wiki/List_of_countries`

7.3.2 Novel Scenarios

Before discussing how light constraints are expressed and validated in our model, it is worth introducing two possible new scenarios: WikiFactory and Miki. Light constraints management turned out to be generic enough to address domain-specific issues we found in these two, unrelated projects.

WikiFactory

WikiFactory [37] is a framework designed to automatically produce domain-oriented wiki sites. The idea came by examining how users use to create similar pages and structures in wiki sites for a specific domain. What the authors observed is that most of the work is completely manual, time-consuming, and error-prone. On the other hand, each domain suggests a set of pages, links and data structures that each wiki site used in that domain should have. An example is a wiki site for a university department, which is supposed to have a page with the list of professors, for each of them a brief description and a list of courses, and for each course information including an enrollment page for the exam. Instead of manually creating such pages, the authors proposed to automatically produce them from an ontological description of the departments and a set of instance data.

WikiFactory is a Java application based on semantic web technologies which takes in input an OWL document describing a domain-oriented wiki site, and produces pages for a specific wiki engine. Even if the very early implementation produces only content for MediaWiki, the architecture is independent from the final wiki engine. The core of the application is the ontological description created by an ontology expert, in charge of writing on OWL document about a specific domain, and a domain expert, in charge of completing such OWL document with data about a specific instance of that domain.

At the first installation, such ontology is actually transformed into a set of consistent pages. An important open issue of WikiFactory is how to preserve the consistency between the ontology and the wiki pages upon page editing. We do not

want to prevent users to freely modify content, in order to preserve The Wiki Way. Still, it would be desirable updating the ontology according to user requests. Light constraints can be really helpful in such scenario: consistency can be described as a light constraint, so that whenever a user changes a page she can be notified about the consequences of the change. As expected, many more issues need to be investigated about the techniques for updating the ontologies, for versioning changes, for solving conflicts and so on, but even this scenario shows the flexibility of a model based on light constraints.

Miki

Miki is the code name of an ongoing effort for creating an infrastructure for collaborative authoring of formalized mathematics on the web. The web already presents examples of web-enabled digital libraries of formalized mathematics [4, 6, 109], but despite the web-friendliness of such libraries for browsing purposes, their authoring process is far from The Wiki Way and is often centralized and managed by the developers of a given proof assistant or theorem prover. Miki aims at importing The Wiki Way in the authoring process of libraries of formalized mathematics.

Interesting challenges to the wiki community are posed by Miki. Some of them are related to the usability of wikis for editing content which requires high interactivity, like proof scripts, but are not relevant to our discussion here. Others are related to the logical consistency of what is shown to the user. Concepts from the library are not isolated, but can be linked together by a *requirement* notion; for example: the proof of a theorem on algebraic ring structures is likely to require the availability of a definition of rings in order to be properly proof checked. If the definition of groups changes, it is likely for the proof to need adjustments as well, or it will probably fail a proof checking test. A user working on such broken pages needs to be aware of their brokenness to avoid him trusting (in her mind) unproven mathematical assertions.

Since libraries of formalized mathematics are also often used for presenting formal mathematics, they also support free form pages (sometimes called *theories*) which

are used to present mathematical results and which contain references to formalized concepts. During rendering those references are inlined and shown to the user. This poses the additional need of verifying the logical consistency of a set of concepts, giving feedback to users who are reading theory pages.

In Miki we can represent the logical consistency of mathematical concepts as a light constraint, and we will be able to address both the above issues. The design of Miki, which is still in early stages of development, is an instantiation of the generic architecture we present in this chapter.

7.3.3 User Experience Requirements

In Section 7.3.1 and Section 7.3.2 we gave evidence of the existence of light constraints in real life collaborative editing tasks. No wiki system we are aware of support them in any way. For this reason, even what does "supporting them" mean is an unanswered question. In this work we give one answer, hoping to foster discussion on this subject in the wiki community.

We claim that an authoring system is said to support light constraints if:

(a) it helps the editing work of authors giving visibility to constraint violations;

(b) it helps the work of tailors (the users which coordinate the collaboration on set of pages) enabling the description of constraints and their association to pages.

Instantiating such a system in the setting of a wiki system poses additional requirements on the way users should interact with it. All boils down to respecting the wiki way of working and is expressed by the following set of requirements:

Requirement 7.3 (Unconstrained Saving) *Authors should not be forced to resolve all constraint violations in order to save a page.*

In apparent contrast with the purpose of constraint support, Requirement 7.3 stresses the fact that constraints are meant to help authors without diminishing their editing freedom.

Requirement 7.4 (Freedom of Constraints Definition) *Tailors should be able to work on constraints and associate them to pages using classical wiki techniques.*

Requirement 7.4 includes providing simplified markup for constraint definitions, and versioning of both constraints and their relationship with pages.

Requirement 7.5 (Constraints Visibility) *Information on constraints should be visible to all users.*

Requirement 7.5 is meant to provide visibility of all information relative to constraints (which are associated to a given page, which are violated and which are not, ...) to all users, i.e. not only to authors during page editing. This would help diminishing the gap between page producers and page consumers (the more is visible that something need to be fixed, the more is likely that someone will fix it), and ease the work of WikiGnomes.

We claim that all the above requirements are fulfillable in a wiki system and in the next sections we describe the skeleton of such a system.

7.4 Data Model

Our proposal for encoding light constraints in wiki system is to represent them as *validators*: computational entities able to decide whether a wiki page fulfills a given light-constraint. Validators will be associated to pages. View, Save, and other actions on pages will be changed to exploit validation outcome. Most notably: Save will become conditional on the validation outcome (or on an explicit "forced saving" required by the author) and View will notify every wiki user of the validation status of the viewed page.

This section and the next one are devoted to describing a generic architecture which implements this proposal. Here we present the concepts and the static entities which characterize it (what we call the *data model*) while in Section 7.5 we focus on the actors which compose it and on how the behaviour of the usual wiki actions is changed to exploit validation.

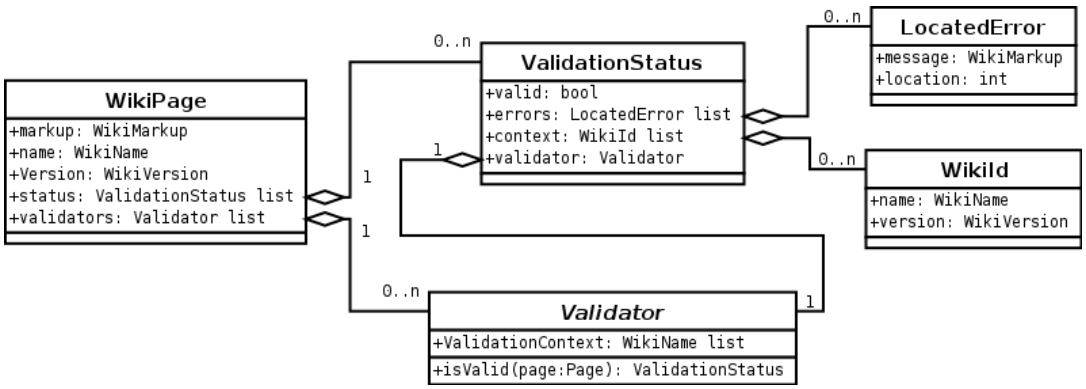

Figure 7.1: UML sketch of the data model

Figure 7.1 is an Unified Modeling Language (UML) sketch of the data model. The basic entity of all wiki systems is the page, which is reported on the left of Figure 7.1. At the very minimum a page is characterized by three properties:

markup a text string containing the actual wiki markup the user sees when editing a page and that is rendered on-the-fly upon page viewing. Its actual syntax is system dependent;

name a text string denoting univocally a page inside the system, the name should follow system-specific conventions (like `CamelCase`) since it is used to ease linking mechanisms;

version a text string denoting the version of a page; over the set of versions a total order should be defined.

Let's consider the spell checking scenario of Section 7.3.1 (an example that will follow us in this section), a sample page might be represented in the minimal data model as follows (using a syntax inspired by the object as record metaphor):

```
Page about = {
  markup = "This peper rocks, follow TheWikiWay";
  name = "AboutThisPaper";
  version = "3.141592";
}
```

A validator is intuitively a function encoding a single light constraint. A validator can be applied (via the `isValid` method) to pages and returns either a statement that the page is valid (with respect to the light constraint encoded by the validator itself) or a statement that it is not. This computational aspect of validator is needed in order to be able to take decisions based on its outcome as it will be done, for instance, for conditional saving. In case a page turns out not to be valid, the validator returns a list of *localized error messages*: textual messages which are bound to particular characters in the wiki markup. This choice is motivated by the need of guiding authors toward the fulfillment of constraints: localized errors are easier to spot than global ones and hence faster to fix (at the very minimum the spotting time is reduced).

Note that the textual part of messages can actually be wiki markup to provide fancier (hence more expressive) messages to the user. In the example above the ideal error message, representable in our data model, would be located at the beginning of the string `peper` and would contain a statement that the word does not spell check, together with a hyperlink for adding the word to the current spell checking exceptions page.

In order to fulfill Requirement 7.4 (see Section 7.3.3), the association among a page and its validators should be part of the information which are editable by users, hence the following property of pages:

validators a list of validators, one for each light constraint, which should be enforced on the page.

In the frequent case of wiki systems supporting hierarchical structuring of the page namespace, the validators property is likely to be inherited to enforce a common set of light constraints to a particular area of a wiki site. Addition and removal of constraints on large page sets, for example, can then be performed changing a property in a single (root) page.

Requirement 7.5 is implemented in the data model by keeping track of the *validation status* with the following property:

status a list of validation statuses, one for each validator, which were associated to the owning page when the last validation attempt has been performed. The key properties of a status are `valid` (a boolean value denoting the success of the `isValid` invocation), `errors` (the list of located errors, which is meaningful only if `valid == false`), and `context` (a validation context, discussed below).

From several of the scenarios discussed in Section 7.3 we learned that light constraints are not always local to a single page. They often need additional information that should be found on wiki pages, or even external to the wiki site.[9] In the spell checking scenario for instance, the validation is parametric on a dictionary external to the wiki site and on an extra page containing additions to the dictionary. That page is likely to be editable by users. A *validation context* represents the set of wiki pages (referenced by their names) on which a validator is parametric. Note that pages referenced from validation contexts are non-versioned, since to better ensure liveness of wiki content validation will always be attempted using their more recent versions.

Validators are parametric in their validation contexts. In a sense, we think about validators as taking in input both the page they should validate and the validation context. This way we can for instance have parts of a wiki site spell checked using a list of geeky words as exceptions and other parts using a list of biological terms.

In order to fulfill Requirement 7.5 all information about the validation status should be available to users. This explains the `context` property of validation statuses. Error messages are not enough to explain to users why a page is invalid. Pages which are part of the validation context of other pages may indeed change, and that can have effects on the validity of other pages. Consider once more the spell checking scenario, a user removing a word from an exceptions list may increase the amount of spell checking errors in other pages. The information on why this page is no longer valid need to be available to users, in this way we record the *actual validation status* as a property of validation status. The actual validation status is

[9]The latter form of additional information should however be minimized in order to preserve the ability of users to influence validation.

a list of page references corresponding to the validation context, together with their version. This way it will always be possible to retrieve the exact set of pages which leaded to a particular validation outcome.[10]

7.5 Architecture

The presence of validators changes substantially the workflow of wikis based on our model. Each operation on a page becomes *parametric* in the set of validators associated to that page. In particular, viewing a page becomes viewing both its actual content and its validation state (supplied with all the relevant information to spot and fix validation errors), while saving a page becomes invoking validators and, if not valid, deciding whether saving it or not.

We designed a general-purpose architecture that allows users to associate and run sets of validators on wiki pages.

Various entities compose such architecture:

roles played by users [11]

visitors and authors users who view or edit the actual content of wiki pages. No particular skills are required nor more expertise than that required by common wiki sites.

tailors users in charge of configuring and selecting validators associated to a given page. Usually, but not necessarily, users playing this role are more experienced than others.

software components:

wiki engine the actual wiki engine working as any other wiki clone does, but also in charge of invoking validators.

[10] Note that changes to external information used by validators can't, in general, be captured in the same way: yet another good reason to keep as much validation information as possible represented as wiki pages

[11] as often happens, the same user can play different roles at different times

validators entities that actually validate page contents, as discussed in Section 7.4.

batch validator stand-alone component which verifies whether or not changes on a single page affect validation of other pages.

In order to explain the purpose of each entity in our architecture, we discuss individually the two main operations of a light-constrained wiki—VIEW and SAVE—focusing on both their differences with the corresponding wiki operations and on the architectural choices behind them. Later we discuss how validation affects other wiki operations, like versioning and diff-ing.

7.5.1 View Action

Our architecture changes the view operation to *annotated viewing* whenever a user accesses a page. Its content is rendered as usual, but is enriched with a detailed report of the validation process. Figure 7.2 summarizes the runtime behaviour of the VIEW action (sequential numbers in the figure denote the sequence of micro-steps which compose the action).

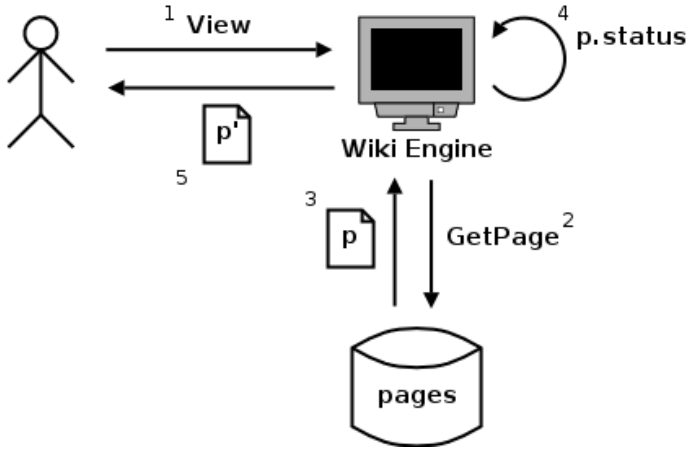

Figure 7.2: Runtime behaviour of the VIEW action

The user involved in such scenario is a common user who simply requires a page

(step 1 in Figure 7.2); the wiki engine retrieves it (steps 2–3) and its associated validation status (step 4) before returning it to the user (step 5).

Some points are worth being remarked about the generalization of our schema. First of all, we have depicted a content repository without dealing with its actual implementation: wiki systems use different techniques to store pages, from MySQL[12] databases (as WikipediA does) to plain text file (as most wikis do), from RDF tuples [86] to Subversion[13] repositories [50]. Moreover, they implement specific solutions to associate metadata to pages (fields in databases, external log file, specific lines in text files, and so forth) and these metadata can be usually customized or extended. We propose to introduce a new class of metadata about the validation status of a page. The key point is that the wiki engine gets pages and retrieves such status, previously set by validators: no matter how these actions are actually implemented.

The analysis of the View action from the user perspective is interesting as well. Few changes are introduced on the behaviour of readers, who access wiki content as they always do, but can now read suggestions from validators or simply ignore them, if not interested. The wiki engine produces a compound page where content and validation outcome are displayed together (see Figure 7.5 for a sample screenshot). It is worth spending some words about the format and detail of such outcome: different pages can be involved in the validation process so that several information need to be displayed, often not stored in the page being validated. Consider for instance, the example of Miki discussed in Section 7.3.2. A page can be invalid because some lemmas referred by that page fail to proof check and, in turn, even these lemmas can be inconsistent because of other related properties. It is very useful to show users such a chain of relationships and consequences. Moreover, errors should be localized, as discussed in Section 7.4. Issues related to the usability and cognitive overhead problems in managing a so huge amount of information are as inevitable as complex, but we consider them out of the scope of the present work.

[12] `http://www.mysql.com`

[13] `http://subversion.tigris.org`

As a final remark, such rich information and, above all, their availability for the whole wiki community simplifies and speeds up the production of pages that fulfill light constraints, since any user can easily discover errors or imperfections and can spontaneously fix them. In a sense, providing a validation feature even for the VIEW action simplifies the life of WikiGnomes and paradoxically improves both content and sharing habits among wiki users.

7.5.2 Save Action

While the actual text editing is not affected by the presence of light constraints, saving is. Our architecture changes such operation into *conditional saving*: whenever a user saves a page, validation is performed and according to its outcome a proper page is returned. Two outcomes are possible: the page is valid, and a simple acknowledgement is return to the user, or it is not, and a detailed report of errors is returned (similar to that shown in Figure 7.5). Then the user can choose whether saving that page or not. Figure 7.3 summarizes the runtime behaviour of the SAVE action.

After submitting a page (step 1 in Figure 7.3) the wiki engine retrieves all the validators associated to that page (steps 2–3) and runs each of them on the submitted content (step 4). The internal structure of validators, as well as the language used to implement them and/or configure the validation itself, are not relevant at this point. What is relevant here is the strong separation between the validation process and the common wiki workflow: such distinction makes it easy to apply a general model to different wiki clones by introducing few modifications and importing external validators or implementing them with few efforts. Note also that no limitation is imposed over the number (and variety) of validators: different kind of light constraints can be checked over the same page, and different communication protocols and validation engines can be exploited.

In case a page is valid, a confirmation message is delivered to the user (step 5) who goes on surfing (or continue editing) normally. On the contrary, in case a page is not valid, two options are provided (still step 5): the user can "forcedly save",

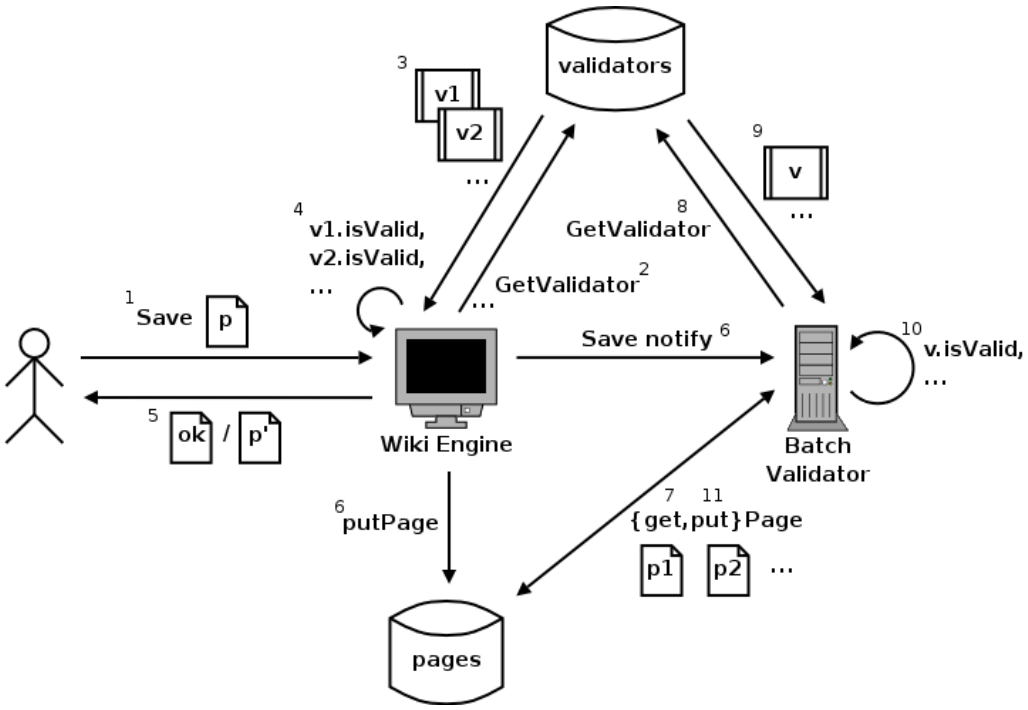

Figure 7.3: Runtime behaviour of the SAVE action

being aware the page violates some light constraints, or can fix errors and try saving again.

"Forced saving" is crucial: it fully adheres to "The Wiki Way"—as users can freely modify content and ignore validators—and allows users to save work in progress pages (not yet valid), or intentionally invalid pages (for instance, as examples of common errors and bad practices). For these reasons we claim that collaboration is not hindered when adopting our approach.

When a user accepts saving a page, two events are triggered (step 6): the new page is stored into the wiki page repository and a notification is sent to a component we call *batch validator*. Note that, according to the data model, storing a page in the repository does not mean only storing its content, but also its validation status. The batch validator is a process running in background that we introduced in order to address context-related issues. As discussed before, validation is not limited to a single and isolated page, but is rather a global process that can involve sets of

pages up to the whole wiki. Therefore running validators only on the submitted content would not be enough, since changes can affect validity of other pages too. Our solution is to notify save events to a listening daemon (the batch validator) and letting it run validators over each page included in the current context. Details about the communication protocol between the wiki engine and the batch validator are not relevant here.

The batch validator proceeds as follows: get all validators associated to all pages in the context of the page being saved (steps 7–9), execute them (step 10), update pages accordingly to the validation outcome (step 11). The latter action of updating does not trigger any further validation. Note that the batch validator works behind the scenes, while the user has simply received a saving confirmation message. This choice is motivated by the possible huge amount of pages involved in validation.

The presence of the batch validator drives us into a very interesting field: the analysis of how versioning is affected by validation. In the classical wiki workflow a new version of a page can be created only by an editing session (actually some wikis allow users to group minor changes or adjacent versions into a single one) but such approach is not enough in our setting. Users, in fact, can be interested in knowing that a page changed its validation state but this can happen without explicit modifications on that page.

Consider the spell checking example: it can happen that, adding a new word to the exceptions list, an existing page becomes valid; the two statuses of that page, before and after having added the new word, are worth be traced and reported to the users. A new version of a document should be created either after an editing session or after an automatic update done by the batch validator. Yet, these two kinds of versions are conceptually different: in a sense, there exist two overlapping and intermixing version trees and users should be able to see both.

Inevitably even the Diff operation changes, since a Diff between two versions does not mean comparing only their content, but even their validation states. At first glance, it means simply producing a Diff page composed by two parts: one devoted to show changes in validation state and another one dealing with content

(as expected, one of these part can be empty). However, a more complex issue need to be addressed, once again because of the validation context. Such a DIFF should provide users precise references to the content modifications that causes that local change, even if they occur in other pages. Obviously the richness and granularity in the DIFF output opens the doors to a series of complex issues related to usability and cognitive overhead, but we consider these aspects out of the scope of this work.

Last but not least, we need to add details about the association between pages and validators, as well as the configuration and encoding of validators themselves. We introduced a specific user role called *tailor*. The term "tailor" indicates the ability of cutting out validators and configuring them for specific (class of) pages. In [64], the authors noticed that, even when the whole community is affected by system customization and tailorability, it is very common that a restricted set of users actually perform such task: although that work primarily focused on software customization, the same observation can be extended wherever a specific and quite difficult configuration task has to be accomplished, and highly-skilled users or domain experts need to be involved. On the contrary, in [65] authors claimed that tailorability should be extended to all the users: yet, differences among user expertises exist and are required to exist, but the customization itself is improved by involving average users too. We are still investigating which level of tailorability is suitable for light-constrained wiki systems, also considering that boundaries among roles are blended in the wiki setting.

Considering the generality of our architecture, a wide spectrum of tailors can exist: on the one edge, a validator can be hard-coded within the system, so that a tailor can at most select validators; on the opposite edge, a validator can be completely programmable, and a tailor can completely decide its internal behaviour; in the middle, validators can be parametrized so that a tailor can set parameters (besides associating them to pages). The extreme solution consists of coding directly the behaviour of validators through a wiki syntax and allowing any user to describe such behaviour.

7.6 Proof of Concept Implementation

We wrote a proof of concept implementation of the architecture described in Section 7.5 which adds validation capabilities to MoinMoin [49]. Its aim is not to extend MoinMoin into a fully general constraint-enabled wiki system, but rather to show the non-invasiveness of a similar extension to a popular wiki system.

The main component of the implementation is a new parser, which in MoinMoin terminology defines one of the possible formats a wiki page (or fragments of it) can be written in. To use the parser—our being called `validate`—it is enough to add a processing instruction at the beginning of a page (or of a delimited fragment). It receives as arguments a list of validators, each of which can be in turn be passed a list of validator-specific arguments. Figure 7.4 shows a snippet of MoinMoin markup of a page which uses our parser. It represents the markup of a wiki page on an hypothetical wiki site used to coordinate paper submissions to a conference on the wiki topic. Line 1 requires the page to be validated by two validators. The former (`abstract_length`) checks that the abstract is no longer than 200 words, while the latter (`spellcheck`) ensures correct spell checking using a page named `WikiWords` as its exceptions list.

The validation status is stored, together with the list of validators associated to a page, encoded in the `extra` field of the `edit-log` file associated to each page. `edit-log` is the place where MoinMoin stores the metadata associated to a page. When a page which uses the `validate` parser is accessed its validation status is retrieved and used to annotate the page markup. Annotations come in two flavours: a validation summary and a set of located errors. The summary is added at the end of the page and reports, for each attached validator, whether the validation has been successful or not and the description of each error. Located errors are reported as links in the markup with (CSS) pop-up descriptions of the errors, pointing to the corresponding error entries in the summary. Figure 7.5 is a screenshot of a MoinMoin page rendered via `validate` showing both the validation summary and one located error (at the beginning of the abstract). A similar feedback has been

```
#format validate abstract_length(200) spellcheck(WikiWords)

#format wiki

= Constrained Wiki: an Oxymoron? =

'''Author(s)''': ["Angelo Di Iorio"] and ["Stefano Zacchiroli"]

'''Abstract''':

  ["The Wiki Way"] is in apparent contrast with any kind of
  editing-time constraint. Nonetheless it is well-known that
  communities of users involved in wiki sites have the habit of
  establishing best authoring practices, and it is a frequent
  need of domain-specific wiki system to enforce some kind of
  well-formedness on page content. A general framework to think
  about the relationship of \WIKI{} system with constraints is
  missing.
  ...
```

Figure 7.4: MoinMoin markup of a page equipped with validators

returned to the user who last edited that page, before he chose to forcedly save.

After markup annotation, the `validate` parser acts as a "proxy" invoking again the internal MoinMoin machinery to discover the appropriate parser for the annotated markup and render it using the abstract formatter which gets passed to parser.

Validators are stored as Python scripts server side, are loaded using the `importPlugin` mechanism of MoinMoin, and are invoked when a page is saved, possibly requiring

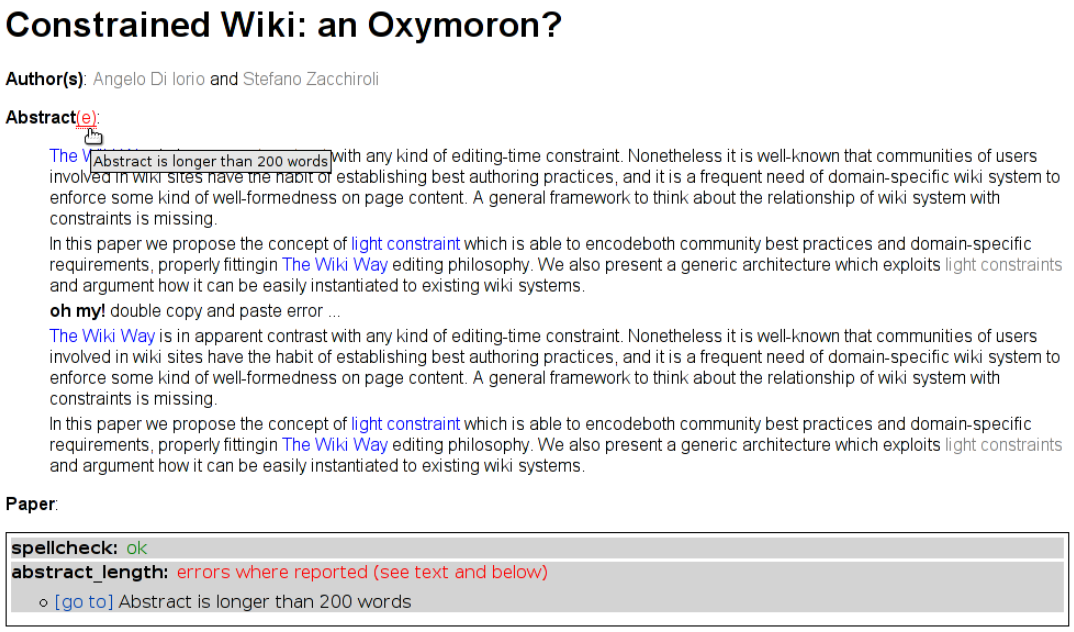

Figure 7.5: Screenshot MoinMoin extended with validation support

forcing by the user in case of validation errors.

The batch validator has been implemented in Python as a daemon with XML-RPC interface. Once notified of a save, it starts digging MoinMoin pages to discover which pages have the page being saved in their validation context.[14] Each of them is then retrieved (using wiki RPC interface [120] `getPage` method), (re-)validated, and stored in case of validation status change (using wiki RPC `putPage`). In order to push toward the wiki the information about validation status changes, the implementation of `putPage` should support the `attributes` argument. In MoinMoin that was not the case, we patched it and reported the bug upstream.

Yet being "proof of concept", our implementation shows that adding support for light constraint to existing wiki systems is far from being challenging. The peculiarities of MoinMoin we exploited are just a few: the extensibility of its markup and metadata, and its wiki RPC interface. All those features are available in the

[14] this is actually implemented naively using wiki RPC interface method `getAllPages`, of course this can be optimized having the batch validator keeping an internal record of validation contexts

implementations of many existing wiki systems. The only parts of MoinMoin code we actually had to patch are the reaction to a "save" request (adding user notification if she attempted to save an invalid page without explicitly forcing the save) and a save time hook which notify the external batch validator. Taken together they sum up to less then 100 lines of code. The batch validator is fully reusable for other wiki systems (assuming they implement the wiki RPC interface).

7.7 Conclusions

The openness and freedom of the wiki editing process has a strong impact on the final pages: they are frequently updated, rich, and continuously improved, but also under-controlled and flawed. We noticed that wiki pages improve their correctness and clearness when a set of rules are enforced by the community or by the wiki system. These rules cannot be strict prohibitions that prevent users from freely expressing their ideas and comments, rather they should help them in doing work that would be otherwise done later or never. In this paper we referred these rules as light constraints and we proposed a general framework to manage them. Our goal is awakening the community to the existence of a strong connection between wikis and constraints, and provide a first general model that can be applied to heterogeneous scenarios.

Basically our solution relies on a strong distinction between the actual wiki engine and a set of validators, in charge of verifying the respect of light constraints associated to the pages: by exploiting validators wiki systems can provide *conditional saving* and *annotated viewing*. The proposed solution does not change the user editing experience, as opposed to other solutions like [44].

In the case of spell checking, each page can be associated to an external validator that actually spell checks content, looking for a dictionary page, whenever that page is saved. In the case of inconsistent and unordered lists of WikipediA, each page can use a validator who knows which other pages have to be consistent with the current one: such validator verifies whether all those lists contain the same elements.

Even the new scenarios we described can be addressed: for WikiFactory, a validator associated to a page recognizes the template for that page and verifies whether that page matches it or not. Similarly in Miki the consistency of an edited page with respect to the mathematical repository is checked by external validators that give a response back to the wiki engine. A point-to-point description of the remaining scenarios is as useless as boring, but it might not be difficult to instantiate the observations above to each of them.

An analysis of our architecture can be completed verifying whether all the requirements for a constraint-enabled wiki are fulfilled. In Section 7.3.3 we identified three such requirements: unconstrained saving (Requirement 7.3), freedom of constraints definition (Requirement 7.4), and constraints visibility (Requirement 7.5). Many times in this chapter we stressed Requirement 7.3 and Requirement 7.5, showing how no strict rules are really imposed on the editing process and showing that all validation information can be given to the users (for instance through the VIEW and DIFF operations and through the localized errors list).

On the contrary, Requirement 7.4 is tricky and not properly addressed. The Wiki Way suggests us to allow any user (or better, any tailor) to freely program validators, directly on the wiki site. Such a solution raises several issues. First of all, it is very difficult to find a language suitable for this purpose, due to the tension among language expressiveness and its simplicity (in term of both syntax and semantics). Scenarios like Miki shows how complex can be validation needs. A second issue is of course security: assuming that a silver bullet language can be found, we need to prevent malicious uses of validators which can be easily provide denial of services. Actually, such approach has already been faced by the so called Community Programmable wikis [29], which allow any user to modify the code of the wiki engine itself, without arriving at satisfying solutions. A compromise solution can be achieved by "tailoring", that is allowing only a subset of trustworthy users to configure and actually write validators. As discussed in Section 7.5, limiting users' tailoring can be, on the one hand, a very good solution to capitalize skilled users work but, on the other hand, a restricted approach which limits average users

potentialities. Note that we do not claim that such a customization is so dangerous to be impossible, rather we think that a more detailed and deeper discussion is required.

Our next step will be investigating such relationship among average users, tailors, languages for programming validators and open editing philosophy. In particular two future directions seem to be equally valid. On the one side, we will try to figure out a general language simple, safe, but enough expressive to allow user to define validators in frequent occurring scenarios. On the other side, we will try to figure out small (different) languages suitable for specific domains (for instance, we are discussing a language to define and verify templates for scenarios similar to WikiFactory).

Our point should be clear now: at first glance constraints and wikis seem to be incompatible, but after a more careful analysis, they can coexist in an interesting and synergetic oxymoron.

Original Contributions

The content-centric view of formal reasoning predates this thesis author work in the HELM project and on the MATITA proof assistant. he idea of light constraints, their development and prototype implementation is an original contribution of this thesis author as a joint work with Angelo Di Iorio.

Related Publications

Part of the work described in this chapter has been previously published in the following papers:

- Angelo Di Iorio and Stefano Zacchiroli.
  Constrained Wiki: an Oxymoron? [52].
  In Proceedings of WikiSym 2006: the 2006 International Symposium on Wikis[15], ACM Press, 2006, ISBN 1-59593-417-0, pages 89–98.

[15] `http://www.wikisym.org/ws2006/`

Appendix A

Summary of Related Software Packages

Being one of the developer of MATITA did not require me only to work on the code base of the proof assistant itself. I have indeed learnt that developing large-sized project—as MATITA arguably is—in OCaml[1] often also requires developing a toolbox of libraries, tools, and best-practices that in other languages come for free. This is probably mainly due to the scientific thrust of the programmers community behind OCaml and the fact that the language, in spite of its age, is not (yet?) so widespread.

Of the several items that I have collected in such a toolbox in the 6 years I have spent working on software of the HELM project, several have grown into standalone software libraries which are used by MATITA but which are meaningful per se and rather loosely coupled with the system. Most of them have been developed collaborating with other members of the HELM team and some of them are written in languages other than OCaml. In many cases, a non negligible part of the work also required maintaining the software packages for the Debian GNU/Linux distribution[2], the distribution of choice of the HELM team.

This appendix is meant to provide an index of those software libraries, as a reference of the work which I have done during my Ph.D., work that does not reasonably fit elsewhere in a thesis dissertation. This appendix also serves as a collection of

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

[2] `http://www.debian.org`

pointers for developers interested in knowing more about some of the technologies that we exploited to implement some of the distinctive features of MATITA.

Licenses. All the software packages listed in this chapter are free software and are distributed either under the terms of the GNU General Public License[3] or under those of the GNU Lesser General Public License[4].

A.1 OCaml HTTP

OCaml HTTP [125] is an OCaml library freely inspired by the `HTTP::Daemon` Perl module[5] that permits to write simple HTTP daemons in OCaml.

The main API permits to define a HTTP daemon specification, which contains, among other parameters, a callback function that is invoked each time a request is received. The callback function will be invoked with an instance of an object representing the received HTTP request and an `out_channel` connected to the remote HTTP client socket.

Then the HTTP daemon can be started invoking the `main` function passing a specification. Each time a client connect to the TCP port bound by the daemon, OCaml HTTP will parse the request and instantiate the `request` object. If all goes well the callback is invoked, otherwise appropriate error messages will be sent back to the client without disturbing the callback.

Several facility functions can be used in the callback to easily send headers, error responses, files, or abstract HTTP `response` objects. The "hard way" can be chosen as well to send data directly on the `out_channel` (especially useful for sending huge amount of data incrementally to the client). The two approaches can also be mixed.

Daemon specifications are also used to specify other parameters governing daemon behaviour like: TCP port and address to bind, way of handling incoming requests (handle all of them in a single process, fork a new process, or spawn a new

[3] `http://www.gnu.org/copyleft/gpl.html`

[4] `http://www.gnu.org/copyleft/lgpl.html`

[5] `http://search.cpan.org/dist/libwww-perl/lib/HTTP/Daemon.pm`

thread for each incoming request), timeout, authentication requirements (username and password for HTTP basic authentication).

The library also contains a tiny implementation of a HTTP client which can be used to retrieve resources via GET HTTP method and to iterate on them (useful for huge resources which cannot be kept in memory at once).

Name: OCaml HTTP

Homepage: `http://www.bononia.it/~zack/ocaml-http.en.html`

License: GNU Lesser General Public License

Debian packages: `http://packages.qa.debian.org/ocaml-http`

Role of this thesis author: Main author, current maintainer, Debian packages maintainer

A.2 LablGtkSourceView

LablGtkSourceView are the OCaml bindings for GtkSourceView[6], a GTK+ widget which extends the standard GTK+ text widgets implementing syntax highlighting, automatic indentation, and other typical features of source code editors.

Using LablGtkSourceView the programmer can instantiate and use GtkSourceView widgets in OCaml programs which use GTK+ through the LablGtk interface.

[6] `http://gtksourceview.sourceforge.net/`

Name: LablGtkSourceView

Homepage: `http://helm.cs.unibo.it/software/lablgtksourceview/`

License: GNU Lesser General Public License

Debian packages: `http://packages.qa.debian.org/lablgtksourceview`

Role of this thesis author: Main author, current co-maintainer with Maxence Guesdon, Debian packages maintainer

A.3 UWOBO

UWOBO [125] is a web-service based XSLT [124] processor.

UWOBO interface is based on HTTP GET method: UWOBO can indeed be contacted with a proper URL containing all parameters needed to request style sheet application.

UWOBO supports not only the usual single XSLT style sheet application to an XML document, but also application of style sheets chains. A XSLT *style sheets chain* is a transformation described providing an ordered list of XSLT style sheets. First style sheet is applied to the input XML document, output of this transformation is the input for the application of the second style sheet and so on until the last one. Last transformation output is the final output sent back to the client.

UWOBO supports all output properties defined in the XSLT recommendation. Output properties are read from the last style sheet of the chain and can be overridden by proper parameters.

UWOBO is fully developed in the OCaml programming language.

Historic remarks. UWOBO was born at the University of Western Ontario, London, Canada. Later it has been further refined and developed at the University of Bologna (hence the name). Current version of UWOBO has been fully reimplemented from scratch at the University of Bologna.

Name: UWOBO

Homepage: `http://helm.cs.unibo.it/software/uwobo/`

License: GNU General Public License

Debian packages: No packages available

Role of this thesis author: Main author, current co-maintainer with the other members of the HELM team

A.4 HTTP Getter

HTTP Getter [125] is a Web-Service used to manage access to the HELM library.

HTTP Getter interface is based on HTTP GET method: it can indeed be contacted with a proper URL containing all parameters needed to fulfill a request.

Using HTTP Getter you can choose which servers build up your own HELM library (or—if you like—your slice of the world wide HELM library).

The Getter then takes care of mapping HELM document URIs to URL pointing to the real documents and uses this abstraction to fulfill user requests.

Using the Getter, the user can perform different kind of actions on the HELM library documents, mainly:

- retrieving documents (including XML CIC documents, XSLT stylesheets, DTDs and more);
- resolving HELM (e.g. in the `cic:` or `nuprl:` URI scheme) URIs to URLs;
- registering new documents adding them to the HELM library;
- listing contents of the HELM library (or of subdirectories of it) in various formats.

The Getter supports CIC documents obtained exporting the library of the Coq proof assistant and NuPRL documents obtained in the same way from the library of the NuPRL proof assistant.

The Getter is fully implemented in the OCaml programming language.

Historic remarks. The Getter was originally written (and rewritten, and rewritten, and ...) in the Perl programming language by Claudio Sacerdoti Coen and this thesis author. Current version of the Getter has been fully reimplemented from scratch.

Name: HTTP Getter

Homepage: `http://helm.cs.unibo.it/software/getter/`

License: GNU General Public License

Debian packages: No packages available

Role of this thesis author: Main author, current maintainer

A.5 Gdome2 XSLT

Gdome2 XSLT is a small C library that implements a minimal XSLT processor that can be used to apply XSLT style sheets to Gdome2 documents. Actually, it just applies some `libxslt`[7] functions to Gdome2 documents, mapping DOM-like trees back and forth between the format used in Gdome2 and the format used in `libxml`[8].

Bindings for the OCaml programming language are also provided. It requires the Gdome2 OCaml bindings that is one of the modules of the GMetaDOM project (see Section A.8).

While the OCaml part is original, the C code is just a slight modification of the C code found in the implementation of the `XML::GDOME::XSLT` Perl module[9].

[7] `http://xmlsoft.org/XSLT/`

[8] `http://xmlsoft.org/`

[9] `http://search.cpan.org/~tjmather/XML-GDOME-XSLT-0.75/XSLT.pm`

Name: Gdome2 XSLT

Homepage: `http://helm.cs.unibo.it/software/gdome_xslt/`

License: GNU Lesser General Public License

Debian packages: `http://packages.qa.debian.org/gdome2-xslt`

Role of this thesis author: Current co-maintainer with Claudio Sacerdoti Coen, Debian packages maintainer

A.6 LablGtkMathView

LablGtkMathView are the OCaml bindings for GtkMathView[10], a GTK+ widget able to render mixed MathML Presentation and BoxML markup.

Using LablGtkSourceView the programmer can instantiate and use GtkMathView widgets in OCaml programs which use GTK+ through the LablGtk interface.

Name: LablGtkMathView

Homepage: `http://helm.cs.unibo.it/mml-widget/` (look for the "OCaml bindings" section)

License: GNU General Public License

Debian packages: `http://packages.qa.debian.org/lablgtkmathview`

Role of this thesis author: Current co-maintainer with Claudio Sacerdoti Coen and Luca Padovani, Debian packages maintainer

[10] `http://gtksourceview.sourceforge.net/`

A.7 WOWcamldebug

WOWcamldebug is a front end that permits to use the OCaml debugger[11] with GVim[12]. You can run it as you usually run `ocamldebug` and it will execute both the OCaml debugger itself, permitting usual interaction, and a GVim window which will be kept synchronized with the current debugging position. The current debugging line is highlighted and the cursor is positioned at the current event position on that line.

Communications are permitted in both directions: from `ocamldebug` to GVim and vice versa. You can simply ignore WOWcamldebug and use your `ocamldebug` terminal as usual. Alternatively you can use GVim and the provided tool-bar for `ocamldebug` interaction which permits the usual `ocamldebug` motion commands (next, step, back-step, and previous) and more fancy actions like printing the value of the identifier at cursor position.

Name: WOWcamldebug

Homepage: `http://www.bononia.it/~zack/wowcamldebug.en.html`

License: GNU General Public License

Debian packages: No packages available

Role of this thesis author: Main author, current maintainer

A.8 GMetaDOM

GMetaDOM [84] is a collection of libraries, each library providing a DOM implementation. Currently available bindings are for C++ and OCaml.

The basic idea is that, given the availability of DOM implementations for the C programming language (like Gdome2), and given the uniformity of the DOM interfaces, bindings for various programming languages based on the C implementation

[11] `http://caml.inria.fr/pub/docs/manual-ocaml/manual030.html`

[12] `http://www.vim.org`

can be built automatically, providing a small number of hand-coded classes and a set of scripts for the automatic generation of the remaining ones.

Furthermore, since a XML description of the DOM interfaces is provided as part of the documentation in the W3C DOM specification, GMetaDOM adopts XSLT as the transformation language for the automatic generation of the interfaces, and uses the `xsltproc`[13] utility as the XSLT processor.

The advantages of such approach should be evident. In particular, for languages like C++ where a number of different alternative DOM implementations are feasible, each with different characteristics like easiness of use, runtime flexibility, resource requirements, the approach of automatic generation permits to create a set of coherent implementations addressing such issues separately, ultimately allowing the developer to choose the library which fits best her needs.

Name: GMetaDOM

Homepage: `http://gmetadom.sourceforge.net`

License: GNU Lesser General Public License

Debian packages: `http://packages.qa.debian.org/gmetadom`

Role of this thesis author: Debian packages maintainer

Related Publications

GMetaDOM has been described at length in the following papers:

- Luca Padovani, Claudio Sacerdoti Coen, and Stefano Zacchiroli.
  A Generative Approach to the Implementation of Language Bindings for the Document Object Model [84].
  In Proceedings of GPCE'04: Third International Conference on Generative Programming and Component Engineering 2004[14], Lecture Notes in Computer Science, Vol. 3286, pages 469–487. Springer-Verlag, 2004.

[13] `http://xmlsoft.org/XSLT`
[14] `http://www.gpce.org/04/`

References

[1] A. A. Adams. Digitisation, representation and formalisation: Digital libraries of mathematics. In J.H. Davenport A. Asperti, B. Buchberger, editor, *Proceedings of Mathematical Knowledge Management 2003*, volume 2594 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2003.

[2] Stuart Aitken, Phil Gray, Tom Melham, and Muffy Thomas. Phases, modes and information flow in theory development. In Nicholas Merriam, editor, *User Interface Design for Theorem Provers*, 1996.

[3] Stuart Aitken, Phil Gray, Tom Melham, and Muffy Thomas. Interactive theorem proving: An empirical study of user activity. *Journal of Symbolic Computation*, 25(2):263–284, 1998.

[4] Stuart Allen, Mark Bickford, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. Fdl: A prototype formal digital library. Technical Report TR2004-1941, Cornell University, 2002.

[5] Alessandro Armando and Daniele Zini. Interfacing computer algebra and deduction systems via the logic broker architecture. In *Proceedings of the Eight Calculemus Symposium*, 2000.

[6] Andrea Asperti, Ferruccio Guidi, Luca Padovani, Claudio Sacerdoti Coen, and Irene Schena. Mathematical knowledge management in HELM. *Annals of Mathematics and Artificial Intelligence*, 38(1-3):27–46, May 2003.

[7] Andrea Asperti, Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. A content based mathematical search engine: Whelp. In

Post-proceedings of the Types 2004 International Conference, volume 3839 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 2004.

[8] Andrea Asperti, Luca Padovani, Claudio Sacerdoti Coen, and Irene Schena. XML, stylesheets and the re-mathematization of formal content. In *Electronic Proceedings of EXTREME Markup Languages 2001*, 2001.

[9] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Crafting a proof assistant. Submitted to the Post-proceedings of the Types 2006 International Conference.

[10] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the Matita proof assistant. Special Issue on User Interface for Theorem Proving. To appear, 2007.

[11] Andrea Asperti and Matteo Selmi. Efficient retrieval of mathematical statements. In Andrzej Trybulec Andrea Asperti, Grzegorz Bancerek, editor, *Proceedings of Mathematical Knowledge Management 2004*, volume 3119 of *Lecture Notes in Computer Science*, pages 17–31. Springer-Verlag, 2004.

[12] David Aspinall. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*. Springer-Verlag, January 2000.

[13] Lennart Augustsson. Compiling pattern matching. In Jean-Pierre Jouannaud, editor, *FPCA 1985: Functional Programming Languages and Computer Architecture, Proceedings*, volume 201 of *Lecture Notes in Computer Science*, pages 368–381. Springer-Verlag, 1985.

[14] Serge Autexier and Christoph Benzmüller, editors. *Proceedings of User Interfaces for Theorem Provers*, Electronic Notes in Theoretical Computer Science. Elsevier Science, 2006. To appear.

[15] Grzegorz Bancerek. On the structure of mizar types. *Electronic Notes in Theoretical Computer Science*, 85(7), 2003.

[16] Gilles Barthe. Implicit coercions in type systems. In *Types for Proofs and Programs: International Workshop, TYPES 1995*, pages 1–15, 1995.

[17] Christoph Benzmüller, Lassaad Cheikhrouhou, Detlef Fehrer, Armin Fiedler, Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, Wolf Schaarschmidt, Jörg H. Siekmann, and Volker Sorge. Omega: Towards a mathematical assistant. In *CADE*, pages 252–255, 1997.

[18] Christoph Benzmüller, Mateja Jamnik, Manfred Kerber, and Volker Sorge. Agent based mathematical reasoning. *Electronic Notes in Theoretical Computer Science*, 23(3):21–33, 1999.

[19] Christoph Benzmüller and Volker Sorge. OANTS – an open approach at combining interactive and automated theorem proving. In Manfred Kerber and Michael Kohlhase, editors, *Symbolic Computation and Automated Reasoning*, pages 81–97. A.K.Peters, 2000.

[20] Yves Bertot. The CtCoq system: Design and architecture. *Formal Aspects of Computing*, 11:225–243, 1999.

[21] Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25:161–194, 1998.

[22] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito editors, editors, *Theoretical Aspect of Computer Software TACS'97, Lecture Notes in Computer Science*, volume 1281, pages 515–529. Springer-Verlag, 1997.

[23] Bruno Buchberger, Gaston Gonnet, and Michiel Hazewinkel, editors. *Special Issue on Mathematical Knowledge Management*, volume 38(1–3) of *Annals of Mathematics and Artificial Intelligence*. Springer Netherlands, May 2003.

[24] Olga Caprotti. Symbolic evaluator service. Technical report, RISC-Linz, Johannes Kepler University, Linz, 2000. Project Report of the MathBrocker Project.

[25] Paolo Casarini and Luca Padovani. The Gnome DOM Engine. *Markup Languages: Theory & Practice*, 3(2):173–190, April 2002.

[26] The Coq proof-assistant.
`http://coq.inria.fr`.

[27] Yann Coscoy. *Explication textuelle de preuves pour le Calcul des Constructions Inductives*. PhD thesis, Université de Nice-Sophia Antipolis, 2000.

[28] Yann Coscoy, Gilles Kahn, and Laurent Thery. Extracting Text from Proofs. Technical Report RR-2459, Inria (Institut National de Recherche en Informatique et en Automatique), France, 1995.

[29] Community programmable wikis. `http://purl.net/net/cpw`.

[30] Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-corn, the constructive coq repository at nijmegen. In *MKM*, pages 88–103, 2004.

[31] W. Cunningham and B. Leuf. *The Wiki way*. Addison-Wesley, New York, 2001.

[32] E. Da Lio, L. Fraboni, and T. Leo. Twiki-based facilitation in a newly formed academic community of practice. In *Proceedings of WikiSym 2005*, San Diego, California, 2005.

[33] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 589–606. Academic Press, 1980.

[34] M. de Jonge. A pretty-printer for every occasion. In Ian Ferguson, Jonathan Gray, and Louise Scott, editors, *Proceedings of the 2nd International Sympo-*

sium on Constructing Software Engineering Tools (CoSET2000), pages 68–77. University of Wollongong, Australia, June 2000.

[35] David Delahaye. *Conception de langages pour décrire les preuves et les automatisations dans les outils d'aide à la preuve: une étude dans le cadre du système* Coq. PhD thesis, Université Pierre et Marie Curie (Paris 6), Décembre 2001.

[36] Davide Delahaye and Micaela Mayero. A Maple mode for Coq. Contribution to the Coq library `http://coq.inria.fr/contribs/MapleMode.html`.

[37] Angelo Di Iorio, Valentina Presutti, and Fabio Vitali. WikiFactory: an ontology-based application to deploy domain-oriented wikis. In *Proceedings of the European Semantic Web Conference*, 2006.

[38] Document Object Model (DOM) Level 2 Specification. Version 1.0. W3C Candidate Recommendation 10 May 2000, `http://www.w3.org/TR/2000/CR-DOM-Level-2-20000510/`.

[39] Dokuwiki. `http://www.splitbrain.org/projects/dokuwiki`.

[40] Herman Geuvers and Gueorgui I. Jojgov. Open proofs and open terms: A basis for interactive logic. In J. Bradfield, editor, *Computer Science Logic: 16th International Workshop, CSL 2002*, volume 2471 of *Lecture Notes in Computer Science*, pages 537–552. Springer-Verlag, January 2002.

[41] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. Edinburgh LCF: a mechanised logic of computation. volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[42] Ferruccio Guidi. *Searching and Retrieving in Content-Based Repositories of Formal Mathematical Knowledge*. PhD thesis, University of Bologna, March 2003. Technical Report UBLCS 2003-06.

[43] Ferruccio Guidi and Claudio Sacerdoti Coen. Querying distributed digital libraries of mathematics. In Therese Hardin and Renaud Rioboo, editors, *Calculemus 2003*, pages 17–30. Aracne Editrice S.R.L., 2003. ISBN 88-7999-545-6.

[44] A. Haake, S. Lukosh, and T. Schummer. Wiki templates: Adding structure support to wikis on demand. In *Proceedings of WikiSym 2005*, pages 41–52, San Diego, California, 2005.

[45] Thomas C. Hales. Introduction to the flyspeck project. In Thierry Coquand, Henri Lombardi, and Marie-Françoise Roy, editors, *Mathematics, Algorithms, Proofs*, number 05021 in Dagstuhl Seminar Proceedings. IBFI, Schloss Dagstuhl, Germany, 2006.

[46] John Harrison. A Mizar Mode for HOL. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 203–220, Turku, Finland, 1996. Springer-Verlag.

[47] John Harrison. Proof style. In Eduardo Giménez and Christine Paulin-Mohring, editors, *Selected Papers 4th Intl. Workshop on Types for Proofs and Programs, TYPES'96, Aussois, France, 15–19 Decemeber 1996*, volume 1512, pages 154–172, Berlin, 1996. Springer.

[48] The HELM on-line library. `http://helm.cs.unibo.it/library.html`.

[49] P. Herman. Moin Moin Wiki. `http://twistedmatrix.com/users/jh.twistd/moin/moin.cgi/`.

[50] Joey Hess. Ikiwiki. `http://ikiwiki.kitenet.net/`.

[51] The HOL Light proof-assistant. `http://www.cl.cam.ac.uk/users/jrh/hol-light/`.

[52] Angelo Di Iorio and Stefano Zacchiroli. Constrained wiki: an oxymoron? In *WikiSym '06: Proceedings of the 2006 international symposium on Wikis*, pages 89–98, New York, NY, USA, 2006. ACM Press.

[53] The Isabelle proof-assistant. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`.

[54] M. L. Jugel and S. J. Schmidt. Snipsnap: the easy weblog and wiki software. `http://www.snipsnap.org/space/SnipGraph`.

[55] Fairouz Kamareddine and Rob Nederpelt. A refinement of de bruijn's formal language of mathematics. *Journal of Logic, Language and Information*, 13(3):287–340, 2004.

[56] E. E. Kim. Purplewiki. `http://purplewiki.blueoxen.net/cgi-bin/wiki.pl`.

[57] Florent Kirchner. Coq Tacticals and PVS Strategies: A Small-Step Semantics. In *Design and Application of Strategies/Tactics in Higher Order Logics*, 2003.

[58] Donald E. Knuth. *The TEXbook*, volume A of *Computers and typesetting*. Addison-Wesley, Reading, MA, USA, 1994.

[59] Aaron Krowne and Nathan Egge. Planetmath: math for the people, by the people. `http://planetmath.org/`.

[60] Christoph Lange and Michael Kohlhase. A semantic wiki for mathematical knowledge management. In *Proceedings of First Workshop on Semantic Wikis: From Wiki to Semantic*, 2006.

[61] Fabrice Le Fessant and Luc Maranget. Optimizing pattern-matching. In *Proceedings of the 2001 International Conference on Functional Programming*. ACM Press, 2001.

[62] Pietro Di Lena. Generazione automatica di stylesheet per notazione matematica. Master's thesis, University of Bologna, 2003.

[63] Zhaohui Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.

[64] Wendy E. Mackay. Patterns of sharing customizable software. In *CSCW '90: Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, pages 209–221, New York, NY, USA, 1990. ACM Press.

[65] Allan MacLean, Kathleen Carter, Lennart Lövstrand, and Thomas Moran. User-tailorable systems: pressing the issues with buttons. In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 175–182, New York, NY, USA, 1990. ACM Press.

[66] Mathematical Markup Language (MathML) Version 2.0. W3C Recommendation 21 February 2001, `http://www.w3.org/TR/MathML2`, 2003.

[67] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[68] William McCune and Larry Wos. Otter - the cade-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997.

[69] The Mizar proof-assistant. `http://mizar.uwb.edu.pl/`.

[70] MKM-IG: the Mathematical Knowledge Management Interest Group. `http://www.mkm-ig.org`.

[71] The Monet Consortium. Monet architecture overview. Technical Report D04, March 2003. Public deliverable of the Monet project.

[72] The MoWGLI Proposal, HTML version. `http://mowgli.cs.unibo.it/html_no_frames/project.html`.

[73] César Muñoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory*. PhD thesis, INRIA, November 1997.

[74] Koji Nakagawa, Akihiro Nomura, and Masakazu Suzuki. Extraction of logical structure from articles in mathematics. In Andrzej Trybulec Andrea Asperti, Grzegorz Bancerek, editor, *Proceedings of Mathematical Knowledge Management 2004*, volume 3119 of *Lecture Notes in Computer Science*, pages 276–289. Springer-Verlag, 2004.

[75] Bill Naylor and Stephen Watt. Meta-stylesheets for the conversion of mathematical documents into multiple forms. *Annals of Mathematics and Artificial Intelligence*, 38(1-3):3–25, May 2003.

[76] R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, 1994. ISBN-0444898220.

[77] Robert Nieuwenhuis and Alberto Rubio. *Paramodulation-based thorem proving*. Elsevier and MIT Press, 2001. ISBN-0-262-18223-8.

[78] The NuPRL proof-assistant. `http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html`.

[79] Steven Obua. Conservative overloading in higher-order logic. In *Rewriting Techniques and Applications*, 2006. To appear.

[80] OMDoc: An open markup format for mathematical documents (draft, version 1.2). `http://www.mathweb.org/omdoc/pubs/omdoc1.2.pdf`, 2005.

[81] Openwiki. `http://www.openwiki.com/`.

[82] Tim O'Reilly. What is Web 2.0: Design patterns and business models for the next generation of software. `http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html`.

[83] Luca Padovani. *MathML Formatting*. PhD thesis, University of Bologna, February 2003. Technical Report UBLCS 2003-03.

[84] Luca Padovani, Claudio Sacerdoti Coen, and Stefano Zacchiroli. A generative approach to the implementation of language bindings for the document object model. In Gabor Karsai and Eelco Visser, editors, *Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*, pages 469–487. Springer-Verlag, 2004.

[85] Luca Padovani and Stefano Zacchiroli. From notation to semantics: There and back again. In *Proceedings of Mathematical Knowledge Management 2006*, volume 4108 of *Lectures Notes in Artificial Intelligence*, pages 194–207. Springer-Verlag, 2006.

[86] Sean B. Palmer. Rdfwiki. `http://infomesh.net/2001/rdfwiki/`.

[87] Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Thèse d'université, Paris 7, January 1989.

[88] The PVS specification and verification system. `http://pvs.csl.sri.com/`.

[89] Aarne Ranta. Grammatical framework: A type-theoretical grammar formalism. *Journal of Functional Programming*, 14(2):145–189, 2004.

[90] Jan Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[91] Alexandre Riazanov. *Implementing an Efficient Theorem Prover*. PhD thesis, The University of Manchester, 2003.

[92] Claudio Sacerdoti Coen. Progettazione e realizzazione con tecnologia XML di basi distribuite di conoscenza matematica formalizzata. Master's thesis, University of Bologna, 2000.

[93] Claudio Sacerdoti Coen. From proof-assistans to distributed libraries of mathematics: Tips and pitfalls. In Andrea Asperti, Bruno Buchberger, and James H. Davenport, editors, *Proceedings of the Second International Conference on*

Mathematical Knowledge Management, MKM 2003, volume 2594 of *Lecture Notes in Computer Science*, pages 30–44. Springer-Verlag, 2003.

[94] Claudio Sacerdoti Coen. *Mathematical Knowledge Management and Interactive Theorem Proving*. PhD thesis, University of Bologna, 2004. Technical Report UBLCS 2004-5.

[95] Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Tinycals: step by step tacticals. In *Proceedings of User Interface for Theorem Provers 2006*, Electronic Notes in Theoretical Computer Science. Elsevier Science, 2006. To appear.

[96] Claudio Sacerdoti Coen and Stefano Zacchiroli. Brokers and Web-services for automatic deduction: a case study. In Therese Hardin and Renaud Rioboo, editors, *Calculemus 2003*, pages 43–57. Aracne Editrice S.R.L., 2003. ISBN 88-7999-545-6.

[97] Claudio Sacerdoti Coen and Stefano Zacchiroli. Efficient ambiguous parsing of mathematical formulae. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *Proceedings of Mathematical Knowledge Management 2004*, volume 3119 of *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, 2004.

[98] Ben Shneiderman. Direct manipulation for comprehensible, predictable and controllable user interfaces. In *IUI '97: Proceedings of the 2nd international conference on Intelligent user interfaces*, pages 33–39, New York, NY, USA, 1997. ACM Press.

[99] Soap version 1.2 part 0: Primer. W3C Recommendation 24 June 2003 `http://www.w3.org/TR/soap12-part0/`.

[100] The OpenMath Society. The OpenMath Standard 2.0. `http://www.openmath.org/standard/om20/omstd20html-0.xml`, June 2004.

[101] Martin Strecker. *Construction and Deduction in Type Theories*. PhD thesis, Universität Ulm, 1998.

[102] Bjarne Stroustrup. *Handbook of Object Technology*, chapter 15. CRC Press, October 1998.

[103] G. Sutcliffe. The CADE-20 Automated Theorem Proving Competition. *AI Communications*, 19(2):173–181, 2006.

[104] Don Syme. 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*, volume 971 of *Lecture Notes in Computer Science*, pages 324–339. Springer-Verlag, 1995.

[105] Koichi Takahashi and Masami Hagiya. Proving as editing HOL tactics. *Formal Aspects of Computing*, 11(3):343–357, 1999.

[106] Laurent Thèry. Colouring proofs: a lightweight approach to adding formal structure to proofs. In David Aspinall and Christoph Lüth, editors, *User Interface Design for Theorem Provers*, 2003.

[107] Laurent Théry, Yves Bertot, and Gilles Kahn. Real theorem provers deserve real user-interfaces. Technical Report Inria Research Report 1684, INRIA, May 1992.

[108] P. Thoeny. TWiki: Enterprise Collaboration Platform. `http://twiki.org`.

[109] Josef Urban. XML-izing Mizar: making semantic processing and presentation of MML easy. In Andrea Asperti, Bruno Buchberger, and James H. Davenport, editors, *Post-Proceedings of the 4th International Conference on Mathematical Knowledge Management, MKM 2005*, volume 3863 of *Lecture Notes in Computer Science*, pages 346–360. Springer-Verlag, 2006.

[110] Universal Resource Identifiers in WWW. RFC 1630, CERN, June 1994.

[111] Markus Wenzel. Type classes and overloading in higher-order logic. In *TPHOLs*, pages 307–322, 1997.

[112] Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics*, pages 167–184, 1999.

[113] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris VII, May 1994.

[114] Freek Wiedijk. The "De Bruijn factor". `http://www.cs.ru.nl/~freek/factor/`, 2000.

[115] Freek Wiedijk. Formal proof sketches. In Wan Fokkink and Jaco van de Pol, editors, *7th Dutch Proof Tools Day, Program + Proceedings*, 2003. CWI, Amsterdam.

[116] Freek Wiedijk. Mmode, a mizar mode for the proof assistant coq. Technical Report NIII-R0333, University of Nijmegen, 2003.

[117] Freek Wiedijk. *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*. Springer, 2006.

[118] WikiGnomes. `http://en.wikipedia.org/wiki/Wikignomes`.

[119] WikipediA, The Free Encyclopedia. `http://www.wikipedia.org/`.

[120] Wiki RPC interface 2, API version 2. `http://www.jspwiki.org/Wiki.jsp?page=WikiRPCInterface2`.

[121] World66. World66 home. `http://www.world66.com/`.

[122] Web services glossary. W3C Working Group Note 11 February 2004 `http://www.w3.org/TR/ws-gloss/`.

[123] Xhtml 1.0 the extensible hypertext markup language (second edition). W3C Recommendation 26 January 2000, revised 1 August 2002, `http://www.w3.org/TR/xhtml1/`.

[124] XSL Transformations (XSLT). Version 1.0. W3C Recommendation, 16 November 1999, `http://www.w3.org/TR/xslt`.

[125] Stefano Zacchiroli. Web services per il supporto alla dimostrazione interattiva. Master's thesis, University of Bologna, 2003.

[126] Zentralblatt MATH. `http://www.emis.de/ZMATH/`.

[127] Jürgen Zimmer and Michael Kohlhase. System description: The mathweb software bus for distributed mathematical reasoning. In *CADE*, pages 139–143, 2002.

[128] Richard Zippel. The MathBus. In *Proceedings of the Workshop on Internet Accesible Mathematical Computation*, 1999.