Crafting a Proof Assistant

Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli

Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7 – 40127 Bologna, ITALY
{asperti,sacerdot,tassi,zacchiro}@cs.unibo.it

Abstract. Proof assistants are complex applications whose development has never been properly systematized or documented. This work is a contribution in this direction, based on our experience with the development of Matita: a new interactive theorem prover based—as Coq—on the Calculus of Inductive Constructions (CIC). In particular, we analyze its architecture focusing on the dependencies of its components, how they implement the main functionalities, and their degree of reusability.
The work is a first attempt to provide a ground for a more direct comparison between different systems and to highlight the common functionalities, not only in view of reusability but also to encourage a more systematic comparison of different softwares and architectural solutions.

1 Introduction

In contrast with automatic theorem provers, whose internal architecture is in many cases well documented (see e.g. the detailed description of Vampire in [16]), it is extremely difficult to find good system descriptions for their interactive counterpart. Traditionally, the only component of these systems that is suitably documented is the *kernel*, namely the part that is responsible for checking the correctness of proofs. Considering that:

1. most systems (claim to) satisfy the so called "De Bruijn criterion", that is the principle that the correctness of the whole application should depend on the correctness of a sufficiently small (and thus reliable) kernel *and*
2. interactive proving *looks* in principle as a less ambitious task than fully automatic proving (this is eventually the feeling of an external observer)

one could easily wonder where the complexity of interactive provers comes from.[1] Both points above—especially the second one—are intentionally provocative. They are meant to emphasize that (1) the kernel is possibly the most crucial, but surely not the most important component of interactive provers and (2) formal checking is just one of the activities of interactive provers, and probably not the most relevant one.

Of course, interactivity should be understood as a powerful integration rather than as a poor surrogate of automation: the user is supposed to interact when the system alone fails. Interaction, of course, raises a number of additional problems that are not present (or not so crucial) in automatic proving:

[1] e.g.: Coq is about 166.000 lines of code, to be compared with 50.000 lines of Otter

– library management (comprising both incomplete proofs management and proof history);
– development of a strong linguistic support to enhance the human-machine communication of mathematical knowledge;
– development of user interfaces and interaction paradigms particularly suited for this kind of applications.

While the latter point has received a renewed attention in recent years, as testified by several workshops on this topic, little or no documentation is available on the two former topics, hindering a real progress in the field.

In order to encourage a more systematic comparison of different software and architectural solutions we must first proceed to a more precise individuation of issues, functionalities, and software components. This work is meant to be a contribution in this direction. In particular we give in Section 2 a high-level architectural description of our interactive theorem prover denominated "Matita",[2] where we made a particular effort to abstract away from the details of our implementation, focusing instead on the overall functionalities offered by the distinct components. We also try to distinguish the components dependent on the logical framework from those that could be possibly shared among different systems (Section 3), providing an estimation of their complexity and of the amount of work that could be required to implement them (Section 4).

Although our architectural description comprises components that (at present) are specific of our system (such as the large use of metadata for library indexing) we believe that the overall design fits most of the existent interactive provers and could be used as a ground for a deeper software comparison of these tools.

As an example Section 5 concludes the paper with a comparison with Coq based on our architectural decomposition. Although the comparison is pretty rough (a deeper one would possibly require the contribution of a developer of the latter system), it is already expressive enough to highlight some points of strength and weakness of the two systems.

2 Architecture

An interactive theorem prover must handle different representations of formulae and proofs, each supporting a different set of operations. In particular we identify five representations that are likely to be necessary in every modern system: completely specified terms, approximated terms (metadata), partially specified terms, content terms, and presentation terms.

Figure 1 shows the components of Matita organized according to the term representation they act on. For each component we show the functional dependencies on other components and the number of lines of source code. Dark gray components are either logic independent or can be abstracted over logical functionalities. Dashed arrows denote abstractions over logic dependent components.

[2] "matita" means "pencil" in Italian: a simple, well known and widespread editing tool among mathematicians.

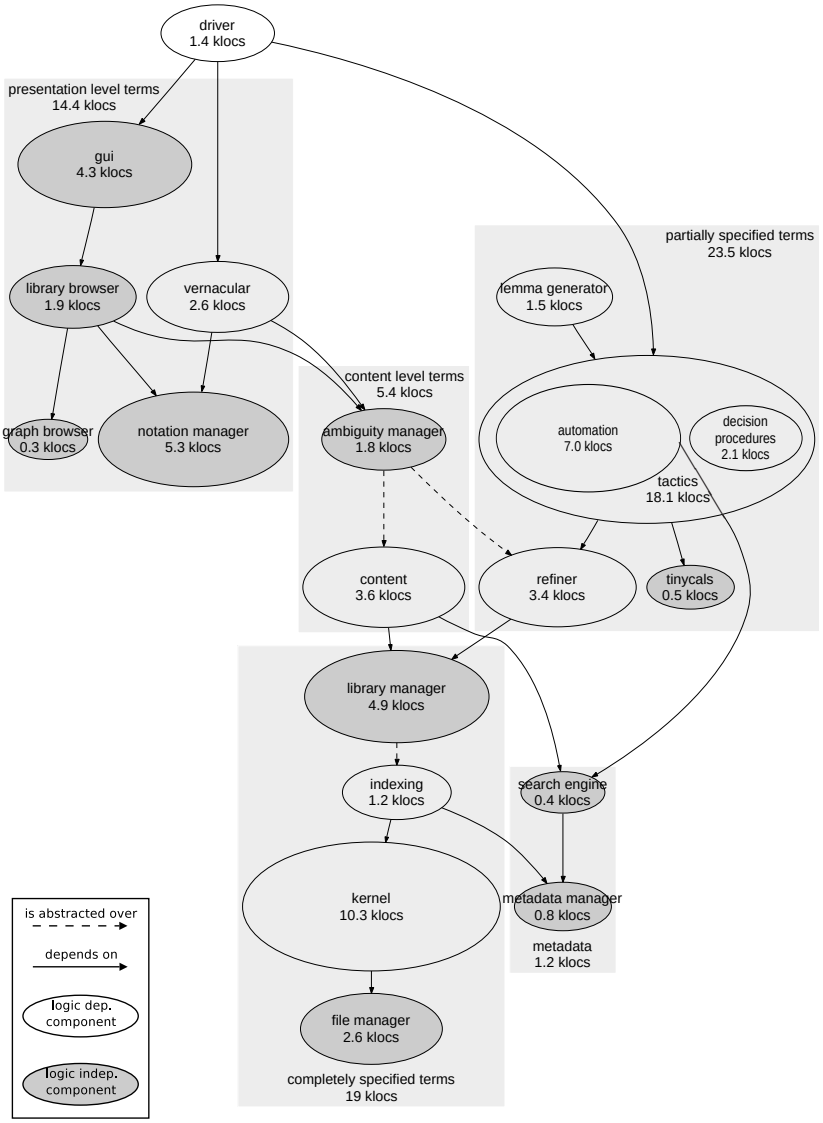

Fig. 1. Matita components with thousands of lines of code (*klocs*)

A normal arrow from a logic dependent component to a dark gray one is meant to be a dependency over the component, once it has been instantiated to the logic of the system. Section 3 will analyze the architecture of Matita from the point of view of reusability and logic independence.

We describe now each term representation together with the related Matita components.

Completely specified terms. Formalizing mathematics is a complex and onerous task and it is extremely important to develop large libraries of "trusted" information to rely on. At this level, the information must be completely specified in a given logical framework in order to allow formal checking. In systems based on the Curry-Howard isomorphism as Matita, proof objects can also be represented as terms of some calculus and the proof-checker is also a type-checker. Thus we will not distinguish proofs and terms any longer when represented as completely specified terms.

According to "De Bruijn criterion", the proof-checker should be as simple as possible. This has a double motivation: on one side simplicity is meant to improve reliability, on the other if the kernel is small we may imagine to write an external independent component to re-check the information. The latter point is possible only if the information is saved in a system independent format, easily accessible from external tools (Matita exports and saves its proof objects in XML).

The *file manager* is responsible for the storage and retrieval of mathematical concepts in the library; in Matita this component is non entirely trivial since the library is conceived as being potentially distributed and the *file manager* is responsible, among other things, to map Uniform Resource Identifiers into Uniform Resource Locators (i.e. names to physical locations).

Concepts to be put in the library may be indexed for retrieval. The interest of using metadata for indexing is that a logic independent metadata-set can be conceived to accommodate most logical frameworks. Thus, a logic dependent *indexing* component extracts metadata from mathematical objects, which may be handled by logic independent searching tools described in the next section.

Finally, the *library manager* component is responsible for maintaining the coherence between related concepts (often automatically generated from other ones) and between the different representations of them in the library (as completely specified terms and as metadata that approximate them).

The actual generation of the derived principles is a logic dependent activity that is not directly implemented by the library manager, that is kept logical independent: the component provides hooks to register and invoke logic dependent lemma generators, whose implementation is provided in a component that we describe later and that acts on partially specified terms.

Metadata. An extensive library requires an effective and flexible search engine to retrieve concepts. Examples of flexibility are provided by queries up to instantiation or generalization of given formulae, combination of them with extra-logical constraints such as mathematical classification, and retrieval up to minor differences in the matched formula such as permutation of the hypotheses or logical equivalences. Effectiveness is required to exploit the search engine as a first step in automatic tactics. For instance, a paramodulation based procedure must first of all retrieve all the equalities in the distributed library that are likely to be exploited in the proof search. Moreover, since search is mostly logic independent, we would like to implement it on a generic representation of formulae that supports all the previous operations.

In Matita we use relational *metadata* to represent both extra-logical data and a syntactic approximation of a formula (e.g. the constant occurring in head position in the conclusion, the set of constants occurring in the rest of the conclusion and the same information for the hypotheses). The logic dependent *indexing* component, already discussed, generates the syntactic approximation from completely specified terms. The *metadata manager* component stores the metadata in a relational database for scalability and handles for the library manager the insertion, removal and indexing of the metadata. The *search engine* component [1] implements the approximated queries on the metadata that can be refined later on if required by logic dependent components.

Partially specified terms. This terms representation allows the omission of subterms, replacing them with untyped linear placeholders or with typed metavariables (in the style of [8,13]). The latter are Curry-Howard isomorphic to omitted subproofs (conjectures still to be proved).

Completely specified terms are often highly redundant to keep the type-checker simple. This redundant information may be omitted during user-machine communication since it is likely to be automatically inferred by the system replacing conversion with unification [19] in the typing rules (that are relaxed to type inference rules). The *refiner* component of Matita implements unification and the type inference procedure, also inserting implicit coercions [3] to fix local type-checking errors. Coercions are particularly useful in logical systems that lack subtyping [10]. The already discussed library manager is also responsible for the management of coercions, that are constants flagged in a special way.

Subproofs are never redundant and if omitted require tactics to instantiate them with partial proofs that have simpler omitted subterms. Tactics are applied to omitted subterms until the proof object becomes completely specified and can be passed to the library manager. Higher order tactics, usually called tacticals and useful to create more complex tactics, are also implemented in the tactics component. The current implementation in Matita is based on *tinycals* [17], which supports a step-by-step execution of tacticals (normally seen as "black boxes") particularly useful for proof editing, debugging, and maintainability. Tinycals are implemented in Matita in a small but not trivial component that is completely abstracted on the representation of partial proofs.

The *lemma generator* component is responsible for the automatic generation of lemmas, triggered by the insertion of new concepts in the library. The lemmas are generated automatically computing their statements and then proving them by means of tactics or by direct construction of the proof objects.

Content level terms. The language used to communicate proofs and especially formulae with the user must also reflect the comfortable and suggestive degree of notational abuse and overloading so typical of the mathematical language. Formalized mathematics cannot hide these ambiguities requiring terms where each symbol has a very precise and definite meaning.

Content level term provides the (abstract) syntactic structure of the human-oriented (compact, overloaded) encoding. In the *content* component we provide

translations from partially specified terms to content level terms and the other way around. The former translation, that loses information, must discriminate between terms used to represent proofs and terms used to represent formulae. Using techniques inspired by [6,7], the formers are translated to a content level representation of proof steps that can in turn easily be rendered in natural language. The representation adopted has greatly influenced the OMDoc [14] proof format that is now isomorphic to it. Terms that represent formulae are translated to MathML Content formulae [12].

The reverse translation for formulae consists in the removal of ambiguity by fixing an interpretation for each ambiguous notation and overloaded symbol used at the content level. The translation is obviously not unique and, if performed locally on each source of ambiguity, leads to a large set of partially specified terms, most of which ill-typed. To solve the problem the *ambiguity manager* component implements an algorithm [18] that drives the translation by alternating translation and refinement steps to prune out ill-typed terms as soon as possible, keeping only the refinable ones. The component is logic independent being completely abstracted over the logical system, the refinement function, and the local translation from content to partially specified terms. The local translation is implemented for constant occurrences by means of call to the search engine.

The translation from proofs at the content level to partially specified terms is being implemented by means of special tactics following previous work [9,20] on the implementation of declarative proof styles for procedural proof assistants.

Presentation level terms. Presentation level captures the formatting structure (layout, styles, etc.) of proofs expressions and other mathematical entities.

An important difference between the content level language and the presentation level language is that only the former is extensible. Indeed, the presentation level language is a finite language comprising standard layout schemata (fractions, sub/superscripts, matrices, ...) and the usual mathematical symbols.

The fact that the presentation language is finite allows its standardization. In particular, for pretty printing of formulae we have adopted MathML Presentation [12], while editing is done using a TEX-like syntax. To visually represent proofs it is enough to embed formulae in plain text enriched with formatting boxes. Since the language of formatting boxes is very simple, many similar specifications exist and we have adopted our own, called BoxML (but we are eager to cooperate for its standardization with other interested teams).

The *notation manager* component provides the translations from content level terms to presentation level terms and the other way around. It also provides a language [15] to associate notation to content level terms, allowing the user to extend the notation used in Matita. The notation manager is logic independent since the content level already is.

The remaining components, mostly logic independent, implement in a modular way the user interface of Matita, that is heavily based on the modern GTK+ toolkit and on standard widgets such as GtkSourceView that implements a programming oriented editor with syntax highlighting or GtkMathView that

implements rendering of MathML Presentation formulae with the possibility of contextual and controlled interaction with the formula.

The *graph browser* is a GTK+ widget, based on Graphviz, to render dependency graphs with the possibility of contextual interaction with them. It is mainly used in Matita to explore the dependencies between concepts, but other kind of graphs (e.g. the DAG formed by the declared coercions) are also shown.

The *library browser* is a GTK+ window that mimics a web browser, providing a centralized interface for all the searching and rendering functionalities of Matita. It is used to hierarchically browse the library, to render proofs and definitions in natural language, to submit queries to the search engine, and also to inspect dependency graphs embedding the graph browser.

The *GUI* is the graphical user interface of Matita, inspired by the pioneering work on CtCoq [4] and by Proof General [2]. It differs from Proof General because the sequents are rendered in high quality MathML notation, and because it allows to open multiple library browser windows to interact with the library during proof development.

The hypertextual browsing of the library and proof-by-pointing [5] are both supported by semantic selection. Semantic selection is a technique that consists in enriching the presentation level terms with pointers to the content level terms and to the partially specified terms they correspond to. Highlight of formulae in the widget is constrained to selection of meaningful expressions, i.e. expressions that correspond to a lower level term, that is a content term or a partially or fully specified term. Once the rendering of a lower level term is selected it is possible for the application to retrieve the pointer to the lower level term. An example of applications of semantic selection is *semantic copy & paste*: the user can select an expression and paste it elsewhere preserving its semantics (i.e. the partially specified term), possibly performing some semantic transformation over it (e.g. renaming variables that would be captured or lambda-lifting free variables).

Commands to the system can be given either visually (by means of buttons and menus) or textually (the preferred way to input tactics since formulae occurs as tactic arguments). The textual parser for the commands is implemented in the *vernacular* component, that is obviously system (and partially logic) dependent.

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 trigging its execution (for instance calling the tactics component if the command is a tactic).

3 Reusability

Proof General provides a generic graphical user interface for interactive theorem provers. By implementing the Proof General Interaction Protocol a new system can be immediately connected to an interface supporting the same protocol, decoupling the system core from the user interface and allowing code reuse.

What makes Proof General possible is the fact that the graphical user interface is mostly logic and system independent. However, the code required to implement a script based graphical user interface as Proof General is only a comparatively small fraction of the system. One of the defy for the future of interactive theorem proving is to provide more and more libraries for logic independent tasks or tasks parametric on the logic. Indexing and searching functionalities as well as notational and rendering support are good candidates.

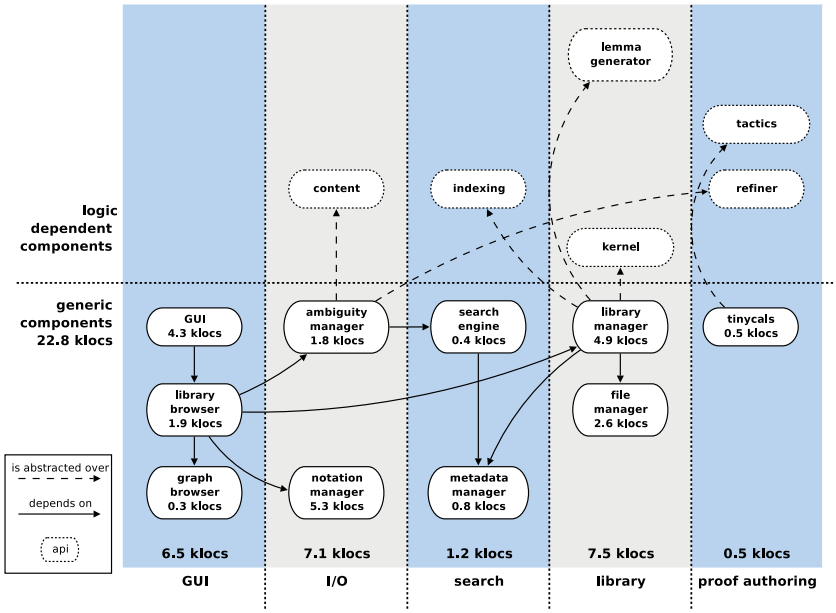

Fig. 2. API of the reusable components.

During the development of Matita we have tried to identify those components that can be reused with little effort in other systems, even if based on a different logic. They have been shown in Figure 1 as dark gray components. Figure 2 focuses on their API, dividing the functionalities of a proof assistant in five classes: visual interaction and browsing of a mathematical library (*GUI* column), input/output of formulae and proofs (*I/O* column), indexing and searching of concepts in a library (*search* column), management of a library of certified concepts (*library* column), and interactive development of proofs by means of tactics and decision procedures (*proof authoring* column). For each functionality, the lower part of the column shows the generic reusable components of Matita and their mutual dependencies. The upper part shows the logic dependent components the programmer needs to implement to instantiate the generic components to a given logic.

If some functionalities are not required (e.g. searching), the component that need to be implemented are pruned (e.g. indexing). Even if the logical system is weaker than the one of Matita, all the components to be implemented should make sense. For instance, if the logical system does not allow metavariables or, more generally, partially specified terms, the refiner can be implemented simply as a call to the type-checker, and unification (also implemented in the refiner component) as a call to the reduction machine implemented in the kernel component. If the calculus does not have reduction, the reduction function can be implemented as the identity function and so on.

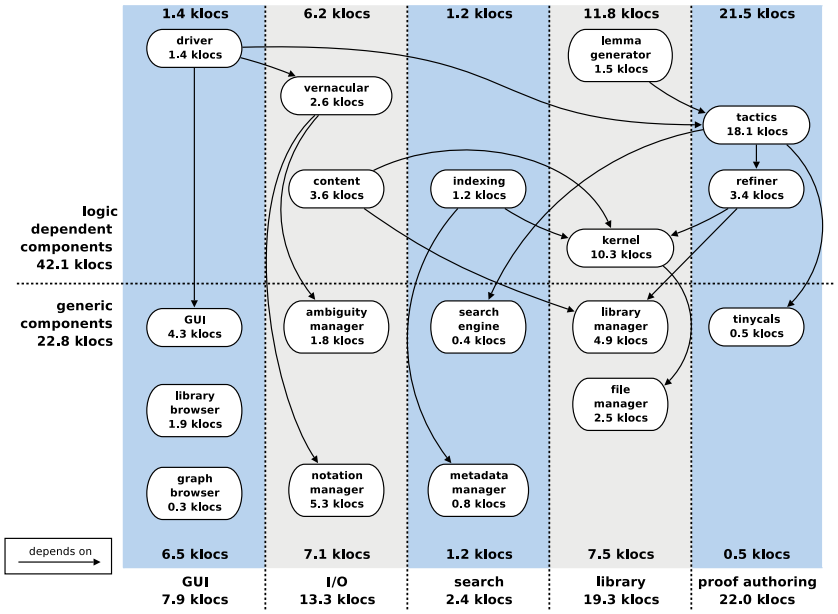

Fig. 3. Implementing the API for CIC in Matita.

Figure 3 is focused on the components implemented in Matita to fulfill the API required by the generic components. The arrows show the dependencies among the logic dependent components and from the logic dependent components to instances of the generic components. The latter dependencies show that the two kinds of components have mutual dependencies that are difficult to untangle, partially justifying the current lack of generic libraries. 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.

Although the amount of reusable code is quite limited (still being more than 22.000 lines of code), the advantages of decoupling the functionality should not

be neglected. In particular 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. Moreover, we expect systems built on top of a collection of reusable components to obtain more quickly some of the functionalities required for early testing, speeding up the development cycle. In the next section we asses this point presenting a timeline for the development of the system.

4 System development

Figure 4 is an hypothetical Gantt-like diagram for the development of an interactive theorem prover similar to Matita. The order in which to develop the components in the figure does not reflect the true history. Indeed, during the development of Matita we delayed a few activities, with major negative impacts on the whole schedule. The duration of the activities in the diagram is an estimation of the time that would be required now by an independent team to reimplement Matita assuming only the knowledge derivable from the literature.

In any case, in the estimated duration of the activities we are considering the time wasted for rapid prototyping: it is not reasonable in a research community to expect the product to be developed for years without any intermediate prototype to play with. For example, we suggest to implement first reduction and typing in the kernel on completely specified terms before extending it to accommodate metavariables (later on required for partially specified terms). This way the kernel of the type-checker can immediately be tested on a library of concepts exported from another system, and different reduction and type-checking algorithms can be compared leading to possibly interesting research results.

Activities related to logic independent components are marked as dashed in the Gantt-like diagram. If those components are reused in the implementation of the system, most functionalities but interactive proof authoring are made available very early in the development. The bad news are that the overall time required to develop the system will not change, being determined by the complexity of the logic dependent components and their dependencies that limit parallelism. Switching to a simpler logic can probably reduce in a significant way the time required to implement the kernel and the refinement component; however, it is likely to have a minor impact on the time required for tactics and decision procedures. The overall conclusion is that the development of an interactive theorem prover is still a complex job that is unlikely to be simplified in a major way in the near future.

The activities of Figure 4 refine the components already presented to improve parallel development and allow rapid prototyping. We describe now the main refinements following the timeline when possible.

We suggest to start developing the kernel omitting support for terms containing metavariables and to add it after the reduction and typing rules for completely specified terms have been debugged. The support for metavariables in the kernel should be kept minimal, only implementing typing rules and unfold-

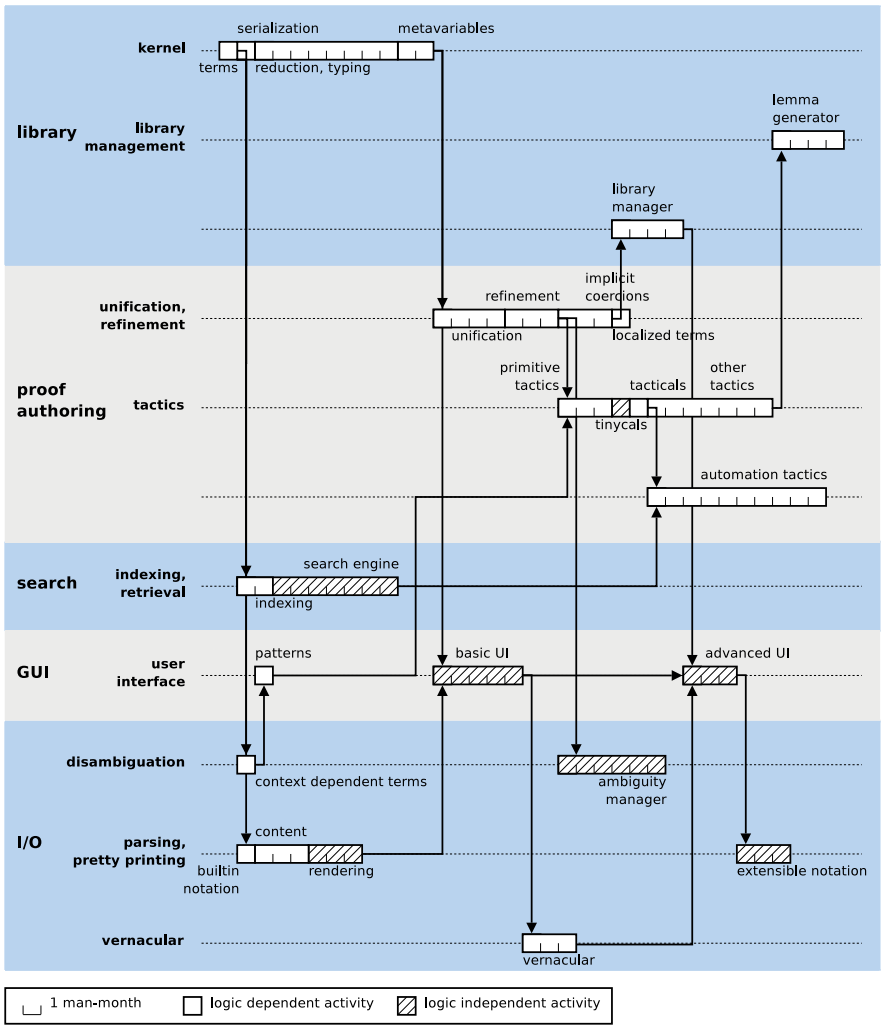

Fig. 4. Gantt-like development schedule of an interactive theorem prover.

ing of instantiated metavariables. The core functionalities on partially specified terms, unification and refinement, are implemented in the refiner component outside the kernel. Completely omitting support for metavariables from the kernel is more compliant to the De Bruijn criterion. However, the kernel code for minimal metavariable support is really simple and small, and its omission forces an almost complete reimplementation of the kernel functionalities in the refiner that is better avoided. In Section 5 we will see this phenomenon when comparing the code size of Matita and Coq.

Context dependent terms are a necessity for passing to tactics arguments that need to be interpreted (and disambiguated) in a context that is still unknown. In Matita context dependent terms are defined as functions from contexts to terms, but other systems adopt different representations.

Patterns are data structures to represent sequents with selected subterms. They are used as tactic arguments to localize the effect of tactics. Patterns pose a major problem to the design of textual user interfaces, that usually avoid them, but are extremely natural in graphical user interface where they correspond to semantic selection (using the mouse) of subterms of the sequent.

A fixed *built-in notation* should be implemented immediately for debugging, followed by the *content* component to map completely (or even partially) specified terms to content and the other way around. Partially specified terms generated by the reverse mapping cannot be processed any further until the refiner component is implemented. Similarly, the reverse mapping of ambiguous terms is delayed until the ambiguity manager is available.

The *rendering* and *extensible notation* activities implement the notation manager component. Initially the machinery to apply extensible notation during rendering is implemented in the rendering activity. A user-friendly language to extend at run time the notation is the subject of the second activity that is better delayed until the interaction mode with the system become clear.

Handling of *implicit coercions* and *localized terms* in the refiner component can be delayed until unification and a light version of refinement are implemented. This way the implementation of tactics can start in advance. Localized terms are data structures to represent partially specified terms obtained by formulae given in input by the user. A refinement error on a localized term should be reported to the user by highlighting (possibly visually) the ill-typed subformula. Localized terms pose a serious problem since several operations such as reduction or insertion of an implicit coercion change or loose the localization information. Thus the refiner must be changed carefully to cope with the two different representations of terms.

The *basic user interface* is an interface to the library that offers browsing, searching and proof-checking, but no tactic based proof authoring. It can, however, already implement proof authoring by direct term manipulation that, once refinement is implemented, can become as advanced as Alf is [11]. The *advanced user interface* offers all the final features of the system, it can be script based and it can present the desired interaction style (procedural versus declarative).

Finally, primitive tactics, that implement the inference rules of the logic, and tacticals are requirements for the development of more advanced interactive tactics and automation tactics, that can proceed in parallel.

5 A comparison of Matita with Coq

We briefly analyze now the code of Coq along the lines of our architectural decomposition, also performing a comparison with Matita. The latter can also

include the size of the code since Coq and Matita share the foundational dialect, the implementation language, and have similar interface and set of tactics.

Not surprisingly, since Matita has been designed and implemented from scratch with most of the functionalities already in mind, the complexity of the code of Matita is greatly reduced with respect to Coq. In particular, the API of the components of Matita comprises about 1.000 functions (that we would like to reduce further), to be compared with the 4.300 functions of Coq. The overall size of the code of Matita is about 65.000 lines of code, against the 166.000 lines of Coq. A refinement of this estimation is given in Figure 5 that compares the size of the corresponding components in Figure 1. Minor differences in lines of code with the previous figures are due to the *utilities* column where we have put miscellaneous code on generic data structures that was previously counted in other components.

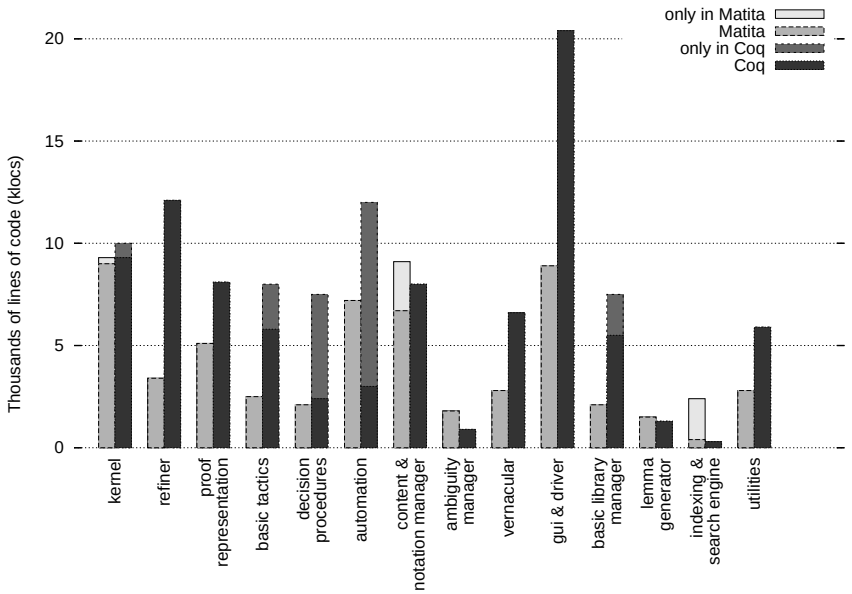

Fig. 5. Comparison of Coq and Matita code size per functionality.

The sizes of the two kernels are almost the same, but while Coq implements also a module system, ours implements a version of CIC extended with metavariables, explicit substitutions, and two contexts for metavariable declaration and instantiation. The latter extensions do not make the kernel sensibly more fragile because of the very few additional lines required.

The refiner of Coq is sensibly bigger due to the need to re-implement type-checking outside the kernel, on the calculus extended with metavariables; more-

over, for historical reasons, Coq metavariables come in two flavors, further increasing complexity. Coq has a complex data structure to represent proof trees, while Matita has just proof terms. While Coq has both variables bound by names and variables bound by position (De Bruijn indexes), Matita has only the latter. In our experience this choice makes the code smaller and hardly more complex.

We have refined the tactics component into *proof representation* and *basic tactics* to show that the size difference is due to Matita not implementing yet all the basic tactics of Coq with their variants. Matita also lacks several domain specific automation tactics of Coq. Both systems have a decision procedure for the first order fragment (actually Coq has two); Matita has a *paramodulation* based resolution tactic that subsumes congruence closure and simple repeated rewriting, implemented in Coq. Proof objects in Matita are the only representation for proofs, while Coq has also proof trees that do not add functionalities, but requires more code.

It has not been possible to clearly distinguish the content and notation manager components of Coq, even if it is clear from the code that intermediate representations for formulae are used. If the distinction cannot be done by an implementor of Coq, this could be a symptom that our division in components is not fully general, or of a design problem in Coq. Only Matita can render proof objects into pseudo-natural language.

The ambiguity manager of Coq is much simpler since it implements only a system of notational scopes, surely more efficient than disambiguation, but also less general and more cumbersome for the user.

CoqIDE, the graphical user interface of Coq, is similar to the one of Matita, but offers less functionalities since the graph browser and library browser components are missing in Coq. Moreover, we have not been able to fully separate the driver, GUI, and library manager components (the *basic library manager* column describes only the functionalities that are clearly separated also in Coq). Actually, the GUI and driver components of Matita are also not clearly separated in the code, we consider this a flaw we intend to fix. The huge difference in code size for the GUI and driver components cannot easily be explained, also considering that the GUI of Matita integrates more functionalities. A partial explanation consists in the fact that the GUI of Matita relies on external libraries.

The indexing and searching functionalities of Matita are more advanced and operate on the whole library, while those of Coq only work on the few developments loaded by the user. In particular, Coq implements no indexing and metadata manager component, and the technique does not scale to big libraries.

The last component not implemented by Coq is the file manager, lacking the concept of a distributed library. However, Coq also implements three additional components that are not in Matita: the first one extracts code from (constructive) proof terms; the second one implements an high level language to extend at run time the set of tactics; the third one is a compiler to efficiently perform term reduction. Of these components, only the code extractor and the high level language seem general enough to be added to our architecture for comparison

with other systems, while the compiler seems too peculiar to be found in other interactive theorem provers based on different logics.

6 Conclusions

In this paper we described a software architecture for an interactive theorem prover. In spite of being an abstraction over the actual architecture of the Matita proof assistant, we expect it to describe the components of any similar system, and the dependencies among them. Being high level, we do not expect the description to be relevant *per se*, but rather to be useful as a blueprint for the comparison of system implementations.

As a proof of concept we attempted a comparison of Matita and Coq, the only other system whose source code we are partially familiar with. It is reassuring that we were able to recognize in the Coq code-base most of the components of our architecture. This paves the way to future comparisons of the internals of each component, better performed by collaborating developers from both teams.

Being Coq and Matita similar systems from the points of view of the logic adopted, the interaction mode, and the functionalities provided, we cannot judge yet the generality of our architecture and its applicability to the study of other systems. We hope however to have posed the ground and raised interest in collaborations with other teams for a better mutual understanding of the alternative software solutions adopted.

We made an effort in spotting those components that are essentially logic independent. With some additional effort, the logic independent components of Matita can be turned into reusable libraries for adoption in other systems. Since we expect other systems to have similar logic independent components, we hope in the future to see the collaborative development of reusable libraries implementing common features.

In the paper we also analyzed the expected benefits of adopting reusable components to shorten the development cycle. The actual gain obviously depends on the complexity of the logic dependent components to be instantiated, that in turn partially depends on the complexity of the logic. For CIC we can give precise estimations based on our experience in the implementation of Matita: while the gain on the programming effort is partially satisfactory, the overall development time for the system by a large team is not shortened at all. On the other hand, working prototypes including even advanced functionalities can be obtained in early development stages, with a positive impact at least on dissemination and system evaluation.

References

1. 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.

2. 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.
3. Gilles Barthe. Implicit coercions in type systems. In *Types for Proofs and Programs: International Workshop, TYPES 1995*, pages 1–15, 1995.
4. Yves Bertot. The CtCoq system: Design and architecture. *Formal Aspects of Computing*, 11:225–243, 1999.
5. Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In *Symposium on Theoretical Aspects Computer Software (STACS)*, volume 789 of *Lecture Notes in Computer Science*, 1994.
6. Yann Coscoy. *Explication textuelle de preuves pour le Calcul des Constructions Inductives*. PhD thesis, Université de Nice-Sophia Antipolis, 2000.
7. 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.
8. 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, CLS 2002*, volume 2471 of *Lecture Notes in Computer Science*, pages 537–552. Springer-Verlag, January 2002.
9. 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.
10. Zhaohui Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.
11. Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806 of *LNCS*, pages 213–237, Nijmegen, 1994. Springer-Verlag.
12. Mathematical Markup Language (MathML) Version 2.0. W3C Recommendation 21 February 2001, `http://www.w3.org/TR/MathML2`, 2003.
13. César Muñoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory*. PhD thesis, INRIA, November 1997.
14. OMDoc: An open markup format for mathematical documents (draft, version 1.2). `http://www.mathweb.org/omdoc/pubs/omdoc1.2.pdf`, 2005.
15. Luca Padovani and Stefano Zacchiroli. From notation to semantics: There and back again. In *Proceedings of Mathematical Knowledge Management 2006*, volume 3119 of *Lectures Notes in Artificial Intelligence*, pages 194–207. Springer-Verlag, 2006.
16. Alexandre Riazanov. *Implementing an Efficient Theorem Prover*. PhD thesis, The University of Manchester, 2003.
17. 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.
18. Claudio Sacerdoti Coen and Stefano Zacchiroli. Efficient ambiguous parsing of mathematical formulae. In Andrzej Trybulec Andrea Asperti, Grzegorz Bancerek, editor, *Proceedings of Mathematical Knowledge Management 2004*, volume 3119 of *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, 2004.
19. Martin Strecker. *Construction and Deduction in Type Theories*. PhD thesis, Universität Ulm, 1998.
20. Freek Wiedijk. Mmode, a mizar mode for the proof assistant coq. Technical Report NIII-R0333, University of Nijmegen, 2003.