Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 226 (2009) 67–87
www.elsevier.com/locate/entcs

A User Interface for a Mathematical System that Allows Ambiguous Formulae

Claudio Sacerdoti Coen[1,2]

Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7 — 40127 Bologna, ITALY

Abstract

Mathematical systems that understand the usual ambiguous mathematical notation need well thought out user interfaces 1) to provide feedback on the way formulae are automatically interpreted, when a single best interpretation exists; 2) to dialogue with the user when human intervention is required because multiple best interpretations exist; 3) to present sets of errors to the user when no correct interpretation exists. In this paper we discuss how we handle ambiguity in the user interfaces of the Matita interactive theorem prover and the Whelp search engine.

Keywords: Overloading, ambiguity, user interface, theorem prover, Matita

1 Introduction

Eventually, all mathematical tools require some input from the user and the input is likely to be a mathematical formula that needs to be parsed and interpreted semantically. In this scenario, the user has some meaning in mind at least for every symbol, constant and variable occurring in the formula, independently from the meaningfulness of the whole formula. Moreover, when the formula is meaningful, the user can still commit an error writing it down. The aim of the mathematical tool is to detect the intended meaning for every constituent of the formula. Once the meaning is determined, the formula must

[1] sacerdot@cs.unibo.it
[2] Partially supported by the strategic project DAMA (Dimostrazione Assistita per la Matematica e l'Apprendimento) of the University of Bologna.

doi:10.1016/j.entcs.2008.12.098

be accepted if meaningful, or otherwise an informative error message should be presented to the user.

Automatically detecting the meaning of constituents of a mathematical formula is a major challenge: mathematical notation is heavily overloaded and its grammar is context dependent and inherently ambiguous. Moreover, ambiguity can be resolved only knowing in advance the intended meaning of all symbols in the context, which partially depends on written and unwritten premises and conventions, but that ultimately boils down to the initial problem itself.

Ambiguity, of which ad-hoc polymorphism is an example, must not be confused and it is not in contrast with parametric polymorphism: the first refers to a formula that may have multiple unrelated interpretations; the second to a formula that is given just one interpretation that may be instantiated to any model of some abstract mathematical structure. Both ambiguity and polymorphism are present in mathematics, but much more attention has been devoted so far to the latter than to the former.

Concretely, implementors of mathematical tools can either avoid ambiguity, by choosing an artificial but unambiguous grammar, or they can develop heuristics. In Sect. 2 we briefly recall the advantages and disadvantages of the former solution. The focus of the paper is, however, on the second one. Moreover, we only focus on ambiguity in formulae, even if more complex forms of ambiguities are possible, e.g. in commands applied to arguments that are ambiguous. The main challenge is that there is the concrete possibility that the heuristics fail to assign the intended meaning to the formula components. This leads to three problematic scenarios that all raise challenging user interface problems.

(i) The system picks a meaningful interpretation for the formula that is not the one intended by the user. The system must provide non-intrusive feedback on the choice.

(ii) The system detects several meaningful interpretations and the heuristics do not automatically pick one. They can, however, rate them. The system must interact with the user to pick the intended interpretation, if it was detected.

(iii) All interpretations detected by the system make the formula meaningless. Nevertheless, the interpretations could be rated. The system must propose the interpretations to the user and show the related error messages.

Additionally, the heuristics themselves can be driven by the user. This can be done explicitly by setting some parameters or implicitly if the system observes and records the interactions with the user. In particular, especially in the

second scenario, the system needs to remember at least the output of the interaction process in order to avoid further interaction if the same command is re-executed. Driving the heuristics and recording user interaction poses additional user interface problems.

The focus of the paper is on the part of the user interface of the Matita theorem prover [3] that deals with disambiguation. The interface is an improvement over the one of the Whelp search engine [2], that predates it and will be updated in the near future.

In the rest of the paper we discuss how the user interface of Matita deals with disambiguation. As far as we know, this is the first paper presenting an user interface of this kind. Starting from the one of Whelp, the current user interface has evolved in the last two years to satisfy users that faced show-stopping situations. Examples of them are when the time spent acting against the system becomes significant or when the system picks wrong interpretations on a regular basis. Currently, no user has faced show-stopping situations for a long time.

A serious evaluation involving a group of users faced with alternative interfaces has not been attempted yet due to a major difficulty: the experiment cannot be repeated in time under the same conditions without accumulation of knowledge by the user. This knowledge biases the experiment. Thus we should use replication in space by involving a large group of users. Nevertheless our current group of users is highly non-omogeneous. For instance, implementors should not take part of the group since, knowing exactly the disambiguation algorithm that is implemented, they infer from error messages more information of what is exposed. Similarly, students that have not received a reasonable training period do not provide an interesting population since they may have difficulties with the logic itself that could be erroneously identified with difficulties related to ambiguity. Finally, replication of the experiment in time, but on different topics, is also unreliable: on the one hand no previous knowledge is accumulated; on the other hand, the set of mistakes made by a user on the formalization of one topic is likely to be very different from that related to another topic. One of the referee suggested a complementary evaluation approach that is not affected by the previous considerations: an analysis of the interface according to an HCI usability framework. This has not been attempeted yet, but we plan to do that in the future.

The main contribution of the paper is not to establish the state of the art nor to drive general principles, but it is to present a design experience and solicit more interest on the community around these themes. We do not attempt a broader analysis of user interfaces for disambiguation mainly because we do not feel that the community has come to the point of drawing

conclusions in the large. On the other hand, we have tried to put transferrable value in the paper by discussing all the previous experiments we tried and why they failed. This is no proof that the current interface behaves well or that it cannot be improved but, on the other hand, the failed experiments constitute a good list of common pitfalls to be avoided.

In Sect. 2 we present further motivations for dealing with mathematical ambiguity. Then in Sect. 3 we present several sources of ambiguity in mathematics before giving in Sect. 4 background material on how ambiguity is handled in Matita and Whelp. The next three sections describe the behaviour of the user interface of Matita in the three scenarios already discussed. Conclusions follow in Sect. 8.

2 Motivations for dealing with standard mathematical ambiguity

The main alternative to the work described in this paper is to simply replace the standard mathematical notation with a completely unambiguous one. This replacement requires a new training period for the human for each new part of the library that has been developed by somebody else. However, the time required is usually negligible with respect to mastering the mathematical content of the library itself. It also requires some form of coordination between developers to avoid notational clashes which prevents reuse. The highest price, however, is the impossibility of communicating the results to non-experts of the system due to esoteric notations that end up replacing the standard ones. Of course, in most cases the system could map the non-standard notations to standard ones, but the price to pay would be that the translated text would be no longer machine readable since information would be lost and the language would become ambiguous. We should also remember that the standard mathematical notation has evolved during the last few centuries to maximise intuition by reusing the same or similar notations for formal concepts that share similar properties. Any alternative ad hoc notation is most probably less effective in this sense.

Finally, it is unrefutable that directly dealing with ambiguity maximises conceptual familiarity with the standard mathematical practice, which is one important HCI design consideration.

However, replacing the standard notation with a non-ambiguous one has a clear advantage also from the user point of view: the system and the human assign the same meaning to every component of the formula. Thus, if the formula is accepted, it is for sure the one intended. Otherwise, if an error is shown, the error is surely meaningful.

To conclude, the decision between the standard language or an ad hoc one is often forced by the scenario where the tool is going to be used. In several situations, only the first solution is viable. One example is applications of theorem provers to education. According to our experience, it is possible to convince mathematical teachers to make students experiment with a proof assistant only if it understands the exact notation used in class. Another example, that motivated us from the beginning, is that of mathematical tools to explore or reuse parts of an unknown mathematical library. Since the notation is usually part of the library, avoiding the traditional mathematical notation makes impossible to issue queries on the library involving formulae. Indeed, we started working on mathematical disambiguation when we developed the Whelp search engine that allows the user, among other things, to look for theorems whose conclusion is an instance or a generalisation of a given formula.

Once again, the main difference between the user interface of Whelp and the one of Matita has been driven by the user interaction scenario of the two tools. In Whelp there is no clear correlation between the previous queries and the current one. On the contrary, in Matita the user works on the same topic for a while and so the heuristics can observe the recent interactions to learn the current user preferences and to drive disambiguation.

3 Ambiguity in mathematics

To fully understand the problem of ambiguity in mathematics from an implementor point of view, it is useful, and not too misleading, to compare a mathematical tool accepting formulae with a compiler. Surprisingly, it becomes evident that ambiguity occurs in every compilation phase.

Lexical analysis is ambiguous since the way tokenisation is performed depends on the use of the symbols. For instance, x_2 can be recognised either as two tokens (when the user is taking the second element of a sequence) or as a single token (when the user is referring to a variable named x_2 to distinguish it from a variable x_1).

The grammar is inherently ambiguous, and there is no way to force the parser by using precedence and associativity rules since the same symbol should be given different precedences according to its semantics in the formula. For instance, equality on propositions (denoted by $=$, a notational abuse for co-implication) has precedence higher than conjunction (denoted by $\wedge$), which is higher than equality on set elements (also denoted by $=$), which is higher than meet for lattice elements (also denoted by $\wedge$). Thus $A = B \wedge P$ can be parsed either as $(A = B) \wedge P$ (a conjunction of propositions) or as $A = (B \wedge P)$ (equality of lattice elements).

Semantic analysis is the phase most affected by ambiguity. First of all mathematical structures usually belong to deep inheritance hierarchies, such as the algebraic (magmas, semigroups, groups, rings, fields, ...) and numerical ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{C}$, ...) ones. Depending on the logic and semantical framework used to represent formulae, a formula that requires subsumption to be understood must be represented either as it is, or by insertion of explicit coercions [7]. Since multiple derivations can usually give meaning to a formula, semantic analysis becomes a one to many relation, at least when it inserts coercions. Secondly, even ignoring inheritance and subsumption, mathematical symbols are often highly overloaded in different mathematical contexts. As a trivial example, $-^{-1}$ is overloaded on semigroup elements (and thus on numbers by instantiation) and on relations (and thus on functions by inheritance). Moreover, x^{-1} can be understood either as x to the power of -1, or as the inverse of x (which is a semantically equivalent, but intensionally different operation).

Another problem in giving semantics to formulae is that the α-conversion equivalence relation, which semantically identifies formulae up to bound variable names, does not hold since it is common practice to reserve names for variables ranging over some set, omitting to specify for quantifiers the sort of the bound symbols. Thus, changing the name of one variable already suggests a different interpretation in different contexts. For instance, $f, g, \ldots$ usually range over functions, $x, y, z, \ldots$ over domain elements and R over relations, suggesting the expected interpretation for f^{-1} and x^{-1} in a context where f and x are implicitly universally quantified. More generally, mathematical texts often start setting up a local context of conventions used for disambiguation, and it is this context that drives disambiguation.

The last phase of a compiler is the translation of the semantically enriched abstract syntax tree into the target language. Loosely speaking, in the case of mathematical systems (and theorem provers in particular) which are based on a logic or a type system, this phase corresponds to the final internalisation (representation) of the formula in the logic. For instance, an equality over rational numbers can be represented using Leibniz equality (the smallest reflexive relation), using a decidable equality over unique representations of rational numbers (e.g. as lists of exponents for the unique factorisation), using an equivalence relation over equivalence classes of non-unique representations (such as pairs of integer numbers representing the enumerator and the denominator), and possibly in many other ways. Orthogonally we must also consider different representations of functions (e.g. as intensional, possibly executable algorithms, or as functional relations). In principle, we could expect that the choice of representation is less a question of ambiguity than of how the formal-

isation is done. Nevertheless, it is not unusual to have different alternative representations in the library of one system and we are convinced that the wealth of alternative representations is a peculiar mathematical feature that humans exploit all the time, moving, often implicitly, from one representation to the next one when needed. This becomes an ambiguity problem as soon as a mathematical library contains alternative representations.

In the rest of the paper the main example of ambiguity that we will show is related to overloading and subsumption in the numerical hierarchy. We could have easily found other examples of overloading totally unrelated to subsumption. For instance, an ongoing formalisation of Sambin's basic picture [12] requires us to overload the formal intersection operator $a \downarrow b$ in four different ways: first of all it is overloaded over observables and over sets of observables; orthogonally, it can be defined in terms of concrete points or directly over observables, yielding four different definitions that are extensionally equivalent, but still intensionally different, only when restricted to singletons and under certain conditions. Since the type-checking rules of the logic do not identify intensionally different definitions, further work to pass from one definition to another one is required when the system does not pick the expected interpretation. Typing conditions usually determine whether the operator must be considered over singletons or over sets. The context of usage in a larger sense determines instead what kind of definition is used and is harder to detect.

We choose subsumption as our running example not only because everyone is familiar with it: more importantly, subsumption in the numerical hierarchy is probably the most difficult example for two reasons. The first one is only technical: operators overloaded on the numerical hierarchy highly constrain each other unless coercions are inserted and coercions can be inserted anywhere, generating a high number of interpretations the system and user must cope with. The second reason is deeper: mathematicians are trained to ignore the difference between "equivalent" representations and thus are often puzzled by error messages deriving from ambiguity at the level of representation.

4 Disambiguation in Matita and Whelp

In a series of previous papers [9,3,10,11] we studied efficient algorithms that exploit type inference algorithms to speed up semantic analysis of ambiguous formulae. Our algorithms partition all possible interpretations of a formula in equivalence classes where every equivalence class contains either one single well-typed interpretation, or a set of interpretations all characterised by the same typing error. The latter equivalence class is represented by a formula containing placeholders such that every possible instantiation of the place-

User Provided Formula:

$\forall a, b, c, d.$

$(a +_1 b) *_1 (c +_2 d) =$

$a *_2 c +_3 a *_3 d +_4 b *_4 c +_5 b *_5 d$

the user does not enter the indexes

(Partial) Interpretation:

$= \mapsto =_{\mathcal{L}} \quad *_1 \mapsto *_{\mathbb{N}} \quad +_1 \mapsto *_{\mathbb{Z}}$

Corresponding term with placeholders:

$\forall a, b, c, d.(a +_Z b) *_N ?_1 =_{\mathcal{L}} ?_2$

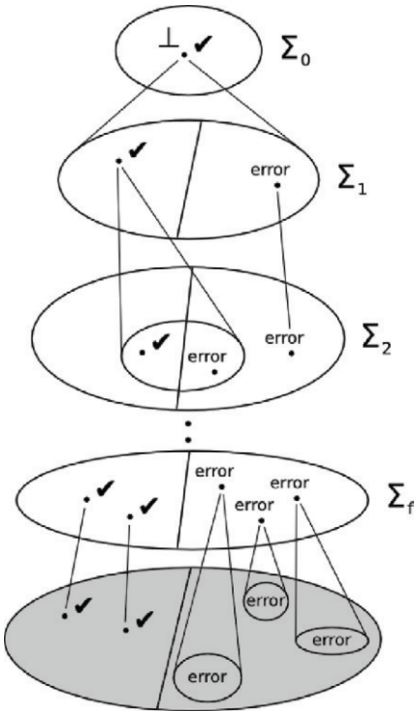
Refinement result:

`Error:` $a +_{\mathbb{Z}} b$ `has type` $\mathbb{Z}$ `but is here used with type` $\mathbb{N}$

Represented terms:

all instances of the formula that respect the partial interpretation constraints. The refinement error applies to all of them.

(a)



(b)

Fig. 1. (a) Partial interpretations (maps from overloaded symbol occurrences to their possible semantics) are useful to concisely represent sets of terms that are all characterised by the same typing error. Note that each symbol occurrence can be given a different meaning. (b) General schema for efficient disambiguation algorithms [11]. Each Σ_i is a set of partial interpretations. If the term with placeholders that corresponds to an interpretation is ill-typed, the interpretation is not refined any more, but it is propagated as it is. If it is not ill-typed (✓ mark), the interpretation is refined in Σ_{i+1} by defining it in any possible way on one more overloaded symbol occurrence. Σ_f is the final set where every interpretation marked with ✓ is now total and uniquely identifies a single term (without placeholders). All other interpretations, marked with `error`, identify ill-typed disjoint sets of terms (without placeholders) which are all instances of the ill-typed partial term corresponding to the interpretation. The set in gray is the set of all terms without placeholders which are ill-typed semantics of the ambiguous formula. This set is never explicitly represented by the system.

holders is an ill-typed formula. In turn, this formula containing placeholders is represented by a partial interpretation which maps some of the overloaded symbols to one of their possible meanings, and all remaining symbols to placeholders (see Fig. 1 (a)).

The main idea, explained in [9] and, formally, in [11] is that the cardinality of the set of all possible interpretations is exponential in the number of overloaded symbols, whereas the cardinality of the partition computed by the algorithm is much smaller. Indeed, it is linear in the number of well-typed interpretations. Moreover, we can compute that partition by consecutive refinements of coarser partitions of equivalence classes of interpretations which

are represented by terms with placeholders which are either ill-typed (not needing further refinement) or which are not known to be ill-typed (needing further refinement only if they contain at least one placeholder), see Fig. 1 (b).

Ambiguities introduced by the lexical analysis and parsing phases are not addressed by our algorithms for semantic analysis, but they could have been previously addressed with parsers recognising GLR grammars (see, for instance, [6]) in order to produce, from the input stream, compact representations of a set of abstract syntax trees to be fed to our semantic analysis. The final output can still be represented by a single partition that satisfies the properties required above.

We are now ready to explain the three scenarios described in Sect. 1 in terms of partitions.

4.1 First scenario: only one well-typed equivalence class.

In this scenario the partition is made of just one equivalence class representing a single well-typed term. In other words, there exists only one interpretation that "makes sense" (is well-typed), and possibly many others which do not. It is thus natural that the system picks the correct one without any interaction with the user, since it is unlikely (but not impossible) that the interpretation the user has in mind is a different one. Sect. 5 addresses the problem of providing non-invasive feedback to the user about the chosen interpretation.

4.2 Second scenario: several well-typed equivalence classes.

In this scenario the partition contains multiple equivalence classes representing well-typed terms with different meanings. What the system does is to rank them according to spatial and temporal considerations. In particular, every time a formula is disambiguated, the system remembers the way each symbol has been interpreted. Such local interpretations are named *aliases* and the most recent alias for a symbol is also recorded explicitly by the system. As we are going to explain, equivalence classes representing correct terms in the partition can then be ranked according to their degree of respect for aliases and in particular for most recent aliases.

Aliases are usually collected without explicit user intervention, but the user is given the possibility to explicitly insert in the script an alias declaration to further constrain the behaviour of the system. The system itself adds aliases declarations to the script in the case of interaction with the user.

In [3] we introduced *passes* to rank interpretations of a formula once the set of aliases and that of recent aliases are given. More precisely, every pass corresponds to an actual run of the algorithm described in the paper and

interpretations generated in one pass are all characterised by some criterion of adherence to the aliases. The criteria of later passes are obviously looser than the one of former passes, so that an interpretation generated in an early pass resembles interpretations for formulae typed recently by the user. Currently, we have adopted the next five passes. Each pass is tried only after the ones that precede it, and only if they failed to produce a valid interpretation. In [3] examples are given to motivate the need for each pass.

(i) do not insert coercions and use only the most recent alias for each symbol, i.e. use only the last recently used interpretations for a symbol

(ii) as the first one, but insert coercions

(iii) do not insert coercions and use only already used aliases, without inserting new ones, i.e. do not automatically pick a new interpretation for a symbol which has not been used yet

(iv) as the third one, but insert coercions

(v) look for all interpretations of a symbol, adding new aliases for the chosen interpretation. Equivalence classes in the partition are ranked according to the pass that produces them.

Multiple correct interpretations generated in the same pass are ranked in the same way. Thus the user interface must collect from the user enough information to pick the right interpretation among those maximally ranked or, on demand, among all rankings. This is the topic of Sect. 6. When there is just one maximally ranked interpretation, the system avoids user interaction but risks picking the wrong one. Thus, as in the first scenario, it is important that the user interface provides non-invasive feedback on the interpretation given to formulae. This is the topic of Sect. 5.

4.3 Third scenario: no well-typed equivalence class

The third scenario is the one where the partition only contains equivalence classes of interpretations that are not well-typed. This means that all possible interpretations contain an error. Our disambiguation algorithm that collects errors in equivalence classes already allows the number of meaningful alternative error messages to be reduced. Nevertheless, this is not sufficient since the user has (hopefully) a single interpretation in mind and, when presented with multiple errors associated to multiple interpretations, he must first spend time to spot the right interpretation before trying to understand the error. When too many interpretations are listed, this procedure can be so annoying that the user stops reading the errors and randomly tries to fix the error ignoring the potentially useful system-provided information.

In [10,11] we addressed the problem by ranking equivalence classes of ill-typed terms pruning out *spurious errors*. A spurious error is an error located in a sub-formula which admits alternative interpretations that assign to the same sub-formula a well-typed interpretation. The idea of a spurious error is that a spurious error is likely to be due to a wrong choice of interpretation, and not to a genuine user error. In a sense, it captures a sort of greedy criterium that seems to be the first one applied by humans when reading formulae: keep reading as much as possible making up your mind until you cannot do that any longer, and try to spot there the error.

Spurious error detection can be efficiently integrated in our efficient disambiguation algorithms and, according to our benchmarks in [11], is effective in reducing the average number of errors to be presented to the user. Nevertheless, we need a lightweight user interface to present the remaining non-spurious errors to the user, possibly ranked according to passes, and to present on demand also the spurious errors in the rare case of false positives [11]. This is the topic of Sect. 7.

5 Disambiguation feedback

Since the mathematical notation is overloaded and interpretations are automatically chosen by the system among the correct ones, it is important to provide feedback to the user on the way formulae are interpreted. We believe that hyperlinking every symbol, constant and notation to the definition of its semantics already provides on demand enough feedback. In the Matita theorem prover [3] and in the HELM/MoWGLI Web interfaces [1], this is achieved with hyperlinks that are followed when the user clicks on a symbol or constant. Moreover, a status line shows the URI of the hyperlink when the mouse is put over the symbol. The status line and the hyperlinks serve different purposes. The first one helps the user to understand what meaning has been assigned to a symbol; the second one is used more rarely when the user has forgot or ignores the exact mathematical definition associated to that meaning.

In place of hyperlinks, there are two other possibilities we know of. The first one is printing the formulae dropping all user-provided mixfix notations. This is also implemented in Matita and can be activated from the `View` menu. Nevertheless, dropping the usual infix notation, a formula can get rearranged in such a way that it becomes difficult to relate it to the original form. Thus this feature is mostly used for debugging. The second alternative is to follow the mathematical tradition and decorate every symbol so as to make explicit its interpretation, as is usually done when using two operators coming from different mathematical structures. This had been implemented in the past

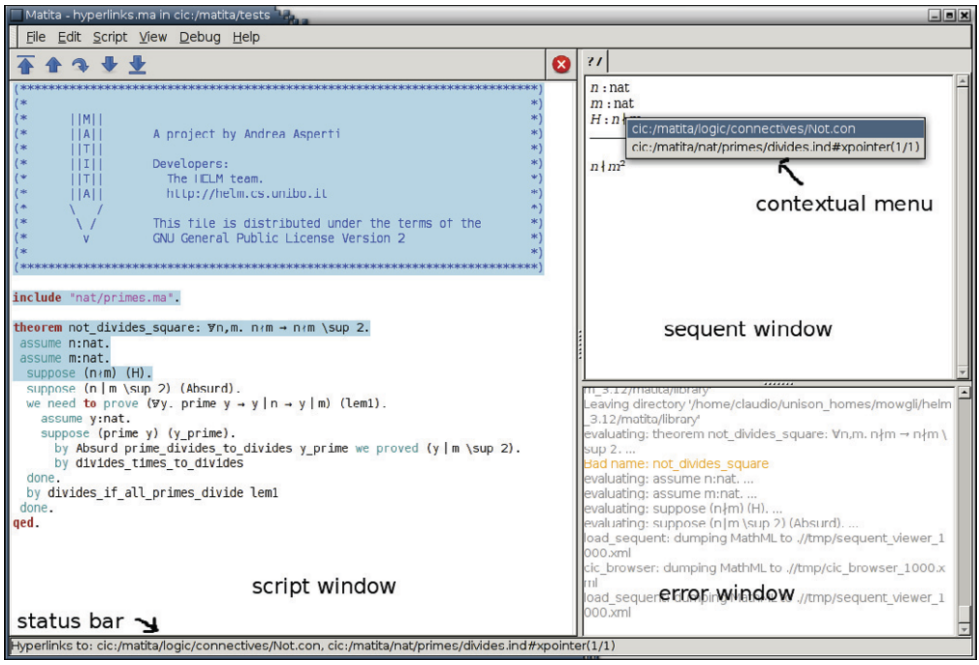


Fig. 2. The user has clicked on the "not divides" symbol in the hypothesis, which hides the formula `Not` (`divides` n m). The status bar lists the hyperlinks for `Not` and `divides` as soon as the mouse is over the symbol. The contextual menu is shown only after left button press on the symbol.

in an early version of the HELM/MoWGLI Web interfaces and we plan to port it to the interface of Matita. The drawback is that switching between decorated and undecorated forms heavily changes the shape of the formula and may provides excessive information. Hence, instead of doing it globally as we previously implemented, we plan to do it locally only on formulae selected by the user, as suggested by one referee.

Some mathematical notations hide more than one symbol, which are independently given a semantics. For these reasons, in Matita it is possible to have hyperlinks to multiple targets, each one identified by its URI. When the user clicks on the hyperlink (see Fig. 2), a contextual menu shows one distinct hyperlink for each URI.

User interfaces such as the one of Matita, which are based on the Proof-General [4] paradigm (whose origins go back to the LEGO mode for Emacs and to the CtCoq system [5]), are characterised by an input buffer (script window) and two output buffers (sequent window and error window), indicated in Fig. 2. Matita also has CIC browser windows (in the foreground in Fig. 3), which are non-modal floating windows used to browse and search in the library of the proof assistant. CIC stands for Calculus of (Co)Inductive Constructions, which is the logical framework implemented in Matita and the language of its library of compiled objects, which can be inspected with the

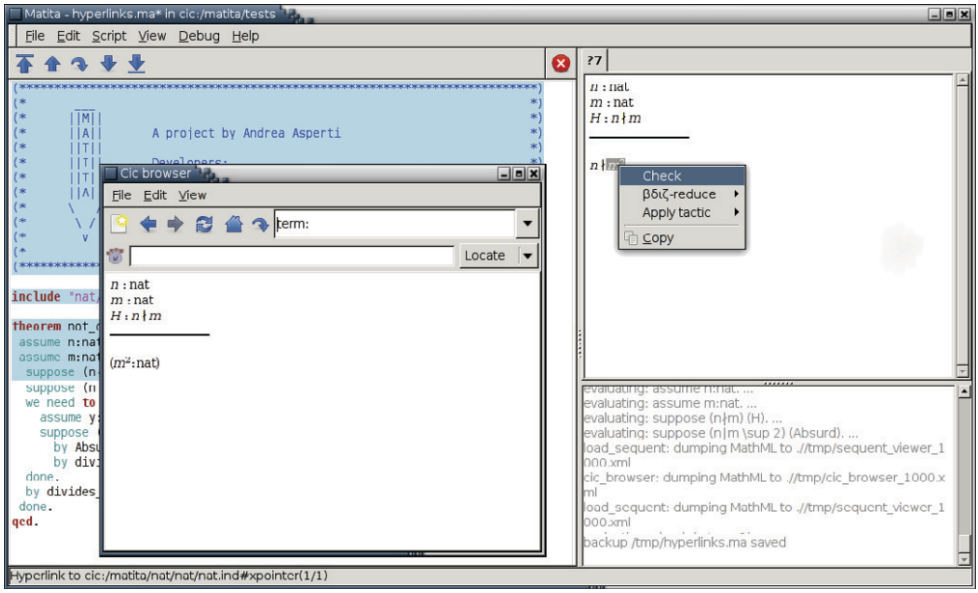


Fig. 3. After semantic selection of the well-formed sub-formula m^2, the user asks to compute its type. The selected term is shown in the CIC browser together with its type and the context it lives in.

CIC browser. The sequent window and the CIC browser are implemented by a widget for MathML Presentation [8], and formulae are translated from their semantics representation in CIC to MathML Content and then to MathML Presentation according to the transformations described in [1], which are responsible for generating the hyperlinks.

Thanks to these transformations, hyperlinks are provided in Matita for the sequent and CIC browser windows. On the other hand, the script window is implemented by a textual widget that shows user provided text. Traditionally, the upper part of the text, which has already been processed by the system, is read-only and highlighted changing the background colour (see again Fig. 2). Formulae contained in the locked text have already been disambiguated for execution. Moreover, in order to localise error messages, the position of all tokens has been recorded during parsing. If the textual widget supports hyperlinks and contextual menus, it is easy to add hyperlinks also to the locked part of the text. This is currently planned for Matita, but not implemented.

Hyperlinks do not help with subsumption. When subsumption is required to type a formula, the system may add coercions in order to record where subsumption has been used, and we would like to inform the user about that in a non-invasive way. However, since coercions are not represented visually in any way (to avoid too much noise), there is in general no place to put a hyperlink. Thus, our current solution is to provide additional feedback

allowing the user to semantically select sub-terms in the sequent window and ask to compute their type as in Fig. 3 in order to understand why coercions were inserted. This parallels the possibility of following hyperlinks for notation and identifiers by clicking on them. What is missing is a mechanism similar to the status bar hints to understand what and where coercions have been inserted. We believe that this could be implemented by modifying the formula to show the coercions when the user holds the mouse over a subformula that has been coerced. Another strategy we have adopted is to add an option to the `View` menu of Matita to temporarily stop hiding of coercions. As for deactivation of notations, this feature is used mainly for debugging and it is not very satisfactory since the feedback is too invasive and the shape of the formula changes too much.

Subformulae must be *semantically selected* to compute their types. Semantic selection [3] is a restriction of selection to well-formed sub-formulae which we provide on top of the MathML widget. It is to be compared with textual selection or graphical selection that allow to select regions of text that are unparsable. We do not plan to provide the same functionality on the script window, since semantic selection is not easily supported by textual widgets and since re-computing the type of the sub-formula requires re-disambiguation of the formula under the same conditions the formula was disambiguated in the first time. These are no longer the conditions the system is in.

6 Choosing an interpretation

As already discussed in the introduction, after disambiguation of a formula there could be multiple equally ranked interpretations, that differ on the interpretation of at least one overloaded notation. User interaction is required to pick one of the maximally ranked interpretations. Fig. 4 shows a very simple example where the user in a new file starts using the infix addition and multiplication notation which are overloaded in the library over integer and natural numbers. The system computes the partition of ranked equivalence classes of interpretations, finding two interpretations with maximal rank. In the first one, all occurrences of the symbols are interpreted over natural numbers, in the second one over integer numbers. Other correct interpretations that receive a lower rank are obtained by considering subsumption between natural and integer numbers. For instance, another possible interpretation is given by $\forall a, b : \mathtt{nat}.\forall c, d : \mathtt{int}.(a +_{\mathbb{N}} b) *_{\mathbb{Z}} (c +_{\mathbb{Z}} d) = a *_{\mathbb{Z}} c + a *_{\mathbb{Z}} d + b *_{\mathbb{Z}} c + b *_{\mathbb{Z}} d$. It receives a lower rank since it is generated only during the fourth disambiguation pass where all aliases and coercions are used, whereas the two maximally ranked are generated during the third one, where no coercions is inserted.

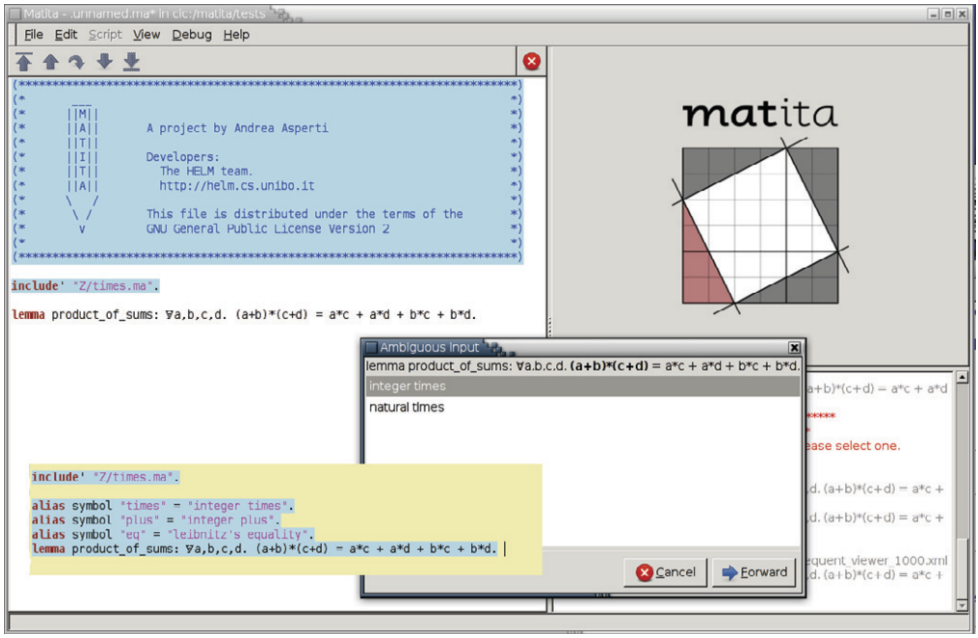


Fig. 4. Ambiguous input. The user is asked for the interpretation of $*$ in the sub-formula $(a + b) * (c + d)$ (which is highlighted in bold). Since the interpretation for $+$ is constrained by the choice, no further questions need to be asked, and the **Forward** button adds the aliases to the script, as shown in the superimposed script snippet (in darker background) that shows the content of the script window after the interaction. The **include'** command activates the infix $+$ and $*$ notation for integer and natural numbers without pre-loading any alias.

Since the system is unable to decide which maximally ranked interpretation is the one expected by the user, it computes a tree of discriminating questions among interpretations. Each node in the tree is a multiple answer question about the meaning of a symbol, where the possible answers range among the meanings used in the set of correct interpretations. The node has a child for each possible answer. The root of the tree is the question that allows the higher number of interpretations to be pruned. Its children are computed recursively according to the same criterion applied to the remaining set of correct interpretations. In our example, we get a degenerate tree with only one node, since one question is sufficient to identify just one maximally ranked interpretation.

Before the introduction of the tree of questions, Matita used to propose to the user one interpretation at a time, showing both the formula with hyperlinks and a table associating each symbol occurrence with its interpretation, similar to the way it is done in the central part of the error window shown in Fig. 5. That interface required a lot of effort from the user who had to inspect globally each interpretation at a time, passing the mouse over every symbol of the formula or reading the whole table.

The next interface we tried used to present a visual tree, similar to the one used to navigate file systems, that allowed to pick interpretations navigating in the tree. The interface was also quite clumsy since there was no more visual feedback on the choices.

The current interface, as the second one, allows the user to take a local decision inspecting only part of the formula but, as the first one, provides an immediate visual feedback on where he is acting in the formula. Moreover, the interface now minimises the number of questions and it recall the familiar interaction style of wizards/assistants, requiring no training period.

Since the system interacts with the user, it is important that the user provided information is recorded somewhere in the script in order to avoid repeating the interaction the next time the script is processed. This is achieved by automatically adding aliases to the script, as shown in the script snippet superimposed to Fig. 4. Aliases where already discussed in Sect. 4.2. The syntax used in the figure is also available to the user to drive the ranking of disambiguations: `alias id "`*`name`*`" = "`*`description`*`"`. The string before the equal sign in an alias declaration is the name of the MathML Content symbol used to give a representation of the notation at the content level. The description after the equal sign was previously associated by the user to a MathML Content to MathML Presentation mapping when declaring the notation. We also associate aliases to identifiers which are represented in MathML Content by themselves. In this case, the syntax becomes `alias id "`*`name`*`" = "`*`URI`*`"`.

Most of the aliases, however, are automatically and deterministically inferred by the system without being explicitly recorded in the script. Indeed, the next time the script will be executed, the very same alias will be generated again. For instance, every time the user gives a new definition, it is assumed that the definition is going to be used in the immediate future and an alias is explicitly declared.

Explicit aliases in the script look similar to Mizar's environments where the user needs to list, at the beginning of an article, all notations (but also definition and theorems) he wants to use. But the syntactic similarity is (partially) misleading: in Matita all definitions, theorem and, potentially [3], notations are always visible and the user does not need to declare in advance which parts of the library he intends to use. On the other hand, like in Mizar, the list of aliases in a script becomes very large when no alias is pre-loaded in advance.

[3] The current implementation of Matita is based on the CamlP5 parser which does not handles GLR grammars. Thus it is currently not possible to pre-load all user notations given in the library. The `include` command of Matita thus performs both pre-loading of user notation and pre-loading of aliases. The `include'` alternative form pre-loads notation alone. We are currently experimenting with alternative GLR grammars for OCaml in order to remove this limitation.

To address this we provide the `include` command that pre-loads all aliases that were active at the end of a previous script. The `include` command looks similar to Coq's `import` or to Isabelle's theory importing machinery and it leads to the same advantages with respect to explicitly listed aliases (see [13], Sect. 4.8 for a short comparison). Even in this case, however, the similarity is only syntactical, since definitions, lemmas and potentially notations can be used anytime in Matita even without including them. The `include` command only pre-loads aliases to set preferences (that can be overridden) on the preferred interpretations for overloaded symbols and notations.

Our approach is very satisfactory from the user point of view, especially when the content of the library is unknown. For instance, students are usually unaware of the organisation of the standard library of Matita and thus fail to include in advance all scripts they will depend on. This is witnessed by the explicit aliases automatically inserted by the system in their scripts every time they use some lemma or definition in a script they forgot to include. Moreover, when the script is executed again, no penalty is spent in searching lemmas in the library since the inserted aliases are tried before. Finally, from the URIs in the aliases it is possible to infer the correct include commands if the user prefers to make all dependencies explicit at the beginning of his file.

7 Error reporting

As already discussed in the introduction, disambiguation of a formula containing an error results in a partition made of ranked equivalence classes of interpretations characterised by the very same error (one for each equivalence class). This is the most difficult scenario for an user interface, since the user is already making a mistake (and thus he can be confused), and we risk showing errors relative to interpretations he does not mean (increasing the confusion) and providing too much information (augmenting the confusion and the time to data-mine the information). In practice, before the introduction of the current interface, we noticed that our users were ignoring the system feedback and were trying to understand the errors without machine help.

The main observation that allowed us to improve the situation came from Ferruccio Guidi, who made us notice that, even when programming in OCaml [4] ,

[4] The most frequently reported error by the OCaml compiler is "This expression has type T_1 but is here used with type T_2" where T_1 and T_2 are inferred by the compiler according to the usage of the bound variables. However, since inference is done globally on the source code, it frequently happens that one of the types is inferred incorrectly, and the error message becomes misleading. On the other hand, the error location is almost always correct. Thus users tend to read the error only if they are unable to immediately spot the problem from the error location.

users tend to ignore the actual error message in favour of the error location only. Indeed, either the error location is misleading, and in this case the error message usually is also, or it provides information that the human mind can perceive and process very quickly via highlighting. Moreover, in most cases the error is likely to be trivial enough when exactly spotted in the formula. On the other hand, even if informative, the error message requires an additional effort and some time to be read and understood. Reading it is of worth only when the mistake is so involved that fixing it required more insight. Thus we changed the user interface in order to present to the user a list of error locations and, only on demand, the error messages relative to that location and, even more lazily, the description of the partial interpretation that is affected by that error and that, very often, can be partially inferred from the error message itself.

A previous interface we tried used to present interpretations first to make the user spot the correct one, so that the error message and location that were shown were granted to be significant. Most of the time, however, spotting the right interpretation processing various kind of information was more difficult then spotting the right error location by cycling between error locations only.

In [11] we tried to quantify how often the error locations spotted in a formula are the correct ones. We did so by randomly inserting errors in 436 theorems from a formalisation where overloading is heavily used. In the 74.1% of the cases the system was able to detect precisely the error location, reporting an average of three different error messages for that location. This confirms that the error location is more informative than the error messages themselves.

Fig. 5 and 6 show our user interface both in the frequent but degenerate case of one single error location, and in the general case. As explained in the captions, the user interface hides by default the error messages (and relative locations) for those interpretations that are unlikely to be the one the user has in mind, according to the ranking (which, in turn, depends on the phase used to constrain the interpretation and on the spurious error criterion).

In the near future we do not plan any significant change to the user interface. On the other hand, we expect to continue working on the improvement of ranking and spurious error detection to further reduce the number of errors to show, at the price of increasing the risk of false negatives.

8 Conclusions

As far as we know, Matita is the only theorem prover that supports arbitrarily overloaded notation and that implements a user interface to cope with ambiguities. We have identified three situations where the user interface plays an

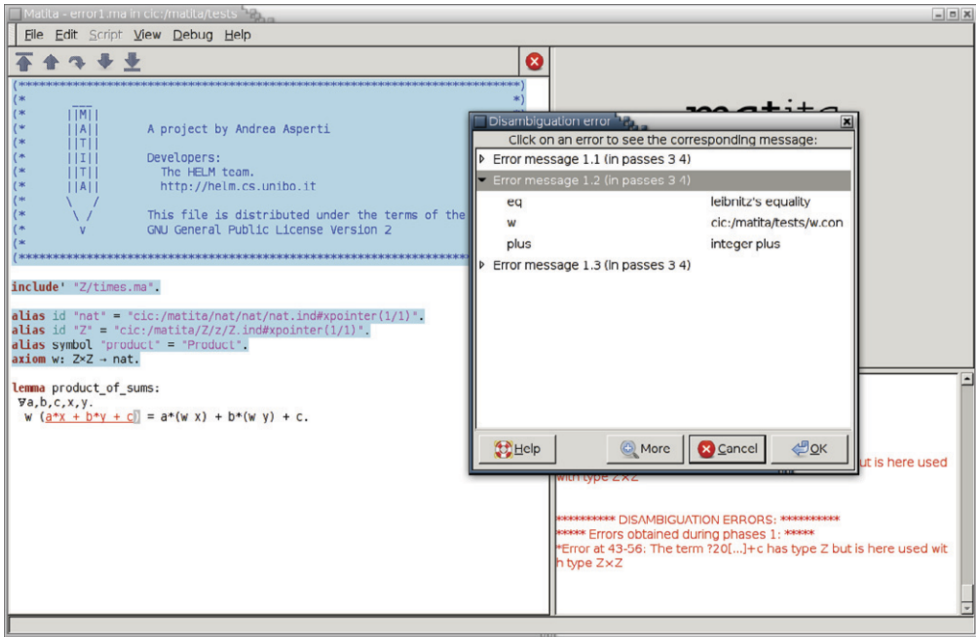


Fig. 5. Error parsing an ambiguous formula. Three partial interpretations are sufficient to represent all possible errors. Moreover, all the errors are located in the same sub-formula $a * x + b * y + c$ which is underlined and highlighted in red. The error message relative to the first interpretation is shown in the error window and a modal window allows the user to see the error messages associated to the other two interpretations. Error messages are always displayed in turn in the error window. By clicking on the small triangles, the user can also see on demand the aliases that make up the interpretation. The latter feature may be necessary to fully understand the error message. Errors relative to passes 1 and 2 are hidden since they also belong to passes 3 and 4. Errors belonging to pass 5 (which completely ignores aliases) as well as errors classified as spurious are not shown by default, but they are shown by pressing button `More`.

important role.

The first one is in providing feedback about the way the system has interpreted symbols. This has been achieved using conventional techniques like hyperlinks and type inference for sub-formulae.

The second situation is user interaction to select one interpretation among those that are deemed equally likely. Our strategy consists of minimising the interaction by building trees of maximally discriminating questions. An additional challenge is the need to record the user choices to avoid repeating the interaction.

The third and most critical situation is that of presenting multiple error messages associated to a wrong formula. Here we designed an interface to progressively provide information to the user on-demand, starting from the less informative (but less confusing) one (i.e. error locations) and moving to the one requiring more effort to be understood by the user.

A preliminary version of the user interface was implemented for the Whelp search engine [2], and re-implemented in Matita, but it was not satisfactory.

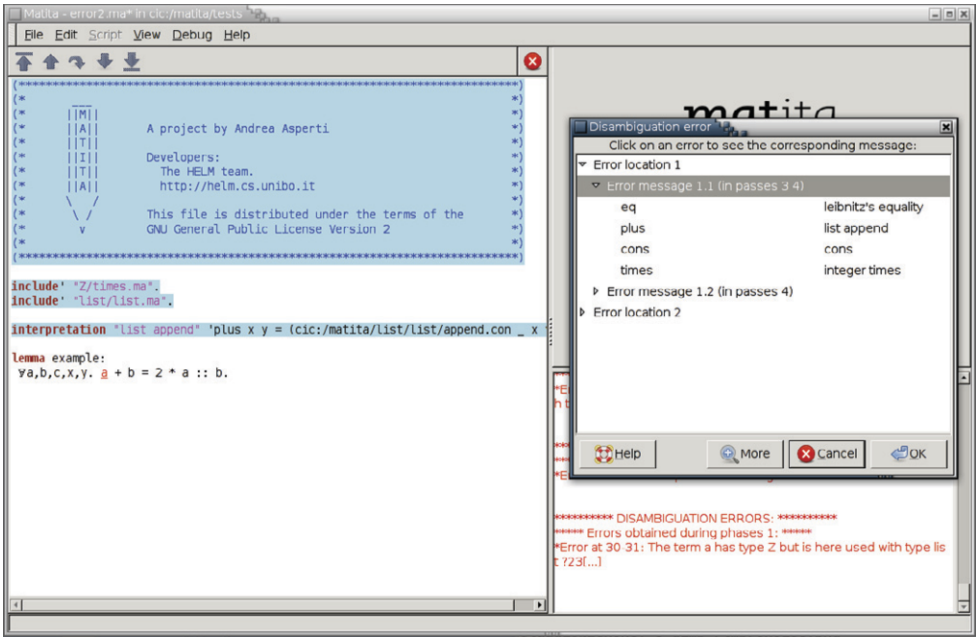


Fig. 6. Error parsing an ambiguous formula. The symbol + has been overloaded also as list append. The symbol :: adds an element (on its left hand side) to a previously existing list (on the right hand side). Since lists are homogeneous, the formula is ill-typed. Interpretations (and corresponding error messages) are categorised by error location first, then by error message. The error location is underlined and highlighted in red when the user clicks on an error location entry in the modal window. Error messages are printed one at a time in the error window when the user clicks on the error message in the modal window.

Matita now implements the new interface described in the paper and we plan to port a modified version of the new user interface also to Whelp. The main difference will be in the use of aliases to rank interpretations. Indeed, in an interactive theorem prover it is possible to understand when the user stops working on some topic to start something completely different. In contrast, a search engine just answers to a stream of queries that could be related or not. Hence the user should be given more explicit control on the management of aliases in order to drop them and start from scratch a new session of queries forgetting the previous disambiguation choices.

As discussed in the introduction, user validation of the system and of alternative variants for its interface is particularly complex, since we do not have a large group of users with uniform skills that can try to formalise the same thing in parallel. Making the same user redo the same formalisation was attempted, but gave biased results since the user has improved his understanding of the domain and the library and tends to commit different mistakes when writing formulae. We also feel that comparing the user experience on different formalisations may be misleading. A more viable alternative, as suggested by one referee, would be to evaluate the system according to some usability

framework. This is currently left as future work. Thus, right now, the main evidence for the benefits of our work is the current degree of satisfaction of our users, to be compared with their degree of insatisfaction with previous versions of the system, where disambiguation often proved to be a show-stopper because of user interface issues. This has no longer been the case since the implementation of the interfaces described in Sect. 6 and Sect. 7.

References

[1] Asperti, A., F. Guidi, L. Padovani, C. Sacerdoti Coen and I. Schena, *Mathematical knowledge management in HELM*, Annals of Mathematics and Artificial Intelligence **38(1-3)** (2003), pp. 27–46.

[2] Asperti, A., F. Guidi, C. Sacerdoti Coen, E. Tassi and S. Zacchiroli, *A content based mathematical search engine: Whelp*, in: *Post-proceedings of the Types 2004 International Conference*, Lecture Notes in Computer Science **3839** (2004), pp. 17–32.

[3] Asperti, A., C. Sacerdoti Coen, E. Tassi and S. Zacchiroli, *User interaction with the Matita proof assistant*, Journal of Automated Reasoning **39** (2007), pp. 109–139.

[4] Aspinall, D., *Proof General: A generic tool for proof development*, in: *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000*, Lecture Notes in Computer Science **1785** (2000).

[5] Bertot, Y., *The CtCoq system: Design and architecture*, Formal Aspects of Computing **11** (1999), pp. 225–243.

[6] Heering, J., P. Klint and J. Rekers, *Incremental generation of parsers*, IEEE Trans. Softw. Eng. **16** (1990), pp. 1344–1351.

[7] Luo, Z., *Coercive subtyping*, J. Logic and Computation **9** (1999), pp. 105–130.

[8] Padovani, L., "MathML Formatting," Ph.D. thesis, University of Bologna (2003), technical Report UBLCS 2003-03.

[9] Sacerdoti Coen, C. and S. Zacchiroli, *Efficient ambiguous parsing of mathematical formulae*, in: A. Asperti, G. Bancerek and A. Trybulec, editors, *Proceedings of Mathematical Knowledge Management 2004*, Lecture Notes in Computer Science **3119** (2004), pp. 347–362.

[10] Sacerdoti Coen, C. and S. Zacchiroli, *Spurious disambiguation error detection*, in: *Proceedings of Mathematical Knowledge Management 2007*, Lecture Notes in Artificial Intelligence **4573** (2007), pp. 381–392.

[11] Sacerdoti Coen, C. and S. Zacchiroli, *Spurious disambiguation errors and how to get rid of them*, Submitted to Journal of Mathematics in Computer Science, special issue on Management of Mathematical Knowledge. Available at the first author's home page. (2008).

[12] Sambin, G., "The Basic Picture: a structural basis for constructive topology," to be published in 2009.

[13] Wenzel, M. and F. Wiedijk, *A comparison of Mizar and Isar*, Journal of Automated Reasoning **29** (2002), pp. 389–411.