Efficient Ambiguous Parsing of Mathematical Formulae*

Claudio Sacerdoti Coen and Stefano Zacchiroli

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

Abstract. Mathematical notation has the characteristic of being ambiguous: operators can be overloaded and information that can be deduced is often omitted. Mathematicians are used to this ambiguity and can easily *disambiguate* a formula making use of the context and of their ability to find the right interpretation.
Software applications that have to deal with formulae usually avoid these issues by fixing an unambiguous input notation. This solution is annoying for mathematicians because of the resulting tricky syntaxes and becomes a show stopper to the simultaneous adoption of tools characterized by different input languages.
In this paper we present an efficient algorithm suitable for ambiguous parsing of mathematical formulae. The only requirement of the algorithm is the existence of a "validity" predicate over abstract syntax trees of incomplete formulae with placeholders. This requirement can be easily fulfilled in the applicative area of interactive proof assistants, and in several other areas of Mathematical Knowledge Management.

1 Introduction

Mathematicians are used to well established and ambiguous notational conventions. To design a software application which have to deal with mathematical formulae, we have to consider this habit.

Sometimes a mathematician who is reading a formula, solves the ambiguity using the notion of *context*: for instance, if f is known to be a scalar value, then f^{-1} is the inverse value $1/f$; if f is a function, then f^{-1} is probably the inverse function of f.

More often the context is not sufficient and the ambiguity is solved by picking the only interpretation that makes sense or the interpretation that is more likely to have one. For instance, without additional information, a mathematician is like to interpret $\phi^2(x)$ as $(\phi \circ \phi)(x)$ although $\sin^2(x)$ is probably interpreted as $(\sin x)^2$.

* The authors have been partially supported by 'MoWGLI: Math on the Web, Get it by Logic and Interfaces', EU IST-2001-33562

Associating precedences to operators does not always solve ambiguity problems as shown by the combined usage of conjuction and equality operators: $A \wedge B = B \wedge A$ is usually interpreted as equality of conjuctions whereas $x = y \wedge y = x$ as conjuction of equalities.

Using unambiguous grammars computer scientists avoid ambiguity problems in programming languages and programmers are used to learn them from scratch along with their ad hoc syntactical rules. Mathematicians can of course do the same with new proof assistants or other Mathematical Knowledge Management (MKM) applications. Notwhithstanding this fact, there are motivations for trying to recognize ambiguous mathematical inputs in the MKM context. In the following paragraphs we try to identify some of them.

Different mathematicians adopt for the same concept different notations. We do not want to give up this possibility for technical reasons. In MKM indeed there are several situations where the user has to write or read a mathematical formula without knowing the context, in which notations are defined. Let us consider a search engine for mathematical results, as the one described in [5]. The user enters a statement to retrieve all its proofs. In order to do so, she has to pick a notation and use symbols and constants that are not fixed by any context. Thus the search engine has to be able to parse the ambiguous sentence, disambiguate it and perform the search modulo the context.

A feature that is usually provided to the user of a Computer Algebra System (CAS) or a proof assistant is that of defining new notations, which are stored in the context. Sooner or later it will happen that a user needs to merge together two contexts where the same notation is used for different concepts. If these systems do not handle overloading, the easiest form of ambiguity, at least one of the two notations is actually masked and the user has to redefine it, creating a break point in the notational uniformity of the whole development. The Mizar[10] library committee has already faced several times these kind of problems, that were solved by changing one of the two notations and updating the whole Mizar library.

Among the topics of MKM there are digitalization and enhancement of already existing mathematical knowledge. Since the notation used in already existing mathematical documents is ambiguous, ambiguity must be addressed in both phases. Morever, since the amount of such documents is huge, we should minimize as much as possible the amount of disambiguation work left to the human.

In this paper we outline an efficient algorithm which can be used to perform ambiguous parsing in applications pertaining to the MKM area. The algorithm can be seen as an improvement of the algorithm for ambiguous parsing used in type based parsers as the one of the Grammatical Framework [11]. Type based parsing imposes a type system over the grammar of the language that must be recognized, and uses the typing judgement to prune wrong interpretations. In the context of mathematical formulae no additional type system needs to be imposed since arity, domain and codomain information associated to mathematical operators can play the role of a type system that supports the required operations.

In Sect. 2 we present the classes of efficient and modular parsing algorithms we are interested in; in Sect. 3 we describe *refinement*, the predicate needed to validate formulae fragments; the algorithm itself is presented in Sect. 4 and analyzed in Sect. 5, where several future research directions are also outlined; final remarks and conclusions are given in Sect. 6.

2 Requirements

Traditionally, compilers have been implemented as pipelines of phases. The first phase, *lexing*, takes in input a stream of characters and produces as output a stream of tokens. These tokens are the input for the *parsing* phase which in turn produces as output one or more abstract syntax trees (or ASTs for short). The *semantic analysis* phase finally takes in input ASTs and enriches their nodes with semantics information. Each phase can fail, aborting the whole process.

The parsing phase is in charge of recognizing if the input phrase pertains to a predefined grammar or not. This grammar is said to be unambiguous if, for any given input phrase, there exists at most one AST; otherwise the grammar is an *ambiguous grammar*.

The output of the parsing phase for an unambiguous grammar is either a *parse error* or the unique AST for the input formula. This property does not hold for ambiguous grammars. In that case the output of the parsing phase is no longer a single tree, but rather *a set of trees*.

Ambiguous grammars are just one aspects of ambiguity, the other one we will consider is introduced by the semantic analysis phase. We say that an AST is an *ambiguous AST* when the semantic analysis phase can associate to it more than one semantics — whatever the semantics is.

When this kind of ambiguity has to be considered, the output of the semantic analysis phase is *a set of semantics* for each AST in the input set. Let us consider the formula $(1+2)*3^{-1}=1$, and suppose we are in the applicative area of proof assistants. This formula exhibits the latter aspect of ambiguity described, since each symbol in it can have many different semantic interpretations:

1. each number can be a natural, integer, rational, real or complex number;
2. "-1" can either be the application of the unary operator "$-$" to 1, or the integer, rational, real or complex number -1;
3. "$=$" can represent many different equalities (Leibniz's polymorphic equality an equivalence relation over natural or real numbers, ...);
4. "$+$" and "$*$" can be defined over naturals, integers, ...;
5. if subtyping does not hold, implicit coercions can be inserted to inject a number of one type in a "super type";
6. since different representations of a concept have different computational properties, several representations of, say, natural numbers may be available.

Similar observations can be made in applicative areas other than that of proof assistants. For instance, some of them applies directly to CASs. In the rest of the paper we want to address the two forms of ambiguities at the same time.

We say that an algorithm performs *ambiguous parsing* when it associates to an input formula E the set of all the semantics for E. This means that we are considering the second and third phases of a traditional compiler as just one macro phase. The motivation is efficiency: we want to be able to detect semantically wrong formulae already during the parsing phase, without generating a multitude of ASTs that later on cannot be assigned any meaning.

At the same time, we do care about the modularity induced by the distinction of the two phases. Thus these are the two requirements that we want our algorithm to satisfy:

1. The algorithm should be *compositional*: the semantic analysis phase should be as separate as possible from the parsing phase.
2. The algorithm should be *efficient*: semantic errors must be detected as early as possible, preventing further parsing and semantic analysis of already incorrect interpretations.

The apparent conflict between the two requirements manifest itself in implementations that fulfil one requirement sacrificing the other. For instance, let us consider again the formula $(1 + 2) * 3^{-1} = 1$. The naive compositional algorithm (NCA for short) performs parsing first, returning a huge set of ASTs, one for each combination of the possible interpretations of each symbol. For instance, one possible output is the tree whose root is equality over real numbers and whose second argument is the Peano number 1. Then semantic analysis proceeds at pruning out the most part of the combinations: for instance, every tree whose root is the equality over real numbers and whose second argument is not a real number is pruned. Unfortunately, this algorithm is not efficient, since several trees are parsed and analyzed even if looking at the root and its second argument already provides enough information to prune them out. However, due to its triviality, NCA is the most commonly used algorithm [11].

Let us now consider the most efficient top-down algorithm, which generates every possible interpretation for the root, and for each interpretation calls itself on the two arguments remembering what the expected type of the argument was. The expected type is used to pick for the second argument of the root the only interpretation that respects the type. Corresponding efficient bottom-up algorithms can be easily built as well. Clearly this algorithm is not compositional, since it performs type checking already during the parsing phase, and just on a fragment of the tree.

In Sect. 4 we propose a compositional and yet efficient algorithm for ambiguous parsing of mathematical formulae, showing that the conflict is only ephemeral. The main assumption underlying the algorithm is the availability of the refinement operation described in the next section. Intuitively, the function allows to detect semantic errors on partial ASTs and thus it can be used already during the construction of the AST set that is the output of the parsing phase.

To the authors knowledge the algorithm is new, despite its apparent simplicity. Its novelty was also confirmed by private communication with Aarne Ranta, who is the main author of Grammatical Framework (GF) [11] and without any doubt one of the major experts in this field.

3 Refinement

The crucial ingredient of our parsing algorithm is the function that performs semantic analysis of the abstract syntax tree of the formula. Its goal is to detect invalid formulae. Since the most common mistake in a formula is the application of an operator to arguments outside its domain, this function is likely to be a sort of type deriver. Indeed, every foundational system used to encode the formulae has its own version of a (weak) "type" system.

Let us consider several kind of formulae encoding used in the mathematical practice and in MKM. Set theory is the most widely used foundation among mathematicians. In principle, every operator in set theory is untyped (or unityped), since every term is a set and set theoretical operations can be applied to any set. For instance, the natural number zero can be defined as the empty set and the natural number one as the set $\{\emptyset\}$. Thus the formula $0 \in 1$ is perfectly valid. Notwithstanding this, mathematicians that work in set theory use to consider natural numbers as a sort of abstract data type, forgetting their concrete definition and considering "well typed" only the applications of operators whose domain is the same or a subset of the set the argument belongs to.

The logical foundation that Computer Algebra Systems are commonly based on is some variant of multi-sorted first order logic. The arity and sorts of the symbols define the type system that prevents wrong applications inside formulae.

The type system becomes a truly first class citizen in mathematical reasoning tools based on type theory. The majority of Proof Assistants (PA), with the significant exception of Mizar[10], are based on type theory, where a type is assigned to every term, and a product (or arrow type) is assigned to operators. The application is well-typed if the type of the argument matches the domain of the product.

Type systems can be classified according to the presence or absence of dependent products. A non-dependent product $T_1 \to T_2$ types functions whose output type T_2 does not depend on the actual argument. On the contrary, a dependent product $\Pi x : T_1.\ T_2(x)$ types functions whose output type $T_2(x)$ is a function of the actual argument x. Dependent type systems allow to express tighter constraints on the arguments of an operator, rejecting a larger number of wrong formulae and making the disambiguation algorithm more effective, but also more complex.

Type derivers are programs that check whether a type can be assigned to a formula, and that can optionally return the computed type. Thus they can be exploited as *pruning functions* for NCA: after generating every AST, the type deriver is used to prune invalid ASTs. To implement our efficient version of the algorithm we need an extension, called *refiner*, of a stronger version of the type deriver able to deal with incomplete formulae. However, our algorithm is fully generic, since it is not based on a particular calculus or a particular type system. The only necessary requirement is the possibility to represent incomplete formulae and the existence of a refiner.

An *incomplete formula* is a formula where non linear placeholders for subformulae occur. Every non linear placeholder $?_i$ where i is a natural number replaces the occurrence of a subformula. Occurrences of equal placeholders replace equal subformulae. In type theory literature, placeholders are called *metavariables* [8, 13, 7].

In type theory for instance, to obtain a typed calculus with metavariables from a typed calculus of closed terms (where no metavariable occurs), metavariable occurrences are added to the term grammar, and a typing context for metavariables is added to the typing rules. The typing context associates a type to each metavariable, that is the type of the term replaced by the metavariable. In the case of dependent products, the type of the metavariable in the typing context is replaced by a sequent, that describes the type of the replaced term and the free variables that (may) occur free in the replaced term, together with their type. For instance, the formula $\forall x : \mathbb{Z}.x = ?_1$ is well typed if the sequent $x : \mathbb{Z} \vdash ?_1 : \mathbb{Z}$ is associated with $?_1$, stating that $?_1$ is a term of type $\mathbb{Z}$ parametric in x also of type $\mathbb{Z}$.

Definition 1 (Refiner). *A* refiner *is a function whose input is an incomplete formula t_1 and whose output is either:*

- *an incomplete formula t_2 where t_2 is a well-typed formula obtained by assigning a type (or a sequent) to each metavariable in t_1 and, in case of dependent types, by instantiating some of the metavariables that occur in t_1;*
- *the special constant ϵ if there exists no well-typed formula t_2 that can be obtained by assigning a type (or a sequent) to each metavariable in t_1 and by instantiating some of the metavariables;*
- *the special constant $\perp$ when neither one of the two previous cases can be verified by the algorithm.*

Whereas type checking is usually decidable, refinement is usually a semidecidable problem. Thus, in the literature, refinement algorithms are usually described to return either a refined term t_2 or the $\perp$ answer [7]. In the Ph.D. thesis of the first author of this paper [12] an algorithm that returns the three previous answers is described as a simple modification of the classical algorithm. Notice that if $refine(t) = \perp$ then no information is actually provided: the formula could either be refinable or not. As a consequence, our disambiguation algorithm can only exploit the other two answers, and the refiner will be more and more useful as long as it minimizes the number of $\perp$ outputs.

A simple example should clarify the usage of the refinement algorithm. Let us consider the incomplete mathematical formula $?_1 = \sqrt{?_2}$ represented in set theory and in a type theory of total functions with dependent products, and let's try to refine it. We use juxtaposition for operator application, following the usual λ-calculus convention.

– *Set theory:* $?_1 = \sqrt{?_2}$ is encoded as $(=\ ?_1\ (sqrt\ ?_2))$ where $=$ is the polymorphic equality over arguments that belongs to one set and *sqrt* is a function whose domain is the set $\mathbb{R}^+_0$ of non negative real numbers and whose codomain is the set $\mathbb{R}$ of real numbers.
  $refine(=\ ?_1\ (sqrt\ ?_2))$ returns $(=\ ?_1\ (sqrt\ ?_2))$ since the term is well-typed once the type $\mathbb{R}$ is assigned to $?_1$ and the type $\mathbb{R}^+_0$ is assigned to $?_2$.
– *Dependently typed type theory of total functions:* $?_1 = \sqrt{?_2}$ is encoded as $(=\ ?_3\ ?_1\ (sqrt\ ?_2\ ?_4))$ where $=$ is the monomorphic ternary operator whose first argument is a type and whose other arguments are terms of that type; *sqrt* is the binary operator whose first argument x is of type $\mathbb{R}$ and whose second argument has dependent type $(\geq\ x\ 0)$ (i.e. it is a proof of $x \geq 0$).
  $refine(=\ ?_3\ ?_1\ (sqrt\ ?_2\ ?_4))$ returns $(=\ \mathbb{R}\ ?_1\ (sqrt\ ?_2\ ?_4))$ since the term is well-typed once that $?_3$ is instantiated with $\mathbb{R}$, the sequent $\vdash ?_1 : \mathbb{R}$ is assigned to $?_1$, the sequent $\vdash ?_2 : \mathbb{R}^+_0$ is assigned to $?_2$ and the sequent $\vdash ?_4 : (\geq\ ?_2\ 0)$ is assigned to $?_4$.

Notice that, in the case of dependent type theories, the refiner sometimes needs to instantiate metavariables that occur in the argument of an operator whose type is a dependent product. For instance, in the previous example the type of the monomorphic equality is $\Pi T : Type.\ T \to T \to Prop$ and $?_3$ is the argument of the formal parameter T. Thus the type $\mathbb{R}$ of the third operator argument $(sqrt\ ?_2?\ _4)$ must match the type $T = ?_3$ and this holds only instantiating $?_3$ with $\mathbb{R}$.

In the next section we will describe our disambiguation algorithm that is based on refinement instead of the simpler type derivation.

4 The Algorithm

Our algorithm for efficient ambiguous parsing is based on two components: a parser and a disambiguator. The *parser* component is similar to the homonymous phase of a traditional compiler pipeline in the sense that it actually checks if the input token stream belongs to a given grammar. The main difference with a traditional parser is that this one, instead of returning a set of ASTs, acts lazily immediately returning a pair. The first projection of this pair is a description of the choices which have to be made in order to obtain a *term* (i.e. an unambiguous interpretation of the input). We call such a description an *interpretation domain*. The second projection of the pair returned by the parser is a function that returns a term once applied to an oracle (called *interpretation*) that guides its choices.

The second component of our algorithm, the *disambiguator*, "drives" the parser by feeding with interpretations its output. Interpretations can be totally or partially defined over the interpretation domain. When the parser is faced with a choice that is not considered in a partial interpretation, it generates a placeholder. A partial interpretation approximates a total one since the generated open term matches the closed term of the total interpretation. The disambiguator builds total interpretations by repeated approximations, using the refiner described in Sect. 3 to prune those approximations that lead to not well-typed terms.

The clear distinction between parser and disambiguator ensures that our algorithm is compositional; the efficiency is in turn granted by the refiner which permits to prune interpretations as soon as they lead for sure to not well-typed terms. Let us analyze more formally the ideas sketched so far.

Let $\mathcal{T}$ be the set of well formed terms and let $\mathcal{S}$ be a set of symbols that can have more than one interpretation (that we call *choice*) in the input.

Definition 2 (Interpretation). *Interpretation domain, interpretation codomain and interpretation are mutually defined in the following way:*

- *an* interpretation domain *is a finite set whose elements are couples* (s, l) *where* s *(the symbol to be interpreted) is an element of* $\mathcal{S}$ *and* l *(the list of possible choices) is a list of functions from interpretations to* $\mathcal{T}$*;*
- *an* interpretation codomain *is a set whose elements are either the element "?" (implicit term) or functions from interpretations to* $\mathcal{T}$*;*
- *an* interpretation *is a total function from interpretation domain to interpretation codomain, subject to the following condition: for each element* $i = (s, [t_1 \;;\; \dots \;;\; t_n])$ *of the interpretation domain, if the interpretation applied to* i *returns a function* t*, then* $t \in \{t_1, \dots t_n\}$*.*

An interpretation ϕ *is* partial *when there exists an interpretation domain item* i *such that* $\phi(i) = ?$*.*

In ML notation Def. 2 is:

```
type symbol                           (* abstract data type *)
type term                             (* abstract data type *)
type interpretation_domain_item =
  Item of symbol * (interpretation -> term) list
and interpretation_codomain_item =
    Implicit
  | Term of (interpretation -> term)
and interpretation =
  interpretation_domain_item -> interpretation_codomain_item
```

Intuitively, an interpretation domain item describes all the possible ways to interpret a given symbol. An interpretation codomain item is either an implicit term (refusal of choosing an interpretation for the symbol) or just one interpretation among the possible ones. Thus, an interpretation is just a kind of oracle that, given a set of choices and a set of answers for each choice, associates to each choice either an answer among the one proposed or the "no choice" term.

We let D range over interpretation domains, ϕ range over interpretations and $t_\uparrow$ over functions from interpretations to terms. Let us now define the parser.

Definition 3 (Parser). *A* parser *is a function that maps its input (a token stream) to a couple* $(D, t_\uparrow)$ *where* D *is an interpretation domain and* $t_\uparrow$ *is a function from interpretations defined over* D *to terms.*

Operationally, the parser can be obtained mechanically from a traditional parser for non-ambiguous grammars, either top down, bottom up, or general, by lifting its output from terms to functions, from interpretations to terms. An example in OCamlYacc[1] syntax should clarify the previous statement. Let us consider the OCamlYacc production that parses the formula $t!$ (the factorial of t) and returns an OCaml value representing the resulting term:

```
expr: expr BANG { Bang ($1) }
```

Applying the mechanical lifting transformation, the production becomes:

```
expr:
  expr BANG {
    let dom, mk_expr = $1 in
    dom, fun interp -> Bang (mk_expr interp) }
```

The first component of the output is the domain of the subexpression. The second component is a λ-abstraction over an interpretation, that becomes the argument of the subexpression (that is now lifted and needs an interpretation in input).

The interesting case is the case of ambiguous parsing, i.e. the case when the parser must make a choice. The solution is simply to add all the possible choices to the output domain, and makes the interpretation function perform the choice. Example:

```
expr:
  expr EQ expr {
    let ((dom1, mk_expr1), (dom2, mk_expr2)) = ($1, $3) in
    (union (union dom1 dom2)
      (EQ,
      [ fun interp ->  (* decidable equality over naturals *)
          APPL [eq_nat; mk_expr1 interp; mk_expr2 interp];
        fun interp ->  (* leibniz's equality *)
          APPL [eq; ?; mk_expr1 interp; mk_expr2 interp] ])),
    fun interp ->
      match interp EQ with
      | Implicit -> ?  (* fresh metavariable *)
      | Term mk_term -> mk_term interp }
```

If the oracle `interp` given in input satisfies Def. 2, then `interp EQ` will be either `Implicit` (refusal to choose) or one of:

1. ```
   Term (fun interp ->  (* decidable equality over naturals *)
     APPL [eq_nat; mk_expr1 interp; mk_expr2 interp] )
   ```
2. ```
   Term (fun interp ->  (* leibniz's equality *)
     APPL [eq; IMPLICIT; mk_expr1 interp; mk_expr2 interp] )
   ```

[1] OCamlYacc is a tool, similar to Yacc, that produces bottom up OCaml parsers from LALR(1) grammars.

From the latter example it should be clear in which sense the parsing phase proceeds "lazily" and also in which sense interpretations behave as oracles for the parsers. The last and more interesting component is the disambiguator.

Definition 4 (Disambiguator). *A* disambiguator *is a function whose input is a couple* $(D, t_\uparrow)$ *where* D *is an interpretation domain and* $t_\uparrow$ *a function from interpretations defined over* D *to terms. The output of the disambiguator is the set* $\{t \mid \exists \phi$ *of domain* D *s.t.* $t = t_\uparrow(\phi)$ *and* t *is well-typed*$\}$

All the couples parser, disambiguator are clearly compositional. We will now present an efficient algorithm satisfying the specification for a disambiguator. The idea of the algorithm is to build the set of interpretations that produces well-typed terms by progressive refinements, starting from the interpretation that always refuses to choose and progressing towards an interpretation that always returns some lifted term. At each refinement step, the function $t_\uparrow$ is applied to obtain an open term, that is immediately fed to the refiner. The refiner function acts as a sort of validator, rejecting partial instantiations that already produce ill-typed open terms. In the case of dependent types it may also provide positive information, in the sense that it may automatically instantiate several metavariables, constraining the set of future choices.

The simplest version of our algorithm — that does not exploit refiner positive information — can be sketched in pseudo-code as follows:

$$
\begin{array}{l}
\textbf{let } \mathit{refine}(t) = \text{a refiner according to Def. 1} \\
\textbf{let } \mathit{update}(\phi, s, t'_\uparrow)(s') = \begin{cases} \mathit{Term}(t'_\uparrow) \ \textbf{if } s' = s \\ \quad \phi(s) \quad \textbf{otherwise} \end{cases} \\
\textbf{let } \mathit{disambiguate}(D, t_\uparrow) = \\
\quad \textbf{let } \phi_0 = \lambda x.\mathit{Implicit} \\
\quad \textbf{let } \varPhi = \begin{cases} \{\phi_0\} \ \textbf{if } \mathit{refine}(t_\uparrow(\phi_0)) \neq \epsilon \wedge \mathit{refine}(t_\uparrow(\phi_0)) \neq \bot \\ \quad \emptyset \quad \textbf{otherwise} \end{cases} \\
\quad \textbf{foreach } (s, l) \in D \\
\qquad \varPhi := \{\phi' \mid \phi \in \varPhi \wedge t'_\uparrow \in l \wedge \phi' = \mathit{update}(\phi, s, t'_\uparrow) \wedge \mathit{refine}(t_\uparrow(\phi')) \neq \epsilon\} \\
\quad \textbf{return } \{\phi \mid \phi \in \varPhi \wedge \mathit{refine}(t_\uparrow(\phi)) \neq \bot\}
\end{array}
$$

The following is an example of the application of the algorithm to the formula $(5/2)!$ where $/$ can be either the integer division $/_\mathbb{N}$ or the real division $/_\mathbb{R}$, 2 and 5 can be natural or real numbers and the factorial operator can be applied only to natural numbers. Only the first iterations of the main loop are reported.

$$
D = \begin{cases} / \mapsto [/_\mathbb{N} : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat} \ ; \ /_\mathbb{R} : \mathtt{R} \to \mathtt{R} \to \mathtt{R}] \\ 2 \mapsto [2 : \mathtt{nat} \ ; \ 2 : \mathtt{R}] \\ 5 \mapsto [5 : \mathtt{nat} \ ; \ 5 : \mathtt{R}] \end{cases}
$$

$$
\phi_0 = \begin{cases} / \mapsto \mathit{Implicit} \\ 2 \mapsto \mathit{Implicit} \\ 5 \mapsto \mathit{Implicit} \end{cases} \qquad \begin{array}{l} refine(t_\uparrow(\phi_0)) = \\ \quad refine(?_1!) = \{?_1 : \mathtt{nat}\} \end{array}
$$

$$
\varPhi_0 = \{\phi_0\}
$$

$$
\phi_1 = \begin{cases} / \mapsto /_\mathbb{N} \\ 2 \mapsto \mathit{Implicit} \\ 5 \mapsto \mathit{Implicit} \end{cases} \qquad \begin{array}{l} refine(t_\uparrow(\phi_1)) = \\ \quad refine((?_1 /_\mathbb{N} ?_2)!) = \{?_1 : \mathtt{nat} \ ; \ ?_2 : \mathtt{nat}\} \end{array}
$$

$$\phi'_1 = \begin{cases} / \mapsto /_{\mathbb{R}} \\ 2 \mapsto Implicit \\ 5 \mapsto Implicit \end{cases} \qquad \begin{aligned} &refine(t_\uparrow(\phi_1)) = \\ &\quad refine((?_1/_{\mathbb{R}}?_2)!) = \epsilon \end{aligned}$$

$$\Phi_1 = \{\phi_1\}$$

5 Analysis and future improvements

The algorithm presented in the previous section is surely efficient according to our notion of efficiency since at each iteration of the *for* loop the set of interpretations $\{\phi' \mid \phi \in \Phi \wedge t'_\uparrow \in l \wedge \phi' = update(\phi, s, t'_\uparrow)\}$ is immediately pruned by means of the check $refine(t_\uparrow(\phi')) \neq \epsilon$. Pruning prevents further parsing of the user provided formula (since parsing is a lazy operation) and reduces the number of applications of the semantic analyser (the refiner in our implementation). More formally, we can try to estimate the computational complexity of the algorithm in order to compare it with that of NCA.

Estimating precisely the cost of the parsing phase is very complex since it is interrupted by pruning. Moreover, the overall time spent by the algorithm is dominated by the semantic analysis. Thus we ignore the parsing time and we define the computational cost of the algorithm as a function of the number of calls to the refiner.

Let Φ_i be the value of the Φ variable at the i-th loop of the algorithm. The number of refine operations invoked is $\Sigma_{i=1}^{|D|}|\Phi_i|$.

The worst case of the algorithm is obtained when pruning always fails. Thus $\Sigma_{i=1}^{|D|}|\Phi_i| > |\Phi_D| = \Pi_{(s,l)\in D}|l|$. The latter formula is the number of abstract syntax trees computed by NCA. Thus, in the worst case, our algorithm is more expensive than the NCA. However, the worst case is extremely unlikely when $|D|$ is big, since it corresponds to a term where a lot of ambiguous symbols occur with the following property: each symbol is completely independent of the others and it can be resolved independently from the other choices.

The optimal case of the algorithm is obtained when pruning reduces the set $\{\phi' \mid \phi \in \Phi \wedge t'_\uparrow \in l \wedge \phi' = update(\phi, s, t'_\uparrow)\}$ to a set of cardinality c where c is the number of valid interpretations. Thus $\Sigma_{i=1}^{|D|}|\Phi_i| = \Sigma_{i=1}^{|D|}c = c|D|$. Since c is usually a small value — how many different valid interpretations of a formula usually hold? — the latter expression is smaller than $\Pi_{(s,l)\in D}|l|$ already for small values of $|D|$, and it becomes smaller and smaller when $|D|$ (the number of ambiguous symbols in a formula) increases.

It is now evident that the computational complexity of the algorithm is greatly influenced by the pruning rate: the more invalid partial terms will be detected, the smaller the $|\Phi_i|$, the lower the overall time spent by the algorithm. In particular, to obtain an average performance close to the optimal case we should minimize $|\Phi_i|$ for each i by pruning invalid interpretations as early as possible.

The choice of the strategy used to pick the next element of the domain D in the `foreach` loop of the algorithm greatly influences the pruning rate. Let us

NA	*PA*	*I*	*NA*	*PA*	*I*	*NA*	*PA*	*I*	*NA*	*PA*	*I*	*NA*	*PA*	*I*	*NA*	*PA*	*I*
1	2	-1	32	33	-1	32	33	-1	128	38	90	512	45	467	1680	35	1645
1	4	-3	32	33	-1	42	13	29	128	38	90	512	41	471	1792	51	1741
1	6	-5	32	33	-1	63	83	-20	128	38	90	896	51	845	2688	54	2634
3	5	-2	32	40	-8	63	83	-20	128	38	90	896	51	845	3584	55	3529
4	6	-2	32	39	-7	63	19	44	160	38	122	896	51	845	3584	54	3530
4	7	-3	32	33	-1	96	37	59	192	108	84	896	49	847	7168	63	7105
8	7	1	32	33	-1	128	40	88	192	111	81	896	47	849	8192	60	8132
14	24	-10	32	33	-1	128	43	85	224	46	178	1024	42	982	14336	62	14274
21	13	8	32	33	-1	128	42	86	224	45	179	1280	44	1236	21504	65	21439
32	32	0	32	33	-1	128	42	86	224	47	177	1280	43	1237	21504	60	21444
32	33	-1	32	33	-1	128	39	89	256	40	216	1280	43	1237	36864	79	36785
32	38	-6	32	33	-1	128	38	90	320	20	300	1536	54	1482	53760	65	53695
32	33	-1	32	39	-7	128	38	90	480	41	439	1536	48	1488	53760	67	53693

NA = Trivial compositional algorithm (number of refinements)
PA = Proposed algorithm (number of refinements)
I = Improvement = $NA - PA$

Table 1. Comparison between our algorithm and NCA.

consider the following trivial example: $(f\ (g\ x)\ (h\ y))$ where all the atoms are ambiguous (i.e. f, g, x, y are all elements of the disambiguation domain D).

The strategy that sorts D according to the visit in preorder of the syntax tree refines in succession the terms $(f\ ?_1\ ?_2)$, $(f\ (g\ ?_3)\ ?_4)$, $(f\ (g\ x)\ ?_4)$ and so on. Since the type of a function constraints the type of its arguments, refining the first term already rejects interpretations of f that are not binary, and refining the second term rejects interpretations of g whose output type does not match the type of the first argument of f. Thus at each step several partial interpretations are pruned. If the type of the function constraints the possible interpretations of the operands to just the choices that are prefixes of the final valid interpretations, then we are facing the optimal case of the algorithm.

Any other strategy that consider a subterm without first considering its parents in the abstract syntax tree yields to less pruning. For instance, the term $(f\ (?_5\ x)\ (?_6\ y))$ can be successfully refined for each interpretation of f, x and y such that f is a binary operator. The reason is that the refiner can always attribute to $?_5$ the type $T_1 \to T_2$ where T_1 is the type expected by f and T_2 is the type of x.

The strategy that sorts D by preorder visiting the syntax tree is not always optimal, but it behaved particularly well in all the benchmarks we did on our implementation, exhibiting an average case really close to the optimal one. Table 1 compares the number of invocations of the refiner using our algorithm and using NCA. The table has been obtained parsing all the statements and definitions of all the theorems that deals with real numbers in the standard library of the Coq proof assistant.

As expected our algorithm performs more refinements than NCA when the number of ambiguous symbols — and thus also the number NA of syntax trees — is small. In this case only a few more refinements are performed. When the size of the domain — logarithmic in the number of NA of syntax trees — grows, the number of refinements performed by our algorithm grows only linearly in the size of the domain. Note that this is the expected behaviour for the optimal case of the algorithm when the number of valid interpretations c is fixed. Indeed, only 5 of the 79 statements admit more than one valid interpretation. We conclude that in practice the average case of our algorithm is close to the optimal case.

Notice also that there is a trivial heuristic to predict whether our algorithm is convenient over NCA: it is sufficient to look at the number NA of syntax trees and apply our algorithm whenever NA is higher than a given threshold (32 in our benchmark).

We should further observe that the computational cost of a refinement is not constant, being a function of the term to refine, and it is extremely difficult to approximate. Still, it is surely small for small terms. Thus the calls to the refiner performed by our algorithm are in the average case cheaper than those performed by NCA, since at least half of our calls are on prefixes of the syntax tree.

Moreover, using preorder visit, the computational cost of the computation of $t_\uparrow(\phi)$ and $refine(t_\uparrow(\phi))$ can be lowered. Indeed, at each iteration of the `for` loop, $t_\uparrow$ is applied to an interpretation ϕ' that differs from ϕ only by instantiating more implicit arguments that are leaves of the generated term tree. Thus, the term returned by $t_\uparrow(\phi)$ is a prefix of the term returned by $t_\uparrow(\phi')$. This suggests that the cost of the computation of $t_\uparrow(\phi')$ could be greatly reduced by changing $t_\uparrow$ so that its exploit the knowledge about the partial result $t_\uparrow(\phi)$. Similarly, when $refine(t_\uparrow(\phi))$ is defined and different from $\perp$, its value can easily be exploited in the refinement of $t_\uparrow(\phi')$.

Combining these two optimizations, we can easily make the cost of the two operations at each iteration negligible, still being compositional. These optimizations have not been implemented yet, they are planned as future work.

Another optimization derives from the positive information computed by the refinement function, that is the map that associates a sequent or an instantiation to each metavariable. For instance, if the refine operation assigns to $?_1$ the type $\mathbb{R} \to \mathbb{R}$, an interpretation that instantiates $?_1$ with logical negation can be rejected without even trying to refine the associated term. This corresponds to remembering in Φ also the refinement map associated with each term and to adding a new pruning test over ϕ' based on the unification of the type of the term generated by $t'_\uparrow$ with the type assigned to s in the map. This optimization is also planned as future work.

An interesting topic that was completely skipped so far is the construction of the interpretation domain by the parser. MKM tools provide at least two sources of information that can be exploited to construct the list of possible choices for each symbol. The first one is related to the ambiguities that arise from the *resolution* of an identifier. Indeed, it is not unusual to have several objects in a distributed library that are given the same name (e.g. "reflexive_property", "domain" or simply "P"). Thus to every identifier we can associate several objects. The parser needs to retrieve for each identifier the list of all the objects whose name is equal to the identifier. The task can easily be performed making the parser consult a search engine for mathematical formulae as the one developed in the MoWGLI project [5] or the similar, but less generic one, developed by the Mizar group [2].

The second source of information is represented by XML notational files that describe the mathematical notation associated with definitions in MKM libraries. Indeed, several projects provide XSLT stylesheets to render mathematical formulae encoded in content level markup as MathML Content, OpenMath or OMDoc. Since there exist large classes of operators that share a similar notation, these stylesheets are usually generated from more concise descriptions of symbols arity and precedence levels [9, 4, 6]. These descriptions could be exploited also in the implementations of our disambiguation algorithm.

Finally we should address the case where the disambiguator returns more than one well-typed term. Depending on the kind of processing required on the terms, the system may either proceed in parallel on all the generated terms, or ask the user to choose the term she was interested in. In the latter case the system can identify all the choices that are still ambiguous, and present to the user a human readable description of each choice.

6 Concluding remarks

Our disambiguation algorithm has been implemented and integrated both in the MoWGLI search engine and in the HELM proof assistant prototype [12]. The former is a Web-Service for retrieving lemmas and definitions from the distributed HELM library by matching their type against a user provided pattern. The lemmas and definitions are XML encodings of those in the library of the Coq proof assistant [3]. We developed a Web interface for the search engine that allows users to enter patterns as mathematical formulae with placeholders.

The usual mathematical notation is available and it is disambiguated using our algorithm. In case of multiple interpretations of the formula the search engine can ask the user to identify the only interpretation she is interested in, or it can perform the search according to each interpretation. For instance, it is possible to look for theorems stating $?_1 + ?_2 = ?_2 + ?_1$ retrieving at once the commutative property for the addition over Peano natural numbers, binary positive numbers, integers, rationals and real numbers.

The performance analysis presented in Sect. 5 is confirmed by our implementation: the time spent in the disambiguation of the formula is negligible with respect to the time spent in the search and in the network transmission.

The HELM proof assistant prototype implements an interactive reasoning tool based on the logic of Coq, adopting the HELM distributed library as its own library. One main difference with respect to the Coq system is the size of the context. Whereas in Coq the context corresponds to only a subset of the whole library, in our prototype all the definitions and theorems in the library are in scope. Thus we must face a much higher degree of ambiguity, since several mathematical notions have been redefined several times in the library of Coq and since there exists several formalizations of the same mathematical concept. Another important difference is that Coq behaves as a compiler, whereas our tool is more interactive. Thus every Coq input must have exactly one interpretation, since in case of multiple valid interpretations it is not possible to ask to the user what interpretation she meant. We observed that disambiguation time is negligible with respect to the validation time of a single proof step.

Comparing our solution with that adopted in the forthcoming release of the Coq system (version V8.0) is surely interesting. Whereas in the previous versions of the system the grammar did not allow overloading, in version 8.0 overloading is admitted thanks to the introduction of *notational scopes*. A scope is a region of text where only some parsing rules are active. For instance, there exists a scope where "*" is interpreted as the multiplication between Peano numbers and another one where it is interpreted as the product of two types. A syntactical device is given to the user to change the current scope, even in the middle of a formula. Scopes are associated with the arguments of the constants, so that when an argument of type **Set** is met the scope that handles "*" as a type product is opened.

The reader should notice that associating a scope to a constant argument is weaker than associating a scope to the type expected for an argument. For instance, the identity function id has type $\Pi T : Set.T \rightarrow T$ and $(id\ nat\ 0)$ is a correct application where the type T is instantiated with the type nat of natural numbers, and 0 has the expected type $T = nat$. However, associating the scope of natural numbers notation to the second argument of id independently from the value of the first argument is a mistake. More generally, we observe that scopes behaves as a new kind of types, much in the spirit of those of Grammatical Framework [11]. This new type system is imposed in parallel with the already existent type system of Coq, that is not exploited. On the contrary, our algorithm is based on the refinement operation provided by the underlying logic of Coq.

For sure, one benefit of this duplicate and weaker type system is its generality: since it is independent from the underlying logic, it can be made part of the notation description and it can be reused with several backends. Nevertheless, there are major drawbacks. First of all, as shown in the previous examples, disambiguation is less effective than that based on our technique, since scopes are chosen by exploiting the context only in a minimal way. More explicitly, imposing a weak type system when a stronger one is available is not appealing at all and requires strong motivations that we do not see.

Secondly, there is a problem of consistency between the two type systems: since the notational types are assigned by the user without any consistency check, it may happen that a wrongly assigned notational type prevents the user from inserting valid Coq terms. Adding another layer of consistency checking is both theoretically and practically complex, especially when compared with the simplicity of the algorithm proposed in this paper.

References

1. A. Asperti, F. Guidi, L. Padovani, C. Sacerdoti Coen, I. Schena. Mathematical Knowledge Management in HELM. In Annals of Mathematics and Artificial Intelligence, 38(1): 27–46, May 2003.
2. G. Bancerek, P. Rudnicki. Information Retrieval in MML. In Proceedings of the Second International Conference on Mathematical Knowledge Management, MKM 2003. LNCS, 2594.
3. The Coq proof-assistant, http://coq.inria.fr
4. P. Di Lena. Generazione automatica di stylesheet per notazione matematica. Master thesis, University of Bologna, 2003.
5. F. Guidi, C. Sacerdoti Coen. Querying Distributed Digital Libraries of Mathematics. In Proceedings of Calculemus 2003, 11th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning. Aracne Editrice.
6. M. Kohlhase. OMDoc: An Open Markup Format for Mathematical Documents (Version 1.1). OMDoc technical recommendation.
7. C. McBride. Dependently Typed Functional Programs and their Proofs. Ph.D. thesis, University of Edinburgh, 1999.
8. C. Munoz. A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory. Ph.D. thesis, INRIA, 1997.
9. W. A. Naylor, Stephen Watt. Meta Style Sheets for the Conversion of Mathematical Documents into other Forms. On-Line Proceedings of the First International Conference on Mathematical Knowledge Management, MKM 2001.http://www.emis.de/proceedings/MKM2001/
10. The Mizar proof-assistant, http://mizar.uwb.edu.pl/
11. A. Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism, manuscript made available in September 2002, to appear in Journal of Functional Programming.
12. C. Sacerdoti Coen. Knowledge Management of Formal Mathematics and Interactive Theorem Proving. Ph.D. thesis, University of Bologna, 2004.
13. M. Strecker. Construction and Deduction in Type Theories. Ph.D. thesis, Universität Ulm, 1998.