Spurious Disambiguation Error Detection

Claudio Sacerdoti Coen* and Stefano Zacchiroli*

Department of Computer Science, University of Bologna
sacerdot@cs.unibo.it   zacchiro@cs.unibo.it

Abstract. The disambiguation approach to the input of formulae enables the user to type correct formulae in a terse syntax close to the usual ambiguous mathematical notation. When it comes to incorrect formulae we want to present only errors related to the interpretation meant by the user, hiding errors related to other interpretations (*spurious errors*). We propose a heuristic to recognize spurious errors, which has been integrated with the disambiguation algorithm of [6].

1 Introduction

In [6] we proposed an efficient algorithm for parsing and semantic analysis of ambiguous mathematical formulae. The topic is particularly relevant for the Mathematical Knowledge Management community since every mathematical assistant sooner or later faces the need of letting its user type formulae. When the user is not acquainted with a system or its library—as it happens when using mathematical search engines [1,3,7]—we cannot assume the knowledge of a language other than the usual corpus of ambiguous mathematical notation.

Our algorithm mimics a mathematician behavior of disambiguating a formula by choosing the only possible interpretation that has a meaning in the current context. However when a formula is not correct, every interpretation is "equally" meaningless. Nevertheless, a mathematician seems to be able to understand which interpretation is more likely, spotting the genuine errors in the formula.

Example 1. If f is known to be a real-valued function on vectors, the formula $f(\alpha \cdot x + \beta \cdot y + z) = \alpha \cdot f(x) + \beta \cdot f(y) + z$ is not correct and a mathematician would probably assert that z is not used properly in the right hand side of the equation. Instead, the algorithm of [6] would return several alternative error messages such as: `in "f(`$\alpha \cdot$`x`$\overrightarrow{+} \ldots \overrightarrow{+}$`z)` $= \ldots$`": x is a vector, but is used as a scalar`.

A possible way out is designing a disambiguation algorithm able to rate the possible interpretations so that the one expected by a mathematician ranks first. Also in those cases were several possible interpretations are meaningful, this approach is necessary to choose automatically among them or to ask the

* Partially supported by the Strategic Project "DAMA: Dimostrazione Assistita per la Matematica e l'Apprendimento" of the University of Bologna

user providing a sensible default. In [2] we proposed such an algorithm that was designed to tackle the case of correct formulae with multiple interpretations. In this paper we address the case of formulae for which no correct interpretation can be found.

Consider again Example 1. We need to find a criterion to identify the given error message as spurious, i.e. as an error relative to an interpretation that is not the one expected by the user. Note that a formula can contain more than one genuine error: they are all the errors in the expected interpretation of the formula. The heuristic criterion we propose is the following.

Criterion 1 (Spurious error detection). *An error is* spurious *when it is localized in a sub-formula F such that there is an alternative interpretation of the* whole *formula such that no error is localized in F.*

Intuitively an error is spurious when no genuine error is spatially co-located with it, i.e. genuine errors are to be found elsewhere. In Example 1 if we interpret all the operators in the left hand side as operations on vectors we do not obtain any error message in the left hand side. Hence the genuine error must be in the right hand side.

The main goal of this paper is the integration of spurious error detection in the efficient algorithm proposed in [6]. We proceed as follows. In Section 2 we formalize the specification of the class of disambiguation algorithms. In Section 3 we provide an improved description of the algorithm proposed in [6], proving that it is a member of the disambiguation algorithm class, while in Section 4 we extend the algorithm with spurious error detection.

2 Disambiguation Algorithm Specification

Traditionally semantic analysis maps an abstract syntax tree (AST) of a formula to a term—its semantics—in some calculus. In an ambiguous setting, semantic analysis rather maps an AST to *a set of* terms; the set can then be rated according to some criterion to identify the best semantics. To represent in a concise way a set of terms sharing a common structure, we use a term containing non linear placeholders in the spirit of [4,5]. We say that a term t' is an *instantiation* of t if it is obtained filling zero or more of its placeholders. For instance $?_1 = ?_2 + ?_2$ represents the set of terms $\{t_1 = t_2 + t_2 \mid t_1, t_2 \text{ terms}\}$; $?_1 = 0 + 0$ and $0 = 0 + 0$ are two instances belonging to that set.

Lemma 1. *If t_1 is an instance of t_2 then the set of instances of t_1 is a subset of the set of instances of t_2.*

Proof. By definition of instantiation. □

Among all the terms that are semantics of a given AST, we are interested only in those that are well-typed. Thus, we are interested in terms with placeholders only when they denote non-empty sets of well-typed instantiations. We assume the existence of a *refiner* $\mathcal{R}(\cdot)$, that is a function from terms to outcomes. An *outcome* is either the distinguished symbol ✓ or an informative error message. The latter is returned when the set of well-typed instantiations of the input term is (known to be) empty. For instance $\mathcal{R}(f(?_1) = 1) = ✓$ whereas $\mathcal{R}(f(?_1) = f + 1) =$ `"f is a function, but is used as a scalar"`. In the latter case the error message is relevant to every possible instantiation; in the former there is no guarantee that every possible instantiation is well-typed. Still, the following lemma holds.

Lemma 2. *A term t without placeholders is well-typed iff $\mathcal{R}(t) = ✓$*

Proof. t is the only instance of itself thus, by definition of $\mathcal{R}(\cdot)$, $\mathcal{R}(t) \neq ✓$ iff t is not well-typed. □

We are now ready to describe the specification of a disambiguation algorithm for an AST t. Let $Dom(t)$ be the set of occurrences of overloaded symbols in t. For each $s \in Dom(t)$, let $\mathcal{D}_s$ be the set of possible choices for s.

An *interpretation* ϕ for t is a partial function $Dom(t) \ni s \mapsto u_s \in \mathcal{D}_s$. Intuitively a (partial) interpretation restricts the set of semantics of t resolving the overloading for the occurrences in the domain of ϕ. When an interpretation is a total function a unique semantics is determined. To formalize this intuition we associate to a partial interpretation ϕ a term with placeholders $[\![t]\!]_\phi$, where all (applications of) occurrences of symbols not in the domain of ϕ have been interpreted as fresh placeholders. For instance, when $\phi = [+_1 \mapsto \textit{point-wise sum}]$, $[\![\texttt{(f+g)(x)=f(x)+g(x)}]\!]_\phi$ denotes $(f+g)(x) = ?_1$. Note that the arguments of the second occurrence of plus have been omitted.

We denote with Φ_t the set of all (partial) interpretations for t and with $\hat{\Phi}_t$ the set of all total interpretations. We call $\bot$ the function everywhere undefined and we denote as $\phi[s \mapsto u]$ the function that maps s to u and behaves as ϕ elsewhere. The set of interpretations is ordered by the usual order on partial functions: $\phi_1 \sqsubseteq \phi_2$ iff $\forall s, \phi_1(s) = u \Rightarrow \phi_2(s) = u$. The minimum of Φ according to $\sqsubseteq$ is $\bot$.

Lemma 3. *$\phi_1 \sqsubseteq \phi_2$ iff $[\![t]\!]_{\phi_2}$ is an instance of $[\![t]\!]_{\phi_1}$.*

Proof. By structural induction on t and by cases on the definition of $[\![\cdot]\!]$. Since, for the sake of brevity, we omitted its definition, the present lemma can be seen as a required property of $[\![\cdot]\!]$. □

Together with Lemma 1, Lemma 3 confirms the intuition that the more overloading is resolved, the smaller the set of semantics.

A disambiguation algorithm partitions the set of semantics of an AST into classes of well-typed terms and classes of terms characterized by the same typing error. Since Lemma 2 holds only for placeholder-free terms, all terms in the well-typed class must have no placeholders. We will use the notion of cover to grasp partitions at the interpretation level, and the notion of typing cover to grasp well-typedness.

We say that a set of interpretations S *covers* a set of interpretations T, written $S \rhd T$, when $\forall \phi \in T, \exists! \phi' \in S, \phi' \sqsubseteq \phi$.

Lemma 4. *If $S \rhd T$ then for each $\phi_1 \in T$ there exists an unique $\phi_2 \in S$ such that $\llbracket t \rrbracket_{\phi_1}$ is an instance of $\llbracket t \rrbracket_{\phi_2}$.*

Proof. By Lemma 3 and the definition of cover. □

Corollary 1. *If $S \rhd \hat{\Phi}_t$ and $\phi_1, \phi_2 \in S, \phi_1 \neq \phi_2$ then the set of instances of $\llbracket t \rrbracket_{\phi_1}$ is disjoint from the set of instances of $\llbracket t \rrbracket_{\phi_2}$.*

Proof. Suppose per absurdum that u is an instance of both $\llbracket t \rrbracket_{\phi_1}$ and $\llbracket t \rrbracket_{\phi_2}$. Let $u' \in \hat{\Phi}_t$ be an instance of u. By Lemma 4 $\phi_1 = \phi_2$, but by hypothesis we know $\phi_1 \neq \phi_2$. □

Theorem 1. *$S \rhd \hat{\Phi}_t$ iff $\{\{u \mid u$ is an instance of $\llbracket t \rrbracket_{\phi}\} \mid \phi \in S\}$ is a partition of $\{u \mid \exists \phi \in \hat{\Phi}_t, u = \llbracket t \rrbracket_{\phi}\}$ (i.e. the set of all semantics of t).*

Proof. The forward implication is by Lemma 4 and Corollary 1. For the converse implication consider an arbitrary but fixed $\phi \in \hat{\Phi}_t$. By hypothesis there is a unique $\phi' \in S$ such that $u = \llbracket t \rrbracket_{\phi}$ is an instance of $\llbracket t \rrbracket_{\phi'}$. Thus $S \rhd \hat{\Phi}_t$. □

We say that a set of interpretations A' is a *refinement* of a set of interpretations A, written $A \Diamond\!\!\rangle A'$ when $A \rhd A'$ and for all $u \in \hat{\Phi}_t$ such that there is a $\phi \in A$ such that u is an instance of $\llbracket t \rrbracket_{\phi}$ there exists a unique $\phi' \in A'$ such that u is an instance of $\llbracket t \rrbracket_{\phi'}$.

Theorem 2. *If $A \cap B = \emptyset$, $A \cup B \rhd \hat{\Phi}_t$ and $A \Diamond\!\!\rangle A'$, then $A' \cup B \rhd \hat{\Phi}_t$.*

Proof. By Theorem 1 $\{\{u \mid u$ is an instance of $\llbracket t \rrbracket_{\phi}\} \mid \phi \in A \cup B\}$ partitions the set of all semantics of t. $\{\{u \mid u$ is an instance of $\llbracket t \rrbracket_{\phi}\} \mid \phi \in A' \cup B\}$ partitions the same set by definition of $A \Diamond\!\!\rangle A'$, where the requirement $A \rhd A'$ is fundamental to avoid interference with B. Hence the thesis by Theorem 1. □

A set S of interpretations is said to be *typing* when for all $\phi \in S$ if $\mathcal{R}(\llbracket t \rrbracket_{\phi}) = \checkmark$ then $\phi \in \hat{\Phi}_t$. In particular a *typing cover* is a cover $S \rhd \hat{\Phi}_t$ that is also typing. Intuitively a disambiguation algorithm returns a typing cover equipped with rating information for its interpretations (that will be called classification).

Theorem 3. *For each typing cover S and for each term u in the set of all semantics of t, u is well-typed iff $\mathcal{R}(\llbracket t \rrbracket_{\phi}) = \checkmark$ where ϕ is the only interpretation in S such that u is an instance of $\llbracket t \rrbracket_{\phi}$.*

Proof. If $\mathcal{R}(\llbracket t \rrbracket_{\phi}) \neq \checkmark$ by definition of $\mathcal{R}(\cdot)$. Otherwise by Lemma 2 and definition of typing cover. □

We also expect something more that cannot be grasped formally: if u is not well-typed then the error message for $\mathcal{R}(\llbracket t \rrbracket_{\phi})$ should also be relevant for u. This property is inherited from the refiner.

Lemma 5. $\{\bot\} \rhd \hat{\Phi}_t$. *It is typing iff $\mathcal{R}(\llbracket t \rrbracket_{\bot}) \neq \checkmark$ or $Dom(t) = \emptyset$.*

Proof. Trivial by definition of $\hat{\Phi}_t$ and $\mathcal{R}(\cdot)$. □

To rate covers, we assume that to each interpretation ϕ is associated a rate $\rho(\phi)$. A rate is an element of a partially ordered set $(A, \preceq)$, such that $\rho(\phi_1) \preceq \rho(\phi_2)$ iff $[\![t]\!]_{\phi_1}$ is more likely to be the intended meaning of t than $[\![t]\!]_{\phi_2}$.

Formally, a *disambiguation algorithm* takes as input an AST t and returns a typing and covering classification Σ. A *classification* Σ is a set of tuples $\langle \phi, o, r \rangle$ such that:

1. for all $\langle \phi, o, r \rangle \in \Sigma$, $o = \mathcal{R}([\![t]\!]_\phi)$, and r belongs to some partially ordered set $(B, \preceq)$;
2. for all $\langle \phi_1, o_1, r_1 \rangle, \langle \phi_2, o_2, r_2 \rangle \in \Sigma$, if $\phi_1 = \phi_2$ then $o_1 = o_2$ and $r_1 = r_2$.

A classification Σ is a *covering classification* if $S_\Sigma = \{\phi \mid \langle \phi, o, r \rangle \in \Sigma\}$ is a cover; it is a *typing classification* when S_Σ is typing.

We choose for B the set {▁, ▄, █} $\times A$ ordered lexicographically by the orders: ▁ $\leq$ ▄ $\leq$ █ and $\preceq$.

Every classification can be partitioned into the set of (so far) successful and the set of failing interpretations as follows:

$$(\Sigma)^{\checkmark} = \{\langle \phi, o, r \rangle \in \Sigma \mid o = \checkmark\}$$
$$(\Sigma)^{✗} = \Sigma \setminus (\Sigma)^{\checkmark}$$

Example 2 (Naive Disambiguation Algorithm). The *naive disambiguation algorithm* (NDA for short) is the disambiguation algorithm that, when applied to an AST t, computes the typing and covering classification $\Sigma = \{\langle \phi, o, r \rangle \mid \phi \in \hat{\Phi}_t,\ o = \mathcal{R}([\![t]\!]_\phi),\ r = \rho'(o, \phi)\}$ where:

$$\rho'(o, \phi) = \begin{cases} \langle █, \rho(\phi) \rangle & \text{if } o = \checkmark \\ \langle ▄, \rho(\phi) \rangle & \text{otherwise} \end{cases}$$

The rating function $\rho'(\cdot, \cdot)$ gives priority to successes over failures; outcomes being equal, it falls back to the interpretation rating.

We call this algorithm "naive" since its computes the typing cover $S_\Sigma = \hat{\Phi}_t \rhd \hat{\Phi}_t$ of maximum cardinality. Its execution is computationally expensive since it invokes the refiner $|S_\Sigma| = |\hat{\Phi}_t| = \prod_{s \in Dom(t)} |\mathcal{D}_s|$ times.

Example 3 (NDA execution). Consider the (non-typable) AST corresponding to $\mathtt{f}(\alpha \cdot \mathtt{x} + \beta \cdot \mathtt{y} + \mathtt{z}) = \alpha \cdot \mathtt{f}(\mathtt{x}) + \beta \cdot \mathtt{f}(\mathtt{y}) + \mathtt{z}$, where `+` is left-associative, $\mathtt{x}, \mathtt{y}, \mathtt{z}$ are globally declared as real vectors, α, β are reals, and `f` is a real-valued function on vectors. The symbol "+" is overloaded on scalar and vector sums; "$\cdot$" is overloaded on scalar and external products.

NDA returns a classification consisting of 2^8 error messages (not necessarily unique), where 2 are the possible choices for each occurrence of overload symbols and 8 is the number of occurrences of "$\cdot$" and "+". The "expected" error message `"z is a vector, but is used as a scalar"` is drowned in a sea of errors like (re-ordered here for reader's sake):

- "x is a vector, but is used as a scalar"
- "y is a vector, but is used as a scalar"
- "z is a vector, but is used as a scalar"
- "$\alpha \cdot$ x is a vector, but is used as a scalar"
- "$\beta \cdot$ y is a vector, but is used as a scalar"
- "$\alpha \cdot$ x $+$ $\beta \cdot$ y is a vector, but is used as a scalar"
- ...
- "f(x) is a scalar, but is here used as a vector"
- "f(y) is a scalar, but is here used as a vector"
- ...

We can only hope that $\rho(\cdot)$ does a great job ranking first the expected interpretation. In practice we are not aware of any rating function that performs well looking only at the interpretations.

3 An Efficient Disambiguation Algorithm

In terms of efficiency we can do better than NDA. The key observation for improvement is that a single invocation of the refiner on a term with placeholders can rule out the whole set of its instances. More precisely, if the refinement of such a term fails, all of its instances are not well-typed (and will fail in the same way). Thus, it is not necessary to compute the largest typing and covering classification as NDA does: intuitively, the smaller the classification, the more efficient the algorithm.

A typing and covering classification can be built incrementally starting from a covering classification. Indeed if a covering classification Σ is not typing it must contain a partial interpretation $\phi \in S_{(\Sigma)^{\checkmark}}$. A more precise classification can be obtained replacing the interpretation ϕ with a set of more instantiated interpretations S such that $S \rhd \{\phi\}$. Since $\phi_1 \sqsubseteq \phi$ for each $\phi_i \in S$, the domain of ϕ_1 (a subset of $Dom(t)$) is bigger than the domain of ϕ. Thus the refinement process ends in a finite number of steps since $Dom(t)$ is finite; moreover it yields a typing classification.

To increase efficiency, we can enforce the invariant that all interpretations $\phi \in S_{(\Sigma)^{\checkmark}}$ share a common domain. Thus at each step we have to extend at once the domain shared by all ϕs. Let Σ be a classification such that the interpretations in S_Σ are defined on the same domain and let $s \in Dom(t)$. We define:

$$\Sigma_s = \{\langle \phi, o, r \rangle \mid \exists \phi' \in S_\Sigma, \exists u \in \mathcal{D}_s, \phi = \phi'[s \mapsto u], o = \mathcal{R}([\![t]\!]_\phi), r = \rho'(o, \phi)\}$$

Lemma 6. *Let Σ be a classification such that the interpretations in S_Σ are defined on the same domain and let $s \in Dom(t)$. $\Sigma \mathrel{\Diamond\!\!\!>} \Sigma_s$.*

Proof. By construction of Σ_s and definition of $\mathrel{\Diamond\!\!\!>}$. □

The refinement process outlined above can now be formally described. At the n-th step we have the covering (not typing) classification Σ_n. Choosing s outside the domain of the ϕs in $S_{(\Sigma_n)^{\checkmark}}$, we obtain the next covering classification

$\Sigma_{n+1} = ((\Sigma_n)^{\checkmark})_s \cup (\Sigma_n)^{\text{✗}}$. Since the functions in $S_{(\Sigma_{n+1})^{\checkmark}}$ are more defined that those in $S_{(\Sigma_n)^{\checkmark}}$ the most natural choice for the initial covering classification is $\Sigma_0 = \{\langle \bot, o, r\rangle \mid o = \mathcal{R}([\![t]\!]_\bot), r = \rho'(o, \bot)\rangle\}$.

Example 4 (Refinement process). Consider the AST of Example 2. Picking occurrences $s \in \mathit{Dom}(t)$ according to the pre-visit order of the AST, the first steps of the refinement process yield the following covering classifications (where for the sake of brevity errors have been substituted by ✗):

$$\begin{array}{llll}
\Sigma_0 = \{\langle \phi_1, \checkmark, \langle \text{🌡}, \rho(\phi_1)\rangle\rangle\} & \text{where } [\![t]\!]_{\phi_1} & = f(?_1) = ?_2 \text{ and } \phi_1 = \bot \\
\Sigma_1 = \{\langle \phi_{11}, \checkmark, \langle \text{🌡}, \rho(\phi_{11})\rangle\rangle, & [\![t]\!]_{\phi_{11}} & = f(?_1 \overrightarrow{+} z) = ?_2 \\
\qquad \langle \phi_{12}, \text{✗}, \langle \text{🌡}, \rho(\phi_{12})\rangle\rangle\} & [\![t]\!]_{\phi_{12}} & = f(?_1 + z) = ?_2 \\
\Sigma_2 = \{\langle \phi_{111}, \checkmark, \langle \text{🌡}, \rho(\phi_{111})\rangle\rangle, & [\![t]\!]_{\phi_{111}} & = f(?_1 \overrightarrow{+} ?_2 \overrightarrow{+} z) = ?_3 \\
\qquad \langle \phi_{112}, \text{✗}, \langle \text{🌡}, \rho(\phi_{112})\rangle\rangle, & [\![t]\!]_{\phi_{112}} & = f(?_1 + ?_2 \overrightarrow{+} z) = ?_3 \\
\qquad \langle \phi_{12}, \text{✗}, \langle \text{🌡}, \rho(\phi_{12})\rangle\rangle\} & [\![t]\!]_{\phi_{12}} & = f(?_1 + z) = ?_2 \\
\Sigma_3 = \{\langle \phi_{1111}, \checkmark, \langle \text{🌡}, \rho(\phi_{1111})\rangle\rangle, & [\![t]\!]_{\phi_{1111}} & = f(\alpha \overrightarrow{\cdot} x \overrightarrow{+} ?_1 \overrightarrow{+} z) = ?_2 \\
\qquad \langle \phi_{1112}, \text{✗}, \langle \text{🌡}, \rho(\phi_{1112})\rangle\rangle, & [\![t]\!]_{\phi_{1112}} & = f(\alpha \cdot x \overrightarrow{+} ?_1 \overrightarrow{+} z) = ?_2 \\
\qquad \langle \phi_{112}, \text{✗}, \langle \text{🌡}, \rho(\phi_{112})\rangle\rangle, & [\![t]\!]_{\phi_{112}} & = f(?_1 + ?_2 \overrightarrow{+} z) = ?_3 \\
\qquad \langle \phi_{12}, \text{✗}, \langle \text{🌡}, \rho(\phi_{12})\rangle\rangle\} & [\![t]\!]_{\phi_{12}} & = f(?_1 + z) = ?_2 \\
\ldots
\end{array}$$

Theorem 4 (Correctness of the Refinement Process). *The above refinement process implements a disambiguation algorithm, i.e. for each AST t, $\Sigma_{|\mathit{Dom}(t)|}$ is a covering and typing classification.*

Proof. By induction on $|\mathit{Dom}(t)|$ we prove that $\Sigma_{|\mathit{Dom}(t)|}$ is covering.
Base case. By Lemma 5 Σ_0 is a covering classification.
Inductive case. Let Σ_n be a covering classification per inductive hypothesis. By definition $\Sigma_{n+1} = ((\Sigma_n)^{\checkmark})_s \cup (\Sigma_n)^{\text{✗}}$. By Theorem 2 and Lemma 6, Σ_{n+1} is covering.

To prove that $\Sigma_{|\mathit{Dom}(t)|}$ is typing the reader can prove by induction that all the ϕs in $S_{(\Sigma_n)^{\checkmark}}$ are defined on a subset of $\mathit{Dom}(t)$ of cardinality n. The thesis follows trivially. □

The above refinement process is parametric in how the next symbol $s \in \mathit{Dom}(t)$ is chosen at each step. In [6] we discussed the implication of such a choice on the computational complexity in terms of numbers of refiner invocations. The best choice corresponds to a pre-visit of the abstract syntax tree t.

We now present the *efficient disambiguation algorithm* (EDA for short) of [6]. It proceeds by recursion on $\mathit{Dom}^{\mathtt{list}}(t)$, which is the list of overloaded symbol occurrences in t obtained in a pre-visit traversal.

$$f(\Sigma, l) = \begin{cases} \Sigma & \text{if } l = [] \\ f((\Sigma_s)^{\checkmark}, tl) \cup (\Sigma_s)^{\text{✗}} & \text{if } l = s :: tl \end{cases}$$

$$\mathit{EDA}(t) = f((\Sigma_0)^{\checkmark}, \mathit{Dom}^{\mathtt{list}}(t)) \cup (\Sigma_0)^{\text{✗}}$$

Theorem 5 (Correctness of EDA). *EDA implements a disambiguation algorithm.*

Proof. By Theorem 4 it is sufficient to prove that the classification returned by EDA is the same returned by the refinement process. We observe that

$$\begin{aligned}
\Sigma_n &= ((\Sigma_{n-1})^{\checkmark})_{s_n} \cup (\Sigma_{n-1})^{\times} \\
&= ((((\Sigma_{n-2})^{\checkmark})_{s_{n-1}} \cup (\Sigma_{n-2})^{\times})^{\checkmark})_{s_n} \cup (((\Sigma_{n-2})^{\checkmark})_{s_{n-1}} \cup (\Sigma_{n-2})^{\times})^{\times} \\
&= ((((\Sigma_{n-2})^{\checkmark})_{s_{n-1}})^{\checkmark})_{s_n} \cup (((\Sigma_{n-2})^{\checkmark})_{s_{n-1}})^{\times} \cup (\Sigma_{n-2})^{\times} && (\dagger) \\
&= (((((\Sigma_{n-2})^{\checkmark})_{s_{n-1}})^{\checkmark})_{s_n})^{\checkmark} \cup \\
&\quad\; (((((\Sigma_{n-2})^{\checkmark})_{s_{n-1}})^{\checkmark})_{s_n})^{\times} \cup (((\Sigma_{n-2})^{\checkmark})_{s_{n-1}})^{\times} \cup (\Sigma_{n-2})^{\times} \\
&= \ldots \\
&= ((\cdots(((((\Sigma_0)^{\checkmark})_{s_1})^{\checkmark})_{s_2})^{\checkmark}\cdots)_{s_n})^{\checkmark} \cup && (\ddagger) \\
&\quad\; ((\cdots(((((\Sigma_0)^{\checkmark})_{s_1})^{\checkmark})_{s_2})^{\checkmark}\cdots)_{s_n})^{\times} \cup \cdots \cup (((\Sigma_0)^{\checkmark})_{s_1})^{\times} \cup (\Sigma_0)^{\times}
\end{aligned}$$

where (†) is justified by the two identities $((\Sigma)^{\times})^{\checkmark} = \emptyset$ and $((\Sigma)^{\times})^{\times} = (\Sigma)^{\times}$. The reader can verify that the pseudo-code of EDA is a recursive formulation of (‡) for $n = |Dom(t)|$. □

Example 5 (EDA execution). Consider the AST of Example 2. EDA yields a smaller classification, containing "just" 6 error messages:

1. "in $\mathtt{f}(?_1 + \mathtt{z}) = ?_2$: z is a vector, but is used as a scalar"
2. "in $\mathtt{f}(?_1 + ?_2 \overrightarrow{+} \mathtt{z}) = ?_3$: $?_1 + ?_2$ is a scalar, but is used as a vector"
3. "in $\mathtt{f}(\alpha \cdot \mathtt{x} \overrightarrow{+} ?_1 \overrightarrow{+} \mathtt{z}) = ?_2$: x is a vector, but is used as a scalar"
4. "in $\mathtt{f}(\alpha \overrightarrow{\cdot} \mathtt{x} \overrightarrow{+} \beta \cdot \mathtt{y} \overrightarrow{+} \mathtt{z}) = ?_1$: y is a vector, but is used as a scalar"
5. "in $\mathtt{f}(\alpha \overrightarrow{\cdot} \mathtt{x} \overrightarrow{+} \beta \overrightarrow{\cdot} \mathtt{y} \overrightarrow{+} \mathtt{z}) = ?_1 + \mathtt{z}$: z is a vector, but is used as a scalar"
6. "in $\mathtt{f}(\alpha \overrightarrow{\cdot} \mathtt{x} \overrightarrow{+} \beta \overrightarrow{\cdot} \mathtt{y} \overrightarrow{+} \mathtt{z}) = ?_1 \overrightarrow{+} \mathtt{z}$: $?_1 + \mathtt{z}$ is a vector, but is used as a scalar"

where (5) is the expected one, while the other errors are spurious. The rating of errors is unchanged with respect to Example 2.

4 A Humane Disambiguation Algorithm

We look for a restriction of Criterion 1 which can be integrated in EDA. The characteristic of EDA (with respect to the general refinement process) is the pre-visit ordering of $Dom(t)$. This implies that:

a. to interpret an occurrence s, every occurrence s' that precedes s in pre-order must be interpreted too;
b. when an interpretation ϕ yields an error, every occurrence s' that follows in pre-order the last occurrence s added to the domain of ϕ will not be interpreted by any interpretation $\phi' \sqsupseteq \phi$.

Together, (a) and (b) imply that not every sub-formula F will be interpreted in any possible way. Actually, (b) is a consequence of (a). This imposes a non negligible restriction of Criterion 1 for efficiency reasons, yielding:

Criterion 2 (Efficient spurious error detection). *An error message relative to an interpretation ϕ of an AST t is* spurious *iff there exists an occurrence $s \in Dom(t)$ and an interpretation ϕ' such that:*

1. $\phi(s) \neq \phi'(s)$;
2. $\phi'(s') = \phi(s')$ *for all s' that precedes s in pre-order;*
3. *ϕ' is total on the occurrences of overloaded symbols occurring in the sub-tree rooted at s;*
4. $\mathcal{R}([\![t]\!]_{\phi'}) = ✓$.

Dropping (2)—imposed by (a)—from the conditions above we obtain a more formal writing of Criterion 1. We now address the issue of integrating Criterion 2 in EDA.

$f(\Sigma, l)$, the core of EDA, does not work directly on t, but rather on the list l, which is an abstraction of the occurrences of overload symbols in t. In l the tree-structure of t has been lost. As a consequence, without changing its input, we cannot make f recognize spurious errors using Criterion 2. As a solution we could make f work by recursion on t by integrating in f a pre-visit traversal. Still, we prefer to avoid binding f to the data type of AST of formulae and to keep separate the construction of $Dom(t)$ from the actual disambiguation.

Therefore we introduce the new $Dom^{\mathtt{tree}}(t)$ datatype which is a tree representation of $Dom(t)$. $Dom^{\mathtt{tree}}(t)$ is a tree which contains only the nodes $s \in Dom(t)$ and preserves the ancestor-descendent relation of t. As a concrete representation of $Dom^{\mathtt{tree}}(t)$ we adopt the well-known first-child/next-sibling representation. This representation allows to implement straightforwardly a pre-visit of the tree recognizing when all children of a given node have been traversed. Note that the pre-visit order is imposed by the efficiency analysis given in [6] and recognizing the end of children traversal is necessary for Criterion 2.

We call the algorithm that recognizes spurious errors the *humane disambiguation algorithm* (HDA for short). It proceeds by recursion on $Dom^{\mathtt{tree}}(t)$ and, at the end of children traversal, lowers the rate of spurious errors. The pseudo code of HDA is given below:

$$g(\Sigma, t) = \begin{cases} \Sigma & \text{if } t = nil \\ g((\Sigma_1)^{✓}, b) \cup p((\Sigma_1)^{✓}, (\Sigma_1)^{✗} \cup (\Sigma_s)^{✗}) & \text{if } t = \begin{array}{l} s \rightarrow b \\ \downarrow \\ c \end{array} \\ \qquad\qquad\qquad \text{where } \Sigma_1 = g((\Sigma_s)^{✓}, c) & \end{cases}$$

$$p(\Sigma_{ok}, \Sigma_{err}) = \begin{cases} \Sigma_{err} & \text{if } \Sigma_{ok} = \emptyset \\ \{\langle \phi, o, r\rangle \mid \langle \phi, o, \langle m, p\rangle\rangle \in \Sigma_{err}, r = \langle 🌡, p\rangle\} & \text{if } \Sigma_{ok} \neq \emptyset \end{cases}$$

$$\begin{aligned} HDA(t) &= (\Sigma')^{✓} \cup p((\Sigma')^{✓}, (\Sigma')^{✗} \cup (\Sigma_0)^{✗}) \\ &\quad \text{where } \Sigma' = g((\Sigma_0)^{✓}, Dom^{\mathtt{tree}}(t)) \end{aligned}$$

g has the same role f had in EDA, while $p(\cdot, \cdot)$ (mnemonic for "prioritize") lowers the rate of spurious errors to 🌡, which is the lowest rating.

Theorem 6 (Correctness of HDA).

1. *HDA implements a disambiguation algorithm.*
2. *An error in a classification returned by HDA is spurious according to Criterion 2 iff it is rated* $\langle \text{🌡}, \rho(\phi)\rangle$.

Proof. We just give a sketch of the proof, which is involved due to the complexity of the code.

(1) By Theorem 5 it is sufficient to prove that the classification returned by HDA is equal to the classification returned by EDA up to rates. Since both algorithms perform a pre-visit of the input tree, we can consider "parallel" executions of them. At the nth step EDA is called on the list $s_n :: tl$ while HDA is called on the tree $\begin{smallmatrix} s_n \longrightarrow b \\ \downarrow \\ c \end{smallmatrix}$. The nodes that EDA will encounter processing tl are the same (and in the same order) of those HDA will encounter processing c at first and then b. The thesis is reduced to a proof by induction on the length of tl that $f((\Sigma_{s_n})^{\checkmark}, tl)$ is equal to $(g((\Sigma_{s_n})^{\checkmark}, c))^{\times} \cup g(g((\Sigma_{s_n})^{\checkmark}, c)^{\checkmark}, b)$ up to rates.

(2) Recursion is never performed on elements of the current classification corresponding to errors. Thus once an error has been down-rated by $p(\cdot,\cdot)$ its rating will never be raised again.

Suppose that at a given iteration $p(\cdot,\cdot)$ lowers the rating of an error ϵ relative to an interpretation $\phi \in (\Sigma_s)^{\times} \cup (g((\Sigma_s)^{\checkmark}, c))^{\times}$. We interpret that as ϵ being located in $\begin{smallmatrix} s \\ \downarrow \\ c \end{smallmatrix}$. The set $\tilde{S} = S_{(g((\Sigma_s)^{\checkmark}, c))^{\checkmark}}$ is not empty since ϵ has been down-rated.

We consider now two cases: either there exists $\phi' \in \tilde{S}$ such that $\phi(s) \neq \phi'(s)$ or not. In the former case s and ϕ' satisfy all the requirements of Criterion 2. In the latter case let $\phi' \in \tilde{S}$. Let $s' \in c$ be the last occurrence that follows s in pre-order such that $\phi(s') \neq \phi'(s')$. Consider now the recursive call on $\begin{smallmatrix} s' \longrightarrow b' \\ \downarrow \\ c' \end{smallmatrix}$ and iterate the above reasoning. Since this time $\phi(s') \neq \phi'(s')$, ϵ is now properly down-rated according to Criterion 2. When the recursive call on c returns ϵ is still correctly down-rated and $p(\cdot,\cdot)$ leaves its rate unchanged. □

Example 6 (HDA execution). Consider again the AST of Examples 2 and 5. The first recursive invocation is $g(\Sigma, \tau)$ where: $\Sigma = \{\langle \bot, \checkmark, \langle \text{🌡}, \rho(\bot)\rangle\rangle\}$ and $\tau = \begin{smallmatrix} + \longrightarrow b \\ \downarrow \\ c \end{smallmatrix}$. g computes

$$\begin{aligned} \Sigma_s = \{&\langle \phi_{11}, \checkmark, \langle \text{🌡}, \rho(\phi_{11})\rangle\rangle, & \text{where } \llbracket t \rrbracket_{\phi_{11}} &= f(?_1 \overrightarrow{+}\, z) = ?_2 \\ &\langle \phi_{12}, \times, \langle \text{🌡}, \rho(\phi_{12})\rangle\rangle\} & \llbracket t \rrbracket_{\phi_{12}} &= f(?_1 + z) = ?_2 \end{aligned}$$

and then calls itself recursively on $(\Sigma_s)^{\checkmark}$ and c yielding

$$\begin{aligned} \Sigma_1 = \{&\langle \phi_{11111}, \checkmark, \langle \text{🌡}, \rho(\phi_{11111})\rangle\rangle, & \text{where } \llbracket t \rrbracket_{\phi_{11111}} &= f(\alpha \overrightarrow{\cdot} x \overrightarrow{+} \beta \overrightarrow{\cdot} y \overrightarrow{+} z) = ?_1 \\ &\langle \phi_{11112}, \times, \langle \text{🌡}, \rho(\phi_{11112})\rangle\rangle, & \llbracket t \rrbracket_{\phi_{11112}} &= f(\alpha \overrightarrow{\cdot} x \overrightarrow{+} \beta \cdot y \overrightarrow{+} z) = ?_1 \\ &\langle \phi_{1112}, \times, \langle \text{🌡}, \rho(\phi_{1112})\rangle\rangle, & \llbracket t \rrbracket_{\phi_{1112}} &= f(\alpha \cdot x \overrightarrow{+} ?_1 \overrightarrow{+} z) = ?_2 \\ &\langle \phi_{112}, \times, \langle \text{🌡}, \rho(\phi_{112})\rangle\rangle\} & \llbracket t \rrbracket_{\phi_{112}} &= f(?_1 + ?_2 \overrightarrow{+} z) = ?_3 \end{aligned}$$

Since $(\Sigma_1)^{\checkmark}$ is not empty, all the errors in $(\Sigma_s)^{\times}$ and $(\Sigma_1)^{\times}$ are recognized as spurious and their rating is lowered to 🌡. In particular the new rating for the error associated to ϕ_{12} will remain the same in the final classification returned by HDA. Errors coming from $(\Sigma_1)^{\times}$ were already recognized as spurious; this is not always the case.

Eventually HDA yields the same errors of Example 5, but rated differently: the expected one—error (5)—is rated $\langle$🌡$, \rho(\phi_5)\rangle$ (ranking first) while the remaining spurious errors are rated $\langle$🌡$, \rho(\phi_i)\rangle$.

5 Conclusions

In this paper we proposed a heuristic criterion to detect spurious errors in ambiguous formulae. An error is spurious when it is not relative to the formula interpretation expected by the user. We integrated the criterion in the efficient disambiguation algorithm of [6].

We also believe that the specification of a disambiguation algorithm (Section 2) and the description of our efficient disambiguation algorithm (Section 3) are an improvement over previous descriptions in the literature.

We have implemented the proposed algorithm in the Matita proof assistant [2] and experimented with it in an ongoing formal development of Lebesgue's dominated convergence theorem in an abstract setting. Actually this formalization effort has motivated the study of spurious error identification since in the abstract setting there are plenty of overloaded operators and it was not unusual to be faced with too many error messages to be useful. In the current implementation in Matita we have decided to hide spurious errors from the user, unless explicitly asked for. This choice has decreased dramatically the amount of error messages, but in the general case is still possible to be faced with more than 1 genuine (i.e. not spurious) error. The problem of how effectively present multiple error messages to the user belongs to the user-interface field and will be discussed in a forthcoming paper.

For efficiency reasons, the criterion implemented in Matita is Criterion 2, that is a restriction of Criterion 1. There are cases of undetected spurious errors in Matita that would have been caught by the more general criterion. Consider for instance the right hand side of the formula given in Example 1. According to Criterion 1 the only two genuine errors are:

- "in $?_1 + \mathtt{z}$: z is a vector, but is used as a scalar"
- "in $?_1 + ?_2 \overrightarrow{+} \mathtt{z}$: $?_1 + ?_2$ is a scalar, but is used as a vector"

however, according to Criterion 2, we also get the errors:

- "in $\alpha \cdot \mathtt{f(x)} \overrightarrow{+} ?_1 \overrightarrow{+} \mathtt{z}$: $\alpha \cdot \mathtt{f(x)}$ is a scalar, but is used as a vector"
- "in $\alpha \overrightarrow{\cdot} \mathtt{f(x)} \overrightarrow{+} ?_1 \overrightarrow{+} \mathtt{z}$: $\mathtt{f(x)}$ is a scalar, but is used as a vector"

Moreover Criterion 1 is debatable itself: are both the above "genuine" errors really genuine? Would mathematicians agree that the second error is spurious since the number of scalars is greater than the number of vectors in the sum? What if there were two scalars and two vectors in the same sum? Or does the order matter? Does the first addend determines the signature of the sum?

Unable to convince ourselves that a general answer to the above questions exists, we claim that Criterion 1 is widely acceptable and never gives false positives. Whether the gap between the two criteria can be reduced without loosing efficiency is an open research direction.

References

1. Andrea Asperti, Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. A content based mathematical search engine: Whelp. In *Post-proceedings of the Types 2004 International Conference*, volume 3839 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 2004.
2. Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 2007. Special Issue on User Interface for Theorem Proving. To appear.
3. G. Bancerek and P. Rudnicki. Information retrieval in MML. In Andrea Asperti, Bruno Buchberger, and James H. Davenport, editors, *Proceedings of the Second International Conference on Mathematical Knowledge Management, MKM 2003*, volume 2594 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
4. Herman Geuvers and Gueorgui I. Jojgov. Open proofs and open terms: A basis for interactive logic. In J. Bradfield, editor, *Computer Science Logic: 16th International Workshop, CSL 2002*, volume 2471 of *Lecture Notes in Computer Science*, pages 537–552. Springer-Verlag, January 2002.
5. César Muñoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory*. PhD thesis, INRIA, November 1997.
6. Claudio Sacerdoti Coen and Stefano Zacchiroli. Efficient ambiguous parsing of mathematical formulae. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *Proceedings of Mathematical Knowledge Management 2004*, volume 3119 of *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, 2004.
7. Zentralblatt MATH. `http://www.emis.de/ZMATH/`.