Math.comput.sci. 2 (2008), 355–378

1661-8270/020355-24, *published online* December 5, 2008
DOI 10.1007/s11786-008-0058-2

Mathematics in Computer Science

Spurious Disambiguation Errors and How to Get Rid of Them

Claudio Sacerdoti Coen and Stefano Zacchiroli

Abstract. The disambiguation approach to the input of formulae enables users of mathematical assistants to type correct formulae in a terse syntax close to the usual ambiguous mathematical notation. When it comes to incorrect formulae however, far too many typing errors are generated; among them we want to present only errors related to the formula interpretation meant by the user, hiding errors related to other interpretations.

We study disambiguation errors and how to classify them into the spurious and genuine error classes. To this end we give a general presentation of the classes of disambiguation algorithms and efficient disambiguation algorithms. We also quantitatively assess the quality of the presented error classification criteria benchmarking them in the setting of a formal development of constructive algebra.

Mathematics Subject Classification (2000). Primary 68T99; Secondary 03B70.

Keywords. Ambiguity, semantic analysis, user interaction, metavariables.

1. Introduction

In [11] 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 [1,3,13] – we cannot assume the knowledge of a language other than the usual corpus of ambiguous mathematical notation.

Our algorithm mimics a mathematician's behavior of disambiguating a formula by choosing the only possible interpretation that has a meaning in the current

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

context. However when a formula is not correct, every interpretation may be considered as "equally" meaningless. Nevertheless, a mathematician seems to be able to understand which interpretation is more likely, spotting the genuine errors in the formula.

Example 1.1. 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 on the right hand side of the equation. Instead, the algorithm of [11] returns several alternative error messages such as: `in "f(α·x⃗+ ... +⃗z) = ...": x is a vector, but is used as a scalar`. The error spotted by the mathematician is just one of them.

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 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.[1]

Consider again Example 1.1. 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 several genuine errors: 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 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 1.1 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 on the right hand side.

Note that a genuine error localized in a formula F does not always say that F is the sole responsible for the overall incorrectness. For instance in `v=x` where `v` is a vector and `x` a scalar, we have either a genuine error localized in `v` (a vector used as a scalar) or another genuine one localized in `x` (a scalar used as a vector). Moreover, a formula may even contain two genuine independent errors at the same time; in this case the errors are localized in disjoint sub-formulae. An example is the conjunction of two statements each containing an error.

[1] A short version of this paper has already appeared in the proceedings of the Mathematical Knowledge Management 2007 conference [6]. In the present version we study 2 alternative criteria and algorithms for the recognition of spurious disambiguation errors, assess their usefulness, and compare them quantitatively.

The main goal of this paper is the integration of spurious error detection in the efficient algorithm proposed in [11]. 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 [11], proving that it is a member of the disambiguation algorithm class, while in Section 4 we extend the algorithm with spurious error detection. Finally, in Section 5 we benchmark the extended algorithms.

2. Disambiguation algorithms

Traditionally semantic analysis maps an abstract syntax tree (AST for short) 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 single term containing non linear placeholders in the spirit of [5, 8]. We say that a term u' is an *instantiation* (or instance) of u if it is obtained filling zero or more of its placeholders.

For example $?_1 = ?_2 + ?_2$ stands for the set of terms $\{u_1 = u_2 + u_2 \mid u_1, u_2 \text{ terms}\}$; $?_1 = 0 + 0$ and $0 = 0 + 0$ are two instances belonging to that set. In Figure 1 is given a graphical intuition of the mapping from terms with placeholders to the corresponding sets of placeholder free instances; the latter sets can overlap. In the previous example $0 = 0 + 0$ is also an instance of $?_1 = ?_2 + ?_1$.

Lemma 2.1. *If u_1 is an instance of u_2 then the set of instances of u_1 is a subset of the set of instances of u_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* $\mathcal{R}(\cdot)$, which is a function from terms to outcomes. An *outcome* is either the distinguished symbol ✓ or an informative error message. The latter is returned if and only if the set of well-typed instantiations of the input term is (known to be[2]) 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 2.2. *A term u without placeholders is well-typed iff $\mathcal{R}(u) =$ ✓.*

[2]For placeholder-free terms (i.e. *closed* terms) the problem reduces to type checking and is decidable; for open terms we do not require decidability, which cannot be achieved in type systems with dependent types.

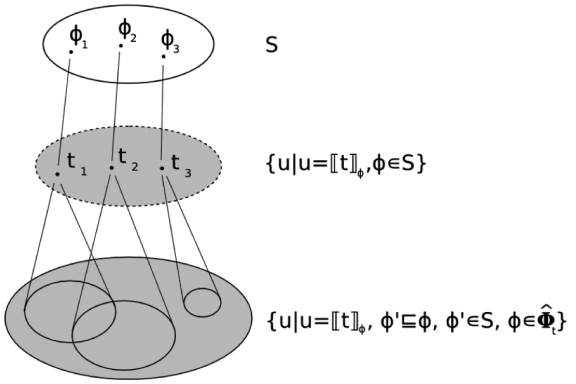

FIGURE 1. Interpretations are in 1-1 correspondence with terms (with placeholders). Terms, hence interpretations, represent sets of ground instances, i.e. fully determined semantics. Since such sets can overlap, a set of interpretations does not partition the set of its semantics.

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

According to our definition, a refiner can report only one error message: when a formula contains more than one typing error, the refiner only signals the first one. The disambiguation algorithms we present in the paper will have the same behavior, since generation of error messages is done by the refiner.

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. A non-overloaded symbol occurring in t is intuitively equivalent to an overloaded symbol s' such that $\mathcal{D}_{s'}$ is a singleton.[3]

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 its domain. 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:

- any occurrence of a non-overloaded symbol s has been assigned its sole semantics;
- all (applications of) occurrences of overloaded symbols not in the domain of ϕ have been interpreted as fresh placeholders;

[3]We do not include non-overloaded symbols in $Dom(t)$ since the computational complexity of the presented algorithms will be a function of the cardinality of $Dom(t)$.

- any occurrence of an overloaded symbol s in the domain of ϕ has been interpreted as $\phi(s)$.

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. In Figure 1 it is shown how interpretations are associated, via open terms, to (possibly overlapping) sets of semantics. We call $\perp$ 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 $\forall s \forall u\ \phi_1(s) = u \Rightarrow \phi_2(s) = u$. The minimum of Φ according to $\sqsubseteq$ is $\perp$.

Since we are only interested in terms that are possible semantics for a given AST t, in the remainder of the paper when we write "u is an instance of v" we also implicitly assume that $u = [\![t]\!]_\phi$ for some (partial) interpretation $\phi \in \Phi_t$. Moreover we will write "u is a *ground* instance of v" when u does not contain placeholders and u is an instance of v.

Lemma 2.3. *$\phi_1 \sqsubseteq \phi_2$ iff $[\![t]\!]_{\phi_2}$ is an instance of $[\![t]\!]_{\phi_1}$.*

Proof. By structural induction on t and by cases on the definition[4] of $[\![\cdot]\!]$. □

Together with Lemma 2.1, Lemma 2.3 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 2.2 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 $\forall \phi \in T, \exists! \phi' \in S, \phi' \sqsubseteq \phi$. We will say that S is a *cover* when $S \rhd \hat{\Phi}_t$. As shown in Figure 2(b), uniqueness is required to think of covers as partitions (see Theorem 2.6 below). However, as shown in Figure 2(a), uniqueness is not sufficient in the general case of S covering T. This will be solved with the introduction of refinements – see Figure 2(c) – whose formal definition follows Theorem 2.6.

Lemma 2.4. *If $S \rhd T$ then for each $\phi_1 \in T$ there exists a unique $\phi_2 \in S$ such that $[\![t]\!]_{\phi_1}$ is an instance of $[\![t]\!]_{\phi_2}$.*

Proof. By Lemma 2.3 and the definition of cover. □

Corollary 2.5. *If $S \rhd \hat{\Phi}_t$ and $\phi_1, \phi_2 \in S, \phi_1 \neq \phi_2$ then the set of instances of $[\![t]\!]_{\phi_1}$ is disjoint from the set of instances of $[\![t]\!]_{\phi_2}$.*

[4]Since, for the sake of brevity, we omitted the definition of $[\![\cdot]\!]$, the present lemma can also be seen as a required property of $[\![\cdot]\!]$.

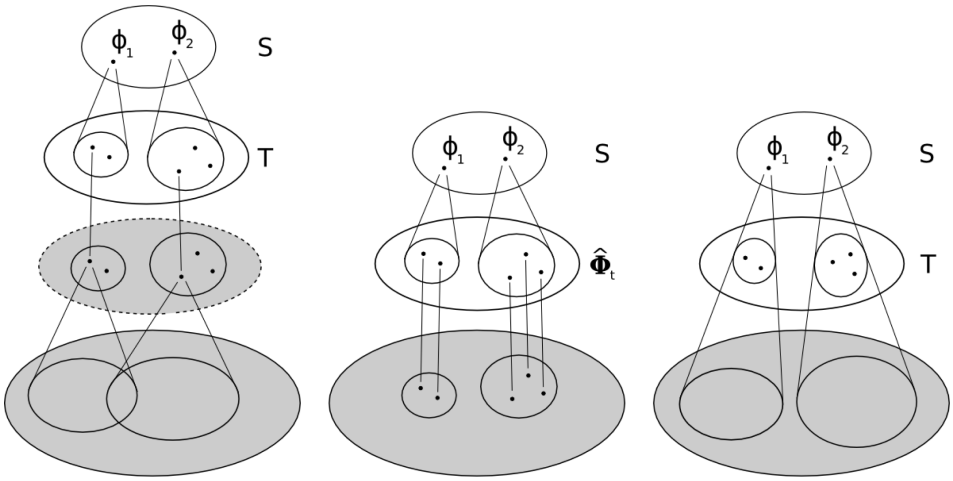

FIGURE 2. (a) When $S \rhd T$, S partitions the set of interpretations T, but not the set of ground instances of T. (b) When $S \rhd \hat{\Phi}_t$, S partitions the set of all semantics. (c) When $S \rhd T$, S partitions both the set of interpretations T and the set of ground instances of T (a subset of the set of all semantics).

Proof. Suppose per absurdum that u is an instance of both $[\![t]\!]_{\phi_1}$ and $[\![t]\!]_{\phi_2}$. Let $\phi \in \hat{\Phi}_t$ such that $[\![t]\!]_\phi$ is an instance of u. By Lemma 2.4, $\phi_1 = \phi_2$, but by hypothesis we know $\phi_1 \neq \phi_2$. □

Theorem 2.6. *$S \rhd \hat{\Phi}_t$ iff $\{\{u \mid u$ is a ground instance of $[\![t]\!]_\phi\} \mid \phi \in S\}$ is a partition of $\{u \mid \exists \phi \in \hat{\Phi}_t, u = [\![t]\!]_\phi\}$ (i.e. the set of all semantics of t).*

Proof. The forward implication is by Lemma 2.4 and Corollary 2.5. 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 $u = [\![t]\!]_\phi$ is a ground instance of $[\![t]\!]_{\phi'}$. Thus $S \rhd \hat{\Phi}_t$. □

We say that a set of interpretations A' is a *refinement* of a set of interpretations A, written $A \diamondsuit A'$ when:

1. $A \rhd A'$
2. for all $\phi \in A$ and $\psi \in \hat{\Phi}_t$ such that $[\![t]\!]_\psi$ is an instance of $[\![t]\!]_\phi$, there exists a unique $\phi' \in A'$ such that $[\![t]\!]_\psi$ is an instance of $[\![t]\!]_{\phi'}$.

As shown in Figure 2(c), when $S \diamondsuit T$ we can think of S as a partition coarser than T. Refinements will play a major role in our disambiguation algorithm that proceeds by iteratively building more and more fine grained refinements. Theorem 2.7, whose intuition is shown in Figure 3(a), is a preliminary step in this

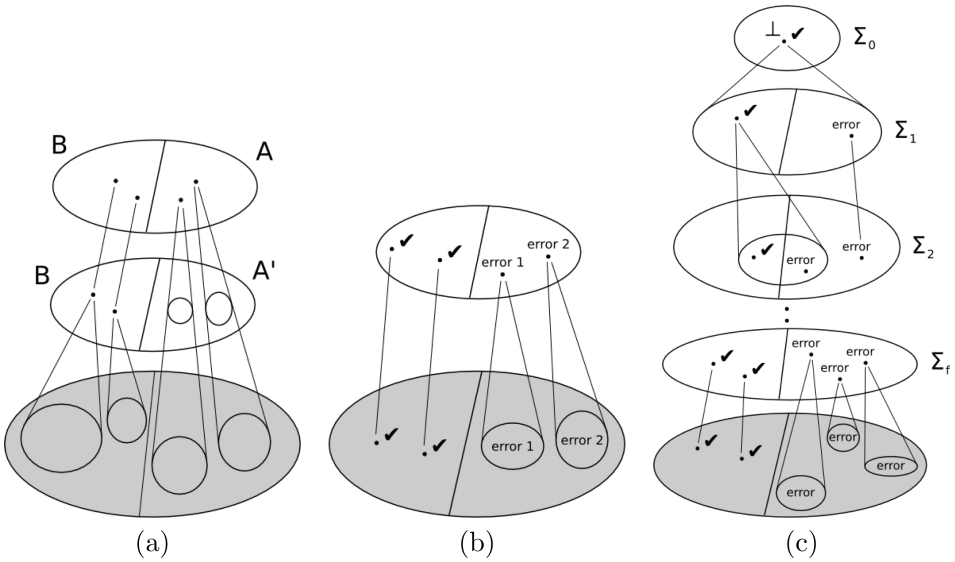

FIGURE 3. (a) Theorem 2.7: when A and B are disjoint and partition the set of theirs ground instances, refining A with A' refines the partition. (b) A typing cover partitions the set of all semantics. A well-typed interpretation represents a well-typed singleton; a non well-typed interpretation represents an equivalence class of semantics which cannot be typed for the same reason. (c) Refinement process: faulty interpretations are propagated, well-typed ones are refined as in (a) until a typing cover is reached (b).

direction, since it shows how to build a more precise refinement by substituting some interpretations (intuitively those so-far correct) with more instantiated ones.

Theorem 2.7. *If* $A \cap B = \emptyset$, $A \cup B \rhd \hat{\Phi}_t$ *and* $A \diamondsuit A'$, *then* $A' \cup B \rhd \hat{\Phi}_t$.

Proof. By Theorem 2.6 $\{\{u \mid u \text{ is a ground instance of } [\![t]\!]_\phi\} \mid \phi \in A \cup B\}$ partitions the set of all semantics of t. $\{\{u \mid u \text{ is a ground instance of } [\![t]\!]_\phi\} \mid \phi \in A' \cup B\}$ partitions the same set by definition of $A \diamondsuit A'$, where the requirement $A \rhd A'$ is fundamental to avoid interference with B. Hence the thesis by Theorem 2.6. □

A set S of interpretations is said to be *typing* when for all $\phi \in S$, if $\mathcal{R}([\![t]\!]_\phi) = \checkmark$ then $\phi \in \hat{\Phi}_t$. In particular a *typing cover* is a cover $S \rhd \hat{\Phi}_t$ that is also typing. We use typing covers as concise representations of typing information for all the semantics of a term (see Figure 3(b) and Theorem 2.8). The output of our disambiguation algorithm is a typing cover equipped with rating information for its interpretations (that will be called classification).

Theorem 2.8. *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 $u = \llbracket t \rrbracket_\phi$.*

Proof. If $\mathcal{R}(\llbracket t \rrbracket_\phi) \neq \checkmark$ by definition of $\mathcal{R}(\,\cdot\,)$. Otherwise by Lemma 2.2 and definition of typing cover. □

We also expect something more that cannot be grasped formally: if S is a typing cover, u is in the set of all semantics of t, ϕ is the only interpretation in S such that u is a ground instance of $\llbracket t \rrbracket_\phi$, and 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.

Intuitively, the set of interpretations that corresponds to the coarsest partition of the semantics of t is the singleton set $\{\bot\}$. The following lemma confirms this intuition and provides necessary and sufficient conditions for this set to be a typing cover.

Lemma 2.9. $\{\bot\} \rhd \hat{\Phi}_t$. *Moreover $\{\bot\}$ 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 a rate $\rho(\phi)$ is associated. A rate is an element of a partially ordered set $(A, \preceq)$, such that $\rho(\phi_1) \preceq \rho(\phi_2)$ iff $\llbracket t \rrbracket_{\phi_1}$ is less likely to be the intended meaning of t than $\llbracket t \rrbracket_{\phi_2}$.

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 $\langle \phi, o, r \rangle \in \Sigma, o = \mathcal{R}(\llbracket t \rrbracket_\phi)$, 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 $\preceq$. We reserve 🌡 for well-type interpretations, 🌡 for genuine errors, and 🌡 for spurious errors. The latter symbol will be used only in Section 4.

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

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

Algorithm 1 (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:

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

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

We call Algorithm 1 "naive" since it 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 2.1 (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 (not necessarily unique) error messages, 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"`
- "$\alpha\cdot\mathtt{x}+\beta\cdot\mathtt{y}$` 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. Efficient disambiguation algorithms

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 $\phi_i \sqsubseteq \phi$ for each $\phi_i \in S$, the domain of ϕ_i (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. Figure 3(c) explains graphically the refinement process.

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 ϕ. 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 a classification Σ extended to s as:

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

Lemma 3.1. *Let Σ be a classification, let $S = \{\phi \in \hat{\Phi}_t \mid \exists \phi' \in S_\Sigma, \phi' \sqsubseteq \phi\}$. If $S_\Sigma \rhd S$ and the interpretations in S_Σ are defined on the same domain then for all $s \in Dom(t)$ we have that $\Sigma \diamondsuit\!\!\gg \Sigma_s$ and $S_{\Sigma_s} \rhd S$.*

Proof. By construction of Σ_s and definition of $\diamondsuit$. The condition $S_\Sigma \rhd S$ is required for uniqueness in the proof of $\Sigma \diamondsuit\!\!\gg \Sigma_s$. □

The previous lemma is better understood in the particular case where Σ is a covering classification. In such a case $S = \hat{\Phi}_t$ and the lemma just says that the extension of a covering classification is still a covering classification. Presented in this form, the lemma generalizes to classifications covering only a subset of $\hat{\Phi}_t$.

The refinement process outlined above and in Figure 3(c) 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 ϕ 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 than 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 3.1 (Refinement process). Consider the AST of Example 1.1. Picking occurrences $s \in Dom(t)$ 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 ✗):

$$\Sigma_0 = \{\langle \phi_1, \checkmark, \langle \text{🌡}, \rho(\phi_1)\rangle\rangle\} \qquad \text{where} \quad [\![t]\!]_{\phi_1} = f(?_1) = ?_2 \text{ and } \phi_1 = \bot$$

$$\Sigma_1 = \{\langle \phi_{11}, \checkmark, \langle \text{🌡}, \rho(\phi_{11})\rangle\rangle, \quad [\![t]\!]_{\phi_{11}} = f(?_1 \overrightarrow{+} z) = ?_2$$
$$\langle \phi_{12}, \text{✗}, \langle \text{🌡}, \rho(\phi_{12})\rangle\rangle\} \quad [\![t]\!]_{\phi_{12}} = f(?_1 + z) = ?_2$$

$$\Sigma_2 = \{\langle \phi_{111}, \checkmark, \langle \text{🌡}, \rho(\phi_{111})\rangle\rangle, \quad [\![t]\!]_{\phi_{111}} = f(?_1 \overrightarrow{+} ?_2 \overrightarrow{+} z) = ?_3$$
$$\langle \phi_{112}, \text{✗}, \langle \text{🌡}, \rho(\phi_{112})\rangle\rangle, \quad [\![t]\!]_{\phi_{112}} = f(?_1 + ?_2 \overrightarrow{+} z) = ?_3$$
$$\langle \phi_{12}, \text{✗}, \langle \text{🌡}, \rho(\phi_{12})\rangle\rangle\} \quad [\![t]\!]_{\phi_{12}} = f(?_1 + z) = ?_2$$

$$\begin{array}{lll}
\Sigma_3 = \{\langle \phi_{1111}, \checkmark, \langle \overset{1}{\bullet}, \rho(\phi_{1111})\rangle\rangle, & \text{where} & [\![t]\!]_{\phi_{1111}} = f(\alpha \overrightarrow{\cdot} x \overrightarrow{+} ?_1 \overrightarrow{+} z) = ?_2 \\
\quad \langle \phi_{1112}, \times, \langle \overset{1}{\bullet}, \rho(\phi_{1112})\rangle\rangle, & & [\![t]\!]_{\phi_{1112}} = f(\alpha \cdot x \overrightarrow{+} ?_1 \overrightarrow{+} z) = ?_2 \\
\quad \langle \phi_{112}, \times, \langle \overset{1}{\bullet}, \rho(\phi_{112})\rangle\rangle, & & [\![t]\!]_{\phi_{112}} = f(?_1 + ?_2 \overrightarrow{+} z) = ?_3 \\
\quad \langle \phi_{12}, \times, \langle \overset{1}{\bullet}, \rho(\phi_{12})\rangle\rangle\} & & [\![t]\!]_{\phi_{12}} = f(?_1 + z) = ?_2
\end{array}$$

...

Theorem 3.2 (Correctness of the refinement process). *The above refinement process implements a disambiguation algorithm, i.e. for each AST t, $\Sigma_{|Dom(t)|}$ is a covering and typing classification.*

Proof. By induction on $|Dom(t)|$ we prove that $\Sigma_{|Dom(t)|}$ is covering.

Base case. by Lemma 2.9 Σ_0 is a covering classification.

Inductive case. let Σ_n be a covering classification per inductive hypothesis. By definition $\Sigma_{n+1} = ((\Sigma_n)^{\checkmark})_s \cup (\Sigma_n)^{\times}$. By Theorem 2.7 and Lemma 3.1, Σ_{n+1} is covering.

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

The above refinement process is parametric in how the next symbol $s \in Dom(t)$ is chosen at each step. In [11] we discussed the implication of such a choice on the computational complexity in terms of numbers of refiner invocations. Our conclusion can be summarized (and slightly generalized) in the following way: the best choices correspond to those strategies (called *efficient*) that always pick the next symbol s so that $|(((\Sigma_n)^{\checkmark})_s)^{\times}|$ is maximized. The rationale of all such strategies is that the more partial terms you rule out, the less you will have to refine later on. The best choice corresponds to the case where the symbol s is either the argument of an already interpreted symbol s', or when s is applied to an already interpreted symbol s'. Intuitively, in both cases, the types of the interpretations of s and s' are mutually constrained, and all interpretations that do not respect this constraint will be pruned.

The actual strategy used in [11] corresponds to a pre-visit of the AST t, which trivially implements the father-children requirement for an efficient strategy. When a node s of the AST is visited, all its choices $\mathcal{D}_s$ must be considered to obtain $((\Sigma_n)^{\checkmark})_s$. Here we have an additional degree of freedom in the combination of the recursive descent on the AST and the consideration of all choices. One possibility is to make a choice and immediately continue recursion on the subtree before considering the next choices; the dual possibility is to immediately classify all choices before recurring on the subtree. The two possibilities correspond respectively to depth-first and breadth-first visits of the choice tree, which is obtained from the AST by replacing every node s with the nodes in $\mathcal{D}_s$, duplicating edges as needed.

Algorithm 2 (Efficient disambiguation algorithm). We now present the *efficient disambiguation algorithm* (EDA for short) of [11]. It proceeds by recursion on $Dom^{\texttt{list}}(t)$, which is the list of overloaded symbol occurrences in t obtained in a pre-visit traversal.

$$f(\Sigma, l) = \begin{cases} \Sigma & \text{if } l = [] \\ f((\Sigma_s)^{\checkmark}, tl) \cup (\Sigma_s)^{\times} & \text{if } l = s :: tl \end{cases}$$
$$\mathrm{EDA}(t) = f\big((\Sigma_0)^{\checkmark}, Dom^{\texttt{list}}(t)\big) \cup (\Sigma_0)^{\times}$$

EDA implements the above breadth-first efficient strategy in a non trivial way (Theorem 3.3). The invariant of the algorithm is that, at the n-th recursive invocation, Σ is equal to $(\Sigma_n)^{\checkmark}$ (where Σ_n is the n-th covering classification of the refinement process, see again Figure 3(c)). That means that Σ only contains the interpretations that are so far well-typed. The function immediately extends all interpretations in Σ (or, equivalently, $(\Sigma_n)^{\checkmark}$) with the head symbol of $Dom^{\texttt{list}}(t)$. Then it splits the well-typed part $(\Sigma_s)^{\checkmark}$, which is passed in the recursive call for further extension, and the non well-typed part $(\Sigma_s)^{\times}$. Since the latter is propagated as it is in the refinement process, the algorithm avoids passing it to the next recursive call. Instead, it will simply merge $(\Sigma_s)^{\times}$ with the result of the recursive invocation. The initial work done by EDA before calling f is required to grant the invariant by immediately pruning non well-typed interpretations from Σ_0 (the coarsest covering classification).

Without loss of efficiency, which is affected only by the visit order of the AST, we could have implemented the depth-first strategy.

Theorem 3.3 (Correctness of EDA). *EDA implements a disambiguation algorithm.*

Proof. By Theorem 3.2 it is sufficient to prove that the classification returned by EDA is the same returned by the refinement process. We observe that

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

where $(\dagger)$ 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 $(\ddagger)$ for $n = |Dom(t)|$. □

Example 3.2 (EDA execution). Consider the AST of Example 1.1. 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 $\mathtt{f}(?_1 + ?_2 \vec{+} \mathtt{z}) = ?_3$: $?_1 + ?_2$ is a scalar, but is used as a vector"
3. "in $\mathtt{f}(\alpha \cdot \mathtt{x} \vec{+} ?_1 \vec{+} \mathtt{z}) = ?_2$: x is a vector, but is used as a scalar"
4. "in $\mathtt{f}(\alpha \vec{\cdot} \mathtt{x} \vec{+} \beta \cdot \mathtt{y} \vec{+} \mathtt{z}) = ?_1$: y is a vector, but is used as a scalar"
5. "in $\mathtt{f}(\alpha \vec{\cdot} \mathtt{x} \vec{+} \beta \vec{\cdot} \mathtt{y} \vec{+} \mathtt{z}) = ?_1 + \mathtt{z}$: z is a vector, but is used as a scalar"
6. "in $\mathtt{f}(\alpha \vec{\cdot} \mathtt{x} \vec{+} \beta \vec{\cdot} \mathtt{y} \vec{+} \mathtt{z}) = ?_1 \vec{+} \mathtt{z}$: $?_1 \vec{+} \mathtt{z}$ 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 1.1.

4. Spurious disambiguation errors

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 imposes the two following requirements:

a. to interpret an occurrence s, every occurrence s' preceding 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 $\phi' \sqsupseteq \phi$.

Taken 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.

To obtain a formal and implementable definition of Criterion 1, we also need to understand what does it mean for an error to be "localized in a sub-formula F". Suppose that a wrong interpretation ϕ' is obtained from a correct interpretation ϕ by making a choice for s. There are at least two heuristics to decide the error cause. According to the *optimistic heuristic*, we assume that the error has only been caused by the last choice. On the other hand, according to a more *pessimistic heuristic*, the error has been caused by the choices of all symbols in the path from the AST root to s. The latter condition makes sense since the type of a function constraints the type of its arguments and vice-versa.

The optimistic heuristic localizes the error in the sub-formula F rooted in s, where s is the last chosen symbol. Thus, an error is not localized in the sub-formula F rooted in s as soon as another interpretation does not localize the error in F. On the other hand, the more pessimistic heuristic localizes the error in every sub-formula F' containing F. Thus, an error is not localized in the sub-formula F rooted in s when every interpretation that is total on the sub-formula rooted in s is correct.

Example 4.1. Consider the (non-typable) AST t corresponding to $\mathtt{f}(\mathtt{x} + \mathtt{y}) + \mathtt{y}^2$, where f is a real-valued function on vectors, the symbol "+" is overloaded on scalars and vector sums, and exponentiation is defined only on scalars.

Let ϕ be the typing partial interpretation $[+_1 \mapsto \textit{scalar sum}; +_2 \mapsto \textit{vector sum}]$ and let ϕ' be ϕ extended with $[\,\cdot^{\,2} \mapsto \textit{scalar exponentiation}]$. Since y is used both as a scalar and as a vector, $[\![t]\!]_{\phi'}$ is not well-typed.

The optimistic heuristic localizes the error in the sub-formula $\mathtt{y}^2$. The more pessimistic heuristic localizes the error in the sub-formulae $\mathtt{y}^2$ and $\mathtt{f}(\mathtt{x}+\mathtt{y})+\mathtt{y}^2$ (but not in $\mathtt{f}(\mathtt{x}+\mathtt{y})$).

The two heuristics, combined with the previously discussed requirements for integration in EDA, yield two different criteria:

Criterion 2 (Prudent 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. ϕ, ϕ' *are both defined on all s' preceding s in pre-order;*
3. $\mathcal{R}([\![t]\!]_{\phi'}) = \checkmark$;
4. ϕ' *is total on the occurrences of overloaded symbols occurring in the sub-tree rooted at s.*

Criterion 3 (Draconian 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. ϕ, ϕ' *are both defined on all s' preceding s in pre-order;*
3. $\mathcal{R}([\![t]\!]_{\phi'}) = \checkmark$.

The two criteria differ only in the fourth requirement.[5] The prudent criterion is based on the pessimistic heuristic, while the draconian on the optimistic heuristic. Dropping from both criteria the second requirement – imposed by Requirement (a) on page 367 – we obtain two different more formal writings of Criterion 1 that differ only in the translation of "*error localized in a sub-formula F*". It is evident that more errors are classified as spurious by the draconian criterion. Thus, the draconian criterion is to be preferred as long as genuine errors are not erroneously classified as spurious. In Section 5 we investigate this.

We now address the issue of integrating the two criteria in EDA: Section 4.1 describes an implementation of Criterion 2, Section 4.2 an implementation of Criterion 3.

[5] In the short version [12] of this paper we have investigated a variant of the prudent spurious error detection criterion where the second requirement was stricter: "$\phi'(s') = \phi(s')$ *for all s' preceding s in pre-order*". The requirement was induced by a prototypical EDA implementation that worked depth-first on the choice tree. The criterion was even more prudent than the current one, but lacked a clear intuition. Practically, too many errors were not classified as spurious by it with no evident reason; hence, it has been dropped in the present version of the paper.

4.1. Prudent spurious error detection

$f(\Sigma, l)$, the core of EDA, does not work directly on t, but rather on the list l, which is a serialization of the occurrences of overload symbols in t. In l the tree-structure of t has been lost. Given that Criterion 2 is defined in terms of sub-trees rooted at overload symbol occurrences, we cannot make f recognize spurious errors using Criterion 2 still working on l. As a solution we could make f work by recursion on (the AST of the formula of) t by integrating in f a pre-visit tree traversal. Still, we prefer to avoid binding f to the AST data type and to keep separate the construction of $Dom(t)$ from the actual disambiguation.

Therefore we introduce the new $Dom^{\texttt{tree}}(t)$ datatype which is a tree representation of $Dom(t)$. $Dom^{\texttt{tree}}(t)$ is a tree which contains only the nodes $s \in Dom(t)$ and preserves the ancestor-descendant relation of t. As a concrete representation of $Dom^{\texttt{tree}}(t)$ 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 sub-trees of a given node have been traversed.

Algorithm 3 (Prudent efficient disambiguation algorithm). We call the algorithm that recognizes spurious errors according to the prudent criterion the *prudent efficient disambiguation algorithm* (P-EDA for short). It proceeds by recursion on $Dom^{\texttt{tree}}(t)$ and, at the end of children traversal, lowers the rate of spurious errors. The pseudo code of P-EDA is given below:

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

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

$$\text{P-EDA}(t) = (\Sigma')^{\checkmark} \cup p\big((\Sigma')^{\checkmark}, (\Sigma')^{\text{✗}} \cup (\Sigma_0)^{\text{✗}}\big) \quad \text{where} \quad \Sigma' = g\big((\Sigma_0)^{\checkmark}, Dom^{\texttt{tree}}(t)\big)$$

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

Theorem 4.1 (Correctness of P-EDA).

1. P-EDA *implements a disambiguation algorithm.*
2. *An error in a classification returned by* P-EDA *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 3.3 it is sufficient to prove that the classification returned by P-EDA 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 P-EDA is called on the tree $\overset{s_n\rightarrow \quad b}{\underset{c}{\downarrow}}$. The nodes that EDA will encounter processing tl are the same (and in the same order) of those P-EDA 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 $(g((\Sigma_{s_n})^{\checkmark}, c))^{\text{✗}} \cup g(g((\Sigma_{s_n})^{\checkmark}, c)^{\checkmark}, b)$ 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 $\phi \in (\Sigma_s)^{\text{✗}} \cup (g((\Sigma_s)^{\checkmark}, c))^{\text{✗}}$. We interpret that as ϵ being located in $\overset{s}{\underset{c}{\downarrow}}$. 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 $\overset{s'\rightarrow \quad b'}{\underset{c'}{\downarrow}}$ 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 4.2 (P-EDA execution). Consider again the AST of Examples 1.1 and 3.2. The first recursive invocation is $g(\Sigma, \tau)$ where: $\Sigma = \{\langle \bot, \checkmark, \langle \text{🌡}, \rho(\bot)\rangle\rangle\}$ and $\tau = \overset{+\rightarrow \quad b}{\underset{c}{\downarrow}}$. g computes

$$\begin{aligned} \Sigma_s = \{&\langle \phi_{11}, \checkmark, \langle \text{🌡}, \rho(\phi_{11})\rangle\rangle, \quad \text{where} \quad [\![t]\!]_{\phi_{11}} = f(?_1 \overrightarrow{+} z) = ?_2 \\ &\langle \phi_{12}, \text{✗}, \langle \text{🌡}, \rho(\phi_{12})\rangle\rangle\} \qquad\qquad [\![t]\!]_{\phi_{12}} = f(?_1 + z) = ?_2 \end{aligned}$$

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

$$\begin{aligned} \Sigma_1 = \{&\langle \phi_{11111}, \checkmark, \langle \text{🌡}, \rho(\phi_{11111})\rangle\rangle, \quad \text{where} \quad [\![t]\!]_{\phi_{11111}} = f(\alpha \overrightarrow{\cdot} x \overrightarrow{+} \beta \overrightarrow{\cdot} y \overrightarrow{+} z) = ?_1 \\ &\langle \phi_{11112}, \text{✗}, \langle \text{🌡}, \rho(\phi_{11112})\rangle\rangle, \qquad\qquad [\![t]\!]_{\phi_{11112}} = f(\alpha \overrightarrow{\cdot} x \overrightarrow{+} \beta \cdot y \overrightarrow{+} z) = ?_1 \\ &\langle \phi_{1112}, \text{✗}, \langle \text{🌡}, \rho(\phi_{1112})\rangle\rangle, \qquad\qquad [\![t]\!]_{\phi_{1112}} = f(\alpha \cdot x \overrightarrow{+} ?_1 \overrightarrow{+} z) = ?_2 \\ &\langle \phi_{112}, \text{✗}, \langle \text{🌡}, \rho(\phi_{112})\rangle\rangle\} \qquad\qquad [\![t]\!]_{\phi_{112}} = f(?_1 + ?_2 \overrightarrow{+} z) = ?_3 \end{aligned}$$

Since $(\Sigma_1)^{\checkmark}$ is not empty, all the errors in $(\Sigma_s)^{\text{✗}}$ and $(\Sigma_1)^{\text{✗}}$ 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 P-EDA. Errors coming from $(\Sigma_1)^{\text{✗}}$ were already recognized as spurious; this is not always the case.

Eventually P-EDA yields the same errors of Example 3.2, but rated differently: the expected one – error (5) – is rated $\langle 🌡, \rho(\phi_5)\rangle$ and ranked first, as well as error (6), while the remaining spurious errors are rated $\langle 🌡, \rho(\phi_i)\rangle$. Thus the algorithm has correctly detected that there is a problem in the topmost addition of the right hand side. Indeed, either the addition does not respect the typing constraint imposed by the left hand side, or that imposed by its second child z. In the latter case, in our prototype the user gets the dual error message: z does not respect the constraint imposed by its parent.

Example 4.3 (P-EDA execution: false negative). Consider now a variation of the previous example: $\mathtt{f}(\alpha\cdot\mathtt{x}+\beta\cdot\mathtt{y}+\mathtt{z}) = \alpha\cdot\mathtt{f}(\mathtt{x})+\beta\cdot\mathtt{f}(\mathtt{y})+2\cdot\mathtt{z}$. The non spurious error messages detected by P-EDA are:

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

The algorithm has correctly detected that there is a problem on the right hand side of the equation, but it has not been "draconian enough" to blame z or the last product. Indeed, errors (2) and (3) show that the last product does not respect at once the typing constraints imposed by its parent and its second child. On the contrary, not recognizing error (1) as spurious is questionable.

4.2. Draconian spurious error detection

Algorithm 4 (Draconian efficient disambiguation algorithm). We call the algorithm that recognizes spurious errors according to the draconian criterion the *draconian efficient disambiguation algorithm* (D-EDA for short). Differently than P-EDA, D-EDA does not require a structured domain; as such, it proceeds by recursion on $Dom^{\mathtt{list}}(t)$, lowering the rate of spurious errors after each domain extension. The pseudo code of D-EDA is given below:

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

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

$$\text{D-EDA}(t) = h\big((\Sigma_0)^{\text{✓}}, Dom^{\mathtt{list}}(t)\big) \cup p\big((\Sigma_0)^{\text{✓}}, (\Sigma_0)^{\text{✗}}\big)$$

$h(\cdot)$ has the same role $f(\cdot)$ had in EDA, while $p(\cdot,\cdot)$ has the same definition it had in P-EDA.

Theorem 4.2 (Correctness of D-EDA).

1. D-EDA *implements a disambiguation algorithm.*
2. *An error in a classification returned by* D-EDA *is spurious according to Criterion* 3 *iff it is rated* $\langle \text{🌡}, \rho(\phi)\rangle$.

Proof. We just give a sketch of the proof.

1. Since $h(\,\cdot\,)$ in D-EDA differs from $f(\,\cdot\,)$ in EDA only in the invocation of $p(\,\cdot\,,\,\cdot\,)$ to rate errors, and since $p(\,\cdot\,,\,\cdot\,)$ does not drop any error, D-EDA implements a disambiguation algorithm because of Theorem 3.3.
2. Recursion is never performed on elements of the current classification corresponding to errors. Thus, as in P-EDA, once an error has been down-rated, its rate will never be raised again.

   $p((\Sigma_s)^{\checkmark}, (\Sigma_s)^{\text{✗}})$ down-rates an error associated to an interpretation $\phi \in S_{(\Sigma_s)^{\text{✗}}}$ iff there exists an interpretation $\phi' \in S_{(\Sigma_s)^{\checkmark}}$ iff $\mathcal{R}([\![t]\!]_{\phi'}) = \checkmark$ (because of (1)) and $\phi'(s') = \phi(s')$ for all s' preceding s in pre-order (because of (1) and the definition of Σ_s), i.e. iff ϕ, ϕ' are as required by Criterion 3. □

Example 4.4 (D-EDA execution). Consider again Example 4.2. D-EDA yields the very same result.

Example 4.5 (D-EDA execution: no false negative). Consider again Example 4.3. D-EDA yields the very same errors, but error (1) is now recognized as spurious since the correct interpretation that yields $\mathtt{f}(\alpha \overrightarrow{\cdot}\, \mathtt{x} \overrightarrow{+} \beta \overrightarrow{\cdot}\, \mathtt{y} \overrightarrow{+} \mathtt{z}) = ?_1 + ?_2$, which is not total on the right hand side, is now enough to down-rate the error.

Example 4.6 (D-EDA execution: false positive?). Consider the AST of $\alpha \cdot \mathtt{f}(\mathtt{x}) + \beta \cdot \mathtt{f}(\mathtt{y}) + \mathtt{z} = \mathtt{f}(\alpha \cdot \mathtt{x} + \beta \cdot \mathtt{y} + \mathtt{z})$, which has been obtained swapping the left and right hand sides of Example 4.4. D-EDA returns only the following two errors as non spurious:

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

On the one hand one can expect that, since before the swap we have blamed the usage of z on the right hand side, we should now blame the usage of z on the left hand side. Accordingly, this spurious error is a false positive. On the other hand, a mathematician that reads formulae from left to right[6] is likely to think at z as a scalar, blaming its usage on the right hand side as D-EDA does.[7]

[6] According to linguistic conventions [9], mathematical formulae are read from left to right or from right to left. Disambiguating formulae in linguistic order is potentially the most effective way to detect spurious errors the way a mathematician does. Unfortunately, the visit in linguistic order is not efficient (in the sense of page 365) since it does not respect the father-children requirement. For instance, the left to right visit of $\alpha \cdot x + (\beta \cdot y + z)$ will refine the term $\alpha \cdot x + (\beta \cdot y +_? z)$ where $+_?$ remains a placeholder and $\beta \cdot y$ is not constrained by $\alpha \cdot x + \cdot$.

[7] A related topic is the support for *naming conventions*, frequently used in mathematics textbooks to implicitly "type" identifier classes as in "*unless otherwise stated we will use* $u, v, w, \ldots$ *for real*

5. Performance of spurious error detection

In this section we quantitatively assess the effectiveness of the presented criteria in correctly identifying and localizing the errors related to the interpretation meant by the user. Our criteria localize errors at sub-formulae, that can be identified by the symbol occurrences they are rooted in. As such, our approach for measuring the success of a criterion is to introduce a number of errors at symbol occurrences and then to evaluate the criterion hit ratio in reporting them (as non spurious) to the user.

Errors spotted by semantic analysis can be classified as follows:

Symbol replacement. The user has written a symbol in place of a different one; this can happen because of a typo or because of a conceptual error.

Extra or missing argument. The user has applied a function or an operator to too many or too few arguments.

Structure alteration. The user has misused a notation, altering the structure of a formula as it is understood by the system. A typical case is when the precedence order expected by the user in a given formula is not the one used by the system.

Example 5.1 (Structure alteration). The precedence of $\wedge$ used as the logical conjunction is lower than that of equality that is in turn lower than that of $\wedge$ used as the lattice meet operator. If the parser is non ambiguous, when applied to $\mathtt{A} \wedge \mathtt{B} = \mathtt{C}$ it will always build either the AST of $(\mathtt{A} \wedge \mathtt{B}) = \mathtt{C}$ or that of $\mathtt{A} \wedge (\mathtt{B} = \mathtt{C})$ not withstanding what the user has in mind.

We believe that this and similar examples motivate in our context the use of GLR parsers as the one described in [10]. GLR parsers can return the set of all ASTs matching the input, and our disambiguation algorithm can be run in parallel on each of them. A possible way to extend spurious error detection criteria in this setting is to decide that errors coming from an AST are spurious if there exists another AST having at least one correct interpretation.

Since structure alteration is better handled with general parsers, we consider in the remainder of the paper only symbol replacement and extra argument errors. We avoid addressing missing argument errors since they yield exactly the same refinement error as extra arguments (i.e. arity mismatch), but are more difficult to benchmark. The reason is that the expected error should be localized in a missing sub-formula; instead, the error is localized in the parent of the missing argument, which is informative enough for the user, but makes the benchmarking code dirtier.

---

vectors". We do not take explicit advantage of naming conventions, as their declaration is not treated in this presentation, but some mathematical proof assistants – e.g. Coq (`http://coq.inria.fr`) – enable users to declare and exploit naming conventions, to support the widespread mathematical practice. Always using the type prescribed by the naming convention is not a good idea, since it does not allow for naming convention exceptions. In Coq the user may force another type only with a casting operator. In our setting we can do better by simply exploiting naming conventions to sort alternative interpretations of a formula: the more they adhere to the naming convention, the higher they rank.

Concretely, we have implemented our disambiguation algorithms in the Matita interactive theorem prover [2], we have picked a set of scripts from its library, and we have automatically introduced at random positions in the AST of every statement or definition 3 errors of the symbol replacement or insertion kinds. Then, we have run two modified versions of the Matita compiler (implementing Algorithms 3 and 4) on the broken scripts,[8] collecting all the errors and error locations reported by the system.

The chosen scripts constitute a development of constructive algebra [7] à la Bishop [4], up to valued group lattices. It includes 152 theorems and definitions, comprising: constructive setoids $(C, \neq)$ with the induced equality $=$; constructive partially ordered set $(C, \not\leq)$ with the induced order relations $\leq$ and $<$; groups $(C, \neq, +, O, -)$; lattices $(C, \wedge, \vee)$; valuations (K, C, μ); topological spaces (X, O); and all the algebraic structures (such as group lattices) that inherit from them. Moreover, the development imports all results on natural numbers $(\mathbb{N}, O, S, +, \cdot, <, \leq, =)$ from the standard library of Matita.

Our choice for this development has been driven by the high amount of overloading due to structure inheritance and reuse of natural number notations. In particular, `0` is overloaded with arity 0 as the neutral element of several structures, and with arity 1 as the operator that returns the opens of a topological set `X`. When used as a symbol argument, for instance in expressions like $(\texttt{0} + \texttt{x}) < \texttt{y}$, the interpretation of `0` forces that of $+$ and $<$. Moreover, `0` is also frequently quantified over the type of ordered sets; as such, `0` as an identifier appears both as a free and a bound name. For all these reasons, `0` is our symbol of choice for introducing errors in the formula by either replacing it for a symbol, or by adding it in argument position.

Table 1 presents the results obtained with the two implemented algorithms and, for comparison, with no spurious error detection at all. Spurious error recognition outputs on the fed scripts are partitioned in the following classes:

Precise. Just one non spurious error location is reported, which is also the one of the introduced error.

Imprecise. Several alternative non spurious error locations are reported, including the one of the introduced error. All the other error locations correspond to false negatives.

False positive. The location of the introduced error is reported as the location of at least one spurious error.

Undetected. No detected error is located at the location of the introduced error.

In general, multiple alternative error messages can be reported at the same location. For this reason, for each of the above classes, Table 1 shows the average and maximum number of errors (and their locations) which are considered genuine by the criterion. These figures tell how many different error messages the user is faced with, assuming that spurious errors are hidden to her. They also suggest

[8] Of the 456 generated scripts, 20 scripts were not actually broken, in the sense that the randomly modified formula still had a correct interpretation.

TABLE 1. Comparison of spurious error recognition criteria.

No Spurious Error Recognition

	scripts		locations		errors	
			avg	max	avg	max
precise	204	46.8%	1.0	1	3.0	3
imprecise	129	29.6%	2.6	6	5.3	13
false-positive	0	0.0%	n/a	0	n/a	0
undetected	103	23.6%	1.2	4	1.5	15
total	436	100.0%	1.5	6	3.3	15

Prudent Criterion

	scripts		locations		errors	
			avg	max	avg	max
precise	250	57.3%	1.0	1	2.9	5
imprecise	81	18.6%	2.1	3	5.0	11
false-positive	2	0.5%	2.5	3	3.5	4
undetected	103	23.6%	1.1	3	1.3	8
total	436	100.0%	1.2	3	2.9	11

Draconian Criterion

	scripts		locations		errors	
			avg	max	avg	max
precise	323	74.1%	1.0	1	3.0	9
imprecise	0	0.0%	n/a	0	n/a	0
false-positive	10	2.3%	1.7	2	2.9	5
undetected	103	23.6%	1.0	2	1.2	6
total	436	100.0%	1.0	2	2.6	9

that the error location is more significant than the error message since multiple messages are associated to the same location. This does not come as a surprise. Consider the formula `f x O` where `f` has arity 1. The algorithms easily detect that `O` is the error, independently from the interpretation of `O`. However, each interpretation potentially yields a different error message, always localized in `O`. For this reason, we decided to define precision in terms of locations.

When spurious error recognition is not in effect (upper part of Table 1), all errors are reported as genuine and the user can be faced with up to 15 errors, or 6 locations, per statement. On the average, the system reports more than one location, forcing the user to waste time to understand where the error actually is. The average number of errors reported as genuine is about 3; remember that in the test cases we have always introduced exactly one error per statement. The error location is undetected in 23.6% of the scripts. This is inherited from EDA, it is orthogonal to spurious error detection, and common to all criteria. In the remaining cases, more than 1/3 of the errors are reported imprecisely.

The draconian criterion (lowest part of Table 1) never reports errors imprecisely, at the price of 10 false positives over 436 tests. Moreover, it also decreases the average and maximum number of errors and locations presented to the user.

Eight of the false positives are similar: they contain a sub-formula $O\ ?\ s$ where s is an overloaded symbol and ? is a user-provided placeholder the system should infer. Now, $O\ ?$ can be interpreted as the set of opens of some topological space (to be inferred), which in turn is identified with its characteristic function. Thus $O\ ?\ ?_1$ is a correct term (meaning $?_1 \in O\ ?$), and all errors are eventually located in s, since no overloaded interpretation of s is an open set.

One of the remaining false positives is related to an error inserted in the definiens of subtraction in a group: $x + -y$ is replaced with $O + -y$. The type of O forces the type of $+$ that is unconstrained being the root of the definiens; thus all errors are reported on $-$ since $O + \cdot$ expects a natural number while y has been declared as a group element. The last false positive is similar.

The prudent criterion (middle part of Table 1) improves over the lack of spurious error detection, without coming close to the draconian performance. The eight equal false positives of the draconian criteria are imprecisely detected by the prudent criterion. The remaining two are still reported as false positives.

6. Conclusions

In this paper we have analyzed the problem of rating errors coming from multiple interpretations of user provided formulae due to symbol overloading. We have introduced the notions of spurious vs genuine errors, and proposed two heuristic criteria for the classification of errors in the two classes. An error is spurious when it is not relative to the formula interpretation expected by the user. The criteria have been specifically thought for integration in an efficient disambiguation algorithm which is also described and proved correct in the paper. We have also shown the resulting algorithms, which have also been implemented and benchmarked in Matita.

We have been motivated to study spurious errors by an on-going formalization in Matita of constructive algebra. Indeed, in such an abstract setting there are plenty of overloaded operators, and without spurious error recognition it is not unusual for users to be annoyed with several error messages at multiple locations.

The benchmarks, based on such formalization, validates the usefulness of spurious error recognition. Indeed, in the current implementation we have decided to hide spurious errors from the user, unless explicitly asked for. This choice has sensibly decreased the amount of error messages, but in the general case it is still possible to be faced with more than one error message, though usually error messages can be grouped according to their locations, whose number is on average 1 using the more demanding criterion. The latter observation paves the way to future studies in the human computer interaction field on the effective presentation of the remaining error messages and their locations.

As expected, the benchmarks confirm that the more demanding criterion is much more effective in pruning spurious errors, at the price of a few false positives. The less demanding criterion is moderately better than no spurious error recognition at all, without being completely false positive free. Practically, nothing hinders the combination of the two criteria to obtain an increasingly verbose error reporting mechanism: first we show genuine errors according to the more demanding criterion; then, upon user request when reported errors do not match the interpretation she has in mind, we add those errors that are genuine only according to the less demanding criterion; finally, if the user is still unsatisfied, she can ask to see all errors.

Acknowledgements

We would like to thank the anonymous referees for having spotted an important mistake in one lemma, and for their stimulating suggestions on how to better present substantial parts of this paper.

References

[1] A. Asperti, 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*, volume 3839 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 2004.

[2] A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 39(2), August 2007. Special Issue on User Interfaces for Theorem Proving.

[3] G. Bancerek and P. Rudnicki. Information retrieval in MML. In A. Asperti, B. Buchberger, and J. 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] E. Bishop and D. Bridges. *Constructive Analysis*. Springer, 1985.

[5] H. Geuvers and G. I. Jojgov. Open proofs and open terms: A basis for interactive logic. In J. Bradfield, editor, *Computer Science Logic:* 16*th International Workshop, CSL* 2002, volume 2471 of *Lecture Notes in Computer Science*, pages 537–552. Springer-Verlag, January 2002.

[6] M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors. *Towards Mechanized Mathematical Assistants,* 14*th Symposium, Calculemus* 2007*,* 6*th International Conference, MKM* 2007*, Hagenberg, Austria, June* 27–30*,* 2007*, Proceedings*, volume 4573 of *Lecture Notes in Computer Science*. Springer, 2007.

[7] R. Mines, F. Richman, and W. Ruitenburg. *A Course in Constructive Algebra*. Springer, 1 edition, December 1987.

[8] C. Muñoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory*. PhD thesis, INRIA, November 1997.

[9] H. Naciri and L. Rideau. Formal mathematical proof explanations in natural language using mathml: An application to proofs in arabic. In *Proceedings of MathML International Conference*, 2002.

[10] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[11] C. Sacerdoti Coen and S. Zacchiroli. Efficient ambiguous parsing of mathematical formulae. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Proceedings of Mathematical Knowledge Management* 2004, volume 3119 of *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, 2004.

[12] C. Sacerdoti Coen and S. Zacchiroli. Spurious disambiguation error detection. In *Proceedings of Mathematical Knowledge Management* 2007, volume 4573 of *Lecture Notes in Artificial Intelligence*, pages 381–392. Springer-Verlag, 2007.

[13] Zentralblatt MATH. `http://www.emis.de/ZMATH/`.

Claudio Sacerdoti Coen
Department of Computer Science
University of Bologna
Mura Anteo Zamboni, 7
I-40127 Bologna
Italy
e-mail: `sacerdot@cs.unibo.it`

Stefano Zacchiroli
Université Paris Diderot
Laboratoire PPS, UMR 7126
175 Rue du Chevaleret
F-75013 Paris
France
e-mail: `zack@pps.jussieu.fr`

Received: December 16, 2007.
Revised: June 4, 2008.
Accepted: June 12, 2008.