Rating Disambiguation Errors[*]

Andrea Asperti and Wilmer Ricciotti

Department of Computer Science, University of Bologna
{asperti,ricciott}@cs.unibo.it

Abstract. Ambiguous notation is a powerful tool developed to deal with the complexity of mathematics without sacrificing clarity or conciseness. In the mechanized parsing of ambiguous terms, a disambiguation algorithm can be used to provide the system with the intelligence necessary to select valid interpretations for the overloaded symbols received in input.

Disambiguation works by means of an incremental analysis of the input term, progressively discarding all invalid interpretations. As a result, if the input term cannot be disambiguated, many errors will be produced, only a handful of which are truly meaningful to the user.

In this paper, we improve the existing technique to classify disambiguation errors by introducing a new heuristic to sort errors from the most meaningful to the least, showing that it can be implemented in a natural way in the existing disambiguation algorithm. We also describe a neat interface to present disambiguation errors to the user, suitable for the use in interactive theorem proving applications.

1 Introduction

One of the most notable features of mathematical notation is ambiguity: for instance, it is possible to overload operators, as long as the intended interpretation of a given formula can be inferred from its context. On the other hand, results to be stored in formal libraries of interactive provers [13], need to be in an unambiguous form.

Ambiguous notation serves an important purpose, hiding redundant information and providing a standardized lexicon through which mathematicians can communicate more easily. It is therefore important that tools for mechanized mathematics be able to bridge the gap between mathematical notation and the unambiguous formalism used by the system. Hence, all interactive theorem provers address the issue of ambiguous notation in some way. Some provers try to resolve the ambiguity at parsing time by means of a deterministic system of interpretation scopes (an approach used in Coq). A more sophisticated technique popularized by the Haskell programming language and extended to theorem proving first in Isabelle, then in Coq and Matita, is that of *type classes* [12]; in this case every notation is associated unambiguously to a certain type class and

[*] The project CerCo acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881.

C. Hawblitzel and D. Miller (Eds.): CPP 2012, LNCS 7679, pp. 240–255, 2012.

parsed as such; each type class provides several possible overloaded instances of the notation, allowing the intended one to be selected later on during semantic analysis.

A different approach supported in Matita [3] since the very beginning, allows the parser to produce an ambiguous abstract syntax tree from ambiguous notation, which is later fed to a component called *disambiguator*, in charge of deriving all the possible well-typed interpretations. This paper is an ideal continuation of two works by Sacerdoti Coen and Zacchiroli about the implementation of disambiguators [9,10]. In particular, we focus on disambiguation errors and how to predict how much informative they are going to be for the user.

While the disambiguator approach has a great flexibility, it turns out that error reporting in this setting has an even more critical status than in type-checking. Usually the disambiguation process fails because of a single mistake by the user: in such a case, a human reader is often able to infer the intended meaning of the user input and spot the mistake; from the system perspective, however, each combination of the possible interpretations of ambiguous parts of the input will yield a different error. That is to say, when the disambiguation process fails, we may be left with dozens of failing interpretations and no clear way to recognize the interesting ones.

Our goal is to allow users to recover their intended interpretation from the heap of failures in order to understand what went wrong. We do this by providing a heuristic criterion to rate how likely an interpretation is to be the one intended by the user. This criterion is implemented as a straightforward addition to Sacerdoti Coen and Zacchiroli's efficient disambiguation algorithm.

The structure of the paper is as follows: Section 2 deals with the notion of disambiguation and provides definitions that will be used in the rest of the paper; Section 3 recalls the efficient disambiguation, which will serve as a basis for the rest of this work. Our contributions are described beginning in Section 4, where we discuss some drawbacks of a previous technique to detect so called *spurious* errors; Section 5 presents an entirely novel criterion providing a quantitative analysis of how much errors can be expected to be relevant and discusses an extension of the disambiguation algorithm yielding this finer classification; Section 6 presents a user interface for reporting disambiguation errors in an orderly way. Both the algorithm and the user interface have been implemented and are used as part of the Matita web application[1].

2 The Notion of Disambiguation

In the handbook approach to compilation, the semantic analysis phase is in charge of associating to an abstract syntax tree (AST) at most one interpretation (or no interpretation at all in the case of a static semantic error). In the context of the formalization of mathematics, however, we are willing to allow the user to employ the standard, ambiguous mathematical syntax, with the maximum degree of flexibility. In this scenario, the ambiguity of concrete syntax is

[1] See `http://matita.cs.unibo.it/matitaweb.shtml`

transferred by the parser to an *ambiguous AST*, where by ‘ambiguous’ we mean that it admits more than one interpretation. The process associating an AST to the set of all its valid interpretations is called *disambiguation*. We will now make these concepts more formal.

In this discussion, we will provide an abstract presentation to avoid sticking to a specific syntax or formalism. We will call *AST* a tree built from primitive nodes in the set $\mathbb{S}$; the set of ASTs will be denoted by $\mathbb{A}$. For every node in $s \in \mathbb{S}$ there exists an associated *interpretation domain* $\mathbb{D}_s$ which we will also regard as primitive.

A node in an AST can be either *ambiguous* or *disambiguated*. An ambiguous node is a bare primitive node (obtained by parsing ambiguous concrete syntax); a disambiguated node is a pair $\langle s, d \rangle$ such that d is an interpretation in $\mathbb{D}_s$. Disambiguated nodes can be an intermediate product of disambiguation, but can also result from parsing of unambiguous concrete syntax (which in turn can be a deliberate choice of the user, or the refinement of ambiguous user syntax by means of disambiguation feedback). A given node may occur multiple times in an AST, therefore we will denote occurrences (i.e. positions in an AST) by $n, n', \ldots$. The lookup operation returning the node at position n in the AST t is denoted $t(n)$.

We call an AST containing occurrences of ambiguous nodes an *ambiguous AST*, and an AST containing only disambiguated nodes an *unambiguous AST*. The set of unambiguous ASTs is denoted $\overline{\mathbb{A}}$. The set of the occurrences of ambiguous nodes in an AST t is called *domain* of t and denoted $dom(t)$. A substitution for an AST t is a finite partial map from the domain of t to disambiguated nodes, such that an occurrence of an ambiguous node s is mapped to a corresponding disambiguated node $\langle s, d \rangle$. The substitution map is lifted from nodes to ASTs in the obvious way. We say that an AST t' is an instance of another AST t, or equivalently that t is a generalization of t' (notation: $t \preceq t'$) if there exists a substitution σ such that $t' = t\sigma$. The following property follows immediately.

Lemma 1. $\preceq$ *is a partial order relation.*

Our intuition tells us that the semantics of an unambiguous AST is unique and that the semantics of an ambiguous AST is the union of the semantics of all its unambiguous instances. For our purposes, we can identify the set of semantics of ASTs with the set of unambiguous ASTs $\overline{\mathbb{A}}$. The semantics of ASTs is then formalized as follows:

Definition 1. *The semantics of ASTs* $[\![\cdot]\!] : \mathbb{A} \to \wp(\overline{\mathbb{A}})$ *is a function associating to any AST the set of all its unambiguous instances*

$$[\![t]\!] = \{t' \in \overline{\mathbb{A}} : t \preceq t'\}$$

Since every unambiguous AST is the only instance of itself, according to the above definition, the semantics of an unambiguous AST is a singleton, regardless of the intepretations of its nodes, as expected. However, this definition is in a sense too loose to be of any use, because it says nothing about the coherence of the interpretations we chose. The most obvious example of incoherence is ill-typedness: if our choices yield an ill-typed AST, that AST must be considered meaningless and thus discarded. In our abstract context, we do not employ any concrete notion of well-typedness, but rather we will assume the existence of an oracle $\mathcal{R}$ deciding whether an AST is *valid* or not: the oracle will return ✓ in the former case, and an informative error message otherwise.

Definition 2. *The disambiguation function* $\mathcal{D} : \mathbb{A} \to \wp(\overline{\mathbb{A}})$ *maps any AST to the set of all its valid interpretations*

$$\mathcal{D}(t) = \{t' \in [\![t]\!] : \mathcal{R}(t') = \checkmark\}$$

A trivial algorithm implementing $\mathcal{D}$ consists of computing the set of all ground instances of the AST to be disambiguated and then filtering through the oracle $\mathcal{R}$. This technique is clearly inefficient, since the number of ground instances of an AST is exponential in the number of its ambiguous nodes.

3 A Disambiguation Algorithm

Efficient implementations of disambiguation operate by incrementally instantiating the original AST, immediately pruning those partial instances which can already be shown to be invalid by the oracle, and iterating the process until no ambiguous nodes are left. Early pruning leads to enormous performance improvements.

In order for this kind of implementation to work, we must relax the definition of $\mathcal{R}$ to allow it to take ambiguous ASTs as input too. In this case, we want $\mathcal{R}$ to always return ✓ if the input AST can be instantiated to a valid unambiguous AST, because only invalid instances should be pruned. When this condition is satisfied, we clearly want as many ASTs as possible to be rejected, to minimize the number of incremental instantiations.

In summary, we would like the oracle to return an error if and only if all the instances of the input AST are invalid. However, in general we are not able to identify all the ambiguous ASTs not admitting valid instantiations: this happens for two reasons:

- while some errors are located in disambiguated parts of the AST and can be immediately recognized as such, other errors, located in ambiguous parts, can only be recognized after disambiguation of some nodes: to detect such errors, it is necessary to first instantiate some ambiguous nodes, making an efficient implementation of $\mathcal{R}$ impossible;
- in the case where all instances of the input AST are invalid, $\mathcal{R}$ should return a single message explaining why all such instances cannot be accepted: unfortunately, it may be the case that all the instances are invalid, but each of them is invalid for a different reason.

Our disambiguation algorithm will therefore assume that ASTs rejected by $\mathcal{R}$ are invalid for all possible instantiations, but the inverse implication will not hold in general.

Property 1. Given a (possibly ambiguous) AST t, if $\mathcal{R}(t)$ returns an error, then all instances of t are not valid.

Property 2. If $\mathcal{R}(t)$ returns an error, that error is meaningful for all instances of t.

Given a set of ASTs Σ, we define the notations $\Sigma^{✓}$ and $\Sigma^{✗}$ as follows:

$$\Sigma^{✓} = \{t | t \in \Sigma \wedge \mathcal{R}(t) = ✓\}$$
$$\Sigma^{✗} = \{\langle t, \mathcal{R}(t)\rangle | t \in \Sigma \wedge \mathcal{R}(t) \neq ✓\}$$

Therefore, $\Sigma^{✓}$ will contain the subset of all the ASTs that are still valid, while $\Sigma^{✗}$ will contain all the invalid ASTs in Σ paired with their error messages.

We can now show the "efficient" disambiguation algorithm originally presented in [9]. It follows the aforementioned criterion of incremental instantiation of the input AST.

```
procedure disambiguate($t$)
begin
  $\Sigma \leftarrow \{t\}^{✓}$; $\Omega \leftarrow \{t\}^{✗}$;
  while ($\Sigma \neq \emptyset \wedge$ next($\Sigma$) $\neq ✗$)
    begin
    $n \leftarrow$ next($\Sigma$);
    $\Delta \leftarrow \{u[n \mapsto \langle u(n), d\rangle] | u \in \Sigma, d \in \mathbb{D}_{u(n)}\}$;
    $\Sigma \leftarrow \Delta^{✓}$; $\Omega \leftarrow \Omega \cup \Delta^{✗}$;
    end
  return $\Sigma, \Omega$
end
```

The algorithm maintains a set Σ of partially disambiguated instances of the input AST t, ensuring that they share the same domain and have not been rejected by $\mathcal{R}$ yet (Σ is initialized as the singleton $\{t\}$, except when t is immediately rejected by $\mathcal{R}$, in which case it is initialized as the empty set, leading to failure). The algorithm is parametric on a procedure `next` taking as input a set of ASTs sharing the same domain and returning an element of that domain (an ambiguous node occurrence), or ✗ if no ambiguous node is left. In the **while** cycle, we choose a node `next`(Σ) from the domain of Σ and instantiate it in all possible ways, obtaining a new set Δ. We then filter out invalid instances obtaining a new set Σ to continue iteration. The cycle stops when either all ambiguous nodes have been instantiated (`next`(Σ) $=$ ✗), meaning the disambiguation was successful, or when Σ is empty, meaning all the instances of t are invalid. Ω is used to collect all the errors produced by $\mathcal{R}$.

To discuss the properties of the algorithm, we introduce a notation to refer to the value of a variable at a specific iteration of a while cycle: we will use v_0 to denote the value of a variable v before the first iteration, and v_i to denote its value at the end of the i-th iteration.

Lemma 2. *In an execution of* `disambiguate`(t)*, for all $i \geq 1$, each AST t' in Δ_i is such that $|dom(t')| = |dom(t)| - i$*

Proof. By induction on i: when $i = 1$, we instantiate a single ambiguous node from the original AST t, and the thesis follows easily; when $i > 1$, we know that Σ_{i-1} is a subset of Δ_{i-1}, so by induction hypothesis every AST in it has a domain of cardinality $|dom(t)| - i + 1$: to compute Δ_i, we instantiate one ambiguous node more, therefore getting a domain of cardinality $dom(t) - i$, as needed.

Lemma 3. *The* `disambiguate` *algorithm terminates after at most* $|dom(t)|$ *executions of the* **while** *cycle.*

Proof. Trivial, since at each iteration, either Σ becomes empty (and the algorithm terminates immediately), or the number of ambiguous nodes in Σ decreases by one (easily proved by means of Lemma 2), eventually reaching 0 and falsifying the guard of the **while** cycle.

Lemma 4. *Given an AST t, for all $t' \in \overline{\mathbb{A}}$ such that $t \preceq t'$ and for all $i \leq |dom(t)|$, there exists t'' such that $t \preceq t'' \preceq t'$ and either $t'' \in \Sigma_i$ and $\mathcal{R}(t'') = \checkmark$, or $t'' \in \Omega_i$ and $\mathcal{R}(t'') \neq \checkmark$.*

Proof. We proceed by induction on i. If $i = 0$, then we choose $t'' = t$ and the statement is satisfied. If $i > 0$, by induction hypothesis there exists t''' such that $t \preceq t''' \preceq t'$ and either $\mathcal{R}(t''') = \checkmark$ and $t''' \in \Sigma_{i-1}$ or $\mathcal{R}(t''') \neq \checkmark$ and $t''' \in \Omega_{i-1}$. If $t''' \in \Omega_{i-1}$, we choose $t'' = t'''$ and get the thesis since $\Omega_{i-1} \subseteq \Omega_i$. If $t''' \in \Sigma_{i-1}$, we choose $t'' = t'''[n_i \mapsto t'(n_i)]$: clearly $t \preceq t'' \preceq t'$ by definition; furthermore, $t'' \in \Delta_i$. If $\mathcal{R}(t'') = \checkmark$, then we proved that $t'' \in \Sigma_i$; otherwise, $t'' \in \Omega_i$. In both cases, the thesis holds.

The two following theorems assert the soundness of the algorithm, respectively saying that the Σ returned by the algorithm is the set of all valid disambiguated instances of the input, and that all invalid disambiguated instances of the input have an invalid generalization in the Ω returned by the algorithm (or equivalently, that Ω contains an error explaining why that AST is invalid).

Theorem 1. *The set Σ returned by* `disambiguate`(t) *is equal to $\mathcal{D}(t)$.*

Proof. We prove that the algorithm returns a Σ such that $\Sigma \subseteq \mathcal{D}(t)$ and $\Sigma \supseteq \mathcal{D}(t)$.

- **$\Sigma \subseteq \mathcal{D}(t)$:** Since only valid ASTs ever enter Σ and the cycle only terminates when the domain of ASTs in Σ is empty (or $\Sigma = \emptyset$), Σ only contains unambiguous valid ASTs, thus $\Sigma \subseteq \mathcal{D}(t)$.
- **$\Sigma \supseteq \mathcal{D}(t)$:** By lemmata 3 and 4, we can prove that at the last execution of the **while** cycle, for all $t' \in \mathcal{D}(t)$, there exists $t'' \preceq t'$ such that $t'' \in \Sigma$; this implies Σ is not empty and t'' is unambiguous (otherwise, the cycle would execute another time). It thus follows that $t'' = t'$ and, consequently, $t' \in \Sigma$.

Theorem 2. *Let t be an AST and Ω the error collection returned by* `disambiguate`(t)*. Then:*

1. *given an AST t' and an error message e such that $\langle t', e\rangle \in \Omega$, for all t'' such that $t' \preceq t''$ we have $\mathcal{R}(t'') \neq \checkmark$;*
2. *for all t' such that $t \preceq t'$ and $t' \notin \mathcal{D}(t)$, there exists an AST t'' such that $t \preceq t'' \preceq t'$ and $\langle t'', \mathcal{R}(t'')\rangle \in \Omega$.*

Proof. Part 1 is trivial (only invalid ASTs enter Ω, and by Property 1 of $\mathcal{R}$, all their instances must also be invalid).
To prove part 2, let k be the number of iterations after which the algorithm terminates. We have $\Sigma = \Sigma_k$ and $\Omega = \Omega_k$: thus by using Lemma 4 with $i = k$, we get a t'' such that $t \preceq t'' \preceq t'$ and either $t'' \in \Sigma$ or $t'' \in \Omega$. In the first case by Theorem 1, $t'' \in \mathcal{D}(t)$. This implies t'' is also unambiguous, thus from $t'' \preceq t'$ we also get $t'' = t'$. But then $t' \in \mathcal{D}(t)$, which falsifies our hypothesis. If instead $t'' \in \Omega$, the thesis follows immediately.

The choice of `next` influences both efficiency and the errors returned by the algorithm. Since in general the interpretation of one node constrains the interpretation of all its children, while the constraints imposed by a node to its parent and siblings are much less restrictive (if they exist at all), and since, as we noted, the constraints we imposed on the validity test are such that no reasonable implementation is allowed to consider the children of an ambiguous node, the `next` function should be implemented by visiting the nodes nearest to the root first, as in a pre-order or level-order (breadth first) traversal.

4 Spurious Errors

When disambiguation is successful, it is generally going to return a small set of choices (most usually, just one), all of which are meaningful: in this case, the errors produced during the disambiguation process can be ignored. If disambiguation fails, however, we want to provide the user with information on what went wrong, by returning to him the Ω set computed by `disambiguate`. This set can easily contain a large amount of invalid interpretations that to the system are equally wrong, even though the user is likely to have committed just one mistake; the vast majority of errors produced by the disambiguation algorithm is *spurious*: they do not correspond to a user error, but are only a technical means to drive the disambiguation algorithm to the correct interpretation (if it exists).

In order to provide the user with more accurate information, a heuristic criterion to distinguish genuine errors from spurious errors was introduced in [10]:

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 located in F.*

The meaning of the criterion is clear: the system should try to interpret the input AST as much as possible and keep as real errors only those that are "unrecoverable", i.e. those for which no alternative valid interpretation exists.

However Criterion 1 lacks a clear implementation, especially because an efficient algorithm (like the one presented in the previous section) does not consider all possible interpretations (and consequently all possible errors) of the input AST. For this reason, the following restriction of the criterion[2], allowing a more obvious implementation, has been suggested in the aforementioned paper (here rephrased to make it agree with our simpler presentation of Section 2):

Criterion 2 (Draconian Spurious Error Detection). *During the disambiguation of t, an error message relative to an instance t' of t is* spurious *iff there exists an occurrence $n \in dom(t)$ and an alternative instance t'' of t such that:*

1. $t'(n) \neq t''(n)$;
2. t', t'' *are both unambiguous on all n' preceding n in pre-order;*
3. $\mathcal{R}(t'') = \checkmark$;

Let us point out the heuristic status of the criterion: the causal relation between the interpretation of a node and an error is informal and cannot be grasped accurately by any disambiguation algorithm. In practice, this means that we may sometimes find genuine errors classified as spurious.

A second suspicious point is the second requirement of the criterion, explicitly stating the traversal algorithm to be used in the disambiguation process. The choice of a pre-order traversal is sensible but arbitrary (a breadth-first traversal also enjoys good properties with respect to the disambiguation algorithm). Indeed, the classification of spurious errors is dependent on the order in which subterms are considered: in some cases, opposite classifications can be performed on semantically equivalent terms, differing only by a commutativity property. The next example shows this anomaly.

Example 1. Consider the concrete syntax

$$(\alpha + \beta) + (\alpha + \gamma) = (\boldsymbol{x} + \boldsymbol{y}) + (\boldsymbol{x} + \boldsymbol{z})$$

where α, β, γ are interpreted to be in $\mathbb{R}$ and $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ in $\mathbb{R}^2$. Assume that the interpretation domain for $+$ contains elements `plusR` and `plusV` representing respectively sum on real scalars and vectors; similarly, the domain for $=$ will contain interpretations `eqR` and `eqV` for equality on scalars and vectors. In a disambiguation algorithm classifying spurious errors, the nodes will be considered and assigned interpretations in the pre-order sequence; disambiguation of the above formula will fail because the two sides of the equality have different types. Immediately before failure, the only instance of the original AST still being processed is

$$(\alpha +_{\texttt{plusR}} \beta) +_{\texttt{plusR}} (\alpha +_{\texttt{plusR}} \gamma) =_{\texttt{eqR}} (\boldsymbol{x} \underline{+} \boldsymbol{y}) +_{\texttt{plusR}} (\boldsymbol{x} + \boldsymbol{z})$$

where the symbol being considered is the underlined one. Both interpretations in its domain will fail, returning errors:

[2] The criterion is called *draconian* because it recognizes as spurious more errors than the *prudent* criterion also proposed in [10]. It is not possible to discuss both the criteria here due to space contraints, but the considerations we are going to draw apply to the prudent criterion as well.

- $(\alpha +_{\texttt{plusR}} \beta) +_{\texttt{plusR}} (\alpha +_{\texttt{plusR}} \gamma) =_{\texttt{eqR}} (\boldsymbol{x} +_{\texttt{plusR}} \boldsymbol{y}) +_{\texttt{plusR}} (\boldsymbol{x} + \boldsymbol{z})$: $\boldsymbol{x}$ has type vector but is here used as a scalar
- $(\alpha +_{\texttt{plusR}} \beta) +_{\texttt{plusR}} (\alpha +_{\texttt{plusR}} \gamma) =_{\texttt{eqR}} (\boldsymbol{x} +_{\texttt{plusV}} \boldsymbol{y}) +_{\texttt{plusR}} (\boldsymbol{x} + \boldsymbol{z})$: $\boldsymbol{x} +_{\texttt{plusV}} \boldsymbol{y}$ has type vector but is here used as a scalar

After generating those errors, no valid interpretation is left and the algorithm will stop. Errors produced by the disambiguation of preceding nodes in the preorder sequence will be flagged as spurious by the draconian criterion, including the errors precluding the `plusV` interpretation for $\alpha + \beta$:

- $(\alpha +_{\texttt{plusR}} \beta) +_{\texttt{plusV}} (\alpha + \gamma) =_{\texttt{eqV}} (\boldsymbol{x} + \boldsymbol{y}) + (\boldsymbol{x} + \boldsymbol{z})$: $\alpha +_{\texttt{plusR}} \beta$ has type scalar but is here used as a vector
- $(\alpha +_{\texttt{plusV}} \beta) +_{\texttt{plusR}} (\alpha + \gamma) =_{\texttt{eqR}} (\boldsymbol{x} + \boldsymbol{y}) + (\boldsymbol{x} + \boldsymbol{z})$: α has type scalar but is here used as a vector

If on the contrary we consider the symmetric equation

$$(\boldsymbol{x} + \boldsymbol{y}) + (\boldsymbol{x} + \boldsymbol{z}) = (\alpha + \beta) + (\alpha + \gamma)$$

the disambiguation will proceed until the only interpretation left is:

$$(\boldsymbol{x} +_{\texttt{plusV}} \boldsymbol{y}) +_{\texttt{plusV}} (\boldsymbol{x} +_{\texttt{plusV}} \boldsymbol{z}) =_{\texttt{eqV}} (\alpha \underline{+} \beta) +_{\texttt{plusV}} (\alpha + \gamma)$$

The system will then return the following errors as meaningful:

- $(\boldsymbol{x} +_{\texttt{plusV}} \boldsymbol{y}) +_{\texttt{plusV}} (\boldsymbol{x} +_{\texttt{plusV}} \boldsymbol{z}) =_{\texttt{eqV}} (\alpha +_{\texttt{plusR}} \beta) +_{\texttt{plusV}} (\alpha + \gamma)$: $\alpha +_{\texttt{plusR}} \beta$ has type scalar but is here used as a vector
- $(\boldsymbol{x} +_{\texttt{plusV}} \boldsymbol{y}) +_{\texttt{plusV}} (\boldsymbol{x} +_{\texttt{plusV}} \boldsymbol{z}) =_{\texttt{eqV}} (\alpha +_{\texttt{plusV}} \beta) +_{\texttt{plusV}} (\alpha + \gamma)$: α has type scalar but is here used as a vector

In this case, the errors about $\boldsymbol{x} + \boldsymbol{y}$ will be flagged as spurious, showing that the notion of spuriousness is not stable under minor syntactic modifications of the input. Arguably, in both versions of the equation, $\boldsymbol{x} + \boldsymbol{y}$ and $\alpha + \beta$ are equally wrong (since they are both responsible for the whole equation being rejected).

5 Error Rating

Example 1 shows that in some cases an error is classified as spurious only because of its position in the formula to be disambiguated, even though a user would recognize it as a real error. We attribute this anomaly to the extreme coarseness of the distinction spurious/non-spurious: if we could establish a *rating* criterion capable of distinguishing more than two degrees of significance, it would be possible to present errors to the user so that the most meaningful come first, followed by the less meaningful in a gradual fashion.

Our intent is therefore to understand what are the features of an error that is meaningful to the user. Typically, a meaningful error tells the user something interesting by contrasting large valid subterms with a single incoherent node; according to this point of view, an erroneous AST should be rated depending on its valid generalizations. Let us call *maximal valid generalization* of a (possibly invalid) AST t an AST t' that is a valid generalization of t and such that all other valid generalizations of t have fewer intepreted nodes than t'. The more nodes are interpreted by t', the better the rating of t should be.

Essentially, when rating erroneous interpretations, we want to privilege those that are closer to being valid because their maximal valid generalizations have more interpreted nodes. This requirement is expressed by the following criterion.

Criterion 3 (Error rating criterion). *Given two erroneous partial instances t_1 and t_2 of the same input AST t, the error for t_1 is* less likely *than the error in t_2 (notation: $t_1 \trianglelefteq t_2$) iff there exists a valid generalization of t_2 whose domain is smaller than the domain of all valid generalizations of t_1. Formally:*

$$t_1 \trianglelefteq t_2 \iff \begin{pmatrix} \exists t'_2 \preceq t_2 : \mathcal{R}(t'_2) = \checkmark \wedge \\ \forall t'_1 \preceq t_1 : \mathcal{R}(t'_1) = \checkmark \implies |dom(t'_1)| \geq |dom(t'_2)| \end{pmatrix}$$

We can also express the rating of an AST by means of a natural number using the following rating function.

Definition 3 (Rating function). *The rating of an AST t (notation: $\varrho(t)$) is defined as the smallest cardinality of the domains of all its valid generalizations. Formally:*

$$\varrho(t) \triangleq \min_{t' \preceq t \wedge \mathcal{R}(t') = \checkmark} |dom(t')|$$

According to this definition, the lower the rating, the more likely an AST is to be what the user originally intended. In particular, a valid unambiguous AST receives a rating of 0.

Lemma 5. *For all t_1, t_2, $t_1 \trianglelefteq t_2$ iff $\varrho(t_1) \geq \varrho(t_2)$.*

Proof.

$\Longrightarrow$**:** By the definition of likelihood according to Criterion 3, there exists a valid generalization t'_2 of t_2 such that for all valid generalizations t'_1 of t_1, $|dom(t'_1)| \geq |dom(t'_2)|$; on the other hand, the definition of ϱ implies that $\varrho(t_1) = |dom(t''_1)|$ for some t''_1. By taking $t'_1 = t''_1$, we have $\varrho(t_1) = |dom(t''_1)| \geq |dom(t'_2)|$, and by the definition of ϱ, $|dom(t'_2)| \geq \varrho(t_2)$. Then the thesis holds by transitivity of $\geq$.

$\Longleftarrow$**:** By the definition of ϱ, let t'_2 be a valid generalization of t_2 such that $|dom(t'_2)| = \varrho(t_2)$. Then, again by definition of ϱ, we know that for all valid generalizations t'_1 of t_1, $|dom(t'_1)| \geq \varrho(t_1)$; then combining the hypothesis we get $|dom(t'_1)| \geq |dom(t'_2)|$; by the definition of likelihood, this yields the thesis.

Corollary 1. $\trianglelefteq$ *is a total order relation.*

Proof. A consequence of $\geq$ being a total order relation, by means of Lemma 5.

The rating of an AST provides a formal, yet very natural way of assessing the significance of an interpretation, even when it is not valid. Anyway, the definition we gave does not provide an immediate method for computing the rating of an AST: enumerating the generalizations of a given AST until a valid one is found could be computationally expensive. Luckily, it is possible to generalize the efficient disambiguation algorithm so that it returns errors sorted depending on their rating.

```
procedure disambiguate_and_rate($t$)
begin
  $\Sigma \leftarrow \{t\}^{\checkmark}$;
  if $\Sigma \neq \emptyset$ then $\Omega \leftarrow []$ else $\Omega \leftarrow [\{t\}^{\text{✗}}]$;
  while $(\Sigma \neq \emptyset \wedge \texttt{next}(\Sigma) \neq \text{✗})$
    begin
    $n \leftarrow \texttt{next}(\Sigma)$;
    $\Delta \leftarrow \{u[n \mapsto \langle u(n), d\rangle] | u \in \Sigma, d \in \mathbb{D}_{u(n)}\}$;
    $\Sigma \leftarrow \Delta^{\checkmark}$;
    if $\Delta^{\text{✗}} \neq \emptyset$ then $\Omega \leftarrow \Delta^{\text{✗}} :: \Omega$;
    end
  return $\Sigma, \Omega$
end
```

It is easy to show that the Σ and Ω returned by disambiguate_and_rate contain exactly the same ASTs as those returned by disambiguate, thus the soundness of this algorithm descends from that of the other one. The whole difference between the two lies not in the content of Ω, but in its structure. First it is not a set anymore, but a list of sets; each element in the list, which we will call *error frame*, is obtained from the failing interpretations in a certain Δ_i: thus all the failing interpretations have the same domain, and are failing after the instantiation of the same node. This allows us to prove the following theorem.

Theorem 3. *The list Ω returned by* disambiguate_and_rate *is sorted by decreasing likelihood, that is, if $\Omega = [\omega^1, \omega^2, \ldots, \omega^m]$, then for all $t_i \in \omega^i$ and $t_j \in \omega^j$ where $i \leq j$, $t_j \trianglelefteq t_i$.*

Proof. It is easy to prove that $\Omega = [\Delta^{\text{✗}}_{k_1}, \Delta^{\text{✗}}_{k_2}, \ldots, \Delta^{\text{✗}}_{k_m}]$, such that $k_i > k_j$ iff $i < j$. Therefore, we will prove that if $i < j$, then $t_i \in \Delta^{\text{✗}}_{k_i}$ and $t_j \in \Delta^{\text{✗}}_{k_j}$ are such that $t_j \trianglelefteq t_i$. We know from the definition of the algorithm that for all h, each $t' \in \Delta_h$ is obtained from some AST t'' in Σ_{h-1} by instantiating a single ambiguous node, and that each AST in Σ_{h-1} is valid. This implies $\varrho(t_i) = |dom(t_i)| + 1$ and $\varrho(t_j) = |dom(t_j)| + 1$. By Lemma 2, we prove that $|dom(t_i)| = |dom(t)| - k_i$ and $|dom(t_j)| = |dom(t)| - k_j$; since $k_i > k_j$, we prove that $\varrho(t_j) \geq \varrho(t_i)$ and by Lemma 5, $t_j \trianglelefteq t_i$.

We still have to decide whether we should prefer, as the implementation of next, an in-order visit or a level-order visit. The issue cannot be addressed exclusively on the basis of efficiency, since it is easy to show ASTs on which the former choice outperforms the latter, and vice-versa. However our tests indicate that our algorithm behaves better when employed with a level-order visit, in the sense that the error ordering produced is closer to the expected one. This is probably due to the fact that the interpretations of node occurrences located nearer to the leaves (and consequently the related error messages) tend to be more significant from the user point of view than those of nodes located nearer to the root of the AST. An in-order visit, on the other hand, alternates deep and shallow node occurrences and should therefore be avoided.

6 Error Reporting

An attractive option for reporting disambiguation errors is to let the user disambiguate manually individual error locations by means of a point-and-click interface, until the amount of disambiguation errors and their quality is judged to be satisfying. Our original intention was to implement such an interface; however our experience as users of interactive theorem provers tells us that the user is frequently incapable of guessing where disambiguation went wrong. The reason is that the combination of overloading with other advanced features of theorem provers, especially dependent types and coercive subtyping, make the disambiguation process quite intricate from a human perspective. Since the user is reviewing errors precisely for the purpose of understanding what went wrong, it is highly likely that a single interaction at a random error location with a point-and-click interface will not be clarifying at all, leading to frustration.

Our rating algorithm was designed having in mind that in this quite unusual case, the system knows better than the user which errors are the best candidates to being genuine, thanks to the rating criterion we proposed. We will employ it to design a user interface abiding by the following requirements:

- errors should be grouped according to their location;
- users should see those errors that are more meaningful first;
- the number of errors shown at the same time should be manageable.

Classification of spurious errors in the style of Section 4 only partially respects these requirements: in particular, errors cathegorized as non-spurious respect the requirements, but the other ones do not. As we saw in Example 1, some errors that are morally non-spurious may be cathegorized as spurious too, meaning they will be intermingled with maybe dozens of uninteresting errors coming from mixed locations.

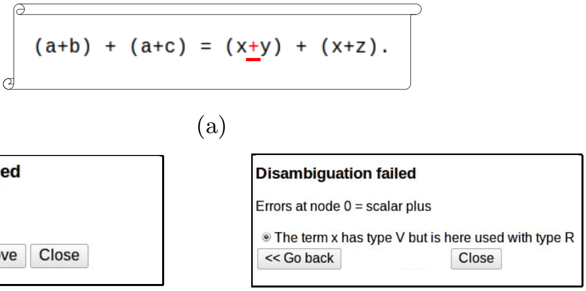
On the other hand, the ω^i sets in the list Ω returned by our algorithm seem to be good candidates for the use in an interface satisfying the aforementioned requirements. All the errors in the same set were produced at the same location in the AST, thus satisfying the first requirement; the ordering of the list Ω asserted by Theorem 3 provides a good basis for respecting the second requirement; finally, partitioning the errors in possibly more than just two sets (spurious and non-spurious) guarantees that our algorithm will perform better also with respect to the last requirement.

Each frame ω^i is obtained by interpreting one ambiguous node occurrence of all ASTs in Σ in all possible ways, filtering by means of $\mathcal{R}$ and keeping only the invalid instances. This has two consequences:

(a) not all interpretations possible for the node may be present in the frame, because some of them may only be shown to be invalid after the AST is instantiated further;

(b) some interpretations for the node may yield more than one invalid AST (in particular, up to one for each AST in Σ).

Given an interpretation for the node occurrence being considered in ω^i, we call the subset of ASTs instantiated with that interpretation a *slice* of ω^i. All the ASTs in the same slice have the same interpretation for the last node considered occurrence, but differ in the interpretation of at least another node.



```
(a+b) + (a+c) = (x+y) + (x+z).
```

(a)

Disambiguation failed

● vector plus
○ scalar plus

(b)

Disambiguation failed

Errors at node 0 = scalar plus

● The term x has type V but is here used with type R

(c)

Fig. 1. Error reporting interface

We propose a user interface composed of two panes. The first pane, called *interpretation pane* (Figure 1(b)), shows, for a given frame, a list of the interpretations of the node occurrence being considered yielding a disambiguation error; we use highlighting directly in the user input (Figure 1(a)) to show the node currently being considered. After choosing an interpretation in the interpretation pane, we are sent to an *error list pane* (Figure 1(c)), reporting the list of all the errors associated with that choice (i.e. a slice of the frame).

The activity diagram in Figure 2 shows the intended user interaction with this interface. The user will initially be shown a list of interpretations from the frame ω^1 (which is the most likely to contain meaningful errors): if the user intended the current node to be associated to one of those interpretations, he will choose it and immediately view the list of related errors; otherwise, he will use the *None of the above* button to switch the view to the following frame and iterate the procedure. After viewing the error list, the user can either be satisfied with the errors shown (when at least one of them explains what went wrong), or decide to go back to the interpretation pane to try selecting a different interpretation or inspecting another frame.

Example 2. Consider again the AST for the expression

$$(\alpha + \beta) + (\alpha + \gamma) = (\boldsymbol{x} + \boldsymbol{y}) + (\boldsymbol{x} + \boldsymbol{z})$$

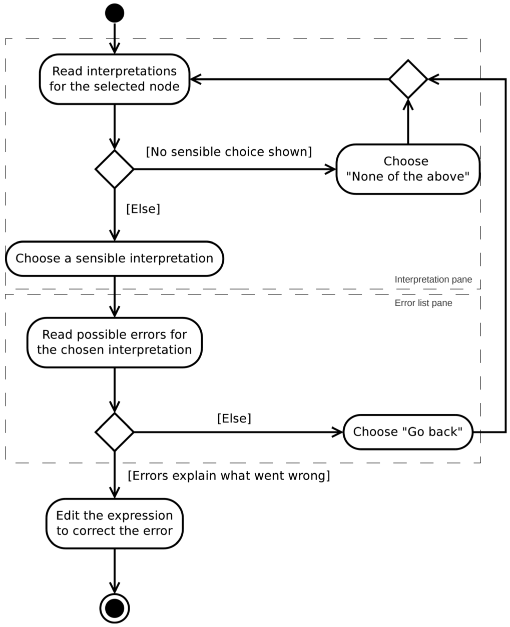

Read interpretations for the selected node
[No sensible choice shown]
Choose "None of the above"
[Else]
Choose a sensible interpretation
Interpretation pane
Error list pane
Read possible errors for the chosen interpretation
[Else]
Choose "Go back"
[Errors explain what went wrong]
Edit the expression to correct the error

Fig. 2. User interaction with the error reporting interface

from Example 1, using for the variables the same types as we used in that example. After using the `disambiguate_and_rate` algorithm with a level-order traversal of the AST, disambiguation will fail and the user interface will highlight a symbol in the original expression:

$$(\alpha+\beta)+(\alpha+\gamma)=(\boldsymbol{x}\underline{+}\boldsymbol{y})+(\boldsymbol{x}+\boldsymbol{z})$$

Among the two possibilities, in the interpretation pane we choose “vector plus”. The interface returns a single error:

- the term $\boldsymbol{x} +_{\texttt{plusV}} \boldsymbol{y}$ has type vector but is here used with type scalar.

We decide that the error is not informative to us: something went wrong in a different part of the AST, therefore we switch to the next error frame.

$$(\alpha+\beta)+(\alpha\underline{+}\gamma)=(\boldsymbol{x}+\boldsymbol{y})+(\boldsymbol{x}+\boldsymbol{z})$$

This time, the interpretation pane only shows the choice “vector plus”, which is not the intended one. We switch again to the next error frame:

$$(\alpha\underline{+}\beta)+(\alpha+\gamma)=(\boldsymbol{x}+\boldsymbol{y})+(\boldsymbol{x}+\boldsymbol{z})$$

After choosing the interpretation “scalar plus”, the error list pane will show us the error

- the term $\alpha +_{\texttt{plusR}} \beta$ has type scalar but is here used with type vector.

Now we realize that the expression we intended to write was

$$f(\alpha + \beta) + f(\alpha + \gamma) = (\boldsymbol{x} + \boldsymbol{y}) + (\boldsymbol{x} + \boldsymbol{z})$$

for a given function f from scalars to vectors. As we noted in Example 1, if we had used the spurious error classification, the error message about $\alpha + \beta$ would have been considered spurious, making it much more difficult to spot it.

7 Conclusions

There have been considerable research efforts devoted to improving the way type errors are reported to the user. Most works are concerned with type systems à la Hindley-Milner, whose type-inference algorithm has been shown to work in a way that is substantially different from how people commonly reason about types. Such works (among which we mention those by Jun, Michaelson, and Trinder [6], Hage and Heeren [5], and Stuckey, Sulzmann and Wasny [11]) are mainly interested in improving the error message that is returned to the user by means of several heuristics. Another relevant proposal by Rittri ([8]) is devoted to the design of an interactive interface that can help explain to the user the source of a type error.

Both kinds of work bear some resemblance to our implementation, in the spirit if not in the letter. Given the fact that the notion of disambiguation error is more general than that of type error, to improve the user experience we are urged to address a different kind of problem: how to help the user to discriminate between genuine errors and spurious errors.

We do this by means of a new algorithm that is capable of partitioning and sorting errors according to their significance. This constitutes a remarkable improvement over the previous technique of spurious error detection, which is only capable of distinguishing two degrees of significance. In addition to this, we also believe that our approach is based on a more understandable principle, which does not involve implementation details such as the order in which nodes in an AST are visited.

Even though the two approaches stem from different analyses of the problem, the solutions have more in common than expected: it can easily be shown that, when an in-order traversal is chosen for the `disambiguate_and_rate` algorithm, the error list returned by it is structured in such a way that the topmost frame contains the genuine errors and the rest of the list contains the spurious ones (according to the draconian criterion). In this sense, the rating of disambiguation errors is a refinement of the discrimination of spurious errors.

We kept our discussion considerably abstract, making only a few weak and plausible assumptions on the structure of ASTs and on the existence of a validity test $\mathcal{R}$; this allows our algorithm and interface to be used in a wide range of applications, including of course interactive theorem provers. In particular, in our implementation of the disambiguation algorithm in Matita, $\mathcal{R}$ operates by first translating the input AST to a term in the foundational language of the system – a variant of the Calculus of (Co)Inductive Constructions extended with metavariables in the style of [4,7] – in such a way that ambiguous nodes and their descendants are replaced by fresh metavariables (this ensures that Property 1 of Section 3 holds). The obtained term is then fed to the refinement facility of the system ([2]) for a typability check.

Our final remarks are about the user experience. The new disambiguation infrastructure has been developed recently as part of the web application version of Matita [1]. Our impression is that it provides a marked improvement over the past, especially because the interface is much less invasive. Due to this change being so new, there could still be room for improvement and we are committed to considering opinions and suggestions coming from the users of the system.

References

1. Asperti, A., Ricciotti, W.: A Web Interface for Matita. In: Jeuring, J., Campbell, J.A., Carette, J., Dos Reis, G., Sojka, P., Wenzel, M., Sorge, V. (eds.) CICM 2012. LNCS, vol. 7362, pp. 417–421. Springer, Heidelberg (2012)
2. Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: A bi-directional refinement algorithm for the calculus of (co)inductive constructions. LMCS 8(1) (2012)
3. Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: The Matita Interactive Theorem Prover. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 64–69. Springer, Heidelberg (2011)
4. Geuvers, H., Jojgov, G.I.: Open Proofs and Open Terms: A Basis for Interactive Logic. In: Bradfield, J.C. (ed.) CSL 2002. LNCS, vol. 2471, pp. 537–552. Springer, Heidelberg (2002)
5. Hage, J., Heeren, B.: Heuristics for Type Error Discovery and Recovery. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) IFL 2006. LNCS, vol. 4449, pp. 199–216. Springer, Heidelberg (2007)
6. Jun, Y., Michaelson, G., Trinder, P.: Explaining polymorphic types. Comput. J. 45(4), 436–452 (2002)
7. Muñoz, C.: A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory. PhD thesis, INRIA (November 1997)
8. Rittri, M.: Finding the source of type errors interactively. Technical report, Department of Computer Science, Chalmers University of Technology, Sweden (1993)
9. Sacerdoti Coen, C., Zacchiroli, S.: Efficient Ambiguous Parsing of Mathematical Formulae. In: Asperti, A., Bancerek, G., Trybulec, A. (eds.) MKM 2004. LNCS, vol. 3119, pp. 347–362. Springer, Heidelberg (2004)
10. Sacerdoti Coen, C., Zacchiroli, S.: Spurious disambiguation errors and how to get rid of them. Journal of Mathematics in Computer Science, Special Issue on MKM 2, 355–378 (2008)
11. Stuckey, P.J., Sulzmann, M., Wazny, J.: Improving type error diagnosis. In: Proceedings of 2004 ACM SIGPLAN Haskell Workshop, Haskell 2004, pp. 80–91. ACM, New York (2004)
12. Wenzel, M.: Type Classes and Overloading in Higher-Order Logic. In: Gunter, E.L., Felty, A.P. (eds.) TPHOLs 1997. LNCS, vol. 1275, pp. 307–322. Springer, Heidelberg (1997)
13. Wiedijk, F. (ed.): The Seventeen Provers of the World. LNCS (LNAI), vol. 3600. Springer, Heidelberg (2006)