Higher-Level Error Messages for OCaml Modules

Malo Monin
Intern
Univ. Rennes
Bruz, France

Florian Angeletti
Cambium
Inria
Paris, France

***Abstract*—Small mistakes in OCaml modules often result in huge error messages that are both hard to read and uninformative, although technically correct. Indeed, some modules contain many thousands of items. Detecting typos in field names can greatly improve error messages by providing a few "did you mean"-like suggestions to the user instead of a long list of missing fields. In order to display those hints in reasonable time, we have explored a way to compute them efficiently using an ingenious combination of commonly known data structures and algorithms.**

***Index Terms*—OCaml, modules, diffing, error messages**

I. Introduction

OCaml is a functional programming language created in 1996 [1]. One of its characteristic features is its module system, which lets the user organize programs into well-separated units of code. A *module* is a collection of named fields that can be values, types, or even other modules and module types. An OCaml program consists of pairs of `.ml` and `.mli` files which correspond respectively to a module implementation and its interface (called a *module type*). Due to the high generality of OCaml modules, related error messages are often very complex, although technically correct. This means one often finds oneself having to decrypt very large error messages that were caused by simple spelling mistakes. Our goal is to produce more readable error messages by detecting those spelling mistakes. To do so, we design and implement an algorithm that detects patchable spelling mistakes and creates error messages accordingly.

Consider the OCaml module types presented in Listing 1, and suppose the user provides a module of type `Implem` where a module of type `Interface` is expected (for example, in the `.ml` file corresponding to a `.mli` file). To a human, it is abundantly clear what the two errors are: a spelling mistake (`lenght` instead of `length`), and a missing field (`succ`). While the compiler is able to detect the missing field properly, it does not notice the spelling mistake and instead considers the value `length` as a missing field as well, which can be confusing to the user. The resulting error message is shown in Figure 1. Ideally, the compiler should be able to detect spelling mistakes and take them into account when generating error messages.

In Section II, we formalize this goal and give a weight function to minimize, as well as time constraints. We then describe the approach we explored in Section III, before discussing its viability in Section IV.

```
module type Implem =
sig
  type t
  val lnegth : t -> int
end
```

```
module type Interface =
sig
  type t
  val length : t -> int
  val succ : t -> t
end
```

Listing 1. Two module types—that of an implementation, and an interface.

```
Signature mismatch:
...
The value length is required but not provided
The value succ is required but not provided
```

Fig. 1. The current error message in the situation presented in Listing 1 (slightly altered for readability).

II. Objective

Given an expected module type (the interface) and the type of a provided module (the implementation), we intuitively want to construct a "patch" from the type of the provided module to the expected module type. Such a patch should be able to perform common operations like renaming fields, changing the type of values, or, if necessary, adding new fields. Ideally, we want to suggest changes that are the easiest possible to implement. In particular, we want to favor name changes to field additions because the former can be applied by changing a few characters only, while the latter requires writing new code. In this section, we formalize the notion of a patch and propose a way to measure its quality.

A. A formalization of patches

Formally, we consider module types as sets of fields. A *field* f is composed of a *name*, denoted $\mathrm{name}(f)$, and a *description* $\mathrm{desc}(f)$ (for a value, its type; for a type, eithers an alias, a constructor, or nothing; etc.). Each field f also has a *kind* ("value," "type," "module," "module type," "class," "class type," or "type extension"), denoted $\mathrm{kind}(f)$. The descriptions of fields of the same kind are partially ordered by a subtyping relation $\preceq$. A field f is said to be *assignable to* a field g if $\mathrm{desc}(f) \preceq \mathrm{desc}(g)$. Because fields of different kinds live within different namespaces, we consider each kind separately. From now on, we suppose that all values have the same kind.

Given an expected module type M and a provided module type $\widetilde{M}$, the patch p we want to construct consists of:

- the set of valid fields alongside their counterpart in the interface

$$\mathcal{V}(p) \quad = \quad \left\{ (f,g) \in \widetilde{M} \times M \;\middle|\; \begin{array}{c} \mathrm{name}(f) = \mathrm{name}(g) \\ \mathrm{desc}(f) \preceq \mathrm{desc}(g) \end{array} \right\}$$

- a set of fields with invalid descriptions alongside their counterparts in the interface

$$\mathcal{D}(p) \quad = \quad \left\{ (f,g) \in \widetilde{M} \times M \;\middle|\; \begin{array}{l} \mathrm{name}(f) = \mathrm{name}(g) \\ \mathrm{desc}(f) \not\preceq \mathrm{desc}(g) \end{array} \right\}$$

- a set of fields to rename alongside their counterparts in the interface

$$\mathcal{N}(p) \quad \subseteq \quad \left\{ (f,g) \in \widetilde{M} \times M \;\middle|\; \begin{array}{l} \mathrm{name}(f) \neq \mathrm{name}(g) \\ \mathrm{desc}(f) \preceq \mathrm{desc}(g) \end{array} \right\}$$

- a set of fields to add

$$\mathcal{A}(p) \quad \subseteq \quad M$$

A patch p must be such that any field of $\widetilde{M}$ appears at most once in $\mathcal{V}(p) \sqcup \mathcal{D}(p) \sqcup \mathcal{N}(p)$, and each field of M appears exactly once in $\mathcal{V}(p) \sqcup \mathcal{D}(p) \sqcup \mathcal{N}(p) \sqcup \mathcal{A}(p)$ (i.e., a field is either valid, has the wrong description, has the wrong name, or is missing). Intuitively, p describes how to morph $\widetilde{M}$ into M by altering the fields of $\widetilde{M}$ to transform them into fields of M. Many such patches may exist for a single $\langle \widetilde{M}, M \rangle$ pair. The next section aims to quantify the quality of a patch.

B. Weight of a patch

Different categories of changes have different implications for the user: adding a field requires writing new code, changing the description of a field likely requires rewriting some code (e.g., changing the type of a function requires changing its implementation), and altering the name of a field only requires editing a few characters. Our goal being to suggest a minimal amount of changes to the user, we should prefer renaming to other categories of changes. Nonetheless, we only want to suggest renames that are likely to be correct. This is why we only consider $\mathcal{A}(p)$ and $\mathcal{N}(p)$ in the weight of a patch p.

As a reminder, the usual *edit distance* between two words w and w', denoted $\mathrm{ED}(w, w')$, is the minimal amount of single character deletions, insertions, or substitutions necessary to transform w into w'. For example, $\mathrm{ED}(\texttt{survey}, \texttt{surgery}) = 2$ (replace `v` with `g`, and add `r`). This is typically calculated using dynamic programming algorithms [2]. Some more complex edit distances might allow character transpositions (i.e., swapping two adjacent characters), or even assign different costs to each edit (e.g., based on the physical distance between keys of a keyboard). In this work, we use the most basic form of edit distance for simplicity.

To prevent favoring changing short names simply because there are fewer characters to modify, we define the *normalized edit distance* between two words w and w', denoted $\overline{\mathrm{ED}}(w, w')$:

$$\overline{\mathrm{ED}}(w, w') \quad = \quad \frac{\mathrm{ED}(w, w')}{|w'|}$$

Moreover, to hinder changing the name of a field too much, we define the *normalized edit distance with a cutoff* $\alpha \in \mathbb{R}_+$ for any two words w and w', denoted $\overline{\mathrm{ED}}_\alpha(w, w')$, as follows:

$$\overline{\mathrm{ED}}_\alpha(w, w') \quad = \quad \begin{cases} \overline{\mathrm{ED}}(w, w'), & \text{if } \overline{\mathrm{ED}}(w, w') \leq \alpha \\ +\infty, & \text{otherwise} \end{cases}$$

$$\widetilde{M} \quad = \quad \left\{ \begin{array}{l} \texttt{val x1 : int} \\ \texttt{val x2 : int} \\ \texttt{val xc : bool} \\ \texttt{val x5 : unit} \end{array} \right\}$$

$$M \quad = \quad \left\{ \begin{array}{l} \texttt{val x1 : int} \\ \texttt{val x2 : float} \\ \texttt{val x3 : bool} \\ \texttt{val x4 : char} \end{array} \right\}$$

$$p \quad : \quad \begin{cases} \mathcal{V}(p) = \{\texttt{val x1 : int}\} \\ \mathcal{D}(p) = \{(\texttt{val x2 : int}, \texttt{val x2 : float})\} \\ \mathcal{N}(p) = \{(\texttt{val xc : bool}, \texttt{val x3 : bool})\} \\ \mathcal{A}(p) = \{\texttt{val x4 : char}\} \end{cases}$$

Fig. 2. Two module types $\widetilde{M}$ and M, and a patch p from $\widetilde{M}$ to M.

Finally, given a patch p, we judge its quality by considering its *weight*, denoted $w(p)$, and defined by

$$w(p) \quad = \quad \left\langle |\mathcal{A}(p)|, \sum_{(f,g) \in \mathcal{N}(p)} \overline{\mathrm{ED}}_{1/2}(\mathrm{name}(f), \mathrm{name}(g)) \right\rangle$$

The weight of a patch is a pair whose first component is the number of fields the patch considers as completely missing, and whose second component is the sum of the normalized edit distances with cutoff $1/2$ corresponding to each rename. For example, the patch in Figure 2 has a weight of $\langle 1, \frac{1}{2} \rangle$. The value of the cutoff was chosen arbitrarily and is probably a good default.

Ideally, we want to minimize the weight for the order $\trianglelefteq$ such that $\langle x, y \rangle \trianglelefteq \langle x', y' \rangle$ if, and only if, both $x \leq x'$ and $y \leq y'$. Sadly, this order is not total, so there might not be a least element. Therefore, we try to minimize the weight for the lexicographic order. This is not too problematic because the second component is bounded thanks to the cutoff.

An interesting subtlety is that some fields of a module can appear in the description of other fields. For example, a type can appear within the type of a submodule, or in that of a value (Listing 2). Essentially, this makes the subtyping relation $\preceq$ dependent on the patch itself. This can be solved by computing the patch in multiple iterations to find a fixed point, each time using the previous patch as a basis. In practice, we only do five iterations, but a future improvement could be to compute an actual fixed point.

C. Time constraint for feedback loops

Minimizing the weight of patches is not the only important metric. Indeed, programmers often use editors that let them preview error messages in real-time, as they are editing source code [3]. For this reason, it is not acceptable to have error messages take too much time to compute. In this work, we impose a total time of less than 500 ms for modules at the

```
module type Provided =
sig
  type me_type
  val x : me_type
end
```

```
module type Expected =
sig
  type my_type
  val x : my_type
end
```

Listing 2. Renaming `me_type` to `my_type` has the effect of making the value `x` well-typed.

boundary between human-written and machine-generated in terms of size. This is in line with the 150 ms average visual stimulus reaction time for humans. In practice, some codebases contain modules with thousands of elements. In most reasonable cases, however, this upper bound is unreached. Moreover, modules that contain hundreds of fields are generally obtained by including smaller modules that can be checked independently.

As explained in Section II.A, a different patch is computed for each kind by only considering the corresponding fields. There are seven different field kinds, leaving us a 70 ms window to compute a patch for each.

III. Efficiently finding spelling mistakes using fuzzy matching

The first step in constructing a patch p is to match fields by name. Unchanged fields are matched silently (they constitute $\mathcal{V}(p)$), and each field whose name is unmodified, but whose description does not respect the subtyping relation, (i.e., each element of $\mathcal{D}(p)$) displays a type error. What remains are the disjoint sets $R \subseteq M$ of expected, but absent, fields, and $L \subseteq \widetilde{M}$ of present, but not required, fields. We are trying to pair the fields of those two sets with each other (i.e., construct $\mathcal{N}(p) \subseteq L \times R$, from which $\mathcal{A}(p)$ can be deduced) in a way that minimizes the weight of the resulting patch. Because patches are computed separately for each field kind, we suppose hereafter that all considered fields have the same kind (i.e., the kind function is constant over $L \sqcup R$).

A. Using a greedy algorithm

As a first attempt, one might be tempted to use a greedy approach: for each field $g \in R$ successively, consider it a rename of the first not-yet-paired, compatible field $f \in L$ such that $\overline{\mathrm{ED}}_{1/2}(\mathrm{name}(f), \mathrm{name}(g))$ is finite. This algorithm is both easy to implement and computationally efficient. However, it fails at producing an optimal patch, as illustrated in Listing 3.

B. Computing a maximum stable marriage

Another way is to compute a stable marriage between L and R, where the order of preference of a field $f \in L$ is given by $g \mapsto \overline{\mathrm{ED}}_{1/2}(\mathrm{name}(f), \mathrm{name}(g))$ (i.e., f prefers g_1 to g_2 if, and only if, $\overline{\mathrm{ED}}_{1/2}(\mathrm{name}(f), \mathrm{name}(g_1)) \leq \overline{\mathrm{ED}}_{1/2}(\mathrm{name}(f), \mathrm{name}(g_2))$), and that of a field $g \in R$ is given similarly by $f \mapsto \overline{\mathrm{ED}}_{1/2}(\mathrm{name}(f), \mathrm{name}(g))$.

As a reminder, given two disjoint sets L and R and, for each element, an order of preference of elements of the other set, a *stable marriage* is a pairing of elements of L with elements of R such that there are no two elements $\ell \in L$ and $r \in R$ that both prefer each other to their respective pairing. The usual Gale–Shapley algorithm [4] is guaranteed to compute a stable marriage of maximal cardinality only if each element ranks all elements from the other set, with a strict order.

```
module type Provided =
sig

  val do_domething : t
end
```

```
module type Expected =
sig
  val undo_something : t
  val do_something : t
end
```

Listing 3. The greedy algorithm may consider `do_domething` a rename of `undo_something`, yielding a patch of weight $\langle 1, \frac{3}{14} \rangle$. The optimal solution, however, pairs `do_domething` with `do_something` in a patch of strictly lesser weight $\langle 1, \frac{1}{12} \rangle$.

```
module type Provided =
sig
  val field_a : 'a
  val field_b : 'a -> 'b
end
```

```
module type Expected =
sig
  val field_1 : 'a
  val field_2 : 'a -> 'b
end
```

Listing 4. In this situation, $\{\texttt{field\_a} \mapsto \texttt{field\_2}\}$ is a stable marriage with weight $\langle 1, \frac{1}{7} \rangle$, maximal for inclusion. Meanwhile, the stable marriage $\{\texttt{field\_a} \mapsto \texttt{field\_1}, \texttt{field\_b} \mapsto \texttt{field\_2}\}$, pairs more fields, resulting in a strictly lesser weight of $\langle 0, \frac{2}{7} \rangle$.

Unfortunately, we are not in this situation: as illustrated in Listing 4, the Gale–Shapley algorithm is not sufficient to reduce the first component of the weight function as much as possible (i.e., pair as many fields as possible). Computing a stable marriage of maximal cardinality in the general case turns out to be NP-hard, which is why we settle for the $3/2$-approximation computed by Király's algorithm [5].

Király's algorithm runs in time linear in the sum of the lengths of the preference lists. It works similarly to the Gale–Shapley algorithm, except it handles ties more carefully. In this regard, it might consider some pairings multiple times, although this is bounded, which guarantees the linear run time.

C. Computing the preference lists

a) *A first approach:* Computing the orders of preference essentially requires computing a matrix of edit distances between elements of L and elements of R (akin to the ranking matrix in [4]). The naïve method, consisting of computing each cell independently using an algorithm such as the Wagner–Fischer algorithm [2], is too slow for our needs. For this reason, we use *tries* to compute each column of the matrix in a single operation, taking inspiration from common fuzzy searching algorithms for fixed dictionaries [6], [7]. This is especially efficient because there are typically fewer missing fields than additional fields.

A trie is a way to represent a map from words to arbitrary data that shares the representation of common prefixes, as illustrated in Figure 3. By constructing a trie $\mathfrak{T}$ mapping names of fields of L to the corresponding fields, we can compute the edit distance between any given word and all names of fields of L simultaneously. In particular, this lets us compute a column of the matrix, corresponding to a field $g \in R$, in a single pass. Sorting the fields of L by increasing distance to g gives us the order of preference of g.

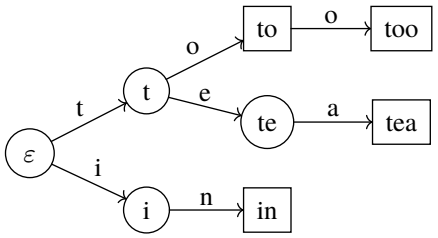

Fig. 3. A trie containing the words "to," "too," "tea," and "in." Terminal nodes are represented with rectangles; each typically contains associated data.

Algorithm 1: Compute column j of distance matrix D.

```
ComputeDistances(T, w, d):
1   if w = ε and T is a terminal node with value i:
2     [D]_{i,j} ← min([D]_{i,j}, d)                ▷ Update distance.
3   if w ≠ ε:
4     ComputeDistances(T, w[1:], d + 1)            ▷ Delete w[0].
5   for each edge -a-> T' of T:
6     ComputeDistances(T', w, d + 1)               ▷ Insert a.
7     if w ≠ ε:
8       if a = w[0]:
9         ComputeDistances(T', w[1:], d)           ▷ Keep a.
10      else:
          ▷ Substitute a for w[0].
11        ComputeDistances(T', w[1:], d + 1)
```

Algorithm 1 is a simplified version of the algorithm one may use to compute a column of the distance matrix. To compute the column j corresponding to a field $r \in R$, call $\textsc{Walk}(\mathfrak{T}, \mathrm{name}(r), 0)$, where $\mathfrak{T}$ is the trie mapping all names of fields of L to the index i of their corresponding lines in the distance matrix (all the distances are initially infinite).

b) *Lazy preference list computation:* One benefit of Király's algorithm is that it considers the elements of each preference list in order, and does not always go through the entirety of each list. We can take advantage of that by computing the elements of the preference lists on demand to prevent unnecessary computations. This requires first rewriting Algorithm 1 as the more general Algorithm 2, which computes all transformations of a word w into the words of a trie T. Algorithm 2 is essentially the implementation of a Levenshtein automata [7].

Given a word w, the tree of transformations $\textsc{Transform}(\mathfrak{T}, w)$ can be searched using an algorithm such as Dijkstra's [8]. Contrary to Boytsov in [7], we search the tree of transformations breadth-first instead of depth-first. This means we get the elements in order of preference directly, and lets us compute only the information we need by extending the boundary just enough to compute the next element every time.

In practice, the transformation tree is searched and pruned using the A* search algorithm [9] with a simple heuristic based on additional metadata attached to each trie (namely, the lengths of the shortest and longest suffixes starting at each subtrie). A set of (T, w) pairs, each alongside a current distance d, (i.e., a priority queue) is enough to represent the state of the search algorithm because the transformation tree is acyclic. Therefore, the search algorithm can easily be converted into an external iterator that is called whenever a new element of the preference list is needed by Király's algorithm. Such an external iterator is created for each field of R.

c) *Limiting the length of the preference lists:* Király's algorithm performs in time linear in the sum of the lengths of the preference lists. As aforementioned, each field typically has a very short preference list. Nevertheless, a maximum length of ten elements, after which no more field is considered, is enforced. This is a fair balance between not affecting the vast majority of cases, and preventing extreme situations from degrading performances too much.

Algorithm 2: Transformations of w into words of a trie T.

```
Transform(T, w):
1   M ← ∅
2   if w = ε and T is terminal:
3     M ← {(T, ε)}
4   if w ≠ ε:
      ▷ Delete w[0].
5     M ← M ⊔ (T, w) -1-> Transform(T, w[1:])
6   for each edge -a-> T' of T:
      ▷ Insert a.
7     M ← M ⊔ (T, w) -1-> Transform(T', w)
8     if w ≠ ε:
9       if a = w[0]:
          ▷ Keep a.
10        M ← M ⊔ (T, w) -0-> Transform(T', w[1:])
11      else:
          ▷ Substitute w[0] for a.
12        M ← M ⊔ (T, w) -1-> Transform(T', w[1:])
13  return M
```

D. Using the greedy algorithm as a fallback

Despite all those optimizations, our implementation of Király's algorithm stays slower than the greedy algorithm mentioned in Section III.A. For this reason, we fall back to the greedy algorithm when the input becomes too large. The empirically determined condition to switch to the greedy algorithm is $\max(|L|, |R|) > 60$.

One thing to note is that the inputs of Király's algorithm are L and R, meaning only the missing fields from the expected module type and the additional (or misspelled) fields from the provided module type matter in terms of complexity. In particular, the size of the expected module type has no impact on the performance of Király's algorithm. This is not exactly the case for the size of the provided module type, as any field of the provided module type that is not present in the expected module type is part of L, and needs to be compared to fields of R. This is a problem on the matter of which closer attention should be brought in the future.

IV. Results

The work presented in this document has been fully implemented in a fork of the OCaml compiler [10] and will be submitted to the main branch as a pull request. After review, and some potential changes and improvements, this change

```
module type Provided =
sig
  val my_float : bool
  val also_floot : float
  val my_bool : bool
  val my_int : int
  val my_other_int : int

  module MyModule : sig
    val inner_int : int
  end
end
```

```
module type Expected =
sig
  val my_float : float
  val also_float : float
  val my_bool : bool
  val my_int : int

  module MyModuleA : sig
    val inner : float
  end
  module MyModuleB : sig
    val inner_int : int
  end
end
```

```
The value also_float is required but not provided
The module MyModuleA is required but not provided
The module MyModuleB is required but not provided
```

```
Try changing value my_float to be a float
Hint: Try renaming value also_floot to also_float
Hint: Try renaming module MyModule to MyModuleB
Try adding a module MyModuleA
```

Listing 5. A situation where a module of type `Provided` was provided where a module of type `Expected` was expected would previously result in the first message, and now results in the second message.

could be included in the next publicly distributed version of the compiler.

We are now able to make smarter suggestions to the user when a provided module does not have the expected type. Listing 5 and Listing 6 are two examples of situations where the error message is noticeably improved.

In this section, we measure the quality of the error messages and the execution time of the algorithm, especially on large inputs. This requires generating large modules, which is done using a Python script that creates OCaml modules containing values with random names (strings of ten to fifteen random identifier characters) and types (built recursively from a pool of type constructors from the standard library), and applies random modifications to them. A modification consists of reordering values, adding a value, removing a value, renaming a value, or changing the type of a value. Those five kinds of modification are assigned equal probabilities. A small example of a generated module (`Expected`) and its modified version (`Provided`) are available in Appendix 1.

A. *Quality of the result*

We will first evaluate the quality of the error messages, by considering the first component of the weight function introduced in Section II.B. In other words, we measure the number of fields the algorithm suggests adding, in opposition

```
module type Provided =
sig
  type me_type
  val me_value : me_type
end
```

```
module type Expected =
sig
  type my_type
  val my_value : my_type
end
```

```
Hint: Try renaming type me_type to my_type
Hint: Try renaming value me_value to my_value
```

Listing 6. A situation where a module of type `Provided` was provided where a module of type `Expected` was expected now results in the message above. This situation is akin to the one presented in Listing 2.

TABLE I

AVERAGE NUMBER (TWO SIGNIFICANT FIGURES) OF NON-PAIRED FIELDS ON RANDOM MODULES OF VARYING SIZE, AND WITH A VARYING AMOUNT OF MODIFICATIONS APPLIED. THE PARENTHESIZED NUMBERS ARE THE EXPECTED AMOUNT OF DELETION MODIFICATIONS. GRAY CELLS: THE GREEDY ALGORITHM IS USED.

Modifs → / Size ↓	10 (2)	20 (4)	50 (10)	100 (20)	200 (40)	500 (100)	1 000 (200)
100	1.9	4.1	9.2				
200	1.9	4.8	10	19			
500	2.2	3.7	10	18	38		
1 000	1.6	4.2	9.3	18	40	100	
2 000	1.6	3.7	8.8	22	40	100	200
5 000	2.1	4.1	11	19	37	100	200

to obtaining them by renaming other fields. About a fifth of the modifications introduced by the generator are value deletions, meaning we expect the first component of the weight function to be about a fifth of the total number of modifications introduced. Table I illustrates that the number of paired items is close to optimal.

It should be noted that the randomly generated names tend to be very far apart, meaning the preference lists often consist of single elements, which does not create situations where the greedy algorithm performs unsatisfyingly. This does not affect the satisfactory nature of those results in the cases where Király's algorithm is used.

B. *Performance*

We now try to measure the compute time of the algorithm. Remember that we imposed a 70 ms window for each run in Section II.C. In other words, all the benchmarks in this section have the goal of being below this 70 ms upper bound.

Figure 4 illustrates that this upper bound is generally respected, with the exception of some extreme cases. One important detail is that the complexity of the whole algorithm does not depend on the module size. Rather, the determining factor is the number of modifications.

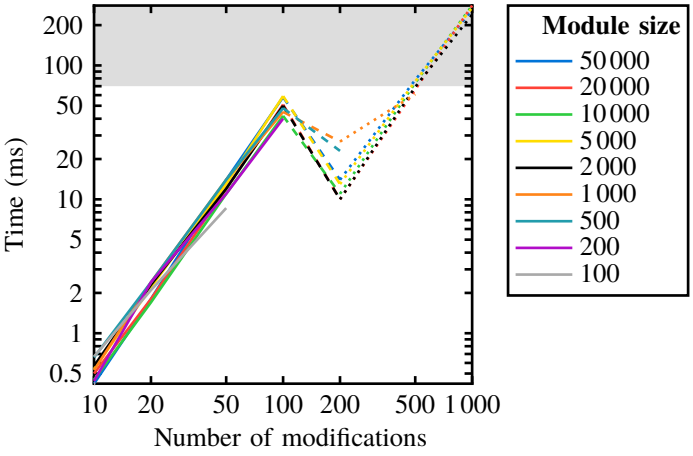

Fig. 4. Average compute time on random modules with varying sizes. Dots: the greedy algorithm is used. The gray area indicates the 70 ms threshold.

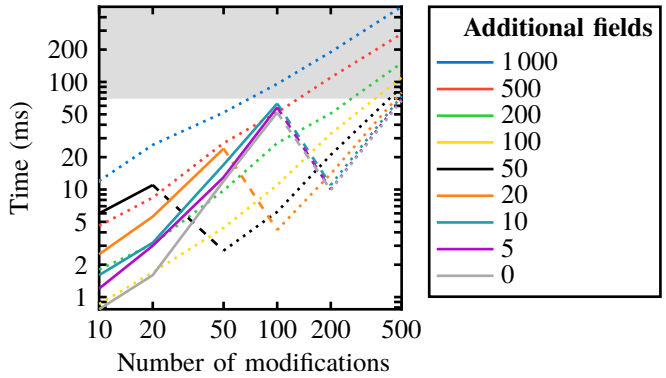

Fig. 5. Average compute time on random modules with varying amounts of additional fields. Dots: the greedy algorithm is used. The gray area indicates the 70 ms threshold.

In Figure 5, we can see that the modules with additional fields compared to the expected interface take more time to analyze. Indeed, each additional field needs to be considered when trying to find a missing field. This result is less satisfactory than the previous one and indicates that additional work is needed to improve this situation.

V. Future works

Future works in this area could focus on the following topics. Performance-wise, more benchmarks could be made in order to more clearly understand the impact of different parts of the algorithm on its performance. This would make it possible to improve the implementation to gain a constant factor. Moreover, the current dependency of the compute time on the number of additional fields is not satisfactory and should be reduced if not eliminated. A possible improvement could be to take into account the placement of fields in sources, as users tend to write fields in the same order in the implementation as in the interface. Feature-wise, diffing modules recursively, as well as considering tree-wise modifications (such as subtree copying and moving) for modules, could help provide more precise suggestions to the user, at the cost of a much higher computational complexity. Moreover, other edit distances could be explored, as well as different ways of normalizing the edit distance (e.g., dividing by the sum of the lengths of both arguments or by their product).

VI. Conclusion

During this internship, we explored a way to improve OCaml module error messages by detecting spelling mistakes efficiently using an ingenious combination of algorithms and data structures. An implementation of this work in a fork of the OCaml compiler has been done, and will soon be proposed as a pull request on the main compiler branch.

References

[1] X. Leroy, J. Vouillon, D. Doligez, D. Rémy, and A. Suárez, *The core OCaml system: compilers, runtime system, base libraries*. (1996). [Online]. Available: https://github.com/ocaml/ocaml

[2] Wikipedia contributors, "Wagner–Fischer algorithm." Accessed: Jun. 05, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Wagner%E2%80%93Fischer_algorithm&oldid=1211775080

[3] Stack Overflow, "Stack Overflow Developer Survey 2023." Accessed: Jul. 17, 2024. [Online]. Available: https://survey.stackoverflow.co/2023/

[4] D. Gale and L. S. Shapley, "College admissions and the stability of marriage," *The American Mathematical Monthly*, vol. 69, no. 1, pp. 9–15, Jan. 1962, doi: 10.1080/00029890.1962.11989827.

[5] Z. Király, "Linear Time Local Approximation Algorithm for Maximum Stable Marriage," *Algorithms*, vol. 6, pp. 471–484, Aug. 2013, doi: 10.3390/a6030471.

[6] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, Mar. 2001, doi: 10.1145/375360.375365.

[7] L. Boytsov, "Indexing methods for approximate dictionary searching: Comparative analysis," *ACM Journal of Experimental Algorithmics*, vol. 16, pp. 1.1–1.91, May 2011, doi: 10.1145/1963190.1963191.

[8] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, pp. 260–271, Jun. 1959.

[9] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, Jul. 1968, doi: 10.1109/TSSC.1968.300136.

[10] M. Monin and F. Angeletti, *MDLC01/ocaml at improved-module-error-messages*. (Jun. 2024). [Online]. Available: https://github.com/MDLC01/ocaml/tree/improved-module-error-messages

Appendix 1. An example of a random module

The playful reader is encouraged to find the seven modifications introduced by the generator.

```
module type Provided = sig
  val rhhpxdt88duy_q'ny : int Seq.t * ('h * 'u)
  val q9f7064sn9liyyza5 : bool -> float
  val l1jijy'10aadvi8fg : int * float
  val xz7k6s7imgky1wo'8 : bool
  val po3qqtfzvifh : unit -> int
  val bbq6'_yt9d2bh70s5 : bool
  val a6ibeit4enp1i : int
  val ab2x0ng_ijmizokpe : int
  val d0l65b5ezhzie : int
  val o867ijoir750hq : 'k list -> int -> unit
  val rue'e0l9vinj0fnv_ : float
  val v489maf18whfdi5 : unit
  val bvytkl99fb3qzgy : unit
  val aqxndc_u1bs'2d : float
  val d_scrmxvv_u'07qh5 : float
  val y40d206ni4f236 : bool
  val bt2'g31ack6aqju : bool
  val j5mbcfg7s_yt9nvp : int
  val u6a010w22kns1zwqb : unit * 'e list
  val egtxu5dzu66g : int * bool
end

module type Expected = sig
  val rhhpxdt88duy_q'ny : int Seq.t * 'h * 'u
  val q9f7064sn9liyyza5 : bool -> float
  val l1jijy'10aadvi8fg : int * float
  val bmq6'_yt92bh7s5 : bool
  val afvf51afh_3fv'r_6 : unit -> unit
  val a6ibeit4enp1i : int
  val ab2x0ng_ijmizokpe : int
  val d0l65b5ezhzie : int
  val o867ijoir750hq : 'k list -> int -> unit
  val rue'e0l9vinj0fnv_ : float
  val v489maf18whfdi5 : unit
  val bvytkl99fb3qzgy : unit
  val aqxndc_u1bs'2d : float
  val d_scrmxvv_u'07qh5 : unit
  val y46206ni4fc36 : bool
  val bt2'g31ack6aqju : bool
  val j5mbcfg7s_yt9nvp : int
  val egtxu5dzu66g : int * bool
end
```