High-level error messages for modules through diffing

Gabriel Radanne, Florian Angeletti

High-level error messages for modules through diffing

Florian Angeletti
Inria
florian.angeletti@inria.fr

Gabriel Radanne
Inria
gabriel.radanne@inria.fr

Modules are one of the most complex features of ML languages. This complexity is reflected in error messages. Whenever two module types are mismatched, it is hard to identify and report the exact source of the error. Consequently, typecheckers often resort to printing the whole module types, and hope that the human user will navigate the sea of definitions.

We propose to improve module error messages by coupling classical typechecking with a diffing algorithm. The typechecker deals with the gritty details of the ML module system whereas the diffing algorithm summarizes the error through a higher level view. The large literature on diffing algorithms allows us to pick and choose the exact algorithm adapted for signatures, functors applications, submodules, etc.

1 Motivation

Typical uses of module are quite simple. Few people use directly higher-order functor with anonymous module arguments and labyrinth of module type definitions. Quite often, the names of functor arguments and parameters even match:

```
module type XT = sig type x end
module type YT = sig type y end
module F(X:XT)(Y:YT) = struct ... end
module X = struct type x = A end
module Y = struct type y = B end
module Result = F(X)(Y)
```

Furthermore, module type errors often happens during code refactoring, where a handful of changes are common:

- Adding a new item
- Removing an old item
- Changing the type of an item

For instance, if we refactor the definition of the functor `F` above and remove the first argument but forget to update the definition of the `Result` module, the previously working code now yields an error

```
module F(Y:YT) = struct ... end
module Result = F(X)(Y)

Error: Signature mismatch:
       Modules do not match: sig type x = X.x = A end is not included in YT
       The type y is required but not provided
```

However, we are not interested in the possibly lengthy mismatch between the module types `XT` and `YT`. We are more interested in the fact that there is an extra functor argument. This combination of hard to decipher module type errors and a small class of common high-level errors makes a good argument for trying to give users a higher-level view in module-level type error messages. We propose to use the tree-like shape of module types to leverage *diffing algorithm* on trees and lists.

Diffing algorithms have a long history in many domains. For linear texts, the Longest Common Subsequence problem [**DBLP:conf/spire/BergrothHR00**] is used for code versioning and wikis. A more general version using edit distances [**DBLP:journals/csur/Navarro01**] is commonly used for spellchecking and bioinformatics. More recently, diffing for trees [**DBLP:journals/tcs/Bill** has found a large application in Web programming for UIs [**reactjs**]. This diversity gives us a very fertile ground to pick diffing algorithms adapted to the exact mismatch at hand, from edit distances for functor applications to tree diffs for signatures with submodules.

We now demonstrate these ideas on functor applications, which we implemented in the OCaml compiler (https://github.com/ocaml/ocaml/pull/9331). In the talk, we will present the larger context, its application to signatures, and some technical details.

2 Optimising edit-distance for functors

As we pointed out before, functors applications are often the source of complex type errors. This is partially due to the contrast between the higher-level view of a functor multi-application

```
module R = F(X_1)...(X_n)
```

and the left-to-right biased view of the typechecker. If type-checking this functor application reports an error at position k, the standard way to report an error would be simply to report the mismatch between expected type of the kth-argument and the kth-parameter. But by doing so, we are losing the context of the functor multi-application.

By considering the whole list of arguments, we can find the smallest patched argument lists that make the multi-application typechecks. A patched argument is here either:

1. An accepted argument from the original argument list
2. A deleted argument that we throw away from the original argument list
3. An additional argument pulled from the expected parameter list
4. A mismatched argument from the original argument list

It is then natural to assign a positive weight to those patched arguments to represent how far away they are from the arguments that the user has written. If only accepted arguments have a zero weight, then we are guaranteed that there is a patched argument list with minimal weight. For instance, in our example

```
module F(Y:sig type y end) = struct ... end
module Result = F(X)(Y)
```

The possible patched arguments would be

1. `[Delete(X); Accept(Y)]`
2. `[Change(X);Delete(Y)]`
3. `[Add(YT);Delete(X);Delete(Y)]`

The smallest change is clearly the first one. Consequently, rather than reporting the mismatch between `XT` and `YT`, we can simply report `X` as an extra-argument:

```
Error: The functor application is ill-typed.
       These arguments:
         X Y
       do not match these parameters:
         functor  (Y : YT) -> ...
  1. The following extra argument is provided X : sig type x = X.x = A end
  2. Module Y matches the expected module type YT
```

It is important to inform users that we consider the functor multi-application as a whole and how we think the code should be corrected. In our experience, these error messages scale to fairly complex uses (as can be seen in the patch linked previously) with high-order, dependent, or even variadic functors, using the full panel of advanced ML features. This work also applies to inclusion tests between two functor signatures.

Our unoptimized implementation uses a variant of the Wagner–Fischer algorithm [**DBLP:journals/jacm/W** with a complexity of $O(\max(|\text{arguments}|, |\text{parameters}|)^2)$ module comparisons in the worse case. This is sufficient in practice, since functors applications are usually short. More efficient algorithms [**DBLP:journals/siamcomp/LandauMS98**] could also be used.

So far, we only considered the Levenshtein distance, which includes insertions, deletions, and substitutions (i.e., mismatchs). We could easily extend our implementation to transpositions but it is unclear how to present such a mix of operations to the user in a clear manner. One case that could be worth investigating is when a simple permutation, without any other operation, is sufficient to make the application typechecks. We could then simply inform the user to reorder the arguments in the right order.

3 Conclusion

We propose a methodology to improve the user experience of error message for modules in ML languages by providing high-level reports. This is done by combining the classical inclusion check algorithm with diffing algorithms, leading to much more focused error messages. In the talk, we will present the application to both functors arguments and signature, explore technical details such as the handling of variadic functors, and compare with existing error messages.