Better typing errors for OCaml

Arthur Charguéraud

Inria

Overview

State of the art

- Dozens of research papers on reporting type errors in ML…
- … none of these ideas ever reached the OCaml compiler!

Motivation

- Get OCaml to produce better error messages, for beginners…
- … and maybe for you, too!

Result

- A patch to the type-checker, providing alternative error messages for ill-typed top-level definitions.

Missing unit argument

```
let x = read_int in    (* missing unit argument *)
print_int x
```

ocamlc

```
File "examples/example_missing_unit_readint.ml", line 2, characters 10-11:
Error: This expression has type unit -> int
       but an expression was expected of type int.
```

ocamlc -easy

```
File "examples/example_missing_unit_readint.ml", line 2, characters 0-9:
Error: The function `print_int' expects one argument of type [int],
       but it is given one argument of type [unit -> int].

You probably forgot to provide `()' as argument somewhere.
```

If reaching a unification error between type `unit -> ?t` and `?u`, then report `You probably forgot to provide `()' as argument somewhere.`

Missing bang

```
let r = ref 1 in
print_int r       (* should be [!r] *)
```

ocamlc

```
File "examples/example_ref_missing_bang.ml", line 2, characters 10-11:
Error: This expression has type int ref
       but an expression was expected of type int.
```

ocamlc -easy

```
File "examples/example_ref_missing_bang.ml", line 2, characters 0-9:
Error: The function `print_int' expects one argument of type [int],
       but it is given one argument of type [int ref].

You probably forgot a `!' operator somewhere.
```

If reaching a unification error between type `?t ref` and `?u`, then report `You probably forgot a `!' operator somewhere.`

Missing rec

```
let facto n =    (* missing [rec] *)
   if n = 0 then 1 else n * facto (n-1)
```

ocamlc

```
File "examples/example_let_missing_rec.ml", line 2, characters 28-33:
Error: Unbound value facto
```

ocamlc -easy

```
File "examples/example_let_missing_rec.ml", line 2, characters 28-33:
Error: Unbound value facto.

You are probably missing the `rec' keyword on line 1.
```

Check whether the unbound variable would have been in the scope if it had been bound by a `let rec` instead of a `let`.

Missing else branch

```
let ordered_list_with x y =
   if x <= y then [x;y]
   else if x > y then [y;x]
```

ocamlc

```
File "examples/example_missing_else.ml", line 3, characters 23-27:
Error: This variant expression is expected to have type unit
       The constructor :: does not belong to type unit
```

ocamlc -easy

```
File "examples/example_missing_else.ml", line 3, characters 22-27:
Error: This expression is the result of a conditional with no else branch,
        so it should have type [unit] but it has type ['a list].
```

If a subterm of a particular language construct does not have the expected type, then explain why this type is expected.

Reducing the left-to-right bias

```
let f b =
  if b then 0 else 3.14 (* should have been 0. *)
```

ocamlc

```
File "examples/example_incompatible_else.ml", line 2, characters 19-23:
Error: This expression has type float but an expression was expected of type
         int.
```

ocamlc -easy

```
File "examples/example_incompatible_else.ml", line 2, characters 2-23:
Error: The then-branch has type [int]
       but the else-branch has type
       [float].
       Cannot unify type [int] with type [float].
```

To type-check a conditional or a pattern matching, first type-check each branch independently, then unify the branch types one by one.

Remaining left-to-right bias

```
let f b x =
  if b
    then print_int x
    else print_float x
```

ocamlc

```
File "examples/example_if_propagate.ml", line 5, characters 21-22:
Error: This expression has type int but an expression was expected of type
         float.
```

ocamlc -easy

```
File "examples/example_if_propagate.ml", line 5, characters 9-20:
Error: The function `print_float' expects one argument of type [float],
       but it is given one argument of type [int].
```

Unification may still perform side-effects accross branches; yet, the error typically involves a free variable, which often is to blame.

Errors for ill-typed applications

```
let _ =
  ignore (Array.make 0.0 20)
```

ocamlc

```
File "examples/example_make_swap.ml", line 2, characters 21-24:
Error: This expression has type float but an expression was expected of type
         int.
```

ocamlc -easy

```
File "examples/example_make_swap.ml", line 2, characters 10-20:
Error: The function `Array.make' expects 2 arguments of types [int]
        and ['a], but it is given 2 arguments of types [float] and [int].
```

If an application fails to type-check, locate the error on the entire application and display: `function `foo' expects arguments of type [bla] and [bla], but it is given arguments of type [bla] and [bla].`

Confusion on arithmetic operators

```
let _ =
  print_float (2.0 + 3.0)     (* should be [+.] instead of [+] *)
```

ocamlc

```
File "examples/example_add_bad.ml", line 2, characters 15-18:
Error: This expression has type float but an expression was expected of type
         int.
```

ocamlc -easy

```
File "examples/example_add_bad.ml", line 2, characters 19-20:
Error: The function `+' expects 2 arguments of types [int] and [int],
       but it is given 2 arguments of types [float] and [float].
```

Errors are no longer reported at a location ahead of the actual error.

Missing parentheses on a negation

```
let _ =
   succ -1       (* missing parentheses around [-1] *)
```

ocamlc

```
File "examples/example_f_minus_one.ml", line 2, characters 3-7:
Error: This expression has type int -> int
       but an expression was expected of type int.
```

ocamlc -easy

```
File "examples/example_f_minus_one.ml", line 2, characters 8-9:
Error: The function `-' expects 2 arguments of types [int] and [int],
       but it is given 2 arguments of types [int -> int] and [int].
```

The new error makes it clear that `` `-' `` is parsed as a binary operator.

Errors on higher-order function calls

```
let _ = List.map (fun x -> x + 1) [2.0; 3.0]
(* should have been [+.] instead of [+], or
   should have been [2;3] instead of [2.0;3.0] *)
```

ocamlc

```
File "examples/example_map_bad.ml", line 1, characters 35-38:
Error: This expression has type float but an expression was expected of type
         int.
```

ocamlc -easy

```
File "examples/example_map_bad.ml", line 1, characters 8-16:
Error: The function `List.map' expects 2 arguments of types ['a -> 'b]
        and ['a list], but it is given 2 arguments of types [int -> int]
        and [float list].
```

The new error explains the type of the anonymous function involved.

Occur-check errors

```
let rev_filter f l =
  List.fold_left (fun x acc -> if f x then x::acc else acc) [] [1; 2; 3]
(* swapped the parameters of the higher-order function *)
```

ocamlc

```
File "examples/example_fold_left_swap_app_2.ml", line 2, characters 43-44:
Error: This expression has type 'a list
       but an expression was expected of type 'a.

       The type variable 'a occurs inside 'a list
```

ocamlc -easy

```
File "examples/example_fold_left_swap_app_2.ml", line 2, characters 2-16:
Error: The function `List.fold_left' expects 3 arguments of types
       ['a -> 'b -> 'a] and ['a] and ['b list],
       but it is given 3 arguments of types ['c -> 'c list -> 'c list]
        and ['d list] and [int list].
```

Summary

- Custom messages for missing `()' and `!' and `rec'.
- Custom messages for subterms of particular constructs.
- Decreased left-to-right bias for `if', `match', and function calls.
- No reporting of errors before their actual locations (binary operators).

- Support for optional and named arguments in function calls.
- No change to errors on top-level definitions involving GADTs.
- No change to module type-checking.

Give it a try!

`https://github.com/charguer/ocaml`

Send feedback!