OCaml Debugging Guide

Fall 2017

Contents

1 Introduction

Nobody likes debugging. Here are the 8 most common but confusing error messages when coding in OCaml (sourced from last year's Piazza), and how to fix them. Use "Ctrl-F" to find your Error!

2 Syntax Errors

2.1 Version 1

```
Error: Line 4 Syntax error: pattern expected.
```

Even theoretically functional OCaml code sometimes throws this error. Often the issue arises when OCaml comes across a "keyword", which is built into the OCaml compiler and can't be used as a variable name. For example:

```
| (x, val) :: [] -> x
```

will throw this error when pattern matching because "val" is an OCaml keyword. To solve, change variable names.

2.2 Version 2

```
type float vector = Vector of float list
Error: Syntax error
```

In general, "type" declarations in ocaml look like

```
type <name> = stuff
```

or

```
type <parameter(s)> name = stuff
```

where "parameters" are things like `'a`. You could write

```
type floatvector = ...
```

but if you want to define something called a "float vector" you'll have to do it by defining

```
type 'a vector = ...
```

and then using this definition in a case where `'a` becomes "float". You can still refer to floats in the type declaration, but the type declaration with both float and vector is not valid OCaml syntax.

3 Type Errors

3.1 Version 1

```
Error: This expression has type ____/1328
       but an expression was expected of type ____/1355
```

This means that you are defining something twice! This error is common when combining OCaml files that could define the same type, variable or other syntax. Look for repeated names in your OCaml code

3.2 Version 2

```
This function has type ___ -> ___ -> ___
___ ___. It is applied to too many arguments.
Maybe you forgot a ';'.
```

This has to do with OCaml needing explicit directions when applying functions, which you can provide with added parentheses. When you write:

```
proc1 proc2 x y  z;;
```

and proc1 and proc2 both take two arguments. You think that you've said "apply proc1 to the args (proc2 x y) and z", but you've really said "apply proc1 to the args 'proc2' and 'x' ", and there's a horrid type-clash. When things like this happen, it's not a bad idea to put parens around the procedure and its args, so that this becomes

```
proc1  (proc2 x y) z;;
```

or even (in an extreme case, with the outer parens unnecessary):

```
(proc1  (proc2 x y ) z ) ;;
```

3.3 Version 3

```
Error: This expression has type 'a _____
          but an expression was expected of type 'b ____
```

The above error abstracts a common type assignment error, where the OCaml compiler finds the wrong or unexpected type during a function call.

One approach to solving such an error is to trace type changes through a function. For example, take this somewhat believable function:

```
let rec broken_function (incoming: 'a list) : 'a list list =
        match incoming with
        |[] ->  []
        |hd::tl -> hd::(broken_function tl);;
```

which should wrap each string in a list of strings inside its own list. The system throws a confusing error:

```
Error: This expression has type 'a but an
expression was expected of type 'a list
The type variable 'a occurs inside 'a list
```

Tracing the type through the function, we see that the function is tail recursive, but oh no! The head is not being wrapped in a list.

```
let rec broken_function (incoming: 'a list) : 'a list list =
        match incoming with
        |[] ->  []
        |hd::tl -> [hd]::(broken_function tl);;
```

4 Others

4.1 Version 1

```
Exception: Invalid_argument "equal: functional value"
```

OCaml cannot directly compare functions! It can only compare function results. Solve this exception by making sure you only ever compare results, and not the abstract functions themselves.

In the great words of Spike:

"OCaml is trying to compare one environment against another, and that means comparing bindings, which means comparing IDs, and values, and comparing values, for Builtins, means comparing [a function] with itself. But that's an attempt to check whether two functions are equal, and that's not allowed in OCaml (or pretty much any other language – it's provably impossible to do!)"

4.2 Version 2

```
Warning 25: bad style, all clauses in this
pattern-matching are guarded.
```

When you write

```
match pair with
| (n, k) when (n < k) -> ...
| (n, k) when (n = k) -> ...
| (n, k) when (n > k) -> ...
```

The things after the "when" are called "guard clauses". Ocaml can't actually tell whether you've got complete matching or not when you've got these things (for strong theoretical computer-science reasons). So to ensure you've got complete matching, they want you to leave at least one clause unguarded. It helps to think about OCaml wanting this:

```
match pair with
| (n, k) when (n < k) -> ...
| (n, k) when (n = k) -> ...
| (n, k) (* when (n > k) *) -> ...
```

where the final commented out clause is there as a reminder of what case it handles.

4.3 Version 3

```
Error: unbound value check_expect
```

This error usually stems from an improper import of CS17.ml. Make sure that you have imported CS17.ml. To do this, at the top of your file write

```
#use "/course/cs017/src/ocaml/CS17setup.ml";;
```

This file path only works if you're running your code on a department machine. If you want to use check expect locally, go to a department machine, email the file to yourself, download it on your own computer (or use scp/sftp if you have that set up) and at the top of your code include the local file.

5 Other Ocaml Weirdness

Here are some other strange features of Ocaml syntax:

5.1 Applying Procedures to Arguments

In Dr. Racket, you got used to calling procedures in the following way:

```
(my-proc arg1 arg2)
```

In Ocaml, you may therefore become used to calling procedures like:

```
my-proc arg1 arg2 ;;
```

This works for most cases, but you must be careful. For example you could conceivably write the following simple code:

```
let num (a : int) : int = a ;;
let opposite (a : int) : int = num -a ;;
```

However, this would throw an error! Here, you had simply intended to apply num to -a, but Ocaml interpreted it as subtracting a from the function num, which is a type clash, because num is a function and not an int. Because of this, when dealing with negative numbers in Ocaml, it is often safer to wrap them in parenthesis to avoid this misinterpretation; your code would then become:

```
let num (a : int) : int = a ;;
let opposite (a : int) : int = num(-a) ;;
```

Another situation in which this problem may arise is when dealing with options. You may wish to add the contents of two options by using pattern matching to extract their contents, and returning a Some of their sum. Again, you would have to be careful to wrap the sum in parenthesis to avoid misinterpretation.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17document by filling out the anonymous feedback form: `http://cs.brown.edu/courses/cs017/feedback`.