Chapter 8

Escaping from computations: exceptions

In some situations, it is necessary to abort computations. If we are trying to compute the integer division of an integer `n` by `0`, then we must escape from that embarrassing situation without returning any result.

Another example of the usage of such an escape mechanism appears when we want to define the `head` function on lists:

```
#let head = function
#     x::L -> x
#   | [] -> raise (Failure "head: empty list");;
Toplevel input:
>     x::L -> x
>        ^
Warning: the variable L starts with an upper case letter in this pattern.
head : 'a list -> 'a = <fun>
```

We cannot give a regular value to the expression `head []` without losing the polymorphism of `head`. We thus choose to escape: we *raise an exception*.

8.1 Exceptions

An exception is a Caml value of the built-in type `exn`, very similar to a sum type. An exception:

- has a *name* (`Failure` in our example),
- and holds zero or one value (`"head: empty list"` of type `string` in the example).

Some exceptions are predefined, like `Failure`. New exceptions can be defined with the following construct:

```
exception <exception name> [of <type>]
```

Example:

```
#exception Found of int;;
Exception Found defined.
```

The exception `Found` has been declared, and it carries integer values. When we apply it to an integer, we get an exception value (of type `exn`):

```
#Found 5;;
- : exn = Found 5
```

8.2 Raising an exception

Raising an exception is done by applying the primitive function `raise` to a value of type `exn`:

```
#raise;;
- : exn -> 'a = <fun>
```

```
#raise (Found 5);;
Uncaught exception: Found 5
```

Here is a more realistic example:

```
#let find_index p =
#  let rec find n =
#    function [] -> raise (Failure "not found")
#           | x::L -> if p(x) then raise (Found n)
#                     else find (n+1) L
#  in find 1;;
Toplevel input:
>           | x::L -> if p(x) then raise (Found n)
>                ^
Warning: the variable L starts with an upper case letter in this pattern.
find_index : ('a -> bool) -> 'a list -> 'b = <fun>
```

The `find_index` function always fails. It raises:

- `Found n`, if there is an element `x` of the list such that `p(x)`, in this case `n` is the index of `x` in the list,
- the `Failure` exception if no such `x` has been found.

Raising exceptions is more than an error mechanism: it is a programmable control structure. In the `find_index` example, there was no error when raising the `Found` exception: we only wanted to quickly escape from the computation, since we found what we were looking for. This is why it must be possible to *trap* exceptions: we want to trap possible errors, but we also want to get our result in the case of the `find_index function`.

8.3 Trapping exceptions

Trapping exceptions is achieved by the following construct:

```
try <expression> with <match cases>
```

This construct evaluates `<expression>`. If no exception is raised during the evaluation, then the result of the `try` construct is the result of `<expression>`. If an exception is raised during this evaluation, then the raised exception is matched against the `<match cases>`. If a case matches, then control is passed to it. If no case matches, then the exception is propagated outside of the `try` construct, looking for the enclosing `try`.

Example:

```
#let find_index p L =
#  let rec find n =
#    function [] -> raise (Failure "not found")
#           | x::L -> if p(x) then raise (Found n)
#                     else find (n+1) L
#  in
#    try find 1 L with Found n -> n;;
Toplevel input:
>let find_index p L =
>                 ^
Warning: the variable L starts with an upper case letter in this pattern.
Toplevel input:
>           | x::L -> if p(x) then raise (Found n)
>                ^
Warning: the variable L starts with an upper case letter in this pattern.
find_index : ('a -> bool) -> 'a list -> int = <fun>

#find_index (function n -> (n mod 2) = 0) [1;3;5;7;9;10];;
- : int = 6

#find_index (function n -> (n mod 2) = 0) [1;3;5;7;9];;
Uncaught exception: Failure "not found"
```

The `<match cases>` part of the `try` construct is a regular pattern matching on values of type `exn`. It is thus possible to trap any exception by using the `_` symbol. As an example, the following function traps any exception raised during the application of its two arguments. Warning: the `_` will also trap interrupts from the keyboard such as control-C!

```
#let catch_all f arg default =
#       try f(arg) with _ -> default;;
catch_all : ('a -> 'b) -> 'a -> 'b -> 'b = <fun>
```

It is even possible to catch all exceptions, do something special (close or remove opened files, for example), and raise again that exception, to propagate it upwards.

```
#let show_exceptions f arg =
#        try f(arg) with x -> print_string "Exception raised!\n"; raise x;;
show_exceptions : ('a -> 'b) -> 'a -> 'b = <fun>
```

In the example above, we print a message to the standard output channel (the terminal), before raising again the trapped exception.

```
#catch_all (function x -> raise (Failure "foo")) 1 0;;
- : int = 0

#catch_all (show_exceptions (function x -> raise (Failure "foo"))) 1 0;;
Exception raised!
- : int = 0
```

8.4 Polymorphism and exceptions

Exceptions must not be polymorphic for a reason similar to the one for references (although it is a bit harder to give an example).

```
#exception Exc of 'a list;;
Toplevel input:
>exception Exc of 'a list;;
>                 ^^
The type variable a is unbound.
```

One reason is that the `excn` type is not a parameterized type, but one deeper reason is that if the exception `Exc` is declared to be polymorphic, then a function may raise `Exc [1;2]`. There might be no mention of that fact in the type inferred for the function. Then, another function may trap that exception, obtaining the value `[1;2]` whose real type is `int list`. But the only type known by the typechecker is `'a list`: the `try` form should refer to the `Exc` data constructor, which is known to be polymorphic. It may then be possible to build an ill-typed Caml value `[true; 1; 2]`, since the typechecker does not possess any further type information than `'a list`.

The problem is thus the absence of static connection from exceptions that are raised and the occurrences where they are trapped. Another example would be the one of a function raising `Exc` with an integer or a boolean value, depending on some condition. Then, in that case, when trying to trap these exceptions, we cannot decide wether they will hold integers or boolean values.

Exercises

Exercise 8.1 *Define the function* `find_succeed` *which given a function* `f` *and a list* `L` *returns the first element of* `L` *on which the application of* `f` *succeeds.*

Exercise 8.2 *Define the function* `map_succeed` *which given a function* `f` *and a list* `L` *returns the list of the results of successful applications of* `f` *to elements of* `L`.