CSCI 2041: Exception Handling

Chris Kauffman

Last Updated:
Fri Nov 30 10:26:08 CST 2018

Logistics

Reading

Practical OCaml: Ch 10
Exception Handling

Goals

- Exception Handling
- Control Flow

A5: Calculon

- Arithmetic language interpreter
- 2X credit for assignment
- 5 Required Problems 100pts
- 5 Option Problems 50pts
- Milestone deadline Wed 12/5
- Final deadline Tue 12/11

Exceptions

- **Exceptions** are a means to alter control flow usually associated with **errors**
- Widely used by most modern programming environments
- Briefly discuss **raising or "throwing" exceptions** and defining our own kinds
- Most often you will need to **handle or "catch" exceptions** raised by library code so will spend significant time on this

Declaration and Raising

- Declare with the keyword `exception` and endow with data via same syntax as algebraic types.
- Raise exceptions with `raise` keyword

```
 1  (* declaration is similar to syntax for algebraic types *)
 2  exception Screwup;;
 3
 4  exception Meltdown of int;;
 5
 6  exception Kersplosion of { radius : float;
 7                             damage : int;   };;
 8
 9  (* keyword raise will raise an exception *)
10  raise Screwup;;
11
12  raise (Meltdown 5);;
13
14  raise (Meltdown 2);;
15
16  raise (Kersplosion{radius=5.46; damage=20});;
17
18  let e = Meltdown 2 in           (* create, don't raise *)
19  raise e;;                       (* now raise *)
```

Two Alternatives

- Recall the assoc operation: look up a value based on a key in a list of pairs like this one

```
let alist = [("Bulbasaur"  ,"Grass");
             ("Squirtle"   ,"Water");
             ("Charmander" ,"Fire");]
;;
```

- Contrast List module's assoc_opt and assoc below in functions from print_kind.ml
- Note that assoc may raise a Not_found exception which should be handled in a try/with block
- Experiment with these two in the REPL

```
1  (* look up kind using assoc_opt *)
2
3  let print_kind1 pok =
4    printf "%s: " pok;
5    let result = assoc_opt pok alist in
6    match result with
7    | None       -> printf "Unknown\n"
8    | Some(kind) -> printf "%s\n" kind
9  ;;
```

```
1  (* look up kind using assoc;
2     catch exceptions *)
3  let print_kind2 pok =
4    printf "%s: " pok;
5    try
6      let kind = assoc pok alist in
7      printf "%s\n" kind;
8    with
9    | Not_found -> printf "Unknown\n"
10 ;;
```

From the REPL

- Both functions work identically
- Print "Unknown" when there is something missing from the list

```
# #use "print_kind.ml";;

# print_kind1 "Squirtle";;
Squirtle: Water
- : unit = ()

# print_kind1 "Charmander";;
Charmander: Fire
- : unit = ()

# print_kind1 "Jigglypuff";;
Jigglypuff: Unknown
- : unit = ()

# print_kind2 "Squirtle";;
Squirtle: Water
- : unit = ()

# print_kind2 "Charmander";;
Charmander: Fire
- : unit = ()

# print_kind2 "Jigglypuff";;
Jigglypuff: Unknown
- : unit = ()
```

Error-Checking

- `assoc_opt` follows the old-school approach
  - run a function
  - check immediately whether it succeded
  - handle errors if things went sideways
- This is how non-exception languages like C deal with errors

```
 1  while(1){
 2    printf("query: ");
 3    result = fscanf(stdin,"%s",buf);
 4    if(result==EOF){                // check for error
 5      printf("end of file\n");
 6      break;
 7    }
 8
 9    char *input = malloc(strlen(buf));
10    if(input == NULL){              // check for error
11      printf("out of memory");
12      exit(1);
13    }
```

1. Error-checking is error-prone and tedious
2. No separation of **error generation** from **error policy**: e.g. have to handle errors immediately where they are identified without broader context: motivates exceptions

A Third, Tempting Alternative

- OCaml does not require exception-generating code to be wrapped in a `try/with` block[1]
- This allows below version `print_kind3` to use `assoc` sans `try/with`
- In a REPL, this appears to have no major effect other than not printing `Unknown` like the previous versions

```
1  (* look up kind but don't
2     catch exceptions *)
3  let print_kind3 pok =
4    printf "%s: " pok;
5    let kind = assoc pok alist in
6    printf "%s\n" kind;
7  ;;
```

```
# print_kind3 "Squirtle";;
Squirtle: Water
- : unit = ()
# print_kind3 "Charmander";;
Charmander: Fire
- : unit = ()
# print_kind3 "Jigglypuff";;
Jigglypuff: Exception: Not_found.
#
```

[1] OCaml uses *unchecked exceptions* like most programming languages aside from Java which also has *checked exceptions* which must be either caught or declared in function prototypes.

Consequences of not Catching

- Despite the innocuous appearance in the REPL, exceptions can have dire consequences in programs
- An **unhandled / uncaught exception** typically ends a program (to the dismay of users)

```
 1  (* main loop which asks for
 2     repeated input *)
 3  let _ =
 4    let quit_now = ref false in
 5    while not !quit_now do
 6      printf "query: ";
 7      let pok = read_line () in
 8      if pok="quit" then
 9        quit_now := true
10      else
11        begin
12          print_kind1 pok;
13          print_kind2 pok;
14          print_kind3 pok  (* !! *)
15        end;
16    done;
17  ;;
```

```
> ocamlc print_kind.ml print_kind_main.ml

> a.out
query: Charmander
Charmander: Fire
Charmander: Fire
Charmander: Fire

query: Bulbasaur
Bulbasaur: Grass
Bulbasaur: Grass
Bulbasaur: Grass

query: Pikachu
Pikachu: Unknown
Pikachu: Unknown
Pikachu: Fatal error: exception Not_found

>
```

Getting Exception Backtraces in OCaml

- A **backtrace** shows what functions were active when an exception was thrown
- Useful when programs crash to help diagnose the path to the error condition
- OCaml **disables backtraces** by default
  - Performance is improved by this decision
  - Most other languages w/ exceptions enable backtraces by default to assist with debugging
- Compile with debugging information: `ocamlc -g`
- Enable backtrace printing in one of two ways
  1. Via environment variable `OCAMLRUNPARAM`

     ```
     > ocamlc -g prog.ml
     > export OCAMLRUNPARAM=b
     > ./a.out
     ```

  2. In source code, call `record_backtrace`

     ```
     Printexc.record_backtrace true;;
     ```

- Exceptions that cause the program to crash produce a listing of the functions that were active at the time of the crash

Example: Backtrace for `print_kind_main.ml`

Not going to edit the source code so enable backtraces via command line

```
> ocamlc -g print_kind.ml print_kind_main.ml # compile with debug info
> export OCAMLRUNPARAM=b                     # set env var to enable backtraces
> a.out                                      # run program
query: Squirtle
Squirtle: Water
Squirtle: Water
Squirtle: Water
query: Jigglypuff                            # not found
Jigglypuff: Unknown
Jigglypuff: Unknown
Jigglypuff: Fatal error: exception Not_found                          # BACKTRACE
Raised at file "list.ml", line 187, characters 16-25                  # origin
Called from file "print_kind.ml", line 35, characters 13-28           # active func
Called from file "print_kind_main.ml", line 17, characters 8-23       # active func
```

Exercise: Exceptions Percolate Up

- Exceptions work their way up the call stack
- On the way up, applicable `try/with` blocks are consulted to see if they can handle the exception
- Note that the raise location may be very different from the handle position and may be many function calls away
- What else can go wrong in the main loop?

```
let _ =                    (* inner_catch.ml *)
  let quit_now = ref false in
  while not !quit_now do
      printf "query: ";
      let pok = read_line () in
      if pok="quit" then
        quit_now := true
      else
        try                      (* begin try *)
          print_kind3 pok        (* may throw *)
        with                     (* exc handling *)
        | Not_found -> printf "Oops!\n";
  done;
;;
```

```
> ocamlc print_kind.ml \
          inner_catch.ml
> a.out
query: Bulbasaur
Bulbasaur: Grass
query: Jigglypuff
Jigglypuff: Oops!
query: Pikachu
Pikachu: Oops!
query: Mewtwo
Mewtwo: Oops!
query: quit
>
```

Answers: End of File

```
> ocamlc print_kind.ml inner_catch.ml

> a.out
query: Squirtle                    # found
Squirtle: Water

query: Pikachu                     # not found
Pikachu: Oops!

query: 123                         # "123" not found
123: Oops!

query: !@#@!%891                   # "!@#@!%891"
!@#@!%891: Oops!

query:                             # Press Ctrl-d
Fatal error: exception End_of_file
>
```

- Pressing `Ctrl-d` sends "End of file" character to indicate no more input. Causes `read_line` to rais an exception
- How can this be "fixed"?

Answers: Several Things May Go Wrong

- `print_kind3` may raise `Not_found`
- `read_line` may raise `End_of_file`
- May want to catch both of them

```
let _ =                              (* separate_catch.ml *)
  let quit_now = ref false in
  while not !quit_now do
      printf "query: ";
      let pok =
        try                          (* begin try *)
          read_line ()               (* may throw *)
        with                         (* exc handling *)
        | End_of_file -> "Default"
      in
      if pok="quit" then
        quit_now := true
      else
        try                          (* begin try *)
          print_kind3 pok            (* may throw *)
        with                         (* exc handling *)
        | Not_found -> printf "Oops!\n";
  done;
;;
```

- Starts getting ugly style-wise, like the C-style of immediate error handling after running a function

There are Many Kinds of Exceptions

Exception types are like algebraic variants

- Can carry data, match individual types in `try/with`
- No warnings for missing a relevant type of exception

```
 1  let _ =
 2    let quit_now = ref false in
 3    while not !quit_now do
 4        printf "query: ";
 5        let pok = read_line () in
 6        if pok="quit" then
 7          quit_now := true
 8        else
 9          try
10            print_kind3 pok
11          with                (* no handlers apply to Not_found  *)
12          | Failure msg -> printf "Error: %s!\n" msg;
13          | Invalid_argument a -> printf "Invalid arg!\n";
14    done;
15
16  ---DEMO---
17  > ocamlc print_kind.ml wrong_exc.ml
18  > a.out
19  query: Pikachu
20  Pikachu: Fatal error: exception Not_found
21  >
```

Handle/Catch Cases are like `match/with`

- Match exception specific kinds to appropriate actions
- May include a "catch-all" case with continue or exit actions

```
 1  let _ =                                (* catch_em_all.ml *)
 2    let quit_now = ref false in
 3    while not !quit_now do
 4      try                                (* begin try *)
 5        printf "query: ";
 6        let pok = read_line () in        (* may throw End_of_file *)
 7        if pok="quit" then
 8          quit_now := true
 9        else
10          print_kind3 pok;               (* may throw Not_found *)
11      with                               (* exc handling *)
12      | Not_found -> printf "Oops!\n";
13      | End_of_file -> printf "Catch more!\n"
14      | exc ->                           (* catch any other exception *)
15         printf "\nSomething went wrong somewhere!\n";
16         let excstr = Printexc.to_string exc in
17         printf "Exception: %s\n" excstr;
18         (* keep looping after reporting exception *)
19    done;
```

REPL Session / Wrap-up

Error generation

- Calling `read_line` returns a string but may raise `End_of_file`
- Calling `print_kind3` may raise a `Not_found`

Error-handling policy **specific to this program**

- Print "Oops" for `Not_found`
- Print "Don't leave.." for `End_of_file`

Other programs can establish different error-handling policies like Quit on `End_of_file`

```
> ocamlc print_kind.ml catch_em_all.ml
> a.out
query: Bulbasaur
Bulbasaur: Grass
query: Pikachu
Pikachu: Oops!
query: Catch more!
query: Catch more!
query: Squirtle
Squirtle: Water
query: quit

>
```

Exceptions separate error generation and error handling allowing program-specific policies to handle the same kinds of errors