SAMFYB toc 4ab23de on Mar 1

1 contributor

143 lines (107 sloc) 4.04 KB

Lecture 13 Exception

Exceptions

We can define new exceptions like this:

```
exception Divide (* a new exception *)
Divide : exn (* "extensible" type: you can add new constructor to it at runtime *)
```

A Real Division using Exception

```
(* divide : real * real -> real
 * req : true
 * ens : divide (r1, r2) = r1/r2 unless r2 is small which raises Divide
 *)
fun divide (r1 : real, r2 : real) : real =
  if Real.abs r2 <= 0.00001 then raise Divide else r1 / r2
```

> Note. `op /` has type `real * real -> real`.

Type-Check. If `e : exn` then `raise e : 'a`.

A more complicated Exception

```
exception Rdivide of real
Rdivide : real -> exn
```

```
fun Rdiv (r1, r2) = if Real.abs r2 <= 0.00001 then raise Rdivide r2 else r1 / r2
```

By this, we've done **error signaling**.

Error Handling

```
e handle p1 => e1
         p2 => e2
         ... (* pattern matching exceptions *)
```

Constraint: `e, e1, ..., en` must have the same type! (So we can think of the entirety as an expression.)

No need to be exhaustive: If no pattern matches an exception raised by `e`, then the original exception percolates out.

The N-Queens Problem

We'll implement the solution using three approaches: Exception, Continuation, Option.

Definitions

A position on the board: `int * int`.

```
(* threat : int * int -> int * int -> bool
 * ens : threat P Q decides whether queens P & Q threatens each other
 *)
fun threat (x, y) (a, b) =
  (x = a) orelse (y = b) orelse (x - y = a - b) orelse (x + y = a + b)

(* conflict : int * int -> (int * int) list -> bool
 * conflict p Q decides whether any queen in Q threatens p
 *)
fun conflict (x, y) = List.exists (threat (x, y))
```

Using Exception Handling

```
exception Conflict

(* addqueen : int * int * (int * int) list -> (int * int) list
 * [current col, board size, existing queens, all queens afterward]
 * local helper try : int -> (int * int) list
 * try j will try place i-th queen in row j or higher
 * may raise conflict if not possible => backtrack
 *)
fun addqueen (i, n, Q) =
  let
    fun try j =
      if conflict i j Q then raise Conflict
                        else if i = n then (i, j) :: Q
                                      else addqueen ((i + 1), n, (i, j) :: Q)
      handle Conflict => if j = n then (* backtrack *) raise Conflict
                                  else try (j + 1)
  in
```

```
    try 1
  end

fun queens n = addqueen (1, n, []) handle Conflict => raise Fail "No Solution."
```

Using Continuation

```
(* addqueen : int * int * (int * int) list -> ((int * int) list -> 'a) (* success *)
 *                                         -> (unit -> 'a) (* failure *) -> 'a
 * local helper try : int -> 'a
 *)
fun addqueen (i, n, Q) sc fc =
  let
    fun try j =
      let
        fun fc_new () = if j = n then fc ()
                                 else try (j + 1)
      in
        if conflict (i, j) Q then fc_new ()
                             else
                               if i = n then sc (i, j) :: Q
                                        else addqueen (i + 1, n, (i, j) :: Q) sc fc_new
      end
  in
    try 1
  end

fun queens n = addqueen (1, n, []) SOME (fn () => NONE)
```

Using Option

Idea. Whenever we raised Conflict, we instead return NONE. Wrap SOME on the value returned by `try`. Case on `try` result, if NONE, then try next row, but if hitting top, return NONE to higher level. If SOME result, then hand back result.