15-150
Spring 2018
Dilsun Kaynar
LECTURE 13
Exceptions
(n-queens example)
Today

• Declaring, raising, handling exceptions to
  • Signal error conditions
  • Control flow of computation
Declaring an exception

Standard ML of New Jersey v110.78

exception Divide;

exception Divide

- Divide;

val it = Divide(-) : exn

stands for extensible
Example 1: raising an exception

exception Divide

(* divide : real * real -> real
   REQUIRES: true
   ENSURES: divide(r1,r2) ==> r1/r2 if r2 is not too close to 0.0
            and raises exception Divide otherwise. *)
fun divide(r1, r2) =
  if Real.abs(r2) <= 0.0001 then raise Divide
  else r1/r2

divide(7.1, 0.0) will get back an error: uncaught exception Divide
Example 2: raising an exception

```
exception Rdivide of real

(* Raising an exception with an argument: *)

(* rdivide : real * real -> real
    REQUIRES: none
    Effects:  rdivide(r1,r2) ==> r1/r2 if r2 is not too close to 0.0
             and raises Rdivide(r1) otherwise.  *)

fun rdivide(r1, r2) =
  if Real.abs(r2) <= 0.0001 then raise Rdivide(r1)
  else r1/r2;

rdivide(1.0, 0.000001) handle Rdivide(r) => code that uses r
```
Types and values

• In scope of an exception named Foo, the expression `raise Foo` causes a runtime error
  - `raise Foo;
    uncaught exception Foo`

• The expression `raise Foo` has type `'a`
  and doesn’t evaluate to a proper value

\[
42 + (\text{raise} \ Foo)
\]

\[
(\text{fn} \ x: \text{int} \Rightarrow 0) \ (\text{raise} \ Foo)
\]
handling

\( e_1 \textbf{handle} \langle \text{exn name} \rangle \Rightarrow e_2 \)

- If \( e_1 \) and \( e_2 \) have type \( t \), so does \( e_1 \textbf{handle} \text{Foo} \Rightarrow e_2 \)
- If \( e_1 \) evaluates to \( v \), so does \( e_1 \textbf{handle} \text{Foo} \Rightarrow e_2 \)
- If \( e_1 \) raises \text{Foo}, \( e_1 \textbf{handle} \text{Foo} \Rightarrow e_2 \Rightarrow e_2 \)
- If \( e_1 \) raises \text{Bar}, so does \( e_1 \textbf{handle} \text{Foo} \Rightarrow e_2 \)
- If \( e_1 \) loops, so does \( e_1 \textbf{handle} \text{Foo} \Rightarrow e_2 \)
The scope of the handler for `Foo` in

```plaintext
e handle Foo => e'
```

is `e`.

Can also combine handlers

```plaintext
e handle Ringerding => e_1'
  | Hatee-hatee-ho => e_2'
  | Wa-pow-pow => e_3'
  | _ => raise NotFox
```

`(e, e_1', e_2', e_3' must have the same type)`
Caution

- Never misspell an exception name because it will be regarded as a variable and match all exceptions

- if E then E1 else E2 handle ...
  The handler will handle only exceptions raised by E2

- case E of P1 => E1 | ... | Pn => En handle ...
  The handler will handle only exceptions raised by En

Don’t forget to use parentheses when necessary
4-queens

column

row

Q
Cannot place 3rd queen
Need to backtrack and undo choice for 2nd queen
Cannot place 4th queen

Need to backtrack and undo choice for 3rd queen
Cannot place 3rd queen

Need to backtrack and undo choice for 3rd queen but there is nothing else

Need to backtrack and undo choice for 2nd queen
Cannot place 2nd queen — we already tried everything else

Need to backtrack and undo choice for 1st queen
Done!
Conflict detection

(* threat : (int*int) -> (int*int) -> bool
  REQUIRES: true
  ENSURES: threat p q ===> true, if position p is threatened
           by a queen at position q;
           False otherwise.
*)

fun threat (x, y) (x',y') =
  (x=x') orelse (y=y') orelse (x+y = x'+y') orelse (x-y = x'-y')

(* conflict : (int*int) -> (int*int) list -> bool
  REQUIRES: true
  ENSURES: conflict p Q ===> true, if position p is threatened
            by any queen in the list of
            positions Q;
            false, otherwise.
*)

fun conflict pos = List.exists (threat pos)

List.exists :('a -> bool) -> 'a list -> bool
n-queens with exceptions

(* addqueen : int * int * (int * int) list -> (int * int) list
   REQUIRES: Q is a list of conflict-free queen positions on an
   n x n board, of the form
   [(i-1, _), (i-2, _), ... (1, _)],
   with 1 <= i <= n   (i=1 means Q is nil).
   ENSURES:  addqueen(i, n, Q) extends Q to a conflict-free placement
   of n queens if that is possible,
   and raises exception Conflict otherwise. *)
fun addqueen(i, n, Q) = 
  let
    fun try j =
      (if conflict (i,j) Q then raise Conflict
       else if i=n then _________
       else ______________________)
        handle Conflict => (if j=n then raise Conflict
                              else try(j+1))
  in
    try 1
  end
n-queens with exceptions

exception Conflict

(* addqueen : int * int * (int * int) list -> (int * int) list *)

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 ____________________________
          handle Conflict => (if j=n then raise Conflict
                                     else try(j+1))
      in
      try 1
  end
exception Conflict

fun addqueen(i, n, Q) =
  let
    fun try j =
      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 raise Conflict
                                  else try(j+1))
      in
        try 1
      end
  in
    try 1
  end
n-queens with exceptions

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 raise Conflict
      else try(j+1))
  in
    try 1
  end

(* val queens : int -> (int * int) list
  REQUIRES: n >= 1
  ENSURES:  queens(n) returns a list of n conflict-free queen
            positions on an n x n board if that is possible,
            and raises exception Conflict otherwise. *)

fun queens(n) = addqueen(1, n, [])
with a success and failure continuation

(* addqueen : int*int*(int*int) list -> ((int*int) list -> 'a) -> (unit -> 'a) -> 'a *)

fun addqueen (i, n, Q) sc fc =
  let
    fun try j =
      let
        fun fcnew () = if j=n then fc() else try(j+1)
      in
        if (conflict (i,j) Q) then __________
        else if i=n then ________________
        else ____________________________
      end
    in
      try 1
    end
with a success and failure continuation

(* addqueen : int*int*(int*int) list -> ((int*int) list -> 'a) -> (unit -> 'a) -> 'a *)

fun addqueen (i, n, Q) sc fc = let
  fun try j = let
    fun fcnew () = if j=n then fc() else try(j+1) in
    if (conflict (i,j) Q) then fcnew()
    else if i=n then ___________________
    else ____________________________
  end in
  try 1 end
with a success and failure continuation

(* addqueen : int*int*(int*int) list -> ((int*int) list -> 'a) -> (unit -> 'a) -> 'a *)

fun addqueen (i, n, Q) sc fc =
  let
    fun try j =
      let
        fun fcnew () = if j=n then fc() else try(j+1)
      in
        if (conflict (i,j) Q) then fcnew()
        else if i=n then sc((i,j)::Q)
        else _________________________________
      end
    in
      try 1
    end
with a success and failure continuation

(* addqueen : int*int*(int*int) list -> ((int*int) list -> 'a) -> (unit -> 'a) -> 'a *)

fun addqueen (i, n, Q) sc fc =
  let
    fun try j =
      let
        fun fcnew () = if j=n then fc() else try(j+1)
      in
        if (conflict (i,j) Q) then fcnew()
        else if i=n then sc((i,j)::Q)
        else addqueen(i+1, n, (i,j)::Q) sc fcnew
      end
    in
      try 1
    end
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 raise Conflict 
                            else try(j+1)) 
  in 
    try 1 
  end 

fun addqueen (i, n, Q) sc fc = 
  let 
    fun try j = 
      let 
        fun fcnew () = 
          if j=n then fc() else try(j+1) 
      in 
        if (conflict (i,j) Q) then fcnew() 
        else if i=n then sc((i,j)::Q) 
        else addqueen(i+1, n, (i,j)::Q) sc fcnew 
      end 
    in 
      try 1 
    end 

side by side

using exceptions

using continuations
n-queens with options

(* addqueen : int * int * (int * int) list ->
   (int * int) list option

REQUIRES: Q is a list of conflict-free queen positions on an
n x n board,
of the form [(i-1, _), (i-2, _), ... (1, _)],
with 1 <= i <= n   (i=1 means Q is nil).
ENSURES: addqueen(i, n, Q) returns SOME(Q'),
where Q' extends Q to
a conflict-free placement of n queens if that is
possible and returns NONE otherwise.)
fun addqueen(i, n, Q) =
  let
    fun try j=
      fun try j=
        case (if conflict (i,j) Q then _________
             else if i=n then _________________
             else ____________________________)
          of NONE => if (j=n) then NONE else try(j+1)
          | result => result
    in
      try 1
    end

n-queens with options
n-queens with options

fun addqueen(i, n, Q) =
  let
    fun try j =
      case (if conflict (i,j) Q then NONE
       else if i=n then SOME((i,j)::Q)
       else addqueen(i+1, n, (i,j)::Q))
      of NONE => if (j=n) then NONE else try(j+1)
       | result => result
    in
      try 1
    end
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 raise Conflict
                         else try(j+1))
  in
    try 1
  end

fun addqueen (i, n, Q) sc fc =
  let
    fun try j =
      let
        fun fcnew () =
          if j=n then fc() else try(j+1)
      in
        if (conflict (i,j) Q) then fcnew()
        else if i=n then sc((i,j)::Q)
        else addqueen(i+1, n, (i,j)::Q) sc fcnew
      end
    in
      try 1
    end

fun addqueen(i, n, Q) =
  let
    fun try j =
      case (if conflict (i,j) Q then NONE
             else if i=n then SOME((i,j)::Q)
             else addqueen(i+1, n, (i,j)::Q))
     of NONE => if (j=n) then NONE else try(j+1)
     | result => result
  in
    try 1
  end