15–150: Principles of Functional Programming

n-Queens

Spring 2025*

Caution: This document implements n-Queens slightly differently than does the lecture code. The overall algorithm is the same conceptually. However, details are different. For instance, columns and rows are interchanged and some of the functions have different types. You should be able to spot those differences without difficulty.

1 The n-Queens Problem

The n-queens problem is to find a placement of n queens on an n-by-n board in which no queen is attacking another queen (along a row, a column, or a diagonal). We can represent the data here very simply. (In this document, we represent rows and columns each via integers in the range 1 to n and we represent board positions as pairs of such integers.)

```
type row = int
type col = int
type pos = row * col
type sol = pos list
```

A value of type `sol` is a "possible solution", i.e., a list of positions; each position is a pair consisting of a row number and a column number. Two positions are in attack if they have the same row number, the same column number, or they lie on the same diagonal. There are two diagonals to check.

The following function, `threat:pos * pos -> bool`, tests if a pair of positions are in attack.

```
fun threat((x,y), (i,j)) =
   (x+y = i+j) orelse (x-y = i-j) orelse (x=i) orelse (y=j)
```

For all `p,q:pos`, `threat (p, q) ==> true` if a queen at `p` attacks a queen at `q`;
`threat (p, q) ==> false` otherwise.

One can now check if a position is attacked by *some* member of a list of positions:

```
(* conflict : pos * pos list -> bool *)
fun conflict (p, nil) = false
  | conflict (p, q::qs) = threat(p, q) orelse conflict(p, qs)
```

`conflict (p, qs) ==> true` if a queen at `p` is attacked by one of the positions in `qs`;
`conflict (p, qs) ==> false` otherwise.

*Adapted nearly verbatim by Michael Erdmann from an earlier document by Stephen Brookes.

And one can define a function that checks if a list of positions is safe, or attack-free (relative to itself):

```
(* safe : pos list -> bool *)
fun safe [ ] = true
  | safe (p::qs) = not(conflict(p, qs)) andalso safe qs
```

It will be helpful to introduce some auxiliary functions:

```
fun rows qs = map (fn (x, y) => x) qs
fun cols qs = map (fn (x, y) => y) qs
```

A list `qs` such that `safe(qs) ==> true` is called a *partial solution.*

A partial solution `qs` is an *n-queens solution* if `rows qs` is a permutation of `[1,...,n]` and every member of `cols qs` is in `[1,...,n]`. This is enough to imply that `qs` has a queen for each row and a queen for each column in the n-by-n board, and there are no attacks.

The infamous "British Museum Algorithm" for the n-queens problem builds a list of all permutations of `[1,...,n]`, treats each such permutation as a possible column assignment for rows 1 through n, and checks each such combination for safety. This is horrendously slow: there are $n!$ permutations of the list `[1,...,n]`, and only a tiny fraction of these are actually attack-free.

Far better is a *backtracking algorithm* that works with partial solutions, starting with the trivial partial solution and trying to grow it by adding queens in successive rows; as soon as we find that the "current" partial solution cannot be extended safely, we "backtrack" to a partial solution for which there were more possibilities to explore.

2 n-Queens with Continuations

Since you have recently worked with continuations, we will first implement backtracking using "failure continuations", then later give an implementation that uses exceptions. Recall that a failure continuation is a function with a dummy argument, of the trivial type `unit`, the type with a single value `( )`, which is pronounced "unit, of type `unit`". For the result type of a continuation here we take an option type; an "answer" is either `SOME qs`, where `qs` is a solution for the n-queens problem; or `NONE`, meaning that there is no solution. We are only searching for *some* solution, if one exists. So the use of options and `SOME` has an intuitively appealing flavor.

```
type ans = sol option
(* SOME(qs) means "found solution qs"; NONE means "no solution". *)

type cont = unit -> ans
(* We let a "failure" continuation be a function from unit to answers. *)
(* The function body is not evaluated unless we apply the function.    *)
```

The key ingredient in our code development is the helper function

```
try : int * row * col list * sol -> cont -> ans
```

which is designed to meet the following specification:

```
(* REQUIRES: 1<=i<=n and A is a sublist of [1,. . .,n]               *)
(*           and qs is a partial solution using rows 1 through i-1   *)

(* ENSURES:                                                         *)
(*  EITHER there is an n-queens solution extending qs               *)
(*         with a queen in row i at a column in A,                  *)
(*         and  try (n,i,A,qs) fc ==> SOME(qs')                     *)
(*         for one such solution qs';                               *)
(*   OR there is no n-queens solution extending qs                  *)
(*        with a queen in row i at a column in A,                   *)
(*           and  try (n,i,A,qs) fc  ==>  fc ( ).                   *)
```

Intuitively, `try` takes as its argument a tuple `(n, i, A, qs)` in which `n` is the board size, `i` is a row index (between 1 and `n`), `A` is a list of column indices (all between 1 and `n`), and `qs` is a "current" partial solution; it returns a function of type `cont -> ans`, that can be applied to a "failure continuation" to obtain an answer.

The design of the SML code for `try(n, i, A, qs) k` is based on the following observations. We assume (as in the spec above) that `1<=i<=n`, `A` is a sublist of `[1,...,n]` and `qs` is a partial solution using rows 1 through `i-1`.

- If `A` is the empty list, there is no n-queens solution extending `qs` as desired (there are no columns for the $\texttt{i}^{th}$ queen).
- If `A` is non-empty, `j` is the first column in `A`, and `B` is the rest of the columns,
  - either a queen in row `i` at column `j` would be attacked by some queen in `qs`, and we should try row `i` and the columns in `B`;
  - or it's safe to extend `qs` with a queen at `(i,j)`; if `i=n` we've reached the final row on the chessboard, and `(i,j)::qs` is a full solution; otherwise, we should try for a solution extending `(i,j)::qs` with a queen in row `i+1` (at any column in `[1,...,n]`); if this fails, we should *backtrack* and try for a solution extending the original `qs` using the columns in `B`.

```
fun try (n, i, A, qs) fc =
    (case A of
        [ ]    =>  fc ( )
      | (j::B) =>  if conflict((i, j), qs)
                   then try (n, i, B, qs) fc
                   else if i = n
                        then SOME( (i, j)::qs )
                        else
                            try(n, i+1,  upto(1, n), (i, j)::qs)
                                (fn ( ) => try (n, i, B, qs) fc))
```

Here `upto(1, n)` evaluates to the list `[1,...,n]`.

Make sure you see how this code implements the algorithm described in the comments above. Check also that if the tuple `(n, i, A, qs)` satisfies the REQUIRES conditions, the recursive calls to `try` will also be on tuples that satisfy the REQUIRES conditions. Finally, notice that in every recursive call to `try`, either the value of `n-i` decreases (and is non-negative), or it stays the same and the length of the column list decreases. This is why the code does not loop forever.

Using `try`, we may define a solver for the n-queens problems:

```
(* queens : int -> ans                                    *)
(* REQUIRES: n>0                                          *)
(* ENSURES: If there is an n-queens solution              *)
(*          then  queens n ==> SOME qs,                   *)
(*          where qs is one such solution.                *)
(*          If there is no solution, queens n ==> NONE.  *)

fun queens n = try (n, 1, upto(1, n), nil) (fn ( ) => NONE)
```

Examples:

```
- queens 4;
val it = SOME [(4,3),(3,1),(2,4),(1,2)] : (int * int) list option
- queens 5;
val it = SOME [(5,4),(4,2),(3,5),(2,3),(1,1)] : (int * int) list option
- queens 3;
val it = NONE : (int * int) list option
- queens 8;
val it = SOME [(8,4),(7,2),(6,7),(5,3),(4,6),(3,8),(2,5),(1,1)]
  : (int * int) list option
- queens 20;
val it =
  SOME
    [(20,11),(19,6),(18,14),(17,7),(16,10),(15,8),(14,19),(13,16),(12,9),
     (11,17),(10,20),(9,18),...] : (int * int) list option
```

3 n-Queens with Exceptions

We now give an exception-based version of the backtracking function that we used to implement the n-queens algorithm previously. We include it here without much commentary, leaving it up to you to try this code out and verify that it works as desired.

```
type row = int
type col = int
type pos = row * col
type sol = pos list

fun threat((x,y), (i,j)) =
   (x+y = i+j) orelse (x-y = i-j) orelse (x=i) orelse (y=j)

fun conflict(p, nil)=false
 |  conflict(p, q::qs) = threat(p, q) orelse conflict(p, qs)

exception Impossible

(* try : int * row * col list * sol -> sol *)

(* REQUIRES: 1<=i<=n and A is a sublist of [1,. . .,n]                    *)
(*           and qs is a partial solution using rows 1 through i-1.       *)

(* ENSURES:                                                               *)
(* EITHER qs extends to a solution with a queen in row i at a column in A, *)
(* and try(n,i,qs,A) computes one such solution;                           *)
(* OR qs cannot be so extended, and try(n,i,A,qs) raises Impossible.       *)

fun try(n, i, A, qs) =
    (case A of
        [ ]    =>  raise Impossible
      | (j::B) =>  if conflict((i, j), qs)
                   then try(n, i, B, qs)
                   else if i = n
                        then (i, j)::qs
                        else try(n, i+1,  upto(1, n), (i, j)::qs)
                               handle Impossible => try(n, i, B, qs))

(* queens : int -> ans                                                    *)
(* REQUIRES: n>0                                                          *)
(* ENSURES:                                                               *)
(* If there is an n-queens solution,                                      *)
(*  queens n    evaluates to SOME(qs),  where qs is one such solution.    *)
(* If there is no solution, queens n evaluates to NONE.                   *)

fun queens n = SOME(try (n, 1, upto(1, n), nil)) handle Impossible => NONE
```