These notes are new to 15-150 in Fall 2019. This semester is the first time we have lectured on this material, and these notes are fresh off the press. Please feel free to email me if you have any suggestions for improvement, or find any typos or inconsistencies.
1 Outline

We will develop solutions to a combinatorial problem with a large search space. Similar techniques are applicable to a pretty wide range of real-world problems, although our chosen problem is more of a “recreational math” classic. An advantage of choosing such a problem is that you should be familiar with the background (chess!) and we can avoid the need for elaborate set-up.

First we’ll design a “direct-style” (pretty conventional) solution to the problem. Then we’ll point out some obvious defects, and we’ll introduce a continuation-style solution that’s intended to avoid inefficiencies. Even though the direct-style design will turn out to be (arguably) less useful in practice, the design is still interesting as a vehicle for the tasteful use of higher-order list-manipulation functions like \texttt{map} and \texttt{foldl} or \texttt{foldr}. Once we’ve developed the continuation-style solution it will be evident that further improvements are possible, to exploit the potential for parallel evaluation.

The main themes are (again)

- Designing higher-order functions
- Recursion
- Using maps and folds
- Specifications, proofs and efficiency

We won’t include correctness proofs here, although in class we discussed why the code is correct in real time, along with the code development. We hope you can fill in the missing details. In class we also showed grid diagrams of some of the results produced by our functions. We encourage you to draw these yourself to help get familiar with the set-up.

Warning: When we show snapshots from an ML session we will usually “sanitize” the type reported by ML to make it more easily readable. In particular, we will introduce some type abbreviations (like \texttt{pos}) and we’ll pretend that ML always knows to use abbreviated type names. (It doesn’t.) Instead ML often reports the full type without abbreviation. In contrast we may sometimes give a specification for a function in which we specify a type that’s not the most general one(!). We’ll do this when we need to use a specific instance of the most general type.
2 The Bishops Problem

The standard chessboard is an 8-by-8 grid of alternating black and white squares. Bishops are chess pieces that attack along diagonals. How many bishops can you put onto an n-by-n chessboard so that each one is threatened by at most one other? For small values of n this is easy to figure out. But you may be surprised to discover that it’s possible to fit 20 bishops on the standard 8-by-8 chessboard. Let’s develop a straightforward way to solve this problem. We’ll begin by introducing some basic types and functions.

The chess board

type pos = int * int;
(* A value (x,y) of type pos represents a square position. *)

fun cart ([ ], B) = [ ]
| cart (a::A, B) = map (fn b => (a, b)) B @ cart (A, B);
(* cart : int list * int list -> pos list
    ENSURES cart (xs, ys) = a list of all pairs (x,y) with x in xs, y in ys *)

fun upto i j = if i>j then [ ] else i::upto (i+1) j;
(* upto : int -> int -> int list
    Requires n >= 1
    Ensures upto 1 n = [1,2,...,n] *)

fun board n = let val xs = upto 1 n in cart (xs, xs) end;
(* board : int -> pos list
    Requires n >= 1
    Ensures init n = a list of the squares of the n-by-n chessboard *)

- board 3;

val it = [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)] : pos list

Obviously the most general type of cart is polymorphic, but we’ll only need to use it with a pair of integer lists.
When bishops attack

fun threat (x,y) (i,j) = if (abs (x - i) = abs (y - j)) then 1 else 0;
(* threat : pos -> pos -> int
   ENSURES threat b c = 1 if bishop at b threatens bishop at c
   threat b c = 0 otherwise *)

fun attacks (b, L) = foldr (fn (c, k) => threat b c + k) 0 L;
(* attacks : pos * pos list -> int
   ENSURES attacks (b, L) = number of threats on b by members of L *)

fun forall p L = foldr (fn (x, t) => (p x) andalso t) true L;
(* forall : (pos -> bool) -> pos list -> bool
   REQUIRES p is total
   ENSURES forall p L = true if every element of L satisfies p
   forall p L = false otherwise *)

fun safe L = forall (fn b => attacks (b, L) <= 2) L;
(* safe : pos list -> bool
   ENSURES safe L = true if no member of L is threatened by more than 1 other
   safe L = false otherwise *)

- safe [(1,1),(1,2),(1,3),(2,1)];
  val it = true : bool

- safe [(1,1),(1,2),(1,3),(2,1),(2,2)];
  val it = false : bool

Check that you understand why the safety check looks for at most 2 threats, rather than at most one! It’s the correct thing to do, given how we wrote the function.
3 A direct-style implementation

Our first implementation will work with states consisting of a pair \( (bs, rest) \) where \( bs \) represents a partial solution — a safe placement of some bishops — and \( rest \) is a list of the remaining squares that are candidates for extending \( bs \). Here is a function \texttt{step} that calculates a list of the safe ways to extend \( bs \) with one more bishop at a square chosen from \( rest \). It uses the obvious helper function for deleting a square. We introduce the predicate \texttt{is_valid_state : state \rightarrow bool} to characterize what we mean by valid states. All state values constructed by our problem solving functions will be valid, and the correctness of our design relies on this fact.

\[
\begin{align*}
\text{type sol} & = \text{pos list}; \\
\text{type state} & = \text{sol} * \text{pos list}; \\
\text{fn is_valid_state (bs, rest)} & = \text{safe bs}; \\
\text{fun del b [ ]} & = [ ] \\
| \quad \text{del b (c::cs)} = \text{if b=c then cs else c::del b cs}; \\
(* \text{del : pos \rightarrow pos list - pos list}
\quad \text{REQUIRES bs is a list with no repeated elements}
\quad \text{ENSURES del b bs = result of deleting b from bs}
*)
\end{align*}
\]

\[
\begin{align*}
\text{fun steps (bs, rest)} & = \\
\quad \text{let}
\quad \quad \text{val R = List.filter (fn b =\rightarrow safe(b::bs)) rest}
\quad \text{in}
\quad \quad \text{map (fn b =\rightarrow (b::bs, del b rest)) R}
\quad \text{end};
\end{align*}
\]

\[
\begin{align*}
(* \text{steps : state \rightarrow state list}
\quad \text{REQUIRES is_valid_state(bs, rest)}
\quad \text{ENSURES steps (bs, rest)} =
\quad \quad \text{a list of all valid states obtainable by}
\quad \quad \text{extending bs safely with one b drawn from rest}
\quad *)
\end{align*}
\]
steps (bs, rest) gives a list of valid states, containing all the partial solutions obtained by extending bs with a bishop at a position in rest and deleting the extra bishop from the remaining squares. Each of these states has form (b::bs, del b rest) where b is in R and safe (b::bs) is true. For example:

- steps ([(1,1),(1,2),(1,3)],[(2,1),(2,2),(2,3)]);
  val it = 
  [[[2,1,1,2,1,3],[2,2,2,3]],
   [(2,3,1,2,1,3),(2,1,2,2)]]

Here there are only two possible extra squares ((2,1) and (2,3)) that are safe to extend with. Similarly,

- steps ([(1,1),(1,2),(1,3)],[(2,2)]);
  val it = [ ]

Now the function that does all the interesting work! The search function maintains a list of candidate states to be explored, and looks for a way to extend one of these states to arrive at a safe placement satisfying the predicate p. The bishops_direct function starts from a singleton list with just the initial state (no bishops so far, all squares as candidates) and looks for a safe placement of a given length. In the specification, we say that position list S extends state (bs, rest) if S is an extension of bs with squares drawn from rest. In other words, S is obtained from bs by consing on some number of squares from rest.

(*
  search : (pos -> bool) -> state list -> (pos list) option
  REQUIRES p is total, every state in L is valid
  ENSURES search p L
  = SOME S, 
    where S is a safe placement satisfying p 
    that extends some state in L, 
    if there is one
  = NONE, 
    otherwise
*)
fun search p [ ] = NONE
| search p ((bs, rest)::xs) = 
  if (p bs) then SOME bs else search p (steps (bs, rest) @ xs);

We use \texttt{search} as the engine that drives our bishops solver. We also introduce a convenient helper to build an "initial" state.

(*
  init : int -> state
  REQUIRES n >= 0
  ENSURES init n = ([ ], board n)
  Note: init n is a valid state
*)

fun init n = ([ ], board n);

(* bishops_direct : int -> int -> (pos list) option
  REQUIRES n>=0, m>=0
  ENSURES bishops_direct n m
  = SOME S,
  where S is a safe placement of length m
  on the n-by-n board,
  if there is one
  = NONE,
  otherwise
*)

fun bishops_direct n m = 
  search (fn bs => (length bs = m)) ([ ], board n)];

Even though we don’t include proof details here, it should be clear that \textit{assuming} the search function satisfies its specification it will follow easily that the \texttt{bishops\_direct} function works as specified.

It might be helpful to think of the "state space" being explored by \texttt{search p ([ ], board n)]} as a directed graph whose nodes are states; each state has an edge to each of the states reachable by safely adding one extra square. (In other words, the graph is “generated” by the \texttt{steps} function.) Essentially, because the search function prepends the successors of the
first state to the list of the remaining states, its behavior is like a depth-
first search of this graph. (Of course the function doesn’t actually build
the graph; this description is merely intended to link up with your intuition
about graph-searching, in case you spotted the analogy.)

Let’s see an example of this code in action:

- bishops_direct 4 8;
val it = SOME [(4,4),(4,1),(2,4),(2,1),(1,4),(1,3),(1,2),(1,1)]
  : pos list option

- safe [(4,4),(4,1),(2,4),(2,1),(1,4),(1,3),(1,2),(1,1)];
val it = true : bool

The previous result came pretty quickly. But the next example takes a
LONG time:

- bishops_direct 6 14;
val it =
  SOME
  [(6,6),(6,4),(6,3),(6,1),(5,6),(5,1),(3,5),
   (3,2),(1,6),(1,5),(1,4),(1,3),(1,2),(1,1)]
  : pos list option

If we attempt to find a placement for 20 bishops on the 8-by-8 board, the
ML runtime system just sits there taking an extremely long time, so much
so that I gave up waiting:

- bishops_direct 8 20;
  .......... taking O(forever)

Nevertheless we can show (using induction) that the functions satisfy their
specifications, so *in principle* (if we were patient enough) a correct answer
should reveal itself eventually. This is hardly satisfactory, and we’ll soon
show how to improve matters.
Using the search function it’s easy to implement a simple-minded iterative algorithm for finding the maximum number of bishops for a given \( n \). We include a `print` instruction to help us visualize what’s happening when the code runs. Unfortunately we will soon see that this method is hopelessly inefficient except for very small value of \( n \).

```ml
fun maximum_bishops_direct n = 
let
  fun loop m =
    (print ("Trying " ^ Int.toString m ^ "\n");
     case (bishops_direct n m) of SOME _ => loop(m+1) | NONE => m-1)
  in
  loop 1
end;

Here are some results:

- `maximum_bishops_direct 4`;

Try 1
Try 2
Try 3
Try 4
Try 5
Try 6
Try 7
Try 8
Try 9
val it = 8 : int

Check that 8 is indeed the maximum number for board size 4-by-4.
Assessment

- Even for reasonably small values of \( n \) and \( m \), the running times of the direct solver are noticeably slow. The result for \texttt{bishops\_direct 4 8} is found quickly, but there’s a definite lag before the result for \( 6\ 14 \) shows up. The time taken by \texttt{maximum\_bishops\_direct 4} is significant — minutes rather than seconds, and definitely not just a few milliseconds.

- This is at least partly caused by the fact that the solver maintains a list of candidates to be potentially explored, and this list can get very large. And even figuring out the list of all safe ways to extend a state is wasteful – there may be many safe extensions but we only care about finding one.

- If we switch to the alternative search function (the one that puts the successor states at the end of the list of pending states) things get even slower! Even for \( 4\ 8 \) the result takes ridiculously long (I gave up!) Why is this happening?

- Surely there must be a way to avoid building candidate lists like this.

- Of course there is...
4 A continuation-style design

We modify the search function (and also change its name). The new `solve` function maintains a predicate and a single state (not a list of states), and has two extra arguments: a “success” continuation and a “failure” continuation. We’ll design the function carefully so that the failure continuation will allow exploration of alternative extensions, if and when needed. Let `answer` be the type `(pos list) option`. Our `solve` function will have the following type:

```
solve : (pos list -> bool) -> state ->
       (pos list -> answer) -> (unit -> answer) -> answer
```

It will satisfy the following specification:

- **REQUIRES**
  - `p` total, `bs` is a safe placement
- **ENSURES**
  - `solve p (bs, rest) s k` = `s L`,
    - where `L` is a safe placement
    - satisfying `p` that extends `bs` with squares from `rest`,
      - if there is one
    - = `k( )`,
    - otherwise

Here is the definition of `solve`. Check how each clause in the function definition is motivated by some obvious reasoning. For example, if `bs` already satisfies `p` then we simply “succeed”, passing `bs` to `s`. Otherwise, if `rest` is empty (and `p(bs)` is `false`), there is no possibility of success, so we “fail” by calling `k` with its dummy argument. And if `rest` is not empty we try finding a satisfactory extension using the first square in `rest` (if it’s safe to do so), with a failure continuation that will, if called, try using the remaining candidate squares. If it isn’t safe to use the first square, we simply try the remaining ones. The ML code design is almost a literal translation of these informally expressed ideas into functional notation!
fun solve p (bs, rest) s k = 
  if p(bs) then s(bs) else 
  case rest of 
    [ ] => k( ) 
  | b::cs => if safe(b::bs) 
    then 
      solve p (b::bs, cs) s (fn () => solve p (bs, cs) s k) 
    else 
      solve p (bs, cs) s k;

Now we can define an answer-finding function by instantiating the success 
and failure continuations in the obvious way:

(*
  bishops_cps : int -> int -> sol option
  REQUIRES n>=0, m>=0
  ENSURES
    bishops_cps n m
    = SOME S,
      where S is a safe placement of length m
      on the n-by-n board,
      if there is one
    = NONE,
      otherwise
*)

fun bishops_cps n m = 
  solve (fn L => length L = m) (init n) SOME (fn ( ) => NONE);

fun maximum_bishops_cps n = 
  let fun loop m = (print ( "Trying Int.toString m \n")); 
    case (bishops_cps n m) of SOME _ => loop(m+1) | NONE => m-1) 
  in 
    loop 1 
  end;

The cps-style functions give responses noticeably faster than the direct-style 
versions from before. And they produce the same answers!
- bishops_cps 4 8;
val it = SOME [(4,4),(4,1),(2,4),(2,1),(1,4),(1,3),(1,2),(1,1)]
  : pos list option

- bishops_cps 6 14;
val it =
  SOME
  [(6,6),(6,4),(6,3),(6,1),(5,6),(5,1),(3,5),
   (3,2),(1,6),(1,5),(1,4),(1,3),(1,2),(1,1)]
  : pos list option

In some cases where direct-style search was painfully long, cps is quicker:

- bishops_cps 8 20;
val it =
  SOME
  [(8,8),(8,5),(8,4),(8,1),(7,8),(7,5),(7,4),(7,1),(6,8),(6,1),
   (3,7),(3,2),(1,8),(1,7),(1,6),(1,5),(1,4),(1,3),(1,2),(1,1)]
  : pos list option

- maximum_bishops_cps 4;
  Trying 1
  Trying 2
  Trying 3
  Trying 4
  Trying 5
  Trying 6
  Trying 7
  Trying 8
  Trying 9
val it = 8 : int (* FAST *)
However, for slightly larger \( n \) we may be able to get a response (unlike before) but only after waiting a while:

\[- \text{maximum_bishops_cps} \; 6;\]

Trying 1
Trying 2
Trying 3
Trying 4
Trying 5
Trying 6
Trying 7
Trying 8
Trying 9
Trying 10
Trying 11
Trying 12
Trying 13
Trying 14
Trying 15

val it = 14 : int (* SLOW *)

How about \( n=8 \)? Try it!

**Assessment**

- The cps-style solver is appreciably faster than the direct-style searcher.
- We make no attempt to calculate asymptotic work, as it is very hard to figure out the details. (Try if you wish!)
- Regardless, there are some worthwhile lessons here in program design methodology.
5 Exploiting polymorphism

We annotated our code so far with types and specifications, chosen to help you understand how the various functions worked and fit together. However, we were unnecessarily specific about the types — especially in the cps design. It turns out that the type answer (our abbreviation for (pos list) option) can be replaced with a type variable... in the type for the solve function.

```plaintext
- solve;
val it = fn
  : (pos list -> bool) -> state ->
     -> (pos list -> 'a) -> (unit -> 'a) -> 'a
```

This means that we can use the solve function in numerous other ways, by picking a specific type for answers and supplying some correspondingly typed continuations. We actually chose answers to be (pos list) option and used SOME : pos list -> (pos list) option and (fn ( ) => NONE) : unit -> (pos list) option as the continuations used by bishops_cps.

As an easy but uninteresting variation, we could have chose bool for the answer type and picked fn _ => true and fn ( ) => false as success and failure continuations. (What would we achieve that way?)

As a more interesting challenge, find a way to write a function that finds a list of all safe placements of m bishops on the n-by-n board:

```plaintext
(* all_bishops : int -> int -> (pos list) list
   REQUIRES n >= 1, m >= 1
   ENSURES all_bishops n m =
       a list of all safe placements
       of m bishops on n-by-n board
*)
```
6 Exploiting symmetry

So far we have completely ignored an important fact about chess.

- When \( n \) is even, the board consists of equal numbers of white squares and black squares. Bishops on white squares are only threatened by other white squares. Similarly for black.

- This means that we don’t really need to keep a single list of bishops of both colors, and it would be an advantage to work independently on the two colored regions.

We’ll use the following helper function to split the initial board into the two colored sub-boards.

\[
\text{fun split } p \ [ ] = (\ [ ] , \ [ ] ) \\
| \text{split } p \ (x::L) = \text{let} \\
| \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{val } (A,B) = \text{split } p \ L \\
| \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{in} \\
| \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if } (p \ x) \ \text{then } (x::A,B) \ \text{else } (A, x::B) \\
| \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{end;}
\]

\[
\text{(* fast_bishops_cps : int } \to \text{ int } \to \text{ (pos list) option} \\
\text{REQUIRES } n \text{ even, } m \text{ even} \\
\text{ENSURES} \\
\text{fast_bishops_cps } n \ m \\
\text{= SOME } bs, \\
\text{where } bs \text{ is a safe placement for } m \text{ bishops on } n\text{-by-}n \text{ board,} \\
\text{if there is one} \\
\text{= NONE, otherwise} \\
\text{*)}
\]
fun fast_bishops_cps n m = 
  let
    val halfm = m div 2
    fun p L = (length L = halfm)
    val (evens, odds) = split (fn (x, y) => (x+y) mod 2 = 0) (board n)
    val (Evens, Odds) =
    (solve p ([ ], evens) SOME (fn () => NONE),
      solve p ([ ], odds) SOME (fn () => NONE)
    )
  in
    case (Evens, Odds) of
      (SOME L, SOME R) => SOME (L@R)
    | (_, _) => NONE
  end;

fun fast_maximum_bishops n =
  let fun loop m = (print ("Trying ^Int.toString m ^ "\n");
      case (fast_bishops n m) of SOME _ => loop(m+1) | NONE => m-1)
  in
    loop 1
  end;

See how the code design here allows for parallel evaluation on the even and the
odd squares (in other words, on white vs black squares). Tuple expressions
can be evaluated in parallel.

Try running this code – you’ll again see fast responses (until the problem
size gets too big!).

• If we are allowed to evaluate independent code in parallel at no cost, this
code seems likely to run even faster. Even when running sequentially
on a single processor it is appreciably quicker than the previous code.

• But it turns out we can exploit symmetry yet again to avoid the need for
parallelism! If we just look for a white solution, it’s easy to turn it into
a black solution with a single map operation (applying a constant-time
function to each square in the white list).

• Exercise: figure out how to do this.
7 Exercises

1. Evaluate \texttt{bishops\_cps 7 17} and draw the solution on a grid. Verify that the result yields a safe placement of 17 bishops on the 7-by-7 board, and that this placement cannot be extended with any other squares safely.

2. See what happens in the direct-style implementation if we change the search function to use a different strategy, e.g.

   \begin{verbatim}
   fun search p [ ] = NONE
   | search p ((bs, rest)::xs) = 
     if (p bs) then SOME bs else search p (xs @ steps (bs, rest));
   \end{verbatim}

   This is no longer “depth-first search”! You should see a difference in results (and in speed!).

3. Make a small number of changes to the basic set-up developed above, that will allow you to solve the n-queens problem: for \( n > 0 \), find a way to put \( n \) queens on the \( n \)-by-\( n \) chessboard so that no queen is attacked by any other. Queens attack along horizontal or vertical lines and along diagonals.

4. Implement the sequential algorithm from the previous section.

5. Implement the all-solutions algorithm for the bishops problem.

6. Revamp your implementations so that they always produce a list of bishop positions that’s sorted with respect to the usual lexicographic order on pairs of integers.

7. Write a function that turns a position list into a string representation of the chessboard, so you can see a 2-dimensional snapshot of your solutions.

8. Looking at the results produced by \texttt{bishops\_cps n} for small values of \( n \), there seems to be a common pattern: there is always an entire column of bishops in the first row, at positions \((1,n),(1,n-1),\ldots,(1,1)\). (These always show up at the end of the list.) Write a function that explicitly looks for a solution with this property.