15-150 Fall 2020

Lecture 13
Direct- and continuation-style programming

1 Topics

- Direct- and continuation-style versions of functions.
- Implementing a backtracking algorithm with continuations

2 Aims

We showed in lecture last week how the use of polymorphism and "success" and "failure" continuations arose naturally when we worked to generalize a solution to a specific problem, aiming for greater flexibility. Our change-making function, with a polymorphic type and two function arguments that represent "continuations" to be used in case of success or failure, is a good example of continuation-style programming. By using a type variable to stand for an "answer" type to be chosen later, serving as the result type for both success continuation and failure continuation, we obtained code that is widely applicable: just pick an actual type of answers and suitable success and failure continuations.

Continuation-style program design may seem harder to grasp at first sight than "direct" style. In direct style we usually deal with functions that "return" a value; in continuation style instead we write functions that pass a value to a continuation. These notes aim to help appreciate the elegance with which we can solve problems in continuation style.

3 What is a continuation?

A function, used as a parameter by another function, and typically used to abstract away from "the rest of the computation", or "what to do to finish a task".

Depending on the way we design our code, a continuation may be called a "success continuation" (to be used when we find a solution to a problem, to turn the solution into the form of an "answer"), or a "failure continuation" (to be used when we find that there is no solution along some exploration path, and we need to "backtrack" to explore alternatives). But often there's no need to classify (success vs. failure), or we might only need one of these two kinds of continuation. The examples to be discussed in these notes are chosen to help clarify these concepts.

4 Direct- and continuation-style functions

A function of type `t1 -> t2` expects to be applied to an argument value of type `t1`, and returns a value of type `t2` (if application of the function to this argument value terminates). We refer to this as a *direct-style* function. Most of the functions introduced so far this semester in class and lab have been direct-style, and we never felt the need to say so explicitly. (When there's not yet an alternative it seems unnecessarily verbose to conjoin an adjective to discriminate!) We can summarize in a slogan: a direct-style function evaluates its argument and returns a value.

Every direct-style function has a continuation-passing *alter ego*, a function written in continuation style whose behavior mimics the direct-style function. Again let's suppose we have a direct-style function `f : t1 -> t2`. A continuation-passing version of `f` is a function that expects to be applied to a value of type `t1` and a continuation of type `t2 -> ans`, where `ans` is a type of "answers" or "final results". Instead of "returning" a value of type `t2`, the cps-function passes a value of type `t2` to the continuation, and thus produces an "answer". Slogan: a continuation-style function evaluates its argument and calls its continuation with a value.

This idea is very general. Every "direct-style" function can be converted into a "continuation-style" version. Writing a function in continuation-style may help to emphasize the control flow, and indeed continuation-style can be a good way to implement a complex pattern of control flow.

To illustrate, let's revisit the factorial function.

Factorial, revisited

Here is the direct-style factorial function `fact : int -> int`.

```
fun fact n = if n=0 then 1 else n * fact(n-1)
```

As before, we know that this function satisfies the specification given by:

> For all `n` ≥ 0, `fact(n)` returns the value of `n!`.

The cps-version of the factorial function,

```
FACT : int -> (int -> 'a) -> 'a
```

is defined as follows:

```
fun FACT n k = if n=0 then (k 1) else FACT(n-1) (fn x => k (n * x))
```

Observe that:

- `fact 0` returns 1, and `FACT 0 k` passes `1` to `k`
- For `n` $\neq 0$, `fact n` makes a recursive call to `fact(n-1)` and, if this returns a value `v`, returns the product of `n` and `v`
- For `n` $\neq 0$, `FACT n k` calls `FACT (n-1)` *with a continuation that*, if passed a value `v`, multiplies it by `n` and passes the product to `k`.

The behavior of the cps-style function `FACT` is given by:

> **Theorem**
> For all `n` ≥ 0, all types `t`, and all continuations `k:int -> t`,
> `FACT n k = k (n!)`.

We can prove this by induction on the value of `n`. Since this is the first time we've seen an example like this, involving a continuation-style function, here is the proof.

- Base case: For `n = 0`. We need to show that for all types `t` and all functions `k:int -> t`, `FACT 0 k = k (0!)`. By definition of `FACT`,

```
FACT 0 k =  if 0=0 then k(1) else FACT(0-1)(fn x => k(0*x))
         =  k(1).
```

  Since `0!=1`, we therefore have `FACT 0 k = k (0!)`, as required.

- Inductive step: Let `n>0`, and suppose as the Induction Hypothesis that

  ```
  (IH): For all types t and all k':int -> t,
        FACT (n-1) k' = k' ((n-1)!).
  ```

  We must show that, for all types `t` and all functions `k:int -> t`, `FACT n k = k (n!)`.

  Let `t` be a type and `k:int -> t` be a function. Then, since `n>0`,

  ```
  FACT n k = FACT (n-1) (fn x => k(n*x))   by def of FACT
  ```

  So let `k'` be `(fn x => k(n*x))`. Then we have

  ```
  FACT n k
    = FACT (n-1) k'               by def of FACT
    = k' ((n-1)!)                 by IH
    = (fn x => k(n*x)) ((n-1)!)   by def of k'
    = k (n*(n-1)!)                by value-substitution
    = k (n!)                      by def of n!
  ```

- That completes the inductive proof.

Comments

(a) The type of `FACT` indicates that we can choose any type of answers that we like. For example, the SML function

```
Int.toString : int -> string
```

converts an integer value into a string. If we want to get a string as an answer, we can use this function as a continuation for `FACT`:

```
FACT 3 Int.toString = Int.toString (6) = "6"
```

(b) Since we know that the direct-style `fact` function satisfies its own spec, we can conclude from the above proof for `FACT` that

> For all $n \geq 0$, and all types `t`, and all functions `k:int -> t`,
> `FACT n k = k (fact n).`

Question

- What happens when `n` < 0? We don't have a definition for `n!` when `n` is negative. Use stepping and evaluational reasoning, show that for all types `t` and all functions `k:int -> t`, `FACT n k` fails to terminate (has an infinite evaluation sequence). Remember that for negative values of `n`, `fact n` also fails to terminate. Is it true that for *all* integers `n`,

  ```
  FACT n k = k (fact n)?
  ```

Continuation-style counting

Here is a direct-style function

```
count : int tree -> int
```

for adding the integers in an integer tree.

```
fun count Empty = 0
 |  count (Node(left, x, right)) =
     (count left) + x + (count right)
```

Notice the control flow in the non-empty-tree case: first make a recursive call on the left subtree, then make a recursive call on the right subtree, then do the arithmetic.

Here is the continuation-passing style version,

```
count' : int tree -> (int -> 'a) -> 'a
```

in which this control flow is made explicit, and also the rôles played by the results of these recursive calls is made explicit:

```
fun count' Empty k = k 0
 |  count' (Node(left, x, right)) =
    count' left (fn m => count' right (fn n => k(m+x+n)))
```

In the non-empty case the continuation on the right-hand side contains an embedded call to `count'` on the right subtree, with a continuation that does the arithmetic before passing the total to the original continuation.

Exercise:
Prove by induction on tree structure that, for all values `T:int tree`, all types `a` and all functions `k:int -> a`,

```
count' T k = k(count T).
```

You can use without proof the fact that for all values `T:int tree`, `count T` returns a value.

Continuation-style, tail recursion, and efficiency

A recursive function definition is said to be *tail recursive* if each recursive call made by the function is in "tail position", which means that it is the last thing the function does before returning.

For example, the factorial function

```
(* fact : int -> int *)
fun fact n = if n=0 then 1 else n * fact(n-1)
```

is not tail recursive, because the recursive call `fact(n-1)` is not in tail position: its result gets multiplied by the value of `n`.

Similarly, the Fibonacci function

```
(* fib : int -> int *)
fun fib 0 = 1
 |  fib 1 = 1
 |  fib n = fib(n-1) + fib(n-2)
```

is not tail recursive, because the call to `fib(n-2)` is not in tail position.

On the other hand, the continuation-style functions

```
(* FACT : int -> (int -> 'a) -> 'a *)
fun FACT n k = if n=0 then k(1) else FACT(n-1) (fn x => k(n * x))

(* FIB : int -> (int -> 'a) -> 'a *)
fun FIB 0 k = k 1
 |  FIB 1 k = k 1
 |  FIB n k = FIB (n-1) (fn x => FIB (n-2) (fn y => k(x+y))
```

are tail recursive.

Tail calls are significant because they can be implemented efficiently in a way that typically saves *space*. Here is an short (and simplified) overview of the main ideas. When a function is called (applied to an argument value), the argument value gets "pushed" onto a *stack*, along with a "return address" that indicates the place it was called from, the place where the result of the call needs to be "returned" to. For tail calls, there is no need to remember the place we are calling from. Instead, one can perform *tail call elimination* by re-using the same stack frame for the new argument value but leaving the return address unchanged; the result of the tail call needs to be "returned" to the original caller.

Here is an illustration of these ideas, based on evaluating a call of the following (tail recursive) function:

```
fun factacc(n:int, a:int):int =
   if n=0 then a else factacc(n-1, n*a)
```

Execution of the call `factacc(3, 1)` (without doing any tail call elimination) looks like:

```
call factacc (3, 1)
  call factacc (2, 3)
   call factacc (1, 6)
    call factacc (0, 6)
    return 6
   return 6
  return 6
return 6
```

In the above display, indentation indicates the growth and shrinking of the stack: the call to `factacc (3,1)` involves pushing the argument pair `(2,3)` onto the stack along with a return address, and so on. There are several return addresses, one for each recursive call, but we end with a series of trivial returns that essentially just pass the value 6 along each time until we reach the return address for the original call. Executing `factacc(n, a)` for some $\texttt{n} > 0$ uses $\texttt{O}(\texttt{n})$ stack space.

If the implementation does tail call elimination, we would end up with an execution that looks like:

```
call factacc (3, 1)
   replace arguments with (2, 3) [same return address]
   replace arguments with (1, 6) [same return address]
   replace arguments with (0, 6) [same return address]
   return 6
```

This reorganization saves space because only the calling function's address needs to be saved, and the stack frame for `factacc` is reused for the recursive calls. Doing things this way takes only $\texttt{O}(1)$ space for `factacc(n, a)`.

In many functional language implementations tail call elimination is done *automatically*, and the programmer need not worry about running out of stack space for extremely deep recursions. Also, the tail recursive variant of a function may be faster than a non-tail recursive variant, but typically only by a constant factor (which would imply the same `O`-class).

Some programmers working in functional languages will rewrite recursive code to be tail-recursive so they can take advantage of this feature. This often requires addition of an "accumulator" argument (`a` in `factacc`).

Fibonacci, revisited

Now the direct- and continuation-style versions of the Fibonacci function. This example is a little more complex, because of the pattern of recursion.

The direct-style Fibonacci function `fib` expects to be applied to an integer and its result type is `int`.

```
(* fib : int -> int *)
fun fib 0 = 1
 |  fib 1 = 1
 |  fib n = fib(n-1) + fib(n-2)
```

It will be helpful to describe the control flow:

> For `n` > 1, `fib(n)` calls `fib(n-1)` (to get a result, say `x`), then calls `fib(n-2)` (to get a result, say `y`), then returns the sum of these results, i.e., the value of `x + y`.

It's easy to prove by induction on `n` that for all `n` ≥ 0, `fib(n)` returns the `n`th Fibonacci number.

The continuation-style Fibonacci function expects to be applied to an integer and an integer-consuming continuation (a function of type `int ->t`, where `t` is the type `t` of what we deem to be "answers").

```
(* FIB : int -> (int -> 'a) -> 'a *)
fun FIB 0 k = k 1
 |  FIB 1 k = k 1
 |  FIB n k = FIB (n-1) (fn x => FIB (n-2) (fn y => k(x+y))
```

The cps version mimics the control flow of the direct-style function:

> For `n` > 1, `FIB n k` calls `FIB (n-1) k'`, where `k'` is a continuation that, when given an integer `x`, calls `FIB (n-2)` with a continuation that, when given an integer `y` passes the sum of `x` and `y` to `k`.

You can "read off" this explanation by looking at the code structure!

We can prove the following specification.

> For all `n` ≥ 0, all types `t`, and all `k:int -> t`,
> `FIB n k` $=$ `k(fib n)`.

Again we can prove this by induction on `n`. The details: an exercise.

5 Backtracking with continuations

We will develop solutions to a combinatorial problem with a large search space. Our techniques are applicable to a 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 avoid the need for elaborate set-up [1].

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 `map` and `foldl` or `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.

Incidentally, we won't just be converting a direct-style function to its cps-style version as in the previous section; the cps-version would actually have the same control flow as the direct-style function and wouldn't be any more efficient. Our continuation-style solver emerges instead after a radical redesign of strategy, and is noticeably more efficient in practice, even though we don't go into work/span analysis in detail.

The main themes are (again): designing and using higher-order functions; maps, folds, and recursion; specifications, proofs and efficiency. We won't include correctness proofs, although in class we did discuss why the code is correct as we introduced it. 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 `pos`) and we may pretend that ML always knows to use abbreviated type names. (It doesn't.) Instead ML often reports the full type without abbreviation. Moreover we may sometimes give a specification for a function in which we specify a type that's not the most general one(!). We may do this when we only need to use a specific instance of the most general type.

[1] You may be familiar with the famous (in some circles, infamous) n-queens problem. We'll tackle a different problem about a different kind of chess piece, but afterwards you should be able to adapt our code development strategy to obtain a nifty functional programming solution for n-queens as well.

6 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. We'll be concerned with the following problem:

> 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

A bishops at `(x, y)` attacks or threatens a bishop at `(i, j)` if and only if the positions are on the same diagonal line. There's an easy way to check this: it happens when the absolute value of `x-i` is equal to the absolute value of `y-j`. We're going to need to add up the number or threats to a given position, so it makes sense to define a function that returns an integer: 0 for no threat, 1 for a threat.

```
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 a bishop at b would threaten a bishop at c
           threat b c = 0 otherwise
*)
```

According to this `threat` function, a bishop at `(x, y)` threatens himself, because `threat (x,y) (x,y) = 1`. This is not a bug, but rather a feature that we need to be aware of...

Next, a function that counts the threats from a list of positions.

```
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
*)
```

Again, note that if `b` is in `L` we include the self-attack in the count.

We will use the following helper function `forall` to check for safety. Its most general type is polymorphic, but we don't need the full power of polymorphism here. (Compare this function with the `exists` function used in an earlier lecture.)

```
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
 *)
```

And here's the safety checker function: it checks for at most 2 threats, because of the over-counting of self-threats, noted above. At most 2 really means that each bishop is attacked by at most 1 *other* bishop. It's always good to keep track of "features" so we know how to get it right!

```
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
*)
```

Some examples to show that we got it right:

```
- 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 again that you understand why the safety check looks for at most 2 threats, rather than at most one[2]. Draw pictures of the 3-by-3 grid to illustrate the two safety checks shown above. Explore some more.

Exercise
Read about the ML type `string`. Write a function

```
display: int -> pos list -> string
```

so that for `n` ≥ 0 and a list `L` of positions `(i,j)` with $1 \leq$ `i,j` $\leq$ `n`, `display n L` returns a string of the form `s1^s2^...^sn`, where each `si` is a string of 0's and 1's ending with `"\n"`. The `j`th character of `si` should be 1 if and only if `(i,j)` is in `L`. For example

```
display 3 [(1,1),(1,2),(1,3),(2,1)] = "111\n100\n000\n"
```

This is a good exercise in using `fold` functions, and the display function will be useful in checking your solutions, later. Just say `print (display n L)`.

Now we have these basic ingredients in place, so we can talk accurately about safety of position lists, we will discuss some strategies for solving the problem. Although we are (of course) doing *functional* programming, it is common to talk about "states" when discussing search algorithms like this one; the term "state" is also (as we know) a major feature in *imperative* programming. Don't get confused; just remember that in math, CS and other areas there's a great deal of terminological overloading! If you prefer, replace the word "state" by something more neutral, with fewer connotations of imperativeness (is that a word?), like "configuration".

[2]You might like to consider an alternative: define `threat` so that it behaves as above except that `threat (x, y) (x, y) = 0`. How would the definitions of the later functions need to change to reflect this?

7 A direct-style implementation

Our first implementation will work with what we'll call *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 `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 from a list. We introduce the predicate `is_valid_state : state -> 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.

```
type sol = pos list
type state = sol * pos list
fn is_valid_state (bs, rest) = safe bs

fun del b [ ]    = [ ]
|   del b (c::cs) = if b=c then cs else c::del b cs
(*
   del : pos -> pos list - pos list
   REQUIRES  bs is a list with no repeated elements
   ENSURES del b bs = result of deleting b from bs
*)


fun steps (bs, rest) =
let
   val R = List.filter (fn b => safe(b::bs)) rest
in
   map (fn b => (b::bs, del b rest)) R
end
(*
   steps : state -> state list
   REQUIRES is_valid_state(bs, rest)
   ENSURES steps (bs, rest) =
         a list of all valid states obtainable by
         extending bs safely with one b drawn from rest
*)
```

`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),(1,1),(1,2),(1,3)],[(2,2),(2,3)]),
   ([(2,3),(1,1),(1,2),(1,3)],[(2,1),(2,2)])]
```

There are only two possible extra squares, `(2,1)` and `(2,3)`, that are safe to extend with. So we get a result list of two states.
Similarly,

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

There's no safe extension using `(2,2)`, so we get the empty list.

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 will use `search` as the engine that drives our bishops solver.
We also introduce a convenient helper function 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 *assuming* the search function satisfies its specification it will follow easily that the `bishops_direct` function works as specified.

It might be helpful to think of the "state space" being explored by `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 `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 results 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 big-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`.

```
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;
Trying 1
Trying 2
Trying 3
Trying 4
Trying 5
Trying 6
Trying 7
Trying 8
Trying 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 `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 `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 llist 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...

8 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 fairly intuitive 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:

```
- 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.

9 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.

```
- 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:

```
(* 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
*)
```

10 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.

```
(* split : (pos -> bool) -> pos list -> pos list * pos list
   REQUIRES p total
   ENSURES split p L = (A, B),
             where A is list of items in L satisfying p,
               and B is list of items in L that don't satisfy p
*)

fun split p [ ] = ([ ], [ ])
|   split p (x::L) = let
                       val (A,B) = split p L
                     in
                       if (p x) then (x::A,B) else (A, x::B)
                     end
```

```
(* fast_bishops_cps : int -> int -> (pos list) option
   REQUIRES n even, m even
   ENSURES
     fast_bishops_cps n m
     = SOME bs,
          where bs is a safe placement for m bishops on n-by-n board,
           if there is one
     = NONE, otherwise
*)

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 evaliuated 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 is 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 achieve efficiency while avoiding 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 suitably chosen constant-time function to each square in the white list).
- Exercise: figure out how to do this. And explain why it's just as efficient to do this, sequentially, as the parallel code discussed above.

11 Exercises

1. Evaluate `bishops_cps 7 17` and draw the solution. Verify that you get a safe placement of 17 bishops on the 7-by-7 board, and that this placement cannot be extended without losing safely.

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

   ```
   fun search p [ ] = NONE
   |   search p ((bs, rest)::xs) =
       if (p bs) then SOME bs else search p (xs @ steps (bs, rest))
   ```

   Not "depth-first search"! You will see different results (and speed!).

3. Make a small number of changes to the basic set-up above, 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. Define a function

   ```
   all_bishops : int -> int -> (pos list -> bool) -> (pos list) list
   ```

   so that for `n`, `m` > 0 and all total predicates `p : pos list -> bool`, `all_bishops n m p` returns a list of all safe ways to place `m` bishops on an `n`-by-`n` board that also satisfy `p`. HINT: find a way to use the continuation-style function that finds a single solution, but be clever about using recursion.

5. 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.

6. Write a function that turns a position list into a string representation of the chessboard, so you can see a 2-dimensional grid of a solution.

7. Looking at the results produced by `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),...,(1,1)`. (These always show up at the end of the list.) Write a function that explicitly looks for a solution with this property.