

Supervision 4

Michaelmas Term 2026 · Michael Lee

●●●● difficulty ★ optional

WORK DUE	A single PDF, 24 hours before the supervision
EXPECTED TIME	5 hours 45 min (+ 30 min for the optional questions)

1 Sequences and laziness

PREREQUISITES Lecture 9 · TIME 2 hours 10 min

The right way to think about lazy data is by the equations it satisfies, not how it is constructed.

QUESTION 1

CONCEPTUAL · 15 MIN

●●●● BASIC

Let `zip_with_s` be a function which applies a binary operator to corresponding elements of two sequences, so that `zip_with_s (+)` applied to `1, 2, 3, ...` and `5, 6, 7, ...` gives `6, 8, 10, ...`. Here is a definition of `zip_with_s`, which you do not need to understand to answer the question:

```

1  let rec zip_with_s opr xs ys = match xs, ys with
2    | Nil, _ -> Nil
3    | _, Nil -> Nil
4    | (Cons (x, xf)), (Cons (y, yf)) ->
5      Cons (opr x y, fun () -> zip_with_s opr (xf()) (yf()))

```

OCaml

Here are two definitions of the Fibonacci sequence.

```

1  let rec fibs1 = Cons (0, fun () ->
2    Cons (1, fun () -> zip_with_s (+) fibs1 (tail fibs1)))
3
4  let rec f a b = Cons (a, fun () -> f b (a+b))
5  let fibs2 = f 0 1

```

OCaml

Consider the sequences

$$\begin{aligned}
 \mathcal{F} &= 0, 1, 1, 2, 3, 5, 8, \dots \\
 \mathcal{F}' &= 1, 1, 2, 3, 5, 8, \dots \\
 \mathcal{F}'' &= 1, 2, 3, 5, 8, \dots
 \end{aligned}$$

(a) Define $\mathcal{F}$, $\mathcal{F}'$, and $\mathcal{F}''$ as instances of some parameterised sequence $F(x, y)$

- (b) Establish two equations, one corresponding to `fibs1`, and the other to `fibs2`.

QUESTION 2

PRACTICAL · 25 MIN

● ● ● ● ● CORE

Write the lazy list whose elements are all ordinary lists of zeroes and ones, namely `[]`, `[0]`, `[1]`, `[0;0]`, `[0;1]`, `[1;0]`, `[1;1]`, `[0;0;0]`, ...

QUESTION 3

PRACTICAL · 25 MIN

● ● ● ● ● CORE

(Continuing question 2.) A palindrome is a list that equals its own reverse. Code the lazy list whose elements are all palindromes of 0s and 1s, namely `[]`, `[0]`, `[1]`, `[0;0]`, `[0;0;0]`, `[0;1;0]`, `[1;1]`, `[1;0;1]`, `[1;1;1]`, `[0;0;0;0]`, ... You can use the list reversal function `List.rev`.

QUESTION 4

CONCEPTUAL · 10 MIN

● ● ● ● ● CORE

- What is the time complexity of appending two lazy lists?
- What is the time complexity of getting the tail of a lazy list?
- When might it be desirable to use a lazy list over an eager list?

QUESTION 5

PRACTICAL · 25 MIN

● ● ● ● ● CORE

Memoisation involves storing the results of computation, so as to avoid recomputation. For example, whenever I try and compute

```
1  f x
```

OCaml

I first look up if I have already performed the computation, and if I have, I just look up the result. Otherwise, I compute the result, and then store it in a table for future lookup.

- Write a program that tests if memoisation is being performed.
- Define `reverseq`, which does not reverse a sequence but instead returns a suspended computation:

```
1  let reverseq xq = fun () ->
2    let rec reverse xq acc = match xq with
3      | Nil          -> acc
4      | Cons(x, xf) -> reverse (xf ()) (Cons(x, fun () -> acc))
5    in reverse xq Nil
```

OCaml

Define also a variant of `appendq` whose second argument is a suspended computation, rather than a lazy list:

```

1  let rec appendq xq yf = match xq with
2    | Nil          -> yf ()
3    | Cons(x, xf) -> Cons(x, fun () -> appendq (xf ()) yf)

```

OCaml

With reference to the following example, explain why memoisation is important for lazy lists.

```

1  let rec until n =
2    if n = 0 then Nil
3    else Cons(n, fun () -> until (n-1))
4  let xq = Cons(1, fun () -> Nil)
5  let yq = until 1000
6  let yqr = reverseq yq
7  let zq = appendq xq yqr
8  let z1 = tail zq
9  let z2 = tail zq

```

OCaml

QUESTION 6

CONCEPTUAL · 30 MIN

● ● ● ● ● CORE

For simplicity, consider the pure, terminating subset of OCaml that only has expressions and values (the question generalises to OCaml programs). Recall that OCaml expressions evaluate to values. Let A and B be OCaml types. Let $t_1 : A \rightarrow B$ and $t_2 : A$ be two OCaml programs. Let t_1 evaluate to v_1 , written $t_1 \rightsquigarrow v_1$ and $t_2 \rightsquigarrow v_2$. Further, since v_1 has a function type, it must have shape $\text{fun } x \rightarrow t$. Function application results in substitution, written $t[t'/x]$ read “ t with t' substituted for x ”. Concretely, if $t = x + 1$, and $t' = 2$, then

$$t[t'/x] = (x + 1)[2/x] = 2 + 1$$

Consider evaluating the application $t_1 t_2$. Consider the following two evaluation strategies

1. Completely evaluate the function, completely evaluate its argument, and then perform substitution

$$\begin{aligned}
 t_1 t_2 &\rightsquigarrow (\text{fun } x \rightarrow t) t_2 && \text{(reduce the function)} \\
 &\rightsquigarrow (\text{fun } x \rightarrow t) v_2 && \text{(reduce the argument)} \\
 &\rightsquigarrow t[v_2/x] && \text{(perform substitution)}
 \end{aligned}$$
2. Completely evaluate the function, and then perform substitution without evaluating the argument

$$\begin{aligned}
 t_1 t_2 &\rightsquigarrow (\text{fun } x \rightarrow t) t_2 && \text{(reduce the function)} \\
 &\rightsquigarrow t[t_2/x] && \text{(perform substitution)}
 \end{aligned}$$

Note that the choice of strategy is a **language** choice, not a user choice. Different languages make different choices.

- (a) Which of the evaluation strategies is call-by-value, and which is call-by-name?

- (b) Write a program that evaluates differently under the two evaluation strategies (even if they eventually agree on the answer, they take different steps to get there), and show how they evaluate in each case.
- (c) Widen the subset of OCaml to include a non-terminating program ω . We can write ω in OCaml as such

```
1  let rec omega () = omega () in omega ()
```

OCaml

Write a program where call-by-value and call-by-name will exhibit different behaviour.

- (d) Invent an approach to simulate the call-by-name evaluation strategy in a language that has chosen the call-by-value evaluation strategy.
- (e) Compare your approach to the transition from lists to lazy lists. What can you say about OCaml's evaluation strategy, and what can you say about the design of lazy lists?

2 Queues and search strategies

PREREQUISITES [Lecture 10](#) · TIME 1 hour 45 min

A careful look at building persistent amortised data structures, and a quick look at search strategies.

QUESTION 7

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

Suppose that we have an implementation of queues, based on binary trees, such that all operations have worst-case time complexity $O(\log n)$.

Outline the advantages and drawbacks of such an implementation compared with the efficient functional queues presented in [Section 10.5](#).

QUESTION 8

CONCEPTUAL · 35 MIN

● ● ● ● ● ADVANCED

This question is inspired by the introductory sections of Anton Lorenzen's *First Order Laziness*, which in turn is inspired by Chris Okasaki's PhD thesis, *Purely Functional Data Structures*.

The version of a queue from the notes performs normalisation whenever `hds` is empty. Another alternative is to perform normalisation whenever `hds` is shorter than `tls`.

- (a) Implement this variant queue
- (b) Assume we have a queue `q`, where the length of `hds` is the same as the length of `tls`, and call this length `n`. Analyse the time complexity of the following program.

```
1  for i = 1 to n do
2    ignore (deq q)
3  done
```

OCaml

MOTIVATION: CONCURRENT SYSTEMS

Note that this example, while contrived, reflects a very real use case: in concurrent and distributed systems, one often faces cases with multiple readers of the same data structure. This program simulates multiple reader threads of the same queue.

(c) In the earlier example, each `deq` potentially involves a normalisation:

```
1 append hds (reverse tls)
```

OCaml

Now consider using lazy lists for `hds` and `tls` (recall the definitions of `reverseq` and `appendq` from question 5)

```
1 appendq hds (reverseq tls)
```

OCaml

might be more efficient in this case. What about without memoisation?

QUESTION 9

CONCEPTUAL · 5 MIN

● ● ● ● ● BASIC

The traditional way to implement queues uses a fixed-length array. Two indices into the array indicate the start and end of the queue, which wraps around from the end of the array to the start. How appropriate is such a data structure for implementing breadth-first search?

QUESTION 10

CONCEPTUAL · 5 MIN

● ● ● ● ● BASIC

Why is iterative deepening inappropriate if $b \approx 1$, where b is the branching factor? What search strategy would make more sense in this case?

QUESTION 11

CONCEPTUAL · 10 MIN

● ● ● ● ● BASIC

An interesting variation on the data structures seen in the lectures is a *deque*, or *double-ended queue*. A deque supports efficient addition and removal of elements on both ends (either its front or back). Suggest a suitable implementation of deques, justifying your decision.

QUESTION 12

PRACTICAL · 30 MIN

● ● ● ● ● CORE

Implement deques as an abstract data type in OCaml (including a type declaration and suitable operations). Estimate the amortised complexity of your solution.

3 Elements of procedural programming

OCaml has references, assignment and while loops. These are important tools in a programmer's arsenal, but they come with high costs: this section is about understanding how to use them, the price one pays for using them, and how to use them well.

QUESTION 13

CONCEPTUAL · 20 MIN

● ● ● ● ● CORE

- (a) Assume e and e' are arbitrary OCaml expressions of type `int`. Is it true that $e + e'$ is always the same as $e' + e$? Either explain why it is true, or write down e and e' that will exhibit different behaviour.
- (b) What if e and e' are pure expressions?

QUESTION 14

CONCEPTUAL · 15 MIN

● ● ● ● ● BASIC

Comment, with examples, on the differences between an `int ref list` and an `int list ref`. How would you convert between the two?

QUESTION 15

PRACTICAL · 10 MIN

● ● ● ● ● BASIC

Write a version of function `power` (Section 1.6) using `while` instead of recursion.

QUESTION 16

PRACTICAL · 10 MIN (+ 20 MIN)

● ● ● ● ● CORE

This question walks you through the creation of a memoised power function, using OCaml's `Hashtbl` module. The important functions are `Hashtbl.create`, `Hashtbl.find_opt` and `Hashtbl.replace`.

- (a) Write `pow : int -> int -> int`, which consults a table before computing and stores what it computes.
- (b) Identify two bad places to create the table, and explain why your choice is better than either of them.
- (c) Is `pow 2 100` any faster if you have already computed `pow 2 99`?
- (d) Is `pow` pure?
- ★ (e) Assume you have access to a constant-time prime factorisation algorithm,

```
val factorise: int -> (int * int) list
```

For example, `factorise 100 = [(2, 2); (5, 2)]`. You can also assume a gcd algorithm

```
val gcd: int list -> int * int list
```

For example, $\text{gcd } [2;4] = (2, [1;2])$, and $\text{gcd } [] = (0, [])$. Can you think of a way to make your solution even faster?

HINT If $\text{pow } 2 \ 100$ is in the table, how much time should it take to compute $\text{pow } 4 \ 50$?

QUESTION 17

PRACTICAL · 5 MIN (+ 10 MIN)

● ● ● ● ● BASIC

(a) Write a function to exchange the values of two `t refs`, `xr` and `yr`.

★ (b) If `t = int`, and you may assume `xr != yr`, can you improve your solution?

QUESTION 18

PRACTICAL · 25 MIN

● ● ● ● ● CORE

Arrays of multiple dimensions are represented in OCaml by arrays of arrays. Write functions to (a) create an $n \times n$ identity matrix, given n , and (b) to transpose an $m \times n$ matrix. Identity matrices have the following form:

$$\begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

QUESTION 19

CONCEPTUAL · 25 MIN

● ● ● ● ● ADVANCED

OCaml enforces abstraction barriers through its powerful module system. Specifically, its module system offers *static sealing*. Consider the following module for the natural numbers

```

1  module type NAT = sig
2    type t
3    val zero : t
4    val succ : t -> t
5  end
6
7  module Nat: NAT = struct
8    type t = int
9    let zero = 0
10   let succ n = n+1
11  end

```

OCaml

The module type ascription `Nat : NAT` seals the module, preventing code from outside the module from peeking into it. This creates a form of unforgeability: the only way to get an element of type `t` is to ask the module for one. We cannot try to forge a value of type `t`, the following example gives a type error:

OCaml

1 Nat.succ 0

Static sealing is a feature offered by the language: attempts to break it are caught by the compiler. If the language does not implement static sealing, it is sometimes possible to build *dynamic* sealing, which offers similar guarantees, but only when the program runs (not when it is compiled).

The idea behind dynamic sealing is simple: there is a type of keys, which cannot be forged, and a type of sealed values. The user can mint new keys. A key can be used to seal values, creating a sealed value. A key can also be used in an attempt to unseal a sealed value. If the key was used to seal the value, then unsealing succeeds, returning `Some v`. Otherwise, it fails, returning `None`.

As described by Filinski in Remark 4.30 of his PhD thesis *Controlling Effects*, and translated into OCaml by Dr. Jeremy Yallop, we can implement dynamic sealing in OCaml.

OCaml

```

1  module type SEAL =
2  sig
3    type sealed
4    type 'a key
5    val make_key : unit -> 'a key
6    val seal : 'a -> 'a key -> sealed
7    val unseal : 'a key -> sealed -> 'a option
8  end
9  module Seal: SEAL = struct
10   type sealed = unit -> unit
11   type 'a key = 'a option ref
12   let make_key () = ref None
13   let seal v r = fun () -> r := Some v
14   let unseal r f = r := None; f (); !r
15 end

```

- (a) How have we implemented keys?
- (b) Why are keys unforgeable?
- (c) Explain how `unseal k (seal v k)` succeeds
- (d) Explain how `unseal k (seal v k')` fails when `k` and `k'` are different