1. Impure Programming

1. In pure functional OCaml, the only way to have a function with an unconstrained return type is to loop forever, i.e.

```
# let rec f x = f x;;
```

But non-pure OCaml adds another way to do this, what is it?

[small]

The most immediately obvious one is raising an exception, but closing the program also works. And there are shorthands for some exceptions

```
# val raise : exn -> 'a = <fun>
# val exit : int -> 'a = <fun>
# val failwith : string -> 'a = <fun>
# val invalid_arg : string -> 'a = <fun>
```

There is also the type-system breaking function

```
# Obj.magic : 'a -> 'b
```

This re-interprets the bits of the argument of type `'a` as type `'b`. It doesn't convert it at all, it just pretends the underlying bits of `'a` are actually `'b`, so using this can completely break type safety and cause crashes. Don't use unless you really know what you are doing. And because of the garbage collector, you cannot rely on addresses staying constant, so you shouldn't use it for address comparisons. See the cycle detection below for more details.

2. Doubly-linked lists are lists which have pointers to the previous and next elements, give a type for doubly-linked lists, and the insertion, fold-left and fold-right functions on them. Make both fold-left and fold-right tail-recursive.

   Note, you will need to look at == operator in OCaml.

[medium]

The code becomes a bit simpler if we first define getters/setters, as we won't have to do so much pattern matching.

```OCaml
type 'a dll = Empty | Node of 'a * 'a dll ref * 'a dll ref

let get_next = function (Node(_, next, _)) -> !next
  | _ -> failwith "get_next";;
let get_prev = function (Node(_, _, prev)) -> !prev
  | _ -> failwith "get_prev";;
let set_next dll n = match dll with
  | Node(_, next, _) -> next := n
  | _ -> failwith "set_next";;
let set_prev dll n = match dll with
  | Node(_, _, prev) -> prev := n
  | _ -> failwith "set_prev";;
```

Adding an element to a list involves breaking the loop between the first and the last element – note this loop must exist otherwise we cannot do tail-recursive fold-right. The highlights indicate the tricky bit of this function: we need to create a new ref cell inside the new node `nn`. Otherwise we would have `get_prev n` and `get_prev nn` aliased, and when we do `set_prev n nn` we will change the previous pointer on both `n` but also `nn` as they are the same reference cell.

```OCaml
let dllcons el = function
  | Empty ->
    let rec n = Node(el, ref n, ref n) in n
  | Node(_, next, prev) as n ->
    let nn = Node(el, ref n, ref !prev)
    in set_next (!prev) nn; set_prev n nn; nn;;
let rec rotate i = function Empty -> Empty
  | Node _ as n when i=0 -> n
  | Node(_, _, prev) when i<0 -> rotate (i+1) !prev
  | Node(_, next, _) -> rotate (i-1) !next;;
let insert el i dll = match i with
  | 0 -> dllcons el dll
  | _ ->
    ignore @@ dllcons el @@ rotate i dll;
    dll;;
```

Note, that because here the type of `dll` is not a reference, only its next/previous pointers are, it is not possible to insert at the front, and update the original list so that it is in the front there. Instead, if you have a doubly-linked list `l` and you run `dllcons a l` then it will be inserted at the end from the point of view of `l`.

The trick for folding is stopping at the correct time. We need to walk at least one element (if there is one), so in the one element case we cannot just check `n==l` but first consider the current element – this is what the `match with` is for.

```OCaml
let dfoldl f acc l =
  let rec inner acc = function
    | Empty -> acc
    | Node(el, next, prev) as n ->
      if n==l
      then acc
      else (inner[@tailcall]) (f acc el) !next
  in match l with Empty -> acc | Node(el, next, prev) -> inner (f acc el) !next;;
let dfoldr f l acc =
  let rec inner acc = function
    | Empty -> acc
    | Node(el, next, prev) as n ->
      if n==l
      then f el acc
```

```
28        else (inner[@tailcall]) (f el acc) !prev
29    in match l with Empty -> acc | Node(el, next, prev) -> inner acc !prev;;
```

To test, let's try making a doubly-linked list of ten elements and converting to plain lists.

```OCaml
let list_of_dll l = dfoldr (fun el acc -> el::acc) l [];;

let dll10 = List.init 10 (fun x -> x)
  |> List.rev
  |> (List.fold_left (fun dll el -> dllcons el dll) Empty);;

list_of_dll dll10;;
```

Note, we could have used records instead of tuples in the `Node` constructor:

```OCaml
type 'a dll = Empty | Node of {
  value: 'a;
  mutable next: 'a dll;
  mutable prev: 'a dll
}
```

3. Discuss how you can detect cycles, and the time and space complexity of this.

[small/medium]

First, note that for general cycle checking, we cannot just check the head element like we did in the `dllfoldl` function, as the current element might not be in the cycle, e.g. the cycle might be $a \to b \to c \to b \to \cdots$.

The simplest way is to just add each node we have seen to a data-structure, ideally it would be a set structure, but how? We cannot just use the value, as it might be infinite. If we just use the address of the ref, then we could do comparisons or hashing – until the garbage collector does a mark and sweep and moves all our objects, and the addresses are no longer valid and items which are in the set behave as if they are no longer in the set.

```OCaml
let r = ref 1
let addressof ref : int = Obj.magic ref;;
addressof r;;
addressof r;;
Gc.compact();;
(* Wait a second *)
addressof r;;
addressof r;;
```

So we have to use a list, which has poor performance for member checking. But I should point out that this limitation is specific to this use case, for normal graph cycle checking, this isn't a problem. And OCaml does a trick for its cycle checking; it only hashes a limited part of the value. This way it probabilistically speeds it up, but then ends up using a list for values that hash to the same location in the hash table.

(Note, I ended up looking this up, then asking, because I didn't realise this trick, see https://discuss.ocaml.org/t/ocaml-cycle-detection-and-garbage-collector/15757/5 for details.)

But the store seen items in a list/set strategy ends up being costly in terms of space and time – let the path to the first repeated element be n, then we use n items of space and an upper bound for time is $n\log(n)$ if we use a tree based set, or n^2 with lists. If we can use a hash table, it could be n.

One classic cycle detection algorithm is known as Floyd's algorithm, or the tortoise and the hare. The idea is very simply that we keep two pointers going through the list, the first advances by one each time, the second advances by two each time.

```OCaml
let floyd_cycle list get_next eq =
  let get_next2 el = Option.bind (get_next el) get_next
  in let rec inner a b = match a, b with
      | Some a, Some b when eq a b -> true
      | Some a, Some b             -> inner (get_next a) (get_next2 b)
      | _                          -> false
  in inner (Some list) (get_next list);;
```

Another algorithm that is worth mentioning for cycle detection, is Brent's algorithm, which in comparison to Floyd's algorithm doesn't evaluate the next function multiple times. Instead it works by keeping a previous pointer, and periodically moving it to the current pointer. The period at which this happens doubles every time, so eventually it will be longer than the cycle length.

```OCaml
let brent_cycle list get_next eq =
  let rec inner len max prev curr =
      match get_next curr with
        | None -> false
        | Some next ->
          eq prev next ||
            if len=max
            then inner 0  (2 * max) curr curr
            else inner (len + 1) max prev next
  in inner 0  1 list list;;
```

4. It is possible to emulate cyclic lists in OCaml using pure programming too, explain how and contrast it with the impure way.

[small/medium]

It is possible to use lazy lists to make cycles, for example

```OCaml
type 'a seq = Nil | Cons of 'a * (unit -> 'a seq);;
let seqnext = function Nil -> None | Cons(_, next) -> Some(next());;

let seq_of_list l =
  List.fold_right (fun v acc -> Cons(v, fun () -> acc)) l Nil;;
```

```
6
7   (* len must be at least 0 *)
8   let mkCycle len =
9     let rec inner n =
10      Cons(n, fun () -> if len<n then inner 0 else inner (n+1))
11    in inner 0;;
12
13  let memeq a b = match a, b with Cons(a, _), Cons(b, _) -> a=b
14    | _ -> false;;
15
16  brent_cycle (mkCycle 10) seqnext memeq;;
17  floyd_cycle (mkCycle 10) seqnext memeq;;
```

However, this does not emulate cyclic lists perfectly because every time it does `Cons(a, f)` it allocates a new cons cell, rather than pointing back to an existing cell. It won't run out of memory because the garbage collector will step in, but this has extra overhead.

To show this, I have created a simple function which just traverses the list up to an upper bound.

```OCaml
1   let rec dlllast dll limit =
2     let rec inner i prev = function
3       | Empty -> prev
4       | Node(this, next, _) ->
5         if i=limit
6         then None
7         else inner (i + 1) (Some this) (!next)
8     in inner 0 None dll;;
9   let rec seqlast seq limit =
10    let rec inner i prev = function
11      | Nil -> prev
12      | Cons(this, next) ->
13        if i=limit
14        then None
15        else inner (i + 1) (Some this) (next())
16    in inner 0 None seq;;
17
18  let test =
19    let pairwise op (a1, b1) (a2, b2) = (op a1 a2, op b1 b2)
20
21    in let extract { Gc.major_collections; minor_collections } =
22      (major_collections, minor_collections)
23
24    in let measure f =
25      let startGc = Gc.stat()
26      in let startTime = Sys.time()
```

```
27      in let _ = f 1000000
28      in let stopTime = Sys.time()
29      in let stopGc = Gc.stat()
30      in let (maj, min) = pairwise (-) (extract stopGc) (extract startGc)
31      in (stopTime -. startTime, maj, min)
32
33    in let cyclic () = measure @@ dlllast dll10
34    in let lazyl () = measure @@ seqlast @@ mkCycle 10
35
36    in let repeats = 100
37    in List.init repeats (fun n ->
38      Printf.printf "\r%3d/%3d" n repeats;
39      flush stdout;
40      (cyclic(), lazyl())
41    )
42    |> List.fold_left
43      (pairwise
44        (fun (t, maj, min) (u, ma, mi) -> (t +. u, maj + ma, min + mi)))
45        ((0.0, 0, 0), (0.0, 0, 0));;
46
47  let ((cycTime, cycMaj, cycMin), (lazTime, lazMaj, lazMin)) = test;;
48  Printf.printf
49    "\r100/100\nMajor GCs %d vs %d\nMinor GCs %d vs %d\nTime %3.4f vs
    %3.4f\n"
50    cycMaj lazMaj
51    cycMin lazMin
52    cycTime lazTime;;
```

Putting it all in a file, compiling and running it:

```
❯ ocamlc cyclic-vs-lazy-loop.ml -o cyclic-vs-lazy-loop
❯ ./cyclic_vs_lazy_loop
100/100
Major GCs 300 vs 300
Minor GCs 1300 vs 4700
Time 3.1115 vs 6.4115
```

5. Why does OCaml not allow one to write `exception Poly of 'a`?

[small]

Polymorphic exceptions would break the type system, because the type of any exception is `exn`, and this doesn't say what `'a` would be in this case.

2. Lazy lists

1. Write a datatype for lazy lists/streams, and a function to create an infinite stream of integers.

[small]

This is pretty simple, note the question doesn't say it has to support finite sequences so for simplicity we do not.

```OCaml
type 'a seq = Cons of 'a * (unit -> 'a seq);;
let rec mkseq n = Cons(n, fun () -> mkseq @@ n + 1);;
let rec take n = function Cons(v, next) ->
  if n <= 0
  then []
  else v::take (n - 1) (next());;
```

2. Write a function to filter the stream.

[small]

```OCaml
let rec filter pred = function Cons(v, next) ->
  if pred v
  then Cons(v, fun () -> filter pred @@ next())
  else filter pred @@ next();;
```

3. You can define the prime number sequence as taking an integer, and removing all multiples of that integer from the rest of the sequence. (Starting at two, of course.) Code the lazy list of primes.

[medium/big]

This is more complicated, the code is small but it is involved. The core idea, is we get the list of integers from 2, filter everything divisible by two, get the next element of that list, filter everything divisible by that element, get the next element, filter everything divisible by that element, and so on.

So the first thing we need to do, is match on the list to get the current prime, return it, as the head, and as the tail, filter everything divisible by that element, and give that to the function again.

```OCaml
let primes =
  let rec inner = function
    Cons(v, next) ->
      Cons(v, fun () -> inner @@ filter (fun n -> n mod v <> 0) @@ next())
  in inner @@ mkseq 2;;
```

I should point out that this is the "Unfaithful Sieve of Eratosthenes", from The Genuine Sieve of Eratosthenes by Melissa E. O'Neill, which contains information on how to make this code more efficient.

See https://www.cs.hmc.edu/~oneill/papers/Sieve-JFP.pdf for more details.

3. Big question

This question is driven at making a standalone program, for which I'll help write part of it. First, some type definitions for the rest of the question:

```OCaml
type var_id = int
type term =
  | Var of var_id
  | List of term list
  | Int of int
  | Bool of bool
  | Str of string;;

type subst = (var_id * term) list
let rec lookup (term: term) (subst: subst) = match term, subst with
  | Var v, (w, wval)::wvs ->
    if v=w then wval else lookup (Var v) wvs
  | _ -> term
let extend (varid: var_id) (value: term) (subst: subst) =
  (varid, value)::subst

type state = subst * var_id;;
type stream = state list;;
type goal = state -> stream;;

(* Some helper functions to abstract away using lists for the stream *)
let fail: goal = fun (s, c) -> [];;
let success: goal = fun (s, c) -> [(s, c)];;
let map = List.map;;
let join = String.concat;;
```

1. Some of the past exam questions give you some code, that has been obfuscated, and ask you to explain it, or make modifications to it. Consider the following program, in which the names have been rewritten by a hungry supervisor:

   ```OCaml
   exception Out_of_ingredients
   let rec cook pantry pizza pasta =
     let deep_dish = lookup pizza pantry
     in let ravioli = lookup pasta pantry
     in match deep_dish, ravioli with
     | Var pepperoni, Var cheese when pepperoni=cheese -> pantry
     | Var pepperoni, _ -> extend pepperoni ravioli pantry
     | _, Var cheese -> extend cheese deep_dish pantry
     | List(flour::base), List(tomato::sauce) ->
         cook (cook pantry flour tomato) (List base) (List sauce)
     | x, y when x=y -> pantry
     | _ -> raise Out_of_ingredients;;
   ```

   Describe what it does. You might want to rewrite it using better names. Also, a good place to start is putting type annotations in places.

   [small/medium]

Let's first add types, and rename the confusing names:

```OCaml
exception No_match
let rec todo (subst: subst) (t1: term) (t2: term) : subst =
  let t1' : term = lookup t1 subst
  in let t2' : term = lookup t2 subst
  in match t1', t2' with
  | Var x, Var y when x=y -> subst
  | Var x, _ -> extend x t2' subst
  | _, Var y -> extend y t1' subst
  | List(x::xs), List(y::ys) ->
      todo (todo subst x y) (List xs) (List ys)
  | x, y when x=y -> subst
  | _ -> raise No_match;;
```

Now it's perhaps clearer that it walks both given terms, ensures if it is a variable then it is not in the substitutions already, and then adds variables to the substitutions. For lists, it does this for each element of the list. A better name for this function is unify, but students may use other reasonable names for it, for example recursive match or similar.

2. Using that code, we have the following function:

```OCaml
let eq a b =
  fun (s, c) ->
    try
      let s' = cook s a b
      in success (s', c)
    with Out_of_ingredients -> fail (s, c);;
```

Can you rewrite `cook` (or your better named version), and `eq` to not use exceptions?

[small]

Here we just need to use options. We cannot just do the following

```OCaml
let rec unify subst t1 t2 =
  let t1' : term = lookup t1 subst
  in let t2' : term = lookup t2 subst
  in match t1', t2' with
  | Var x, Var y when x=y -> subst
  | Var x, _ -> extend x t2' subst
  | _, Var y -> extend y t1' subst
  | List(x::xs), List(y::ys) ->
      unify (unify subst x y) (List xs) (List ys)
  | x, y when x=y -> subst
  | _ -> [];;
```

because for lists, this doesn't work as expected, because it continues to unify the rest of the lists. For example

```OCaml
unify [] (List[Int 1; Var 0]) (List[Int 2; Int 3]);;
(* Returns `[(0, Int 3)]` instead of `[]` as expected. *)
cook [] (List[Int 1; Var 0]) (List[Int 2; Int 3]);;
```

So we need to instead use the optional type, using `Some` when we would normally return a value, and `None` when we would just raise the exception. If we wanted to distinguish multiple exceptions, we could use the `type ('a, 'b) result = Ok of 'a | Error of 'b;;` type.

```OCaml
let rec unify (subst: subst) (t1: term) (t2: term) : subst option
=
  let t1' : term = lookup t1 subst
  in let t2' : term = lookup t2 subst
  in match t1', t2' with
  | Var x, Var y when x=y -> Some subst
  | Var x, _ -> Option.some @@ extend x t2' subst
  | _, Var y -> Option.some @@ extend y t1' subst
  | List(x::xs), List(y::ys) ->
      Option.bind (* bind : 'a option -> ('a -> 'b option) -> 'b option *)
        (unify subst x y)
        (fun s -> unify s (List xs) (List ys))
  | x, y when x=y -> Some subst
  | _ -> None;;
```

Then the equality becomes

```OCaml
let eq (a: term) (b: term) : goal =
  fun (s, c) ->
    match unify s a b with
      | None -> fail (s, c)
      | Some s' -> success (s', c);;
```

3. For the last part of the question, we have some more definitions, and the `var_id` in the state becomes obvious – it represents the first free variable (hence why we are using integers for variable identifiers, rather than say strings.

```OCaml
let fresh (f: term -> goal) : goal =
  fun (s, c) -> f (Var c) (s, c + 1);;
let mplus (a: stream) (b: stream) : stream = a @ b;;
let rec bind (s: stream) (g: goal) : stream =
  match s with
  | [] -> []
  | sub::subs -> mplus (g sub) (bind subs g);;

let disj (g1: goal) (g2: goal) : goal =
  fun state -> mplus (g1 state) (g2 state);;

let conj (g1: goal) (g2: goal) : goal =
  fun state -> bind (g1 state) g2;;
```

This logic programming language returns a list of all the possible results at once, discuss how you could make it not do so, and how you can then change the search strategies.

To help with this question, I have included some examples of use. For a simple example, we have the code:

```OCaml
let test =
  let query =
    fresh @@ fun p -> fresh @@ fun q ->
      let l1 = List [p; Int 7]
      in let l2 = List [Int 9; Int 7]
      in disj
          (conj
            (eq q (Int 5))
            (eq l1 l2))
          (eq q (Int 1))
  in query ([], 0);;
```

which returns

```OCaml
val test : stream = [([(0, Int 9); (1, Int 5)], 2);
                     ([(1, Int 1)], 2)];;
```

and this means that there are two solutions to the logic expression

$$(q = 5 \wedge [p, 7] = [9, 7]) \vee q = 1$$

either p is 9 and q is 5, or q is 1 and it doesn't matter what p is.

To give another example, which will help for the search strategy part, consider representing a kind of binary tree where only the leaves hold values, using list terms, for example:

```OCaml
let t1 = (List [List [List [Int 1; Int 2]; List [Int 3; Int 4]];
                List [List [Int 5; Int 6]; List [Int 7; Int 8]]]);;
```

The expression to walk all nodes (leaves and branches) of the tree can be written as follows.

```OCaml
(* Walk a nested list *)
let rec walk term out =
  disj
    (fresh @@ fun l -> fresh @@ fun r ->
      conj
        (eq term (List[l; r]))
        (disj (walk l out) (walk r out)))
    (eq term out);;
```

For convenience, I will also define some pretty printing functions, you don't necessarily need to understand them though they are not complicated, just a bit tedious.

```OCaml
(* Walks substitutions recursively until no more variables can be
   resolved -- assumes no recursive substitutions *)
let rec make_ground subst term : term = match term with
  | Var v -> (match (List.assoc_opt v subst) with
    None -> term | Some t -> make_ground subst t)
  | List l -> List (List.map (make_ground subst) l)
  | t -> t;;

(* Formats a term *)
let rec string_of_term = function
  | Var v -> Printf.sprintf "V%d" v
  | Int i -> Printf.sprintf "%d" i
  | Str s -> String.escaped s
  | Bool b -> if b then "true" else "false"
  | List l -> "[" ^ String.concat "; " (List.map string_of_term l) ^ "]";;

let print_all_mappings v streams =
  streams
  |> map (fun (subst, _) -> string_of_term @@ make_ground subst v)
  |> join "\n"
  |> print_endline;;
```

Finally we can walk the tree and print the variable binding to the second argument of `walk`.

```OCaml
walk t1 (Var 0) ([], 1)
|> print_all_mappings (Var 0);;
```

```
1
2
[1; 2]
3
4
[3; 4]
[[1; 2]; [3; 4]]
5
6
[5; 6]
7
```

```
8
[7; 8]
[[5; 6]; [7; 8]]
[[[1; 2]; [3; 4]]; [[5; 6]; [7; 8]]]
```

[medium]

I felt the supervision material was already getting to be quite a lot, so I didn't make this a coding question – in any case the interesting bit here is identifying where to do the change, not so much doing the change.

To make this lazy, we would again just code up a lazy list datatype (by the way there is `Seq` but you aren't expected to use it in your exams, instead demonstrating you can create your own). Then code up lazy versions of `mplus` and `bind`.

OCaml
```
type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
type stream = state seq;;

let join sep seq =
  let rec inner isep acc = function
    Nil -> acc | Cons(v, next) -> inner sep (acc ^ isep ^ v) (next())
  in inner "" "" seq;;
let rec map f = function
  Nil -> Nil | Cons(v, next) -> Cons(f v, fun () -> map f @@ next())

let fail = fun (s, c) -> Nil
let success = fun (s, c) -> Cons((s, c), fun () -> Nil);;

let rec (@=) a b = match a, b with
  | Cons(av, an), b -> Cons(av, fun () -> an() @= b)
  | Nil, b -> b;;

let mplus a b = a @= b;;
let rec bind s g = match s with
  | Nil -> Nil
  | Cons(sub, subs) -> mplus (g sub) (bind (subs()) g);;
```

For search strategies, we can consider where streams are potentially very large or infinite, and we might be neglecting other elements of the stream. In the function `mplus` we have to exhaust the first stream before we get to the second one, if instead we modify it then we can get a fairer traversal.

OCaml
```
let rec alternate a b = match a, b with
  | Nil, b -> b | a, Nil -> a
  | Cons(av, an), Cons(bv, bn) -> Cons(av, fun () -> alternate b (an()));;
let mplus a b = alternate a b;;
```

We could also improve fairness by making the conjunction and disjunction take lists rather than just be binary operators. This avoid an expression representing $(a \wedge b) \wedge c$ taking from a and b half as often as from c.

Note, this is the transpose function, except this doesn't use append because it uses a queue instead.

```ocaml
(* OCaml Queue module is not functional. *)
let rec alternate (seqs: 'a seq seq) : 'a seq =
  let q: 'a seq Queue.t = Queue.create()
  in let rec inner = function
    | Nil -> (match Queue.take_opt q with
      | None -> Nil
      | Some s -> inner @@ Cons(s, fun () -> Nil))
    | Cons(Nil, next) -> inner @@ next()
    | Cons(Cons(a, an), bn) ->
      Cons(a, fun () -> Queue.add (an()) q; inner @@ bn())
  in inner seqs;;

let rec seq_of_list = function
  | [] -> Nil
  | x::xs -> Cons(x, fun () -> seq_of_list xs);;

let rec list_of_seq = function
  | Nil -> []
  | Cons(v, next) -> v::(list_of_seq @@ next());;

let any (g: goal list) : goal =
  fun state ->
    alternate @@ map (fun g -> g state) @@ seq_of_list g;;

let rec bind s g = match s with
  | Nil -> Nil
  | Cons(sub, subs) -> mplus (g sub) (bind (subs()) g);;

let all (g1: goal) (g2: goal) : goal =
  fun state -> bind (g1 state) g2;;
```

3.1. Bonus

This uses code from the "Big question" section, optionally along with the lazy list version, to make a type checker for the simply typed lambda calculus. First, we will need some convenience functions.

```ocaml
let any (g: goal list) : goal =
  List.fold_left disj fail g;;

let all (g: goal list) : goal =
  List.fold_left conj success g;;
```

The simply typed lambda-calculus has the typing rules

$$\frac{\Gamma, \text{arg} : \alpha \vdash \text{body} : \beta}{\Gamma \vdash \text{fn arg} \to \text{body} : \alpha \to \beta}\,(\text{fun})$$

$$\frac{\Gamma \vdash \text{func} : \alpha \to \beta \quad \Gamma \vdash \text{arg} : \alpha}{\Gamma \vdash \text{func arg} : \beta}\,(\text{app})$$

$$\frac{}{\Gamma, x : \alpha \vdash x : \alpha}\,(\text{var. found})$$

$$x \neq y \frac{\Gamma \vdash x : \beta}{\Gamma, y : \alpha \vdash x : \beta}\,(\text{var. not found})$$

Which we can easily express as a series of logic rules in our simple logic programming system.

```OCaml
let rec typecheck gamma term typ =
  any [
    (* fun *)
    (fresh @@ fun var -> fresh @@ fun body ->
      fresh @@ fun a -> fresh @@ fun b ->
      all [
        eq term @@ List[Str "fun"; var; body];
        eq typ @@ List[a; Str "->"; b];
        typecheck ((var, a)::gamma) body b]);
    (* app *)
    (fresh @@ fun fn -> fresh @@ fun arg ->
      fresh @@ fun a -> fresh @@ fun b ->
      all [
        eq term @@ List[fn; arg];
        eq typ b;
        typecheck gamma fn (List[a; Str "->"; b]);
        typecheck gamma arg a]);
    match gamma with
    | [] -> fail
    | (x, a)::rest ->
      any [
        (* var found *)
        all [eq x term; eq a typ];
        (* var not found *)
        typecheck rest term typ]
  ];;
```

We will also write a couple of functions to pretty-print the resulting type, to make it easier to read.

```ocaml
(* Creates a module for mapping variables (integers) *)
module VarMap = Map.Make(struct include Int end);;

(* Maps a number to a type, e.g. 0 to 'a, 25 to 'z, 26 to 'aa, 27 to 'ab etc *)
let format_var n =
  let char_of_var n = Char.chr (Char.code 'a' + n)
  in let rec inner n acc =
    if n < 26
    then char_of_var n :: acc
    else inner (n / 26 - 1) @@ char_of_var (n mod 26) :: acc
  in "'" ^ (String.of_seq @@ List.to_seq @@ inner n []);;

(* Map each variable to 'a, 'b, etc *)
let map_var vars v = match VarMap.find_opt v vars with
  | None ->
    let mapping = format_var (VarMap.cardinal vars)
    in (VarMap.add v mapping vars, mapping)
  | Some s -> (vars, s);;

(* Formats a type (which is assumed to be only `List[type; Str"->" type]` or
   `Var v`) as a readable string *)
let rec format_type vars term = match term with
  | List[List l; Str "->"; r] ->
    let (vars', l') = format_type vars (List l)
    in let (vars'', r') = format_type vars' r
    in (vars'', Printf.sprintf "(%s) -> %s" l' r')
  | List[l; Str "->"; r] ->
    let (vars', l') = format_type vars l
    in let (vars'', r') = format_type vars' r
    in (vars'', Printf.sprintf "%s -> %s" l' r')
  | Str s -> (vars, s)
  | Var v -> map_var vars v
  | _ -> failwith "format_type";;

let format_results query results =
  map (fun ((subst, _): state) ->
    let (_, ty) = format_type VarMap.empty (make_ground subst query)
    in ty) results
  |> join "\n" |> Printf.printf "Results:\n%s\n";;
```

Now we can try some examples! This fails because it has an unbound variable

```
let t0 = typecheck [] (Str"x") (Var 0) ([], 1);;
format_results (Var 0) t0;;
```

```
Results:
```

If we bind the variable, it works:

```
let t1 = typecheck [(Str"x", (Var 0))] (Str"x") (Var 0) ([], 1);;
format_results (Var 0) t1;;
```

```
Results:
'a
```

For a different expression that has no unbound variables:

```
let t2 = typecheck [] (List[Str"fun"; Str"x"; Str"x"]) (Var 0) ([], 1);;
format_results (Var 0) t2;;
```

```
Results:
'a -> 'a
```

And for a more complicated expression:

```
let t3 = typecheck
  []
  (* Encoding the term `fun x -> fun y -> fun z -> x y (z y)` *)
  (List[Str"fun"; Str"x";
    List[Str"fun"; Str"y";
      List[Str"fun"; Str"z";
        List[
          List[Str"x"; Str"y"];
          List[Str"z"; Str"y"]]]]])
  (Var 0)
  ([], 1);;
format_results (Var 0) t3;;
```

```
Results:
('a -> 'b -> 'c) -> 'a -> ('a -> 'b) -> 'c
('a -> 'b -> 'c) -> 'a -> ('a -> 'b) -> 'c
```

Note, because we do not do occurs checks, the following also works (not pretty-printing it because the pretty-printer would try to expand it forever):

```
let t4 = typecheck
  []
  (* Encoding the term `fun x -> x x` *)
  (List[Str"fun"; Str"x"; List[Str"x"; Str"x"]])
  (Var 0)
  ([], 1);;
```

```
val t4 : stream =
  Cons
  (([(7, List [Var 7; Str "->"; Var 8]); (3, List [Var 7; Str "->"; Var 8]);
      (4, Var 8); (6, Str "x"); (5, Str "x");
      (0, List [Var 3; Str "->"; Var 4]); (2, List [Str "x"; Str "x"]);
      (1, Str "x")],
    9),
  <fun>)
```

Which does represent the infinite type `'a = 'a -> 'b` as expected. To fix this, you would have to look at adding an "occurs check" into the `extend` function.

If you want to learn more about this kind of logic programming, you could read more about "microkanren", or read the book "Reasoned Schemer" – it is an excellent book, I recommend it. So are the rest of that series.