Up

Module Csp

```
module Csp: sig .. end
```

This is a concurrency library modelled after Communicating Sequential Processes by C. A. R. Hoare and inspired by libraries such as JCSP, C++CSP and PyCSP, as well as Occam. It provides any-to-any synchronous channels with alternation, poisoning and permissions. It also provides primitives for starting processes.

Representation

```
exception PoisonException
```

Thrown when reading from or writing to a poisoned channel.

```
type 'a guard
```

Represents a guard (for example for reading and writing) that can be used in a select.

```
type ('a, 'b) channel
```

Represents a channel for transmitting messages of type 'a. The permissions for the channel handle is in 'b.

```
type on
```

A reperesentation of "true" in the type system.

```
type off
```

A reperesentation of "false" in the type system.

```
type 'a chan = ('a, on * on * on) channel
```

A shorthand for channel handles with all permissions: to read, to write and to poison, in that order.

Channel interaction

```
val new_channel : unit -> ('a, on * on * on) channel
```

Creates a channel with all permissions. Any number of processes can read from and write to this channel as desired. A message is always sent from exactly one process to one other process.

```
val select : 'a guard list -> 'a
```

Blocks until one of the guarded processes become ready, then becomes the guarded process. If multiple guards are ready, one is chosen (pseudo) randomly. Note that this is a stronger guarentee than the corresponding CSP construct, which only specifies that an arbitrary one will be chosen. For any single channel, processes that are waiting to read or write are also served on a first come, first served basis. If there are no guards in the list or if at least one of the guards are accociated with a poisoned channel, PoisonException is thrown.

`val read_guard : ('a, on * 'b * 'c) channel -> ('a -> 'd) -> 'd guard`

A read-guarded process becomes ready when there is somebody waiting to write on the channel.

`val write_guard : ('a, 'b * on * 'c) channel -> 'a -> (unit -> 'd) -> 'd guard`

A write-guarded process becomes ready when there is somebody waiting to read on the channel.

`val poison : ('a, 'b * 'c * on) channel -> unit`

Poisons the channel.

`val read : ('a, on * 'b * 'c) channel -> 'a`

Receives a value from the channel. `read c` is equivalent to `select [read_guard c (fun x -> x)]`.

`val write : ('a, 'b * on * 'c) channel -> 'a -> unit`

Sends a value via the channel. `write c` is equivalent to `select [write_guard c v (fun _ -> ())]`.

```
(* Alternation. *)
Csp.select [
    read_guard c1 (fun x -> print_string ("read " ^ x ^ " from c1"));
    read_guard c2 (fun x -> print_string ("read " ^ x ^ " from c2"));
    write_guard c3 7 (fun x -> print_string ("wrote " ^ x ^ " to c3"));
]
```

Starting processes

These processes run in a shared memory environment (implemented using the standard OCaml threading libraries).

`val parallel : (unit -> unit) list -> unit`

Runs a list of functions as processes in parallel. It will only return once all these processes have finished. If any of the processes is thrown, it will be rethrown from this function once all processes have finished. Even if multiple exceptions are thrown, only the exception of the first processes, in the order of the process list, will be rethrown.

```
(* Read from two channels in parallel and sum the values. *)
let parallel_add i1 i2 o () = while true do
    let i1o = Csp.channel () in
    let i2o = Csp.channel () in
    Csp.parallel [
        (fun () -> Csp.write i1o (Csp.read c1));
        (fun () -> Csp.write i2o (Csp.read c2));
        (fun () -> Csp.write o (Csp.read i1o + Csp.read i2o));
    ] done
```

Channel permissions

The channel permissions control what you can do through a handle. These are enforced statically by the type system. Each of the functions return a handle that only has the advertised permissions, and none of them add new permissions.

```
val read_only : ('a, on * 'b * 'c) channel ->
       ('a, on * off * off) channel

val read_write_only : ('a, on * on * 'b) channel ->
       ('a, on * on * off) channel

val read_poison_only : ('a, on * 'b * on) channel ->
       ('a, on * off * on) channel

val write_only : ('a, 'b * on * 'c) channel ->
       ('a, off * on * off) channel

val write_poison_only : ('a, 'b * on * on) channel ->
       ('a, off * on * on) channel

val poison_only : ('a, 'b * 'c * on) channel ->
       ('a, off * off * on) channel


(* Chosing permissions on a "need to know" basis. *)
let rec counter c n () =
    if n == 0 then Csp.poison c else begin
        (* Csp.read c   <-- would be a compile time error. *)
        Csp.write c n;
        counter c (n - 1)
    end
let rec printer c () = begin
        print_endline (string_of_int (Csp.read c));
        printer c ()
    end
let _ =
    let c = Csp.channel () in
    Csp.parallel [
        counter (Csp.write_poison_only c) 42;
        printer (Csp.read_only c);
    ]
```