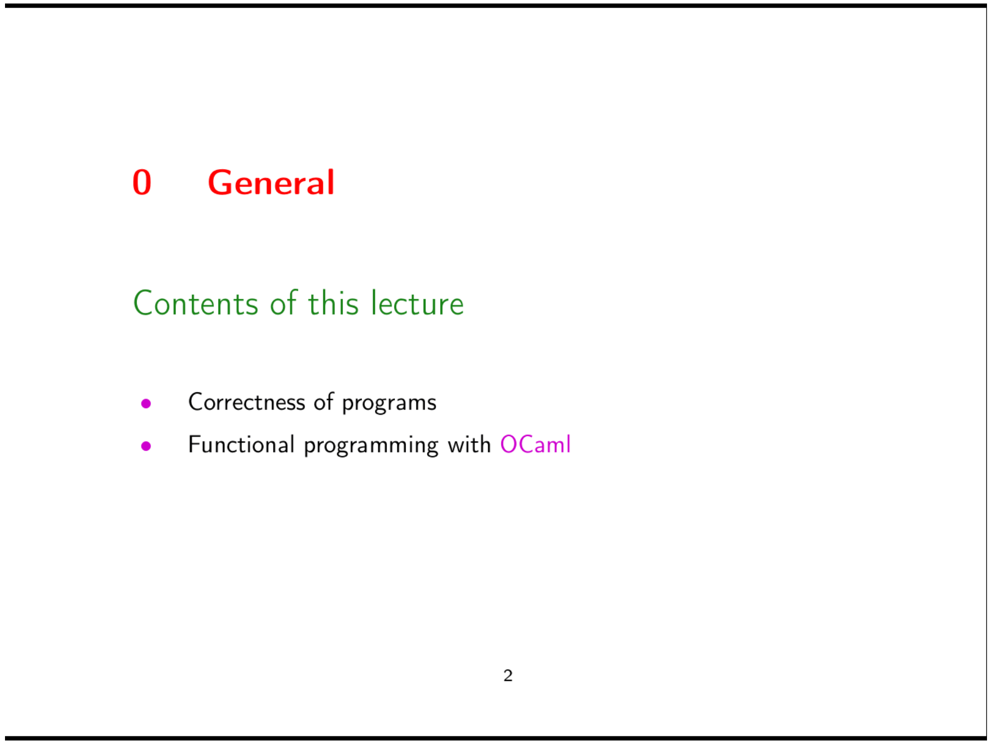
Script **generated by TTT**

Title:	Seidl: Functional Programming and Verification (01.02.2019)
Date:	Fri Feb 01 08:42:43 CET 2019
Duration:	71:34 min
Pages:	22

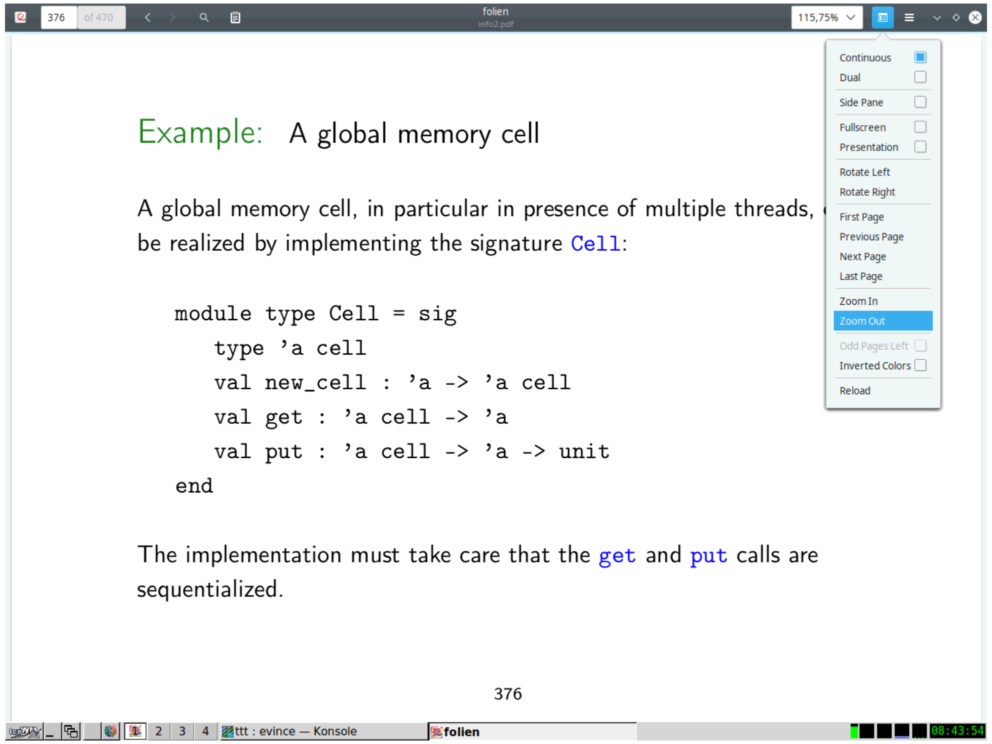


0 General

Contents of this lecture

- Correctness of programs
- Functional programming with OCaml

2



Example: A global memory cell

A global memory cell, in particular in presence of multiple threads, be realized by implementing the signature Cell:

```
module type Cell = sig
   type 'a cell
   val new_cell : 'a -> 'a cell
   val get : 'a cell -> 'a
   val put : 'a cell -> 'a -> unit
end
```

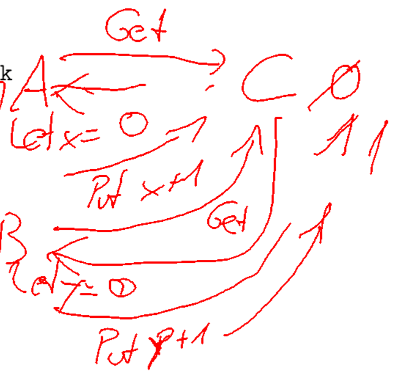
The implementation must take care that the get and put calls are sequentialized.

376

Example: Locks

Often, only one at a time out of several active threads should be allowed access to a given resource. In order to realize such a mutual exclusion, locks can be applied:

```
module type Lock = sig
   type lock
   type ack
   val new_lock : unit -> lock
   val acquire : lock -> ack
   val release : ack -> unit
end
```



Execution of the operation acquire returns an element of type ack which is used to return the lock:

```
type ack = unit channel
type lock = ack channel
```

For simplicity, ack is chosen itself as the channel by which the lock is returned.

```
let acquire lock = let ack = new_channel ()
                   in  sync (send lock ack);
                       ack
```

The unlock channel is created by acquire itself

```
let release ack = sync (send ack ())
```

... and used by the operation release.

```
let new_lock () = let lock = new_channel ()
               in let rec acq_server () =
                           rel_server (sync (receive lock))
                       and rel_server ack =
                           sync (receive ack);
                           acq_server ()
               in create acq_server ();
                  lock
```

acq ack

Core of the implementation are the two mutually recursive functions acq_server and rel_server.

acq_server expects an element ack, i.e., a channel, and upon reception, calls rel_server.

rel_server expects a signal on the received channel indicated that the lock is released ...

Now we are in the position to realize a decent deadlock:

```
let dead  = let l1 = new_lock ()
            in let l2 = new_lock ()
   ...
```

Idea

Again, a server is realized using an accumulating parameter, now maintaining the number of free resources or, if negative, the number of waiting threads ...

```
module Sema = struct open Thread open Event
type sema = unit channel option channel
let up sema = sync (send sema None)
let down sema = let ack = (new_channel () : unit channel)
                in sync (send sema (Some ack));
                   sync (receive ack)
...
```

Idea

Again, a server is realized using an accumulating parameter, now maintaining the number of free resources or, if negative, the number of waiting threads ...

```
module Sema = struct open Thread open Event
type sema = unit channel option channel
let up sema = sync (send sema None)
let down sema = let ack = (new_channel () : unit channel)
                in sync (send sema (Some ack));
                   sync (receive ack)
...
```

```
...
let new_sema n = let sema = new_channel ()
    in let rec serve (n,q) =
       match sync (receive sema)
        with None -> (match dequeue q
Empty        with  (None,q) -> serve (n+1,q)
             | (Some ack,q) -> sync (send ack ());
                               serve (n,q))
        | Some ack -> if n>0 then (sync (send ack ());
                                   serve (n-1,q))
                      else serve (n, enqueue ack q)
    in create serve (n,new_queue()); sema
end
```

Apparently, the queue does not maintain the waiting threads, but only their back channels.

Idea

Again, a server is realized using an accumulating parameter, now maintaining the number of free resources or, if negative, the number of waiting threads ...

```
module Sema = struct open Thread open Event
type sema = unit channel option channel
let up sema = sync (send sema None)
let down sema = let ack = (new_channel () : unit channel)
                in sync (send sema (Some ack));
                   sync (receive ack)
...
```

type msg = Acq of ack channel | Rel

```
...
let new_sema n = let sema = new_channel ()
    in let rec serve (n,q) =
       match sync (receive sema)
        with None -> (match dequeue q
             with  (None,q) -> serve (n+1,q)
              | (Some ack,q) -> sync (send ack ());
                                 serve (n,q))
        | Some ack -> if n>0 then (sync (send ack ());
                                     serve (n-1,q))
                        else serve (n, enqueue ack q)
    in create serve (n,new_queue()); sema
end
```

Apparently, the queue does not maintain the waiting threads, but only their back channels.

8.2 Selective Communication

A thread need not necessarily know which of several possible communication rendezvous will occur or will occur first.
Required is a non-deterministic choice between several actions ...

Example: The function

```
add : int channel * int channel * int channel -> unit
```

is meant to read integers from two channels and send their sum to the third.

```
let copy (in, out1, out2) = forever (fun () ->
    let x = sync (receive in)
    in select [
       wrap (send out1 x)
            (fun () -> sync (send out2 x));
       wrap (send out2 x)
            (fun () -> sync (send out1 x))
       ]
    ) ()
```

Apparently, the event list may also consist of send events — or contain both kinds.

```
type 'a cell = 'a channel * 'a channel
...
```

```
...
let get (get_chan,_) = sync (receive get_chan)
let put (_,put_chan) x = sync (send put_chan x)
let new_cell x = let get_chan = new_channel ()
              in let put_chan = new_channel ()
              in let serve x = select [
         wrap (send get_chan x) (fun () -> serve x);
         wrap (receive put_chan) serve
    ]
    in
       create serve x;
       (get_chan, put_chan)
```

```
...
let get (get_chan,_) = sync (receive get_chan)
let put (_,put_chan) x = sync (send put_chan x)
let new_cell x = let get_chan = new_channel ()
              in let put_chan = new_channel ()
              in let serve x = select [
        wrap (send get_chan x) (fun () -> serve x);
        wrap (receive put_chan) serve
    ]
    in
       create serve x;
       (get_chan, put_chan)
```

→ The leaves are basic events.

→ A wrapper function may be applied to any given event.

→ Several events of the same type may be combined into a choice.

→ Synchronization on such an event tree activates a single leaf event. The result is obtained by successively applying the wrapper functions from the path to the root.

Example: A Swap Channel

Upon rendezvous, a swap channel is meant to exchange the values of the two participating threads. The signature is given by

```
module type Swap = sig
   type 'a swap
   val new_swap : unit -> 'a swap
   val swap : 'a swap -> 'a -> 'a event
end
```

In the implementation with ordinary channels, every participating thread must offer the possibility to receive and to send.

Timeouts

Often, our patience is not endless.

Then, waiting for a send or receive event should be terminated ...

```
module type Timer = sig
   set_timer : float -> unit event
   timed_receive : 'a channel -> float -> 'a option event
   timed_send    : 'a channel -> 'a -> float -> unit option event
end
```

```
module Timer = stuct open Thread open Event
   let set_timer t = let ack = new_channel ()
                  in let serve () = delay t;
                                    sync (receive ack)
                  in create serve (); send ack ()

   let timed_receive ch time = choose [
       wrap (receive ch) (fun a -> Some a);
       wrap (set_timer time) (fun () -> None)
   ]

   let timed_send ch x time = choose [
       wrap (send ch x) (fun a -> Some ());
       wrap (set_timer time) (fun () -> None)
   ]
end
```

8.3 Threads and Exceptions

An exception must be handled within the thread where it has been raised.

```
module Explode = struct open Thread
let thread x = (x / 0);
       print_string "thread terminated regularly ...\n"
let main = create thread 0; delay 1.0;
           print_string "main terminated regularly ...\n"
end
```

... yields

```
> /.a.out
Thread 1 killed on uncaught exception Division_by_zero
main terminated regularly ...
```

The thread was killed, the Ocaml program terminated nontheless.

Also, uncaught exceptions within the wrapper function terminate the running thread:

```
module ExplodeWrap = struct open Thread open Event open Timer
let main = try sync (wrap (set_timer 1.0) (fun () -> 1 / 0))
           with _ -> 0;
           print_string "... this is the end!\n"
end
```