Script generated by TTT

Title: Seidl: Functional Programming and Verification (08.02.2019)

Date: Fri Feb 08 08:29:31 CET 2019

Duration: 67:30 min

Pages: 16

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 nonetheless.

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
```

Then we have

```
> ./a.out
Fatal error: exception Division_by_zero
```

Caveat

Exceptions can only be caught in the body of the wrapper function itself, not behind the `sync` !

8.4 Buffered Communication

A channel for buffered communication allows to send without blocking. Receiving still may block, if no messages are available. For such channels, we realize a module `Mailbox`:

```
module type Mailbox = sig
   type 'a mbox
   val new_mailbox : unit -> 'a mbox
   val send :      'a mbox -> 'a -> unit
   val receive : 'a mbox -> 'a event
end
```

For the implementation, we rely on a server which maintains a queue of sent but not yet received messages.

Then we implement:

```
module Mailbox =
struct open Thread open Queue open Event
   type 'a mbox = 'a channel * 'a channel
   let send (in_chan,_) x     = sync (send in_chan x)
   let receive (_,out_chan)   = receive out_chan
   let new_mailbox () = let in_chan  = new_channel ()
                        and out_chan = new_channel ()
...
```

```
   ...
       in let rec serve q = if (is_empty q) then
                  serve (enqueue (
                  sync (Event.receive in_chan)) q)
          else select [
                 wrap (Event.receive in_chan)
                      (fun y -> serve (enqueue y q));
                 wrap (Event.send out_chan (first q))
                      (fun () -> let (_,q) = dequeue q
                                 in serve q)
               ]
       in create serve (new_queue ());
          (in_chan, out_chan)
end
```

... where `first : 'a queue -> 'a` returns the first element in the queue without removing it.

unit event

```
   ...
       in let rec serve q = if (is_empty q) then
                  serve (enqueue (
                  sync (Event.receive in_chan)) q)
          else select [
                 wrap (Event.receive in_chan)
                      (fun y -> serve (enqueue y q));
                 wrap (Event.send out_chan (first q))
                      (fun () -> let (_,q) = dequeue q
                                 in serve q)
               ]
       in create serve (new_queue ());
          (in_chan, out_chan)
end
```

... where `first : 'a queue -> 'a` returns the first element in the queue without removing it.

unit event

unit event

```
   ...
       in let rec serve q = if (is_empty q) then
                  serve (enqueue (
                  sync (Event.receive in_chan)) q)
          else select [
                 wrap (Event.receive in_chan)
                      (fun y -> serve (enqueue y q));
                 wrap (Event.send out_chan (first q))
                      (fun () -> let (_,q) = dequeue q
                                 in serve q)
                ]
       in create serve (new_queue ());
          (in_chan, out_chan)
end
```

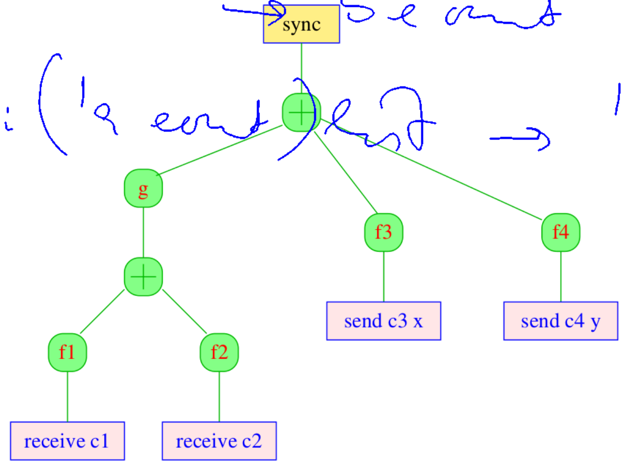
... where `first : 'a queue -> 'a` returns the first element in the queue without removing it.

wrap: 'a event → ('a → 'b)

In general, there could be a tree of events:

→ 'b event

select : ('a event list) → 'a



```
   ...
       in let rec serve q = if (is_empty q) then
                  serve (enqueue (
                  sync (Event.receive in_chan)) q)
          else select [
                 wrap (Event.receive in_chan)
                      (fun y -> serve (enqueue y q));
                 wrap (Event.send out_chan (first q))
                      (fun () -> let (_,q) = dequeue q
                                 in serve q)
                ]
       in create serve (new_queue ());
          (in_chan, out_chan)
end
```

... where `first : 'a queue -> 'a` returns the first element in the queue without removing it.

8.5 Multicasts

For sending a message to many receivers, a module Multicast is provided that implements the signature Multicast:

```
   module type Multicast = sig
      type 'a mchannel and 'a port
      val new_mchannel : unit -> 'a mchannel
      val new_port : 'a mchannel -> 'a port
      val multicast : 'a mchannel -> 'a -> unit
      val receive : 'a port -> 'a event
end
```

The operation new_port generates a fresh port where a message can be received.

The (non-blocking) operation multicast sends to all registered ports.

```
module Multicast = struct open Thread open Event
module M = Mailbox
type 'a port = 'a M.mbox
type 'a mchannel = 'a channel * 'a port channel

let new_port (_, req)    = let m = M.new_mailbox() in
                             sync (send req m); m
let multicast (send_ch,_) x = sync (send send_ch x)
let receive mbox              = M.receive mbox
    ...
```

```
      ...
   let main  = let mc = new_mchannel ()
       in let thread i = let p = new_port mc
          in while true do let x = sync (receive p)
                           in print_int i; print_string ": ";
                              print_string (x^"\n")
                        done
       in  create thread 1; create thread 2;
           create thread 3; delay 1.0;
           multicast mc "Hallo!";
           multicast mc "World!";
           multicast mc "... the end.";
           delay 10.0
   end
end
```

Summary

- The programming language Ocaml offers convenient possibilities to orchestrate concurrent programs.
- Channels with synchronous communicatino allow to simulate other concepts of concurrency such as asynchronous communication, global variables, locks for mutual exclusion and semaphors.
- Concurrent functional programs can be as obfuscated and incomprehensible and concurrent Java programs.
- Methods are required in order to systematically verify the correctnes of such programs ...

Perspectives

- Beyond the language concepts discussed in the lecture, Ocaml has diverse further concepts, which also enable object oriented programming.
- Moreover, Ocaml has elegant means to access functionality of the operating system, to employ graphical libraries and to communicate with other computers ...

$\Longrightarrow$ Ocaml is an interesting alternative to Java.