

## Script generated by TTT

Title: Seidl: Functional Programming and Verification (25.01.2019)

Date: Fri Jan 25 08:31:19 CET 2019

Duration: 88:24 min

Pages: 17

**Case 4:**  $x = x1::xs \wedge y = y1::ys \wedge x1 > y1$ .

We deduce:

```
sorted (merge x y) = sorted (merge (x1::xs) (y1::ys))
                  = sorted (y1 :: merge x ys)
                  = ...
```

**Case 4.1:**  $ys = []$

We deduce:

```
... = sorted (y1 :: merge x [])
    = sorted (y1 :: x)
    = sorted x
    = true
```

## Discussion

ke 59

- Again, we have implicitly assumed that all calls of `app`, `rev` and `rev1` terminate.
- Termination of these can be proven by induction on the length of their first arguments.
- The claim:

$$\text{rev } x = \text{rev1 } x \ []$$

follows from:

$$\text{app } (\text{rev } x) \ y = \text{rev1 } x \ y$$

by setting:  $y = []$  and assertion (1) from [example 1](#).

## Discussion

- Again, we have assumed for the proof that all calls of the functions `sorted` and `merge` terminate.
- As an additional techniques, we required a sorrow [case distinction](#) over the various possibilities for arguments in calls.
- The case distinction made the proof longish and cumbersome.

// The case  $n = 0$  is in fact superfluous.

// since it is covered by the cases 1 and 2

o canlop

## 8 Parallel Programming

The threads library `threads.cma` supports the implementation of systems using more than a single ...

### Example

```
module Echo = struct open Thread
  let echo () = print_string (read_line () ^ "\n")
  let main    = let t1 = create echo ()
                in join t1;
                print_int (id (self ()));
                print_string "\n"
end
```

367

## 8 Parallel Programming

The threads library `threads.cma` supports the implementation of systems using more than a single ...

### Example

```
module Echo = struct open Thread
  let echo () = print_string (read_line () ^ "\n")
  let main    = let t1 = create echo ()
                in join t1;
                print_int (id (self ()));
                print_string "\n"
end
```

367

## 8 Parallel Programming

The threads library `threads.cma` supports the implementation of systems using more than a single ...

### Example

```
module Echo = struct open Thread
  let echo () = print_string (read_line () ^ "\n")
  let main    = let t1 = create echo ()
                in join t1;
                print_int (id (self ()));
                print_string "\n"
end
```

367

## Comments

- The module `Thread` collects basic functionality for the creation of concurrency.
- The function `create: ('a -> 'b) -> 'a -> t` creates a new thread with the following properties:
  - The thread evaluates the function for its argument.
  - The creating thread receives the thread `id` as the return value and proceeds independently.
  - By means of the functions: `self : unit -> t` and `id : t -> int`, the own thread id can be queried and turned into an `int`, respectively.

368

## Further useful Functions

- The function `join: t -> unit` blocks the current thread until the evaluation of the given thread has terminated.
- The function `kill: t -> unit` stops a given thread (not implemented);
- The function `delay: float -> unit` delays the current thread by a time period in seconds;
- The function `exit: unit -> unit` terminates the current thread.

369

*scanlop -I thread unix.cma  
Caveat*

- Within the interactive environment, threads can be enabled via the option `#thread;; !`
- Alternatively, we can compile with the option `-thread` :  
`> ocamlc -thread unix.cma threads.cma Echo.ml`
- The library `threads.cma` requires auxiliary functionality from the library `unix.cma`.  
`//` for Windows, the situation might be different
- The program can then be tested via the call  
`> ./a.out`

370

## 8.1 Channels

Threads communicate via channels.

The module `Event` provides basic functionality for the creation of channels, sending and receiving:

```
type 'a channel
new_channel : unit -> 'a channel
type 'a event
always : 'a -> 'a event
sync : 'a event -> 'a
send : 'a channel -> 'a -> unit event
receive : 'a channel -> 'a event
```

372

- Each call `new_channel()` creates another channel.
- **Arbitrary** data may be sent across a channel !!!
- `always` wraps a value into an `event`.
- Sending and receiving generates `events` ...
- **Synchronization** on event returns their `values`.

```
module Exchange = struct open Thread open Event
let thread ch = let x = sync (receive ch)
                in print_string (x ^ "\n");
                sync (send ch "got it!")
let main = let ch = new_channel () in create thread ch;
           print_string "main is running ... \n";
           sync (send ch "Greetings!");
           print_string ("He " ^ sync (receive ch) ^ "\n")
end
```

373

## Discussion

- `sync (send ch str)` exposes the `event` of sending to the outside world and `blocks` the sender, until another thread has read the value from the channel ...
- `sync (receive ch)` blocks the receiver, until a value has been made available on the channel. Then this value is returned as the result.
- Synchronous communication is one alternative for exchange of data between threads as well as for orchestration of concurrency  
⇒ `rendezvous`
- In particular, it can be used to realize asynchronous communication between threads.

374

In the example, `main` spawns a thread. Then it sends it a string and waits for the answer. Accordingly, the new thread waits for the transfer of a `string` value over the channel. As soon as the string is received, an answer is sent on `the same` channel.

## Caveat

If the ordering of the sequence of `send` and `receive` is not carefully designed, threads easily get blocked ...

Execution of the program yields:

```
> ./a.out
main is sending ...Greetings!
He got it!
>
```

375

In the example, `main` spawns a thread. Then it sends it a string and waits for the answer. Accordingly, the new thread waits for the transfer of a `string` value over the channel. As soon as the string is received, an answer is sent on `the same` channel.

## Caveat

If the ordering of the sequence of `send` and `receive` is not carefully designed, threads easily get blocked ...

Execution of the program yields:

```
> ./a.out
main is sending ...Greetings!
He got it!
>
```

375

## Example: A global memory cell

Eine globale Speicherzelle, insbesondere in Anwesenheit mehrerer Threads sollte die Signatur A global memory cell, in particular in presence of multiple threads, can 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: A global memory cell

Eine globale Speicherzelle, insbesondere in Anwesenheit mehrerer Threads sollte die Signatur A global memory cell, in particular in presence of multiple threads, can 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

This task is delegated to a `server` thread that reacts to `get` and `put`:

```
type 'a req = Get of 'a channel | Put of 'a
type 'a cell = 'a req channel
```



The channel transports requests to the memory cell, which either provide the new value or the back channel ...

*cell*

377