Functional Programming and Verification

Prof. Dr. H. Seidl, N. Hartmann, R. Vogler
WS 2018/19

Exercise Sheet 13
Deadline: 10.02.2019

General Information

Detailed information about the lecture, tutorials and homework assignments can be found on the lecture website[1]. Solutions have to be submitted to Moodle[2]. Make sure your uploaded documents are readable. Blurred images will be rejected. Use Piazza[3] to ask questions and discuss with your fellow students.

Submission

Upload your solutions as a single file named 'hw13.ml' to moodle. Since submissions are tested automatically, solutions that do not compile or do not terminate within a given time frame cannot be graded and thus result in 0 Points. If you do not manage to get all your stuff compiling and/or terminating, comment the corresponding parts of the file!

Threaded Code

In order to compile and execute multithreaded code, use `ocaml` resp. `ocamlc` with arguments `-I +threads unix.cma threads.cma` or `utop -require threads.posix`.

Assignment 13.1 (L) Hellish Counting

In this assignment, you are going to realize concurrent counting.

1. As a first step, implement a function `spawn_counter : int -> Thread.t` that spawns a new thread. This thread should then print all numbers from 0 to the passed argument to the standard output. Print the thread's id in addition to the current number, so that you can identify who is responsible for the output.

2. Write a function `run_counters : int -> int -> unit` that, when called with `run_counters m n`, spawns `m` counters, each counting to `n`. Make sure `run_counters` does not return before all the counters have stopped.

3. Discuss the output you expect for calls of `run_counters m n` with different values of `m` and `n`. Then, check it out!

As a next step, the threads shall now be synchronized, such that all threads take turns with their output. First all threads print 0, then all threads print 1 and so on. Use channels for communication between the threads. Make sure they shutdown correctly and are joined by the main thread.

4. Implement a new version of `spawn_counter` and `run_counters` such that the counters take turns counting.

[1] https://www.in.tum.de/i02/lehre/wintersemester-1819/vorlesungen/functional-programming-and-verification/
[2] https://www.moodle.tum.de/course/view.php?id=44932
[3] https://piazza.com/tum.de/fall2018/in0003/home

Suggested Solution 13.1

All implementations are in *p13_sol.ml*. A couple of things can be observed:

- Output of different counters (threads) can be interleaved arbitrarily. Function calls (e.g. a single `print_endline`) are not atomic and can be interrupted by another thread.
- Buffering seems to be an issue, as sometimes the output of some threads is not shown when the program ends (this becomes more obvious with larger `n`). Although `print_newline` and `print_endline` (among others) are supposed to flush the output, OCaml seems to have some serious issues to do standard output across threads. Adding a small delay (100 ms) right before the thread ends, seems to help.
- `Printf.printf` seems to be broken entirely, as output becomes very irregular and slows down tremendously. In some cases, `Printf.printf` does lead to a deadlock of the program.
- Functions have to be tail-recursive in order to prevent stack overflows.
- A setup organized by the main thread (star topology) is very easy to realize by notifying the threads round robin and giving them a means to signal back that they are done counting.
- In a setup without a designated leader (ring topology) the correct shutdown is quite difficult. Signaling the neighbor to shutdown once `n` is reached does not work for the last thread to end counting, as it will wait on a rendezvous on the channel forever. If we assign this specific behavior to one thread beforehand, this is easy to circumvent. In case of homogeneous threads, one way to perform an organized shutdown is by shrinking the ring whenever one thread finishes counting. To do so, the thread sends its own predecessor (the channel it is receiving on) to its successor over the channel. This is repeated for every finished counter until one thread's send and receive channels are equal (this is the singleton case).

Assignment 13.2 (L) Blog-Server

In this assignment you are going to simulate a server where users can post their own `blog`. As a `blog` we consider a list of `posts` (`string`s). We define these types:

```
type blog = string list
type user = string
type pass = string
type message = Post of user * pass * string
             | Read of user * blog channel
type t = message channel
```

Clients communicate with the server using messages through a single channel:

- A user can publish a new post on her block by sending a `Post` message to the server. The message has to contain the user's name, password and text to be published. If username and password are correct, the server appends the text to the user's blog. Messages with incorrect credentials or non-existing users are simply ignored.

- To read a user's blog, a `Read` message with the corresponding user has to be sent to the server. Furthermore, the message has to contain a channel on which the server sends the requested blog or an empty list if no such user exists.

Implement these functions:

1. `start_server : (user * pass) list -> t` starts a server on its own thread. As an argument the function receives the list of registered users and their corresponding passwords.

2. `post : t -> user -> pass -> string -> unit` publishes a new blog post (last argument) in the given users blog.

3. `read : t -> user -> blog` requests a user's blog from the server.

Example:

```
let s = start_server [("userA", "passA"); ("userB", "passB")] in
post s "userB" "passB" "Welcome to my OCaml blog.";
post s "userA" "passA" "My name is A and I'm starting my own blog!";
post s "userB" "12345" "I am a hacker attacking B's blog now!";
post s "userB" "passB" "You can have threads in OCaml!";
read s "userB"
(* returns:
  ["Welcome to my OCaml blog."; "You can have threads in OCaml!"]
*)
```

Assignment 13.3 (L) How about the Future?

A *future* represents the result of an asynchronous computation. Imagine a time-consuming operation, that is relocated to another thread, then the main thread keeps some kind of "handle" to check whether the operation in the other thread has finished, to query the result or as a means to do other operations with the result. This "handle" is what we call a *future*.

Implement a module `Future` with a `type 'a t` that represents a future object. Furthermore, perform these tasks:

1. Implement `create : ('a -> 'b) -> 'a -> 'b t`, that applies the function given as the first argument to the data given as second argument in a separate thread. A future for the result of this operation is returned.

2. Implement `get : 'a t -> 'a` that waits for the asynchronous operation to finish and returns the result.

3. Implement `then_ : ('a -> 'b) -> 'a t -> 'b t` such that a call `then_ f x` returns a future that represents the result of applying `f` to the result of the computation referred to by `x`. The application of `f` must again be asynchronous, so `then_` must not block!

4. Extend your implementation with exception support, such that if an asynchronous operation throws, this exception is passed to the thread where the future resides and can be caught there.

5. Implement `when_any : 'a t list -> 'a t` that constructs a future that provides its result once any of the given futures has finished its computation. Make sure that `when_any` does not block!

6. Implement `when_all : 'a t list -> 'a list t` that constructs a future that corresponds to a list of all the results of the given futures.

7. Find additional useful functions for the `Future` module and implement them.

Hint: You may define other types and functions if you feel the need to do so.

Assignment 13.4 (H) Parallel Functions [4 Points]

1. Implement a function `par_unary` such that a call `par_unary f` with argument `f : 'a -> 'b` returns a new function `f' : 'a list -> 'b list` that applies the function `f` to a list of inputs in parallel (each input is processed in its own thread).

2. Implement a function `par_binary` such that a call `par_binary f` with argument `f : 'a -> 'b -> 'c` returns a new function `f' : 'a list -> 'b list -> 'c list` that applies the function `f` to lists of inputs in parallel (each input is processed in its own thread). Example:

```
let inc = par_unary (fun x -> x + 1) in
let add = par_binary (+) in
inc (add (inc [1;2;3]) [4;2;7])
(* computes: [7;6;12] *)
```

Assignment 13.5 (H) Arrays [6 Points]

Extend the idea of the *memory cell* introduced in the lecture to model *arrays*. Implement a module `Array` with a suitable type `t` and these functions:

1. `make : int -> 'a -> 'a t` that creates an array with given size. All elements are initialized with the given value.

2. `destroy : 'a t -> unit` that cleans up the array by stopping the array thread.

3. `size : 'a t -> int` returns the size of the array.

4. `set : int -> 'a -> 'a t -> unit` updates the entry at the given position with the given value. If the position is out of the array's bounds, throw an `OutOfBounds` exception.

5. `get : int -> 'a t -> 'a` retrieves the entry at the given position. If the position is out of the array's bounds, throw an `OutOfBounds` exception.

6. `resize : int -> 'a -> 'a t -> unit` resizes the array to the new given size. If the new size is less than the old size, the superfluous elements are chopped off, while the rest stays unchanged. If the new size is greater than the old size, the new elements are appended and initialized with the given value.

Do **not** use OCaml's arrays for your implementation.

Assignment 13.6 (H) Document Server [10 Points]

In this assignment, you are going to simulate a document server. The server is started on a separate thread and is reached through a corresponding channel. The server has to manage user accounts (name and password) and documents. Each document has a unique id, an owner and a list of viewers (users allowed to view the document in addition to the document's owner). Operations require the user to authenticate, if the user account does not exist or the password is incorrect, an `InvalidOperation` is thrown.

Implement:

1. A type `t` representing the channel to the server. Choose a type for the channel that fits your implementation's needs. You may define additional types.
2. Function `document_server : unit -> t` that starts the server thread.

Then implement these functions where the first two arguments are the *username* and *password* used for authentication:

3. `add_account : string -> string -> t -> unit` that creates a new account with given *username* and *password*. Throw `InvalidOperation` if the user does exist already.
4. `publish : string -> string -> string -> t -> int` where the arguments are *username*, *password*, *document*, *server* and the function returns the document's unique id. Remember to throw an `InvalidOperation` if the authentication is incorrect.
5. `view : string -> string -> int -> t -> string` returns the document with the given id or throws `InvalidOperation` if it does not exist or the user is not allowed to view the document.
6. `add_viewer : string -> string -> int -> string -> t -> unit` adds a new user (4. argument) to the list of users allowed to view the given document (3. argument). Throws `InvalidOperation` if no such document or user exists. Furthermore, ownly the document's owner may add viewers.
7. `change_owner : string -> string -> int -> string -> t -> unit` changes the owner of the given document (3. argument) to another user (4. argument). Again, exception `InvalidOperation` is thrown if the user or document does not exists or is not called by the owner.