Tutorial 9 and 10, CS2104

1 Executing Threads

Execute the following example with help of the abstract machine.

```
local A B C in
   thread if A then B=true else B=false end end
   thread if B then C=false else C=true end end
   A=false
end
```

Just sketch execution, the important point here is to understand how threads are created and executed.

2 Threads

Give the values for the variables after execution has terminated:

```
thread if X==1 then Y=2 else Z=2 end end
thread if Y==1 then X=1 else Z=2 end end
X=1
```

and also for:

```
thread if X==1 then Y=2 else Z=2 end end
thread if Y==1 then X=1 else Z=2 end end
X=2
```

Solution.

- `X=1`, `Y=2`, and `Z=2`.
- `X=2`, `Z=2`, and `Y` remains unassigned.

3 Guess What?

Consider the procedure definition:

```
proc {Guess X}
   if X==42 then skip else skip end
end
```

What does `Guess` do? How does

```
thread {Guess X} ⟨s⟩ end
```

execute? When is the statement ⟨s⟩ executed?

Solution. `{Guess X}` suspends the executing thread until `X` is bound. Hence, the statement ⟨s⟩ is executed only after `X` is bound.

4 A Browse Server

Write an agent that displays each received message in the Browser window.

Solution.

```
fun {NewAgent0 Proc}
   S P = {NewPort S}
in
   thread
      {ForAll S Proc}
   end
   P
end
proc {BrowseProcess M}
   {Browse M}
end
BrowseAgent = {NewAgent0 BrowseProcess}
```

5 A Filter Server

Write an agent that receives a message on a port, checks whether the message is okay, and if it is okay forwards it to another agent. Checking is defined by a unary function testing whether the message is to be forwarded or not.

Solution.

```
fun {NewFilterAgent F A}
   %% F is filter function
   %% A is receiving agent
   proc {FilterProcess M}
      if {F M} then {Send A M} end
   end
in
   {NewAgent FilterProcess}
end
```

6 Thread Control

Suppose the following scenario: at some point in your program you create a number of threads. You want to make sure that your program continues execution only if all threads created have terminated. How can you achieve this?

Solution. Each thread created binds as its last statement a dataflow variable. The program only continues execution if all variables used by the threads are bound.

```
fun {WaitThread P}
   thread {P} true end
end
proc {WaitAll Ps}
   %% Run all thread, collect variables in Ws
   Ws = {Map Ps WaitThread}
in
   {ForAll Ws Wait}
end
```

Here `Wait` is the `Guess` procedure from above and `ForAll` applies `Wait` to all elements of `Ws`. For example, in order to run statements $\langle s \rangle_1$ and $\langle s \rangle_2$ concurrently and wait until the execution of both has terminated, we write:

```
{WaitAll [proc {$} ⟨s⟩1 end
          proc {$} ⟨s⟩2 end]}
```

7 Agents With State

In the lecture we discussed agents without state (implemented by a unary procedure taking the message as argument). An agent with state can be implemented by a function that takes two arguments: the previous state and the received message. The function then returns the new state. Try to come up with an abstraction that creates (similar to `NewAgent` from the lecture) an agent with state. This abstraction also needs to take the initial state as an argument.
Try to model a bank accoung agent with this abstraction. The bank account agent should be able to handle withdrawals, deposits, and balance checks.

Solution. For the agent abstraction with state, please consult the lecture notes. A processing function for a bank account agent can be modeled as follows, where the state of the agent is the account's balance and the initial balance is `0`:

```
fun {BankProcess S M}
   case M
   of withdraw(N) then S-N
```

```
   [] deposit(N)  then S+N
   [] balance(N)  then N=S S
   end
end
```

8 `WaitOr` and `WaitSome`

One problem that occurs quite often in practice is to wait until at least one out of two variables becomes bound. For this purpose, Oz provides the procedure `{WaitOr X Y}`. It suspends until `X` or `Y` become bound.
Write a procedure `{WaitSome Xs}` that suspends the executing thread until at least one variable in `Xs` becomes bound.

Solution. The idea is to create a thread for each element of `Xs` that suspends until the element is bound. If it becomes bound, the thread binds a variable shared among all thread (here `Y`). Execution then continues as soon as `Y` is bound:

```
proc {WaitSome Xs}
   Y
in
   {ForAll Xs proc {$ X} thread {Wait X} Y=true  end end}
   {Wait Y}
end
```

9 Sending with Timeout

Implement a function `{SafeSend P M T}` that takes a port `P`, a message `M`, and a time value (in milliseconds) `T`. It sends a pair `M#Ack` to the port `P`. If the receiver acknowledges receipt by binding `Ack` within `T` milliseconds, the function returns `true`. Otherwise, the functions returns `false`.

Solution. We use the function `{IsDet X}` which returns `true`, if `X` is bound (determined) and `false` otherwise. In particular, `IsDet` does not suspend! We use it to check whether the acknowledgment has arrived.

```
fun {SafeSend P M T}
   Ack TimeOut
in
   {Send P M#Ack}
   thread {Delay T} TimeOut=true end
   {WaitOr Ack TimeOut}
   {IsDet Ack}
end
```

A different solution (by Ali Ghodsi) is to use ports instead of `IsDet`. The solution is as follows:

```
fun {SafeSend P M T}
   Ack Result
   Port Stream
in
   Port={NewPort Stream}
   {Send P M#Ack}
   thread {Delay T}   {Send Port false} end
   thread {Wait  Ack} {Send Port true}  end
   Stream.1
end
```

10 Implementing `WaitOr`

Do you have an idea how you could implement `WaitOr` yourself. Hint: You can do it with threads and an agent.

Solution. See above.