Summary of CPSC 418

Tom Wang

Fall, 2024

1 Erlang

1.1 Processes

The built-in function spawn creates a new process. The first argument is the module and function to run, and the second argument is a list of arguments to pass to the function. The function returns a process identifier, which can be used to send messages to the process.

Example:

```
hello(N) when is integer(N), N >= 0 ->
    [ spawn(fun() -> io:format(
                "hello world from process ~b~n", [I])
            end)
        || I <- lists:seq(1,N)
    ].
```

Here we used a list comprehension to spawn N processes, each of which prints a message to the console. The syntax `[Expr || I <- List]` creates a list by evaluating Expr for each element I in List.

1.2 Message Passing

1.2.1 Sending Messages

`Pid ! Expr` will evaluate Expr and send the result to the process with identifier Pid. The message can be any Erlang term.

1.2.2 Receiving Messages

`receive` is used to receive messages. It has the following syntax:

```
receive
    Pattern1 -> Expr1;
```

```
    Pattern2 -> Expr2;
    ...
end
```

The receive block will wait until a message is received that matches one of the patterns. The message is then bound to the variables in the pattern, and the corresponding expression is evaluated.

1.2.3 Communication

The communication is asynchronous, meaning that the sender does not wait for the receiver to process the message. This allows for parallel execution of processes.

If you need pid1 to wait for a response from pid2, you can use the following pattern:

```
pid2 ! {self(), Message},
receive
    {pid2, Response} -> Response
end
```

A start finish protocol can be implemented as follows:

```
pid2 ! {self(), start},
receive
    {pid2, ready} -> ok
end,
pid2 ! {self(), finish},
receive
    {pid2, done} -> ok
end
```

1.2.4 Timeouts

You can add a timeout to the receive block to prevent it from waiting indefinitely. The syntax is as follows:

```
receive
    Pattern1 -> Expr1;
    Pattern2 -> Expr2;
    ...
after
    Timeout -> Expr
end
```

If no message is received within the timeout period, the block will evaluate the expression after the after keyword.

1.3 Head and Tail Recursion

Erlang does not have loops. Instead, it uses recursion to iterate over lists. The two most common types of recursion are head recursion and tail recursion.

1.3.1 Head Recursion

In head recursion, the recursive call is the last operation in the function. This means that the function must wait for the recursive call to return before it can return.

Example:

```
sum([H|T]) -> H + sum(T);
sum([]) -> 0.
```

1.3.2 Tail Recursion

In tail recursion, the recursive call is the first operation in the function. This allows the function to return immediately after making the recursive call.

Example:

```
sum(List) -> sum(List, 0).
sum([H|T], Acc) -> sum(T, H + Acc);
sum([], Acc) -> Acc.
```

Notice that since we do not need to wait for the recursive call to return, we can pass the result of the recursive call directly to the next call. This allows the function to return immediately after making the recursive call. Thus it will not consume stack space.

2 Reduce and Scan

2.1 Reduce

The reduce operation is a common operation in parallel programming. It is used to combine a list of values into a single value. The operation is associative, meaning that the order in which the values are combined does not matter. The reduce operation is defined as follows:

$$\text{reduce}(f, [x_1, x_2, \ldots, x_n]) = f(f(\ldots f(f(x_1, x_2), x_3) \ldots, x_{n-1}), x_n)$$