CSci 2041
Advanced Programming Principles
L14: Expression Evaluation: Lazy Evaluation, Part 2

Eric Van Wyk

Fall 2014

Relation to coroutines and generators

- A coroutine is a procedure that runs concurrently with other coroutines.
- They can suspend execution, handing control over to another coroutine. like "lock"
- This allows two coroutines to interleave their execution, communicating back and forth between them.

Relation to coroutines and generators

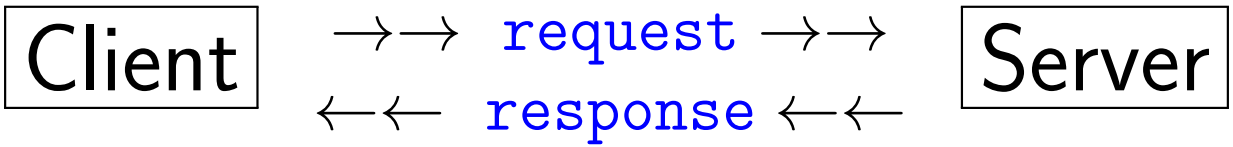
- For example, a client-server architecture in which
  - the client sends requests to a server,
  - which then sends back a response

- Python's `yield` statement and generators are examples of a restricted form of coroutines.

http://pyzh.readthedocs.org/en/latest/the-python-yield-keyword-explained.html

- We will see how both can be realized using lazy streams.

Client-Server architecture



- A sequence of requests is generated and sent from the client to the server.
- These requests are effected by previously received responses.
- For example, a client function `next_request` takes the last response from the server and generates the next request..
- The server generates a sequence of responses based on the requests that it gets.
- For example, a server function `next_response` takes the last request from the client and generates the next response to send to it.

Client server architecture using lazy streams

We can define a `client` and `server` as intertwined consumers and producers of streams of data.

A `client` consumes a stream of responses and produces a stream of requests.

A `server` consumes a stream of requests and produces a stream of responses.

Since these streams are lazily evaluated we need not create, for example, the entire stream of requests before the stream of responses can be produced.

The generation of them can be interleaved.

See implementation in `client_server.ml`.

Memory usage

- How much memory to these streams take?
  - They represent infinite lists, so perhaps a lot of memory is consumed.
- But our OCaml implementation is quite memory efficient.
  - After a response (or request) is consumed it is deallocated by the garbage collector. 用過就丟
  - The rest of the stream is essentially just a function waiting to generate more data.
    So it takes very little memory as well.

Generators in Python

A generator is a form of coroutine that responds to calls by returning values to the calling coroutine.

It is limited in that it can only yield control back to the calling coroutine, not to a different one.

Consider the use of Python's `yield` statement.

See example in `generators.py` in the code examples repository.

Simulating laziness in strict languages

Out lazy streams were an attempt to get some of the benefits of lazy evaluation of lists in a call by value/eager/strict language.

But how do we do this more generally?

Consider defining a `lazy` type constructor so that

- `int lazy` is a type for a lazily evaluated integer.
- `mk_lazy :: (unit -> 'a) -> 'a lazy` takes an expression and delays its evaluation
- `force :: 'a lazy -> 'a` causes the expression to be evaluated.

Can we do this in OCaml using what we've learned so far of the language?

To answer that question let's consider a possible implementation:

```
type 'a lazy = Unevaluated of (unit -> 'a)
let mk_lazy e = Unevaluated e
let force e = match e with
  | Unevaluated f -> f ()
```

But this doesn't save the evaluated value for use later.

```
let l1 = mk_lazy
     ( fun () ->
         (print_endline "A lazy computation" ;
          sqrt 16.0) )
let l2 () = force l1 +. force l1
```

As seen in

```
utop # l2 () ;;
A lazy computation
A lazy computation
- : float = 8.
```

We need side effects.

We need the first call to `force` to change the state of the lazy value from an unevaluated value to an evaluated value.

```
type 'a lazy = Unevaluated of (unit -> 'a)
             | Evaluated of 'a
```

And thus we need mutable references.

```
let l1 = ref (mk_lazy
     ( fun () ->
         (print_endline "A lazy computation" ;
          sqrt 16.0) ) )
```

We need a way to change the reference to a lazy value from an unevaluated expression to a value so that the second access of that reference finds the value, not an unevaluated expression.

Effects

- OCaml provides a safe form of references that can be changed.
- In using this feature we are stepping outside of the pure functional programming model.
- A pure functional language is one without side effects (assignments).

References in OCaml

- type constructor: `ref`
  e.g. `int ref` is a reference (safe pointer) to an `int` `value`

- value construction: `ref`
  e.g. `let sq = ref 4`

- access: `!`
  e.g. `!sq` evaluates to `4`

- assignment: `:=`
  e.g. `sq := !sq + 5`

We will return to OCaml references and side-effects in general after we've discussed one more form of expression evaluation: parallel evaluation.