CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

Lenguajes de Programación

▸ Implementing Lazy Evaluation

Federico Olmedo
Ismael Figueroa

Evaluation strategies of programming languages

Evaluation strategies determine *when* to evaluate the argument of a function call.

- Eager (or strict) evaluation:

  Function arguments are evaluated before proceeding to evaluate the function body.

  (The formal argument of the function is bound to the value of the actual argument.)

- Lazy (or non-strict) evaluation:

  Function arguments are evaluated only if they are required during the evaluation of the function body.

Eager Evaluation

Expression	Environment
`{{fun {x} {{fun {y} {+ y 2}} {+ x 1}}} {+ 4 5}}`	`[]`
`{{fun {y} {+ y 2}} {+ x 1}}`	`[x->9]`
`{+ y 2}`	`[x->9, y->10]`
`12`	

Arguments are interpreted first, and then added to the environment.

Lazy Evaluation

Expression	Environment
`{{fun {x} {{fun {y} {+ y 2}} {+ x 1}}} {+ 4 5}}`	`[]`
`{{fun {y} {+ y 2}} {+ x 1}}}`	`[x->"{+ 4 5}"]`
`{+ y 2}`	`[x->"{+ 4 5}", y->"{+ x 1}"]`
`???`	

Argument evaluation is deferred, until they are "needed".
But we must still keep static scoping.

```
[x->"{+ 4 5}",
 y->"{+ x 1}" where x->"{+ 4 5}"]
```

Show a program to verify whether this works

Lazy Evaluation

```
{+ y 2}
```

```
[x->"{+ 4 5}",
 y->"{+ x 1}" where x->"{+ 4 5}"]
```

PAST PROBLEM: The evaluation of functions is deferred until their application.
To handle this, we close them over the bindings (i.e. environment) at their definition location.

CURRENT PROBLEM: Now we have to defer the evaluation of arbitrary expressions (actual params.) We handle it the same way, by closing actual parameters over their respective environments (to be able to reduce them, when actually needed)

∴ Environment will now map the formal parameter of a function to the the actual parameter, *plus its environment* (known as the *closure* of actual parameter)

An interpreter for a lazy version of our language

```
;; Values of expressions
(deftype Value
  (numV n)
  (closureV id body env)
  (exprV expr env))
```

An **expression closure** defers the evaluation of an arbitrary expression, bound to the environment at the time of definition

An interpreter for a lazy version of our language

```
;; Values of expressions
(deftype Value
  (numV n)
  (closureV id body env)
  (exprV expr env))
```

```
;; interp :: Expr Env -> Value
;; Evaluates an expression in a give environment,
;; using static scoping and lazy evaluation.
(define (interp expr env)
  (match expr
```

Before, with eager evaluation:

```
(interp e env)
```

An interpreter for a lazy version of our language

```
;; Values of expressions
(deftype Value
  (numV n)
  (closureV id body env)
  (exprV expr env))
```

```
[(app f e) (def (closureV the-arg the-body closed-env) (interp f env))
           (def new-env (extend-env the-arg (exprV e env) closed-env))
           (interp the-body new-env)]))
```

But sometimes this doesn’t work as expected…

```
> (run '((fun (x) x) 3))
(exprV (num 3) (mtEnv))

> (run '((fun (x) (+ x x)) 3))
def: no matching clause for (exprV (num 3) (mtEnv))

> (run '((fun (x) x) (fun (y) y)))
(exprV (fun 'y (id 'y)) (mtEnv))

> (run '(((fun (x) x) (fun (y) y)) 3))
def: no matching clause for (exprV (fun 'y (id 'y)) (mtEnv))
```

An interpreter for a lazy version of our language

```
;; Values of expressions
(deftype Value
  (numV n)
  (closureV id body env)
  (exprV expr env))
```

```
;; interp :: Expr Env -> Value
;; Evaluates an expression in a give environment,
;; using static scoping and lazy evaluation.
(define (interp expr env)
  (match expr
    [(num n) (numV n)]
    [(fun id body) (closureV id body env)]
    [(id x) (env-lookup x env)]
    [(add l r) (num+ (strict (interp l env)) (strict (interp r env)))]
    [(sub l r) (num- (strict (interp l env)) (strict (interp r env)))]
    [(if0 c t f) (if (num-zero? (strict (interp c env)))
                     (interp t env)
                     (interp f env))]

    [(app f e)
     (def (closureV the-arg the-body closed-env) (strict (interp f env)))
     (def new-env (extend-env the-arg (exprV e env) closed-env))
     (interp the-body new-env)]))
```

These are the *strictness points* of the language, where we *force* reduction of expression closures

An interpreter for a lazy version of our language

```
;; Values of expressions
(deftype Value
  (numV n)
  (closureV id body env)
  (exprV expr env))
```

```
;; strict :: Value -> Value [without exprV]
;; Further reduces a Value to a numV or closureV
(define (strict v)
  (match v
    [(exprV expr env) (strict (interp expr env))]
    [_ v]))
```

The *strict* function recursively forces evaluation of expression closures, until a value is returned.

The lazy evaluator turned out not to be so efficient…

PROBLEM: With previous interpreter, function arguments are evaluated only if needed, but if needed multiple times, they are evaluated multiple times. E.g. in

```
{{fun {x} {+ x x}} {+ 4 5}}
```

argument `{+ 4 5}` is evaluated twice.

⚠ **This kind of lazy evaluation is known technically as** <u>call by name</u>

But, we can fix it!

SOLUTION: Cache the value of actual argument when reduced for the first time (and directly retrieve its value in any subsequent use)

- We cache the value in the expression closure

```
;; Values of expressions
(deftype Value
  (numV n)
  (closureV id body env)
  (exprV expr env cache))
```

- For caching we use Racket boxes (mutable vectors of size 1)

```
(box 5)                          ⇝ '#&5

(unbox (box 5))                  ⇝ 5

                                 ⇝ 6
(let ([b (box 5)])
(set-box! b 6)
(unbox b))
```

Interpreter for an efficient lazy language

```
;; interp :: Expr Env -> Value
;; Evaluates an expression in a give environment,
;; using static scoping and lazy evaluation.
(define (interp expr env)
  (match expr
    [(num n) (numV n)]
    [(fun id body) (closureV id body env)]
    [(id x) (env-lookup x env)]
    [(add l r) (num+ (strict (interp l env)) (strict (interp r env)))]
    [(sub l r) (num- (strict (interp l env)) (strict (interp r env)))]
    [(if0 c t f) (if (num-zero? (strict (interp c env)))
                     (interp t env)
                     (interp f env))]

    [(app f e)
     (def (closureV the-arg the-body closed-env) (strict (interp f env)))
     (def new-env (extend-env the-arg (exprV e env (box #f)) closed-env))
     (interp the-body new-env)]))
```

Expression closures are initialized with a dummy cache value.

Interpreter for an efficient lazy language

```
;; strict :: Value -> Value [without exprV]
;; Further reduces a Value to a numV or closureV
(define (strict v)
  (match v
    [(exprV expr env cache)




     [_ v]))
```

This kind of lazy evaluation is known technically as call by need

Evaluation strategies of programming languages

- Eager (or strict) evaluation:

  Function arguments are evaluated before proceeding to evaluate the function body. Also known as

[Scheme, C, Java] **Call by value**

since formal arguments are bound to the value of actual arguments

- Lazy (or non-strict) evaluation:

  Function arguments are evaluated only if they are required during the evaluation of the function body. There are two variants:

[ALGOL 60]	**Call by name:**	Function arguments are evaluated every time they are required [Interpreter Slide 9].
[Haskell, R]	**Call by need:**	Function arguments are evaluated only once (upon the first time they are required), and cached for further use [Interpreter Slide 13].

Takeaways

Implementing lazy evaluation requires binding the formal arguments of functions to the *closure* of their actual arguments

Lazy evaluation must be implemented with caution to avoid repeated evaluation of functions' actual arguments [call-by-need]

Lecture material

Bibliography

- Programming Languages: Application and Interpretation (1st Edition)
  Shriram Krishnamurthi [Download]
  Chapter 8

Source code

- Lazy evaluation, call by name [Download]
- Lazy evaluation, call by need [Download]