CS450

Structure of Higher Level Languages

Lecture 14: Thunks and promises

Tiago Cogumbreiro

Today we will learn...

1. Learn about delayed evaluation
2. Promises and their implementation
3. Streams of data

> Acknowledgment: Today's lecture is inspired by Professor Dan Grossman's wonderful lecture in CSE341 from the University of Washington: (Video 1) (Video 2) (Video 3) (Video 4) (Video 5)

Module 2: recap

- Learned advanced Functional Programming principles
- Developed a library of function combinators
- The effect of tail-call optimization and how to use them
- Refactored code to reduce code redundancy
- Refactored code to improve runtime performance

Module 3:
Lazy evaluation

Module 3

Lazy evaluation

- Using functions to delay computation
- Lazy evaluation as a form of controlling execution
- Lazy evaluations as data-structures
- Functional patterns applied to delayed

Delayed evaluation

Recall the evaluation order

Function application

The evaluation of function application can be called ***eager***

- Evaluating a function application, first evaluates each argument before evaluating the body of the function.

Condition

The evaluation of `cond` can be called ***lazy***, in the sense that a branch of `cond` is only evaluated when its guard yields true (and only the one branch is evaluated).

How to encode an `if-then-else`?

```
(define (factorial n)
  (cond [(= n 0) 1]
        [else (* n (factorial (- n 1)))]))
```

Example

```
(define (if b then-branch else-branch)
  (cond [b then-branch] [else else-branch]))

(define (factorial n)
  (if (= n 0) 1 (* n (factorial (- n 1)))))

(factorial 10)
```

> What is wrong with this implementation?

How to encode an `if-then-else`?

```
(define (factorial n)
  (cond [(= n 0) 1]
        [else (* n (factorial (- n 1)))]))
```

Example

```
(define (if b then-branch else-branch)
  (cond [b then-branch] [else else-branch]))

(define (factorial n)
  (if (= n 0) 1 (* n (factorial (- n 1)))))

(factorial 10)
```

> What is wrong with this implementation? Why `(factorial 10)` does not terminate?

Our implementation of `if` is too eager

Because our `if` is a function, applying evaluates the `then-branch` and the `else-branch` before choosing what to return.

Which, means our factorial no longer has a base case, and, therefore, it does not terminate.

```
= (factorial 0)
= (if (= 0 0) 1 (* 0 (factorial (- 0 1))))
= (if #t 1 (* 0 (factorial (- 0 1))))
= (if #t 1 (* 0 (factorial -1)))
= (if #t 1 (* 0 (if (= 0 -1) (= 0 -1) (* -1 (factorial (- -1 1))))))
= ...
```

> Any idea how we can work around this limitation?

Using `lambda`s to delay computation

UMASS BOSTON

We can use a zero-argument lambda to hold each branch, as a `lambda` delays computation!

```
(define (if b then-branch else-branch)
   (cond [b (then-branch)] [else (else-branch)]))

(define (factorial n)
  (if (= n 0) (lambda () 1) (lambda () (* n (factorial (- n 1))))))

(factorial 10)
```

Thunks: zero-argument functions

The pattern of using zero-argument functions to delay evaluation is called a ***thunk***. You can use thunk as a verb which is a synonym of delaying evaluation.

- `(lambda () e)` delays expression `e`
- `(e)` evaluates thunk `e` and calls that thunk

Using thunk

Racket offers `(thunk e)` as a short-hand notation for `(lambda () e)`; both notations are equivalent.

```
(define (if b then-branch else-branch)
   (cond [b (then-branch)] [else (else-branch)]))

(define (factorial n)
  (if (= n 0) (thunk 1) (thunk (* n (factorial (- n 1))))))

(factorial 10)
```

Functional patterns: promises

Repeated delayed computation

> In functional programming, there are cases where you have an intertwined pipeline of functions where a thunk might be carried around. Since, we aim at side-effect free programming models, it is wasteful to compute a thunk multiple times, when at most one would do.

Example

```
(define (runner count thunk call-back)
  (cond [(≤ count 0) (call-back (thunk) thunk)] ; invokes thunk once, and passes it along
        [else (call-back count thunk)]))        ; does not invoke thunk once
```

> It might not possible to know, at the function-level, if `thunk` was already called, as it depends on the caller and, in this case, on `call-back` as well.

Promises: memoize delayed computation

- `(delay e)` delays the evaluation of an expression (yielding a thunk)
- `(force e)` caches the result of evaluating `e`, so that multiple applications of that thunk return the result.

Did you know?

- Memoization: optimization technique that caches the result of an expensive function and returns the cached result
- Haskell does not share the same evaluation model as we have in Racket. Instead, ***all*** expressions of the language are lazily evaluate.
- The idea of memoized delayed evaluation provides an elegant way to parallelize code. The concept is usually known as a ***future***.
- The idea of memoized delayed evaluation (promises) is also very important in asynchronous code (networking, and GUI), eg in JavaScript, in Python

Example: delay/force

Thunks

```
(define (thunk-repeat n th)
  (cond [(≤ n 0) (void)]
        [else
          (th)
          (thunk-repeat (- n 1) th)]))

(thunk-repeat 3 (thunk (sleep 1) 3))
```

Promises

```
(define (promise-repeat n prom)
  (cond [(≤ n 0) (void)]
        [else
          (force prom)
          (promise-repeat (- n 1) prom) ]))

(promise-repeat 3 (delay (sleep 1) 3))
```

Promises versus thunks

UMASS BOSTON

Accessor

- Promises: must call function `force`
- Thunks: call the object itself

Evaluation count

- (`force p`) evaluates the promise at most **once**; subsequent calls are cached
- (`thnk`) calling a thunk evaluates its contents **each and every time**

Implementing promises

Implementing promises: state

> Promises are usually implemented with mutable references. Can we get away with implementing promises without using mutation?

A promise has two states:

1. when the thunk has not been run yet
2. when the thunk has been run at least once

A promise must hold:

- the thunk we want to cache
- the empty/full status

We need to separate the operations that mutate the state, from the ones that query the state.

Implementing promises: operations

Function (`force c`) can be though of a few smaller operations:

1. checking if the promise is empty
2. if the promise is empty, update the promise state to full and store the result of the thunk
3. if the promise is full, does nothing to the promise state, and returns the cached result

Let us separate the operations that change the state from the one that return the value.

- Function (`promise-sync p`) returns a new promise state. When the promise is empty, it computes the thunk and stores it in a full promise. When the promise is full, it just returns the promise given.
- Function (`promise-get p`) can only be called when the promise is full and returns the result of the promise.

Immutable promise implementation

```
(struct promise (empty? result))
(define (make-promise thunk) (promise #t thunk))
(define (promise-run w)
  (define th (promise-result w))
  (th))
(define (promise-get p)
  (cond [(promise-empty? p) (error "promise: call (promise-sync p) first.")]
        [else (promise-result p)]))
(define (promise-sync p)
  (cond [(not (promise-empty? p)) p]
        [else (promise #t (promise-run p))]))
```

Example of immutable promises

Immutable Promises

```
(define (promise-repeat n prom)
  (cond [(≤ n 0) (void)]
        [else
          (promise-repeat (- n 1) (promise-sync prom))]))
(promise-repeat 3 (make-promise (thunk (sleep 1) 3)))
```

Standard promises

```
(define (promise-repeat n prom)
  (cond [(≤ n 0) (void)]
        [else
          (force prom)
          (promise-repeat (- n 1) prom) ]))

(promise-repeat 3 (delay (sleep 1) 3))
```