CS450

Structure of Higher Level Languages

Lecture 10: TypedRacket, thunks, and promises

Tiago Cogumbreiro

Benchmark evaluation

- Unoptimized `foldr`
- Tail-recursive `foldr`

```
Processing a list of size: 1000000
-------------------------------------------------------
Throughoutput (unopt): 7310 elems/ms
Mean (unopt): 136.8±7.56ms
-------------------------------------------------------
Throughoutput (tailrec): 12349 elems/ms
Mean (tailrec): 80.98±1.49ms
Speed-up (tailrec): 1.7
```

A speed improvement of 1.7

What if we use foldl + reverse?

What if we use foldl + reverse?

- Instead of creating nested functions,
- We reverse the list and apply foldl

```
(define (foldr step base-case l)
  (foldl step base-case (reverse l)))
```

What if we use foldl + reverse?

- Instead of creating nested functions,
- We reverse the list and apply foldl

```
(define (foldr step base-case l)
  (foldl step base-case (reverse l)))
```

Simpler implementation!

But is it faster?

Rev+fold runs the slower (0.7)

```
Processing a list of size: 1000000
-------------------------------------------------------
Throughoutput (unopt): 7310 elems/ms
Mean (unopt): 136.8±7.56ms
-------------------------------------------------------
Throughoutput (tailrec): 12349 elems/ms
Mean (tailrec): 80.98±1.49ms
Speed-up (tailrec): 1.7
-------------------------------------------------------
Throughoutput (rev+foldl): 4846 elems/ms
Mean (rev+foldl): 206.34±3.33ms
Speed-up (rev+foldl): 0.7
```

Conclusion

We learned to generalize two reduction patterns (foldl and foldr)

- **Pro:** generalizing code can lead to a central point to optimize code
- **Pro:** generalizing code can reduce our code base
  (less code means less code to maintain)
- **Con:** one level of indirection increases the cognitive code
  (more cognitive load, code harder to understand)

Easier to understand (self-contained)

```racket
(define (reverse l)
  (define (rev accum l)
    (match l
      [(list) accum]
      [(list h l ...) (rev (cons h accum) l)]))
  (rev l (list)))
```

Harder to understand (what is foldl?)

```racket
(define (reverse l)
  (define (on-elem elem accum)
    (cons elem accum))
  (foldl on-elem (list) l))
```

Module 4:

Lazy evaluation

Module 4

Lazy evaluation

- TypedRacket: typing annotations on top of Racket
- Using functions to delay computation
- Lazy evaluation as a form of controlling execution
- Lazy evaluations as data-structures
- Functional patterns applied to delayed

TypedRacket

Typing a factorial example

```
1. #lang typed/racket
2. (: factorial (-> Number Number))
3. (define (factorial n)
4.   (cond [(= n 0) 1]
5.         [else (* n (factorial (- n 1)))]))
```

- ***Line 1.*** Specifies that we are using TypedRacket with `typed/racket`
- ***Line 2.* Function signature:** `(: name-of-function (-> TypeOfArg1 TypeOfArg2 ReturnType))`
- **Running a script is the same as with Racket.** Run `racket script.rkt` as usual!
- Run `racket -I typed/racket` to start the TypedRacket REPL.
- Write `#lang typed/racket/no-check` to disable type-checking (not advised).

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`?

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

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

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

Example

```
(: if (All [T] (-> Boolean T T T)))
(define (if b then-branch else-branch)
  (cond [b then-branch] [else else-branch]))

(: factorial (-> Number Number))
(define (factorial n)
  (if (= n 0) 1 (* n (factorial (- n 1)))))

(factorial 10)
```

The type `(All [T] (-> Boolean T T T))` specifies a **type-parameter**, akin to a generic of Java or a template parameter of C++

> What is wrong with this implementation?

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

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

Example

```
(: if (All [T] (-> Boolean T T T)))
(define (if b then-branch else-branch)
  (cond [b then-branch] [else else-branch]))

(: factorial (-> Number Number))
(define (factorial n)
  (if (= n 0) 1 (* n (factorial (- n 1)))))

(factorial 10)
```

The type `(All [T] (-> Boolean T T T))` specifies a **type-parameter**, akin to a generic of Java or a template parameter of C++

> 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 lambdas to delay computation

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

```
(: if (All [T] (-> Boolean (-> T) (-> T) T)))
(define (if b then-branch else-branch)
   (cond [b (then-branch)] [else (else-branch)]))
(: factorial (-> Number Number))
(define (factorial n)
  (if (= n 0) (lambda () 1) (lambda () (* n (factorial (- n 1))))))

(factorial 10)
```

- Now each parameter is of type `(-> T)`, a function that takes 0 arguments and returns a value of type `T`

Typechecking errors

```
foo.rkt:8:2: Type Checker: Polymorphic function `if'
             could not be applied to arguments:
Argument 1:
  Expected: Boolean
  Given:    Boolean
Argument 2:
  Expected: (-> T)
  Given:    One
Argument 3:
  Expected: (-> T)
  Given:    (-> Number)

Result type:     T
Expected result: Number

in:(if (= n 0) 1 (lambda () (* n (factorial (- n 1))
```

#	*Value*	*Expected*	*Given*
1.	`(= n 0)`	`Boolean`	`Boolean`
2.	`1`	`(-> T)`	`One`
3.	`(lambda)`	`(-> T)`	`(-> Number)`
	return	`T`	`Number`

- Ignore row when same type (Arg 1)
- Ignore when type param `T` vs `Number`
- Only interesting is `Argument 2`, `(-&gt; T)` vs `One`, which is the type of `1`

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.

```
(: if (All [T] (-> Boolean (-> T) (-> T) T)))
(define (if b then-branch else-branch)
  (cond [b (then-branch)] [else (else-branch)]))

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

(factorial 10)
```

Thunks for delayed evaluation

- You can use `(lambda () some-computation)` to delay the evaluation of `some-computation`.
- To initiate `some-computation` you need to run the lambda.
- You can use `(thunk some-computation)` as a short-hand notation.
- Using a lambda to delay computation also lets you run the same computation multiple times.

```
(: f (-> Number))
; (define (f) (displayln "Called (f)") 0) ; <- equivalent
(define f
  (thunk
    (displayln "Called (f)")
    0
  )
)
(f)   ; Outputs: Called (f)
(f)   ; Outputs: Called (f)
```

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

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

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

Promises

```racket
(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

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))
```