CS450

Structure of Higher Level Languages

Lecture 27: Semester Wrap Up & Parallelism

Tiago Cogumbreiro

Today we will...

1. Review what we learned in CS450
2. Remind anonymous feedback
3. Answer Homework Assignment Questions
4. Learn about functional parallelism with futures
5. Fill-in the Course Evaluation Form

My goal with CS450 was to teach you ...

1. Fundamental concepts behind most programming languages

- functional programming, delayed evaluation, control flow and exceptions, object oriented systems, monads, macros, pattern matching, variable scoping, immutable data structures

2. A framework to describe language concepts

- λ-calculus and formal systems to specify programming languages
- functional programming and monads to implement specifications

3. A methodology to understand complex systems

- (formally) specify and implement each programming language feature separately
- understand a complex system as a combination of smaller simpler systems
- implement and test features independently

Piazza review

- 30 students online on average
- 224 questions asked; 100% questions were answered; instructor answered 88% (198 questions)
- 1562 contributions (posts, edits, responses, follow ups, comments); instructor made 40% (668 contributions)

Maximum student's

- contributions made: 123
- questions asked: 36
- posts viewed: 225
- days online: 97

What I would like to improve in CS450

1. **How to test code and ask questions?**
   - Hardly anyone shared tests in the forum. Should tests be an exercise?
   - Questions are incomplete (lack stack traces, are incomplete). Should I teach how to ask questions?
2. **Tests do *not* prove correctness!**
   - If a solution breaks because of a new test, this solution was **incomplete**!
   - The autograder is your friend and so are tests.
3. **Did the students really learn the assignments, or just passed learned to pass the tests?**
   - Add design document per homework assignment?
   - Should we have a midterm?
4. **Should we improve homework text?**
   - Succinct homework assignments to motivate participation, yet not everyone happy.
   - Should we add labs to support homework assignments?

Anonymous Feedback

Open-ended suggestions to your instructor

(all optional and anonymous)

tinyurl.com/cs450-feedback

Or, email me: Tiago.Cogumbreiro@umb.edu

Homework assignment questions

HW7 question

> Thread 337: What is the major difference between an eff and an eff-op?

HW7 question

> Thread 337: What is the major difference between an eff and an eff-op?

Answer

Let us look at `hw7-util.rkt`:

```
(struct eff (state result) #:transparent)
(struct eff-op (func))
```

- `eff` is the return of effectful operations (introduced in Lecture 13; revisited in Lectures 15 and 17)
- `eff-op` a structure that holds an effectful operation, takes a *state* (eg, a `heap`) and produces an `eff` (introduced in Lecture 18, slide 29)

Examples of effectful operations `eff-op`: `eff-bind`, `eff-pure`, `env-put`, `env-get`, `env-push`

HW7 question

> Thread 342: How do I test for if? How do I know if the term is curried?

$$\frac{e_c \Downarrow_E \texttt{\#f} \quad \blacktriangleright \quad e_f \Downarrow v_f}{(((\texttt{if}\ e_c)\ e_t)\ e_f) \Downarrow_E v_f}\ \texttt{(E-if-f)} \qquad \frac{e_c \Downarrow_E v \qquad v \neq \texttt{\#f} \quad \blacktriangleright \quad e_t \Downarrow v_t}{(((\texttt{if}\ e_c)\ e_t)\ e_f) \Downarrow_E v_t}\ \texttt{(E-if-t)}$$

HW7 question

> Thread 342: How do I test for if? How do I know if the term is curried?

$$\frac{e_c \Downarrow_E \texttt{\#f} \quad \blacktriangleright \quad e_f \Downarrow v_f}{(((\texttt{if}\ e_c)\ e_t)\ e_f) \Downarrow_E v_f}\ \texttt{(E-if-f)} \qquad \frac{e_c \Downarrow_E v \qquad v \neq \texttt{\#f} \quad \blacktriangleright \quad e_t \Downarrow v_t}{(((\texttt{if}\ e_c)\ e_t)\ e_f) \Downarrow_E v_t}\ \texttt{(E-if-t)}$$

Answer

1. Use pattern matching with nested a pattern before the branch for apply. See Thread 300.
2. Terms being evaluated are **always** curried.

HW8 question

Thread 334: What does $\lambda(\mathsf{this}, \mathsf{x} \cdots).[\![e]\!]$ mean?

$$\mathtt{J}[\![\mathtt{function}(x \cdots)\ \{e\}]\!] =$$
$$\mathsf{alloc}\ \{\texttt{"\$code"} : \lambda(\mathsf{this}, \mathsf{x} \cdots).\mathtt{J}[\![e]\!], \texttt{"prototype"} : \mathsf{alloc}\ \{\}\}$$

HW8 question

> Thread 334: What does $\lambda(\mathsf{this}, \mathsf{x} \cdots).[\![e]\!]$ mean?

$$\mathtt{J}[\![\mathtt{function}(x \cdots)\ \{e\}]\!] =$$
$$\mathsf{alloc}\ \{\texttt{"\$code"} : \lambda(\mathsf{this}, \mathsf{x} \cdots).\mathtt{J}[\![e]\!], \texttt{"prototype"} : \mathsf{alloc}\ \{\}\}$$

Answer

Generate a lambda, whose

1. *parameters* are $\mathsf{this}, \mathsf{x} \cdots$, so translate the original parameters $x, \cdots$ and add a variable `this`
2. *body* is the translation of e

HW8 question

> Thread 338: What is `js-set!`? (Lecture 26, slide 7)

```
(let ([js-set!
  (lambda (o f d)
    (begin (set! o (update-field (deref o) f d)) d
  (alloc (object
      ["$code"
        (lambda (this x y)
          (begin (js-set! this "x" x)
                 (js-set! this "y" y)))]
      ["prototype" (alloc (object))])))
```

HW8 question

> Thread 338: What is `js-set!`? (Lecture 26, slide 7)

```
(let ([js-set!
  (lambda (o f d)
    (begin (set! o (update-field (deref o) f d)) d
  (alloc (object
      ["$code"
        (lambda (this x y)
          (begin (js-set! this "x" x)
                 (js-set! this "y" y)))]
      ["prototype" (alloc (object))])))
```

Answer

- The generated code did not fit the slide, think of it as the translation of `(set! o.f a)`. I have highlighted in yellow the code being generated.

HW8 question

> Thread 343: What is the difference between `$proto` and `prototype`?

HW8 question

> Thread 343: What is the difference between `$proto` and `prototype`?

Answer

See Lectures 24 and 25.

1. `$proto` is a field used for looking up the super object (the parent); works on any object. In JavaScript this is `__proto__`, in LambdaJS this is `$proto`.
2. `prototype` is the field of every `function`, used by `new` to initialize the `$proto` field of created objects

```
function A () {this.a = 1;}
A.prototype = {"__proto__": {"b": 10, "c": 10, "a": 10}, "b": 20}
a = new A; // {a: 1, *b: 20, *c: 10}
```

Functional parallelism

Parallelism with asynchronous evaluation

UMASS BOSTON

> The idea is similar to `delay/force`

1. `(future t)` evaluates a thunk `t` in another task, possibly by another processor
2. Calling `(future t)` returns a *future value* `f`, a place holder to a parallel computation
3. One can await the termination of the parallel task with `(touch f)`, which blocks the current task until the task evaluating the future thunk terminates. Consecutive `(touch f)` are nonblocking.

```
(define f (thunk (sleep 2) 99)) ;; Spawns a task T1
(assert-equals? (touch f) 99)   ;; Blocks until T1 terminates and returns 99
(touch f)                       ;; We know that T1 has terminated
```

A parallel fold

```
(define (par-reduce f init v lo hi)
  (if (< (- lo hi) threshold)
    ;; Base case, call sequential version
    (foldl f init (vector-view v lo hi))
    ;; Otherwise, divide array into two and spawn another task
    (let* ([mid (floor (+ (/ lo 2) (/ hi 2)))]
           [l (future (thunk (par-reduce f init v lo mid)))]
           [r (par-reduce f init v mid hi)])
      (f (touch l) r))))
```

Map-reduce example

```
(f
  (f
    (foldl f 0 [ 0  ...   64])  ; Task 1
    (foldl f 0 [64  ...  128])) ; Task 2
  (foldl f 0 [ 128  ...  192])) ; Task 3
```

Example of parallel reduce

```
(define (f x y)
  (/ (- (+ (- (* x 2) y y 25) x y 56) x 36) 2))
(define (do-par l)
  (par-reduce f 0 (list→vector l)))
(define (do-seq l)
  (foldl f 0 l))
```

Example of parallel reduce

```
(define (f x y)
  (/ (- (+ (- (* x 2) y y 25) x y 56) x 36) 2))
(define (do-par l)
  (par-reduce f 0 (list→vector l)))
(define (do-seq l)
  (foldl f 0 l))
```

Output

```
Processing a list of size: 10000

 * Serial version *
Throughoutput: 25 elems/ms
Mean: 402.03±9.89ms

 * Parallel version *
Throughoutput: 25 elems/ms
Mean: 392.76±13.2ms
```

Parallelism in Racket

Let us try Clojure!

Parallel reduce

```clojure
(defn do-reduce [f l treshold]
  (proxy [RecursiveTask] []
    (compute []
      (if (⩽ (count l) treshold)
        ;; if the vector is small enough,
        ;; we just reduce over them
        (reduce f 0 l)
        ;; otherwise, we split the vector roughly in two
        ;; and recursively run two more tasks
        (let [half (quot (count l) 2)
              f1 (do-reduce f (subvec l 0 half) treshold)
              f2 (do-reduce f (subvec l half) treshold)]
          ;; do half the work in a new thread
          (.fork f2)
          ;; do the other half in this thread and combine
          (f (.compute f1) (.join f2)))))))
```

Demo

- Clojure 1.10
- OpenJDK 1.8.0_191
- Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
- 4 cores
- list with 1,000,000 elements

Demo

- Clojure 1.10
- OpenJDK 1.8.0_191
- Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
- 4 cores
- list with 1,000,000 elements

```
Serial version
"Elapsed time: 2769.94558 msecs"
Parallel version
"Elapsed time: 755.341055 msecs"
```

3.7× Increase!

Demo 2

> Let us vary the parameter being used...

Demo 2

> Let us vary the parameter being used...

```
Serial version
"Elapsed time: 101.96357 msecs"
Parallel version
"Elapsed time: 219.819163 msecs"
```

2.0× slower!

Parallel overhead is significant!

Demo 3

- Let us vary the size of the data being used: **100,000 elements** rather than 1,000,000

Demo 3

UMASS BOSTON

Let us vary the size of the data being used: **100,000 elements** rather than 1,000,000

```
Serial version
"Elapsed time: 179.724932 msecs"
Parallel version
"Elapsed time: 182.837934 msecs"
```

Data size is also significant!

Thank you!