

# Parallel Computations on Immutable Data Structures

Florian Biermann  
fbie@itu.dk

IT University of Copenhagen & UCAS

2016-06-02



IT University  
of Copenhagen

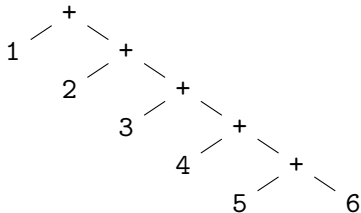
# A Sequential Expression

(+ 1 (+ 2 (+ 3 (+ 4 (+ 5 6)))))

## A Sequential Expression

```
(+ 1 (+ 2 (+ 3 (+ 4 (+ 5 6)))))
```

The expression tree:



# What Is The Problem?

All operations are performed sequentially, even though it would be perfectly ok to compute

`(+ 1 (+ 2 3))`

and

`(+ 4 (+ 5 6))`

in parallel. The result would be the same.

# What Is The Problem?

All operations are performed sequentially, even though it would be perfectly ok to compute

`(+ 1 (+ 2 3))`

and

`(+ 4 (+ 5 6))`

in parallel. The result would be the same.

We can re-formulate the expression and obtain the same result!

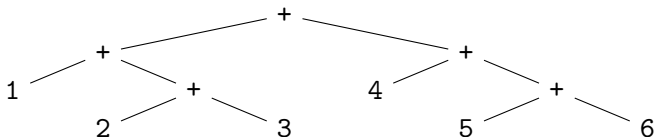
# A Parallel Expression

`(+ (+ 1 (+ 2 3)) (+ 4 (+ 5 6)))`

# A Parallel Expression

$(+ (+ 1 (+ 2 3)) (+ 4 (+ 5 6)))$

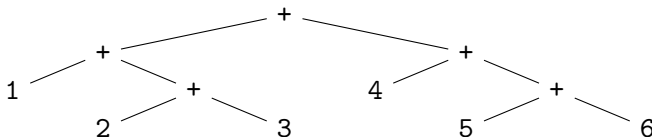
The expression tree:



# A Parallel Expression

$(+ (+ 1 (+ 2 3)) (+ 4 (+ 5 6)))$

The expression tree:



Why is this possible? Because + is an associative operation!

$$(+ \ x \ (+ \ y \ z)) = (+ \ (+ \ x \ y) \ z)$$



# Generalizing Parallel Expressions

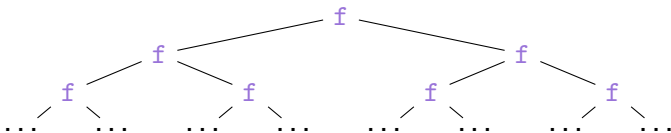
We can do this for any function  $f$  iff

$$(f\ a\ (f\ b\ c)) = (f\ (f\ a\ b)\ c)$$

# Generalizing Parallel Expressions

We can do this for any function  $f$  iff

$$(f\ a\ (f\ b\ c)) = (f\ (f\ a\ b)\ c)$$



# Another Problem

Mapping a function `f` over a `cons` list.

```
(: seq-map (All (A B)
  (-> (-> A B) (Listof A) (Listof B))))
```

# Another Problem

Mapping a function `f` over a `cons` list.

```
(: seq-map (All (A B)
  (-> (-> A B) (Listof A) (Listof B))))

(define (seq-map f xs)
  (match xs
    ['() '()]
    [(cons x xs) (cons (f x) (seq-map f xs))]))
```

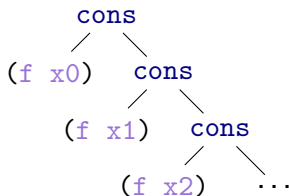
# Another Problem

Mapping a function `f` over a `cons` list.

```
(: seq-map (All (A B)
  (-> (-> A B) (Listof A) (Listof B))))

(define (seq-map f xs)
  (match xs
    ['() '()]
    [(cons x xs) (cons (f x) (seq-map f xs))]))
```

Expression tree of `seq-map`:

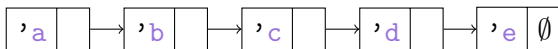


# What Is The Problem?

`seq-map` is not the problem. It is the only way to handle a `cons` list. **The real problem is the single-linked list!** It is inherently sequential:

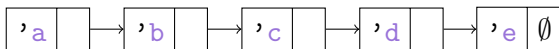
# What Is The Problem?

`seq-map` is not the problem. It is the only way to handle a `cons` list. **The real problem is the single-linked list!** It is inherently sequential:



# What Is The Problem?

`seq-map` is not the problem. It is the only way to handle a `cons` list. **The real problem is the single-linked list!** It is inherently sequential:



The parallel expressions we have seen so far are *trees*. Maybe we can develop a tree data structure that we can use to implement a parallel version of `map`?



# A New Parallel Data Type

We represent the list as a tree instead:

```
(define-type (CatListof A) (U (leaf A)
                               (cat A)))

(struct (A) leaf ([a : A]))
(struct (A) cat  ([l : (CatListof A)]
                  [r : (CatListof A)]))
```

# A New Parallel Data Type

We represent the list as a tree instead:

```
(define-type (CatListof A) (U (leaf A)
                               (cat A)))

(struct (A) leaf ([a : A]))
(struct (A) cat  ([l : (CatListof A)]
                  [r : (CatListof A)]))
```

(leaf a) produces a singleton `CatList`.

(cat l r) concatenates two `CatList` instances.

# A New Parallel Data Type

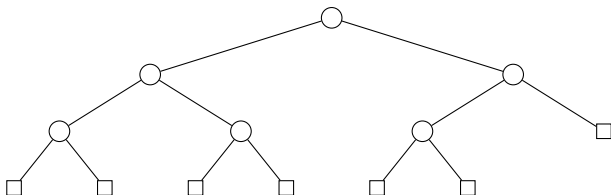
We represent the list as a tree instead:

```
(define-type (CatListof A) (U (leaf A)
                               (cat A)))

(struct (A) leaf ([a : A]))
(struct (A) cat  ([l : (CatListof A)]
                  [r : (CatListof A)]))
```

(leaf a) produces a singleton CatList.

(cat l r) concatenates two CatList instances.



# Let's Implement Map!

```
(: par-map (All (A B)
  (-> (-> A B) (CatListof A) (CatListof B))))
```

# Let's Implement Map!

```
(: par-map (All (A B)
  (-> (-> A B) (CatListof A) (CatListof B))))

(define (par-map f xs)
  (match xs
    [(leaf x) (leaf (f x))]
    [(cat l r) (cat (par-map f l)
                     (par-map f r))]))
```

# Let's Implement Map!

```
(: par-map (All (A B)
  (-> (-> A B) (CatListof A) (CatListof B))))

(define (par-map f xs)
  (match xs
    [(leaf x) (leaf (f x))]
    [(cat l r) (cat (par-map f l)
                     (par-map f r))]))
```

**Note:** This function is not *truly* parallel yet. We have just created a *possibility* for parallelism.

# Parallelism in Racket

In Racket we use a fork/join style model of parallel computations. Fork is called `future` and join is called `touch`:

```
(: in-parallel (All (A B C)
  (-> (-> A B) (-> A C) A (Pairof B C))))
```

# Parallelism in Racket

In Racket we use a fork/join style model of parallel computations. Fork is called `future` and join is called `touch`:

```
(: in-parallel (All (A B C)
  (-> (-> A B) (-> A C) A (Pairof B C))))
```

```
(define (in-parallel f g x)
  (let ([fx (future (lambda () (f x)))]
        [gx (g x)]))
    (cons (touch fx) gx)))
```



# Parallelism in Racket

In Racket we use a fork/join style model of parallel computations. Fork is called `future` and join is called `touch`:

```
(: in-parallel (All (A B C)
  (-> (-> A B) (-> A C) A (Pairof B C))))
```

```
(define (in-parallel f g x)
  (let ([fx (future (lambda () (f x)))]
        [gx (g x)]))
    (cons (touch fx) gx)))
```

Racket has a built-in scheduler for `futures`, so we give a lot of control to the run-time.

```
(: future (All (A) (->(->A) (Futureof A))))
(: touch (All (A) (->(Futureof A) A)))
```

# Parallelizing Map, For Real

```
(: par-map (All (A B)
  (-> (-> A B) (CatListof A) (CatListof B))))
```

# Parallelizing Map, For Real

```
(: par-map (All (A B)
  (-> (-> A B) (CatListof A) (CatListof B))))

(define (par-map f xs)
  (match xs
    [(leaf x) (leaf (f x))])
```

# Parallelizing Map, For Real

```
(: par-map (All (A B)
  (-> (-> A B) (CatListof A) (CatListof B))))

(define (par-map f xs)
  (match xs
    [(leaf x) (leaf (f x))]
    [(cat l r)
     (let ([fl (future ;; Map l in parallel.
                      (lambda () (par-map f l)))]
```

# Parallelizing Map, For Real

```
(: par-map (All (A B)
  (-> (-> A B) (CatListof A) (CatListof B))))

(define (par-map f xs)
  (match xs
    [(leaf x) (leaf (f x))]
    [(cat l r)
     (let ([fl (future ;; Map l in parallel.
                      (lambda () (par-map f l)))]
           [fr (future ;; Map r in parallel.
                      (lambda () (par-map f r)))]))])])
```

# Parallelizing Map, For Real

```
(: par-map (All (A B)
  (-> (-> A B) (CatListof A) (CatListof B))))

(define (par-map f xs)
  (match xs
    [(leaf x) (leaf (f x))]
    [(cat l r)
     (let ([fl (future ;; Map l in parallel.
                      (lambda () (par-map f l)))]
           [fr (future ;; Map r in parallel.
                      (lambda () (par-map f r)))]])
       (cat (touch fl) (touch fr)))]))
```

The recursive call is wrapped in a `future`!

# Pro and Contra

The good parts:

# Pro and Contra

The good parts:

- ▶ The build-in scheduler decides on the number of threads to use.



# Pro and Contra

The good parts:

- ▶ The build-in scheduler decides on the number of threads to use.
- ▶ A thread that is blocked by `touch` is de-scheduled until the `touched future` is done, no locking, spinning etc.

# Pro and Contra

The good parts:

- ▶ The build-in scheduler decides on the number of threads to use.
- ▶ A thread that is blocked by `touch` is de-scheduled until the `touched future` is done, no locking, spinning etc.
- ▶ As a result, we do not have to manually decide **how** to split up work across hardware threads.

# Pro and Contra

The good parts:

- ▶ The build-in scheduler decides on the number of threads to use.
- ▶ A thread that is blocked by `touch` is de-scheduled until the `touched future` is done, no locking, spinning etc.
- ▶ As a result, we do not have to manually decide **how** to split up work across hardware threads.

The bad parts:

# Pro and Contra

The good parts:

- ▶ The build-in scheduler decides on the number of threads to use.
- ▶ A thread that is blocked by `touch` is de-scheduled until the `touched future` is done, no locking, spinning etc.
- ▶ As a result, we do not have to manually decide **how** to split up work across hardware threads.

The bad parts:

- ▶ A list has  $O(n)$  memory overhead, a balanced tree  $O(n \log n)$ .

# Pro and Contra

The good parts:

- ▶ The build-in scheduler decides on the number of threads to use.
- ▶ A thread that is blocked by `touch` is de-scheduled until the `touched future` is done, no locking, spinning etc.
- ▶ As a result, we do not have to manually decide **how** to split up work across hardware threads.

The bad parts:

- ▶ A list has  $O(n)$  memory overhead, a balanced tree  $O(n \log n)$ .
- ▶ If the tree is not balanced, we lose parallelism – mitigate by balancing algorithm!

# Pro and Contra

The good parts:

- ▶ The build-in scheduler decides on the number of threads to use.
- ▶ A thread that is blocked by `touch` is de-scheduled until the `touched future` is done, no locking, spinning etc.
- ▶ As a result, we do not have to manually decide **how** to split up work across hardware threads.

The bad parts:

- ▶ A list has  $O(n)$  memory overhead, a balanced tree  $O(n \log n)$ .
- ▶ If the tree is not balanced, we lose parallelism – mitigate by balancing algorithm!
- ▶ Maybe we spawn **excessively many futures** and put too much strain on the scheduler?

# An Improved Variant

If the size of the tree is too small, then the overhead of constructing futures dominates the run-time cost. We can handle this problem by using lists as leaves instead of scalar values.

# An Improved Variant

If the size of the tree is too small, then the overhead of constructing `future`s dominates the run-time cost. We can handle this problem by using lists as leaves instead of scalar values.

We call such a tree a **Rope**:

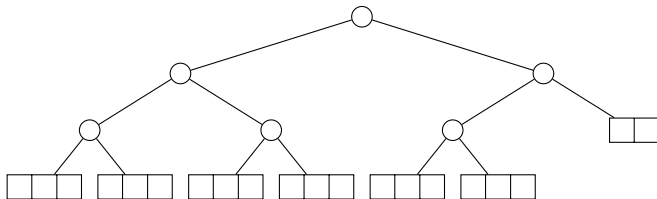
```
(define-type (Ropeof A) (U (leaf A)
                           (cat A)))

(struct (A) leaf ([as : (Listof A)]))
(struct (A) cat  ([l  : (Ropeof A)]
                  [r  : (Ropeof A)]))
```



## Ropes: Trees With Lists as Leaves

We must define a **maximum size**  $s_{\max}$  for leaves. In this example,  $s_{\max} = 3$ :



Parallel higher-order functions on ropes.

# Live Coding Session

Parallel higher-order functions on ropes.

- ▶ I code on the big screen and show you how to do functional programming in Racket.

# Live Coding Session

Parallel higher-order functions on ropes.

- ▶ I code on the big screen and show you how to do functional programming in Racket.
- ▶ You help with ideas and suggestions and experiment yourselves!

# Live Coding Session

Parallel higher-order functions on ropes.

- ▶ I code on the big screen and show you how to do functional programming in Racket.
- ▶ You help with ideas and suggestions and experiment yourselves!
- ▶ All code we write will be available on
  - ▶ [github.com/fbie/parallel-functional-programming](https://github.com/fbie/parallel-functional-programming)

# Live Coding Session

Parallel higher-order functions on ropes.

- ▶ I code on the big screen and show you how to do functional programming in Racket.
- ▶ You help with ideas and suggestions and experiment yourselves!
- ▶ All code we write will be available on
  - ▶ [github.com/fbie/parallel-functional-programming](https://github.com/fbie/parallel-functional-programming)
- ▶ Want to start now and code along?
  - ▶ Download the code from [2/ropes-lecture.rkt](#)

# Live Coding Session

Parallel higher-order functions on ropes.

- ▶ I code on the big screen and show you how to do functional programming in Racket.
- ▶ You help with ideas and suggestions and experiment yourselves!
- ▶ All code we write will be available on
  - ▶ [github.com/fbie/parallel-functional-programming](https://github.com/fbie/parallel-functional-programming)
- ▶ Want to start now and code along?
  - ▶ Download the code from [2/ropes-lecture.rkt](#)
- ▶ You can download Racket from
  - ▶ [racket-lang.org](https://racket-lang.org)

# New Syntax: Recursive Let

Standard `let`:

```
(let ([x 1]
      [y (+ 2 x)]))
  (* y x))
```



# New Syntax: Recursive Let

Standard `let`:

```
(let ([x 1]
      [y (+ 2 x)]))
  (* y x))
```

Undefined identifier: `x`

# New Syntax: Recursive Let

Standard `let`:

```
(let ([x 1]
      [y (+ 2 x)]))
  (* y x))
```

Undefined identifier: `x`

Solution: `letrec`, recursive let-binding:

```
(letrec ([x 1]
         [y (+ 2 x)]))
  (* y x))
```

Allows bindings to reference one-another.

# De-sugaring Recursive Let

```
(letrec ([x 1]
         [y (+ 2 x)])
  (* y x))
```

is equal to

```
(let ([x 1])
  (let ([y (+ 2 x)])
    (* y x)))
```

# Take-Aways From This Lecture

# Take-Aways From This Lecture

- ▶ Immutable data structures are automatically thread-safe!

# Take-Aways From This Lecture

- ▶ Immutable data structures are automatically thread-safe!
- ▶ Choose trees over lists.

# Take-Aways From This Lecture

- ▶ Immutable data structures are automatically thread-safe!
- ▶ Choose trees over lists.
- ▶ Use sequential data structures at the leaves.

# Take-Aways From This Lecture

- ▶ Immutable data structures are automatically thread-safe!
- ▶ Choose trees over lists.
- ▶ Use sequential data structures at the leaves.
- ▶ Use the built-in scheduler.



# Take-Aways From This Lecture

- ▶ Immutable data structures are automatically thread-safe!
- ▶ Choose trees over lists.
- ▶ Use sequential data structures at the leaves.
- ▶ Use the built-in scheduler.
- ▶ Use higher-order functions for re-usable solutions.

# Take-Aways From This Lecture

- ▶ Immutable data structures are automatically thread-safe!
- ▶ Choose trees over lists.
- ▶ Use sequential data structures at the leaves.
- ▶ Use the built-in scheduler.
- ▶ Use higher-order functions for re-usable solutions.

**Reading material available on the GitHub page:**

# Take-Aways From This Lecture

- ▶ Immutable data structures are automatically thread-safe!
- ▶ Choose trees over lists.
- ▶ Use sequential data structures at the leaves.
- ▶ Use the built-in scheduler.
- ▶ Use higher-order functions for re-usable solutions.

## Reading material available on the GitHub page:

- ▶ Herb Sutter: **The Free Lunch is Over, Dr. Dobb's Journal**

# Take-Aways From This Lecture

- ▶ Immutable data structures are automatically thread-safe!
- ▶ Choose trees over lists.
- ▶ Use sequential data structures at the leaves.
- ▶ Use the built-in scheduler.
- ▶ Use higher-order functions for re-usable solutions.

## Reading material available on the GitHub page:

- ▶ Herb Sutter: **The Free Lunch is Over, Dr. Dobb's Journal**
- ▶ Michael Erdmann: **Parallelism, Cost Graphs, and Sequences, lecture notes**

# Take-Aways From This Lecture

- ▶ Immutable data structures are automatically thread-safe!
- ▶ Choose trees over lists.
- ▶ Use sequential data structures at the leaves.
- ▶ Use the built-in scheduler.
- ▶ Use higher-order functions for re-usable solutions.

## Reading material available on the GitHub page:

- ▶ Herb Sutter: **The Free Lunch is Over, Dr. Dobb's Journal**
- ▶ Michael Erdmann: **Parallelism, Cost Graphs, and Sequences, lecture notes**
- ▶ Hans-J. Boehm et al.: **Ropes: an Alternative to Strings** (optional)

# Questions?

Slides and code available at  
[github.com/fbie/parallel-functional-programming](https://github.com/fbie/parallel-functional-programming).

# Questions?

Slides and code available at  
[github.com/fbie/parallel-functional-programming](https://github.com/fbie/parallel-functional-programming).

Any questions? E-mail me at [fbie@itu.dk](mailto:fbie@itu.dk).

# Questions?

Slides and code available at  
[github.com/fbie/parallel-functional-programming](https://github.com/fbie/parallel-functional-programming).

Any questions? E-mail me at [fbie@itu.dk](mailto:fbie@itu.dk).

**Thank you for your attention!**