Parallel Computations on Immutable Data Structures

Florian Biermann

fbie@itu.dk

IT University of Copenhagen & UCAS

2016-06-02

IT University of Copenhagen

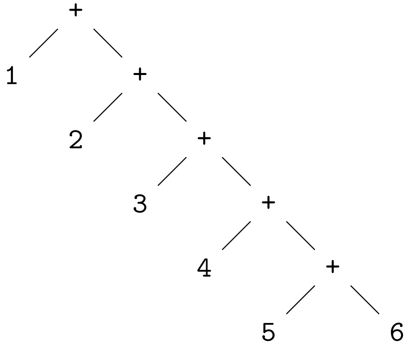
A Recursive Function

$\sum_{i=1}^{6} i$ = `(+ 1 (+ 2 (+ 3 (+ 4 (+ 5 6)))))`

A Recursive Function

$$\sum_{i=1}^{6} i \ = \ \texttt{(+ 1 (+ 2 (+ 3 (+ 4 (+ 5 6)))))}$$

The expression's call-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.

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.

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

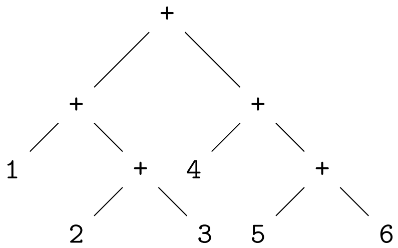
A Parallel Expression

$\sum_{i=1}^{6} i$ `= (+ (+ 1 (+ 2 3)) (+ 4 (+ 5 6)))`

A Parallel Expression

$\sum_{i=1}^{6} i$ `=  (+ (+ 1 (+ 2 3)) (+ 4 (+ 5 6)))`

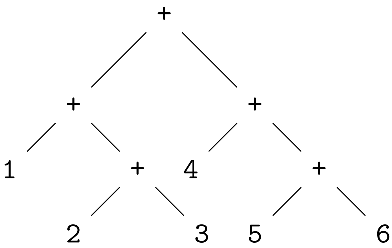
The expression's call-tree:



A Parallel Expression

$$\sum_{i=1}^{6} i \quad = \quad \texttt{(+ (+ 1 (+ 2 3)) (+ 4 (+ 5 6)))}$$

The expression's call-tree:



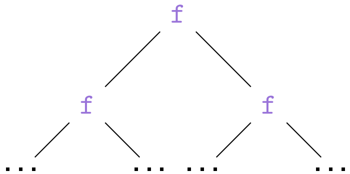
Why is this possible? Because + is an associative operation!

`(+ 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)
```



Another Problem

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

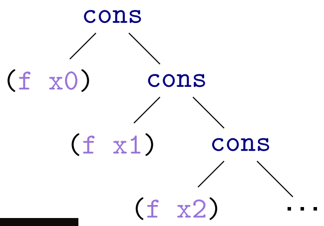
```
(: seq-map (All (A B)
     (-> (-> A B) (Listof A) (Listof B))))
(define (seq-map f xs)
  (match xs
    ['() '()]
    [(cons x xs) (cons (f x) (seq-map f xs))]))
```

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))]))
```

Call-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.

All parallel expressions we saw so far were 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.

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))]))
```

Is this efficient? There are a few things to be aware of:

- A list has $O(n)$ memory overhead, a tree $O(n \log n)$.
- If the tree is not balanced, we lose parallelism.
- It is not truly parallel yet. This is the next step!

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 `future`s, 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))))
(define (par-map f xs)
  (match xs
    [(leaf x) (leaf (f x))]
    [(cat l r)
       (let ([l0 (future ;; Map l in parallel.
                   (lambda () (par-map f l)))]
             [r0 (future ;; Map r in parallel.
                   (lambda () (par-map f r)))])
         (cat (touch l0) (touch r0)))]))
```