Lecture Notes:
Functional Programming in Typed Racket

Florian Biermann
fbie@itu.dk

2016-06-17

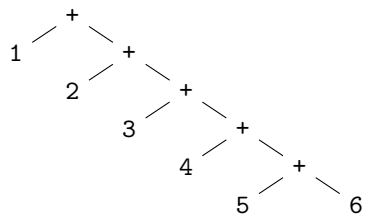
This document focuses on the practical side of parallel programming. Erdmann [2] covers this topic, including cost semantics, much more thoroughly.

1 Sequential and Parallel Expressions

Imagine you write this expression in Racket:

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

The Racket syntax already makes it clear how the corresponding abstract syntax tree (AST) looks like:



The AST shows that there is a sequential dependency between each invocation of `+`, which is the long spine going from the upper left to the lower right. If your program runs on a single core processor, this is a good solution. Today's modern processors however are multi-core processors. Can we change the expression to make use of more than one processor?

As it turns out, this is entirely possible. Consider the following expression instead:

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

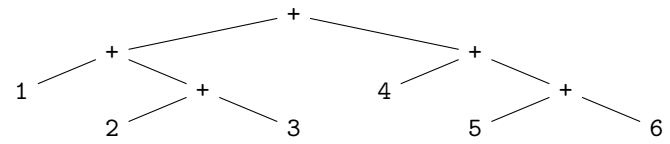
Note that we still use the same number of `+` operations. It will compute the same number, because `+` is an *associative operation*. Associativity means that the order in which the operations are applied does not matter for the result, and therefore it holds that

$$\texttt{(+ a (+ b c))} = \texttt{(+ (+ a b) c)}$$

or

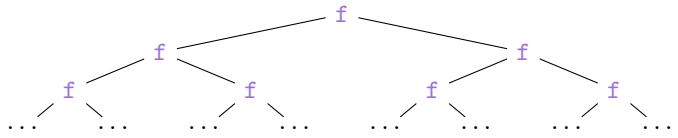
$$a + (b + c) = (a + b) + c$$

if you prefer mathematical notation. This is how the AST looks like for the reformulated expression:



Now, instead of a tree with a long spine, we have a somewhat balanced binary tree. The branches of trees are independent from each other, otherwise the tree would be a cyclic graph. We can make use of this independence, because independent sub-trees means independent calculations and independent calculations means that there is an opportunity for parallelism!

We can reformulate any sequential expression that calls the same function `f` recursively if it holds that `(f a (f b c)) = (f (f a b) c)`, i.e. that `f` is an associative operation:

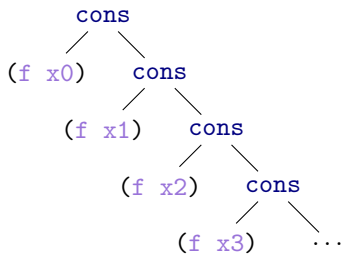


2 Sequential and Parallel Functions

Take a look at the code for sequentially mapping a function over the standard cons-list (or linked list) in Racket:

```
(: 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))]))
```

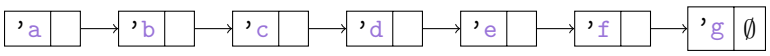
And here is a hypothetical call-tree for `seq-map`:



Again, you see the for sequential computations typical long spine in the tree. Can we turn this sequential function into a parallel function?

Exercise: try to do it now!

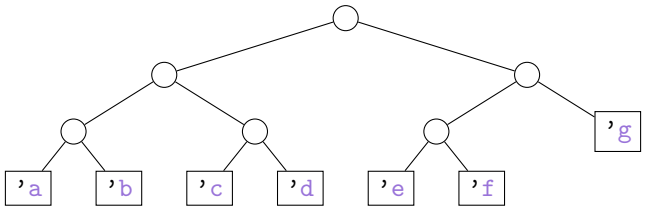
You will quickly see that there is no way to efficiently combine two lists without adding new sequential operations. This is because the `cons` constructor is of type `(All A (->A (Listof A)))`. It does not know how to combine two lists. Therefore, we cannot write a parallel version of `seq-map` on lists. This becomes even more obvious if you visualize the linked list':



What can we do instead? We can create a new data structure that supports parallelism by its very nature. How would such a data structure look like? The parallel expression for the sum from 1 to 6 was a binary tree, so let's try to use a tree to represent our list. We call it `CatListof A`, where *cat* is short for concatenation:

```
(define-type (CatListof A) (U (leaf A)
                              (cat A)))
(struct (A) leaf ([a : A]))
(struct (A) cat  ([l : (CatListof A)]
                  [r : (CatListof A)]))
```

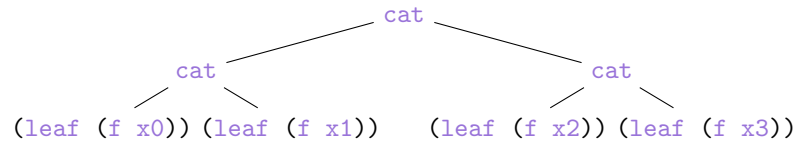
`(leaf a)` produces a singleton `(CatListof A)`. `(cat l r)` concatenates two `(CatListof A)` instances and produces a new instance of `(CatListof A)`. Here is an illustration of an instance of `(CatListof A)` where circles are `cat` instances and rectangles are `leaf` instances:



Now, we can implement a map-function on `(CatListof A)` instances:

```
(: cat-map
  (All (A B) (-> (-> A B) (CatListof A) (CatListof B))))
(define (cat-map f xs)
  (match xs
    [(leaf x) (leaf (f x))]
    [(cat l r) (cat (cat-map f l)
                    (cat-map f r))]))
```

This is still not parallel. However, the only way we can write a map-function for `(CatListof A)` produces a call-tree that bears a *possibility for parallelism*:



3 Parallelism in Racket

Racket and Typed Racket use a fork/join style model of parallel computations. Fork is called `future` and join is called `touch`. We can use it to compute two functions in parallel as follows:

```
(: 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)))
```

Here, the function the function `g` is computed on the same thread and `f` is computed on a different thread.When you call `future` with a lambda expression, Racket creates a new object of type `(Futureof A)`. This is a representation of an ongoing computation (i.e. the lambda expression) which possibly runs in parallel. If and on which processor core this future is computes is determined by the built-in scheduler. There is no need to explicitly create and manage threads in Racket.

In order to get the computed value, you need to synchronize the thread that wants to read the value with the thread that computes the value. This you do by calling `touch` with the instance of `(Futureof A)` from which you want to retrieve the value. If the thread that was computing the future is already done, the thread that is reading the value will just continue. If the future is not yet computed however, the reading thread will block until the computation is done.

This are the types of `future` and `touch`:

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

4 Truly Parallel Functions

Let us return to implementing a parallel version of a map-function for `(CatListof A)`. We now have all the tools we need: a data-structure already laid out for parallelism and a way to express parallel computations in the language. So let's look at how the code for `cat-map` looks like if we apply `future` and `touch`:

```
(: 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)))]))
```

As you can see, we keep the overall structure of the function. The base-case for leaves remains completely untouched. Notice that the types of `cat-map` and `par-map` are identical. The difference is that, instead of calling `par-map` recursively directly, we wrap the recursive call in a lambda-expression which we pass to the `future` constructor. After we have created a future for the left and the right sub-tree of the list, we wait for both computations using `touch` and then we combine the two resulting lists using `cat`.

We call parallelism on aggregate data structures *data-parallelism*, because the parallelism stems from the fact that we apply the same function to many different inputs in parallel.

By creating a new future at each node, we make sure that we exploit as much parallelism as possible. Also, we do not need to worry about how many threads we want to launch or how we want to chunk up the data, because Racket's scheduler takes care of that automatically. This does not come for free, though. Creating many futures for small trees might outweigh the benefit of parallelization, maybe even makes the operation slower.

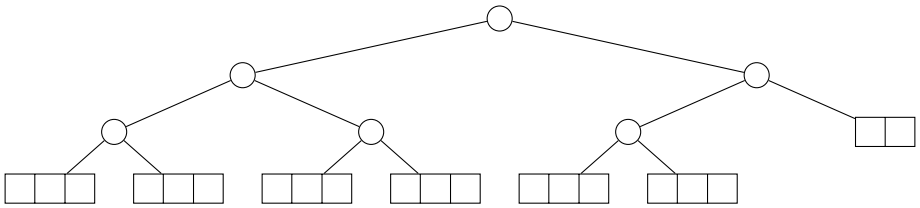
5 Limiting Parallelism with Ropes

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 to limit the degree of parallelism for small input sizes. We can use a data structure called a *Rope*, which is a binary tree with sequential lists (or arrays) as leaves [1]:

```
(define-type (Ropeof A) (U (leaf A) (cat A)))
(struct (A) leaf ([as : (Listof A)]))
(struct (A) cat  ([l  : (Ropeof A)]
```

```
   [r   : (Ropeof A)]))
```

We need to define a value $s_{\max}$ that limits the length of the leaf arrays. Here is an illustration of such a rope, where the $s_{\max} = 3$:



Choosing the a good value for $s_{\max}$ is important. If we set $s_{\max}$ too low, our algorithms will probably create excessively many `future` instances, but if we set $s_{\max}$ too high, we will not expose enough parallelism to gain any performance benefits.

Finding a good value for $s_{\max}$ depends a lot on the application. Do you expect to map mostly long-running functions on your collection? Will this function have roughly the same run-time for each input or will it vary heavily? There are many factors that you need to consider.

How do we implement *map* for this data structure? It is actually quite similar to our `CatListof A` implementation. But because the leaves of the rope are not scalar values but another collection data type, we must rely on using also a higher-order *map* function for this data structure. Since we use the built-in `cons` list, we can simply use the default `map` function that comes with it:

```
(: rope-map
  (All (A B) (-> (-> A B) (Ropeof A) (Ropeof B))))
(define (rope-map f rope)
  (match rope
    [(leaf as) (leaf (map f as))]
    [(cat l r) (cat (rope-map f l)
                    (rope-map f r))]))
```

This is a straight-forward implementation of *map* on ropes. The base case for leaves does not apply the function `f` directly. Instead, it calls `map` on `cons` lists and passes `f` on to the call.

The recursive case for `cat` simply decomposes the rope into its subtrees and calls itself recursively just as we did it for the `CatListof A`.

We can now proceed and write a parallel version of this function. All we need to do is, again, to wrap the recursive calls to `rope-map` in a lambda expression and pass it to `future`. We call this version of `rope-map` simply `rope-pmap`, where p stands for "parallel". Notice again that the types of both functions are the same. Notice furthermore that we again do not change the base case for `leaf`, because here, we explicitly prefer sequential execution:

```
(: rope-pmap
  (All (A B) (-> (-> A B) (Ropeof A) (Ropeof B))))
(define (rope-pmap f rope)
  (match rope
    [(leaf as) (leaf (map f as))]
    [(cat l r) (let [(l0 (future (lambda ()
                                   (rope-pmap f l))))
                     (r0 (future (lambda ()
                                   (rope-pmap f r))))]
                 (cat (touch l0) (touch r0)))]))
```

If we want to proceed to parallelize more functions on ropes, we simply can first implement the simplest sequential version of the function that we can think of. Then, in the next step, we can simply wrap the recursive call in a lambda expression and pass it to a call to `future`. We do not need to worry about thread safety, because the sub-trees are completely disjoint, so the computations will not interfere with each other.

Here is another example, the implementations for the rope versions of *reduce*, namely `rope-reduce` and `rope-preduce`:

```
(: rope-reduce
  (All (A) (-> (-> A A A) (Ropeof A) A)))
(define (rope-reduce f rope)
  (match rope
    [(leaf as) (list-reduce f as)]
    [(cat l r) (f (rope-reduce f l)
                  (rope-reduce f r))]))

(: rope-preduce
  (All (A) (-> (-> A A A) (Ropeof A) A)))
(define (rope-preduce f rope)
  (match rope
    [(leaf as) (list-reduce f as)]
    [(cat l r) (let [(l0 (future (lambda ()
                                   (rope-preduce f l))))
                     (r0 (future (lambda ()
                                   (rope-preduce f r))))]
                 (f (touch l0) (touch r0)))]))
```

Does this mean that we can parallelize *all* higher-order functions that we can think of? It turns out that this is not the case. For instance, the general *fold* function is of type $(B \to A \to B) \to B \to [A] \to B$, where $[A]$ denotes a collection of values of type A. We can write this type in Typed Racket for `cons` list as `(All (A B) (->(->B A B) B (Listof A) B))`.

We have learned that we can parallelize functions that are *associative*. Therefore, in order to parallelize *fold*, we must show that `f` is commutative. If you look at its type, it becomes clear that it is not. Let us assume that we have a

rope with a left and a right subtree. If we compute *fold* for the left side and for the right side, following the definition of *fold*, we end up with two values for each side, both of type B. But our function f is of type $B \to A \to B$, which means we do not know how to combine two values of type B.

Therefore, we cannot parallelize *fold* unless we add a second function parameter of type $B \to B \to B$. It turns out, that this is not necessary, because we can instead use *map* and *reduce* to achieve the same result.

References

[1] Hans-J Boehm, Russ Atkinson, and Michael Plass. Ropes: an Alternative to Strings. 1995. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.9450`.

[2] Michael Erdmann. Parallelism, Cost Graphs, and Sequences. Lecture Notes, 2016. URL `http://www.cs.cmu.edu/~15150/resources/lectures/19/Parallelism.pdf`.