Lecture Notes:
Functional Programming in Typed Racket

Florian Biermann
fbie@itu.dk

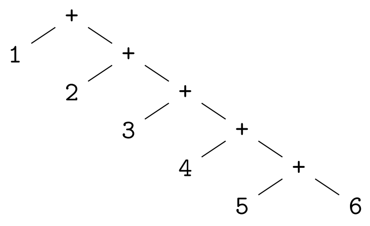
2016-05-26

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)))
```

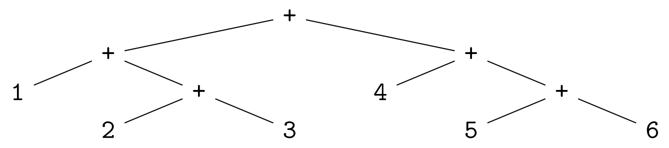
Note that we still use the same number of `+` operations. It will compute the same number, because `+` is an *associative operation*. Associativity means that the order in which the operations are applied does not matter for the result, and therefore it holds that

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

or

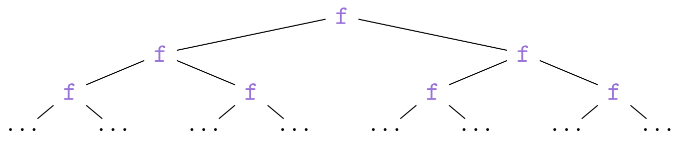
$$a + (b + c) = (a + b) + c$$

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:

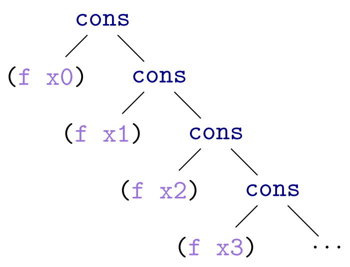


2 Sequential and Parallel Functions

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))]))
```

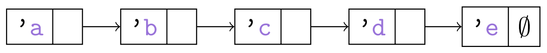
And here is a hypothetical call-tree for `seq-map`:



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!

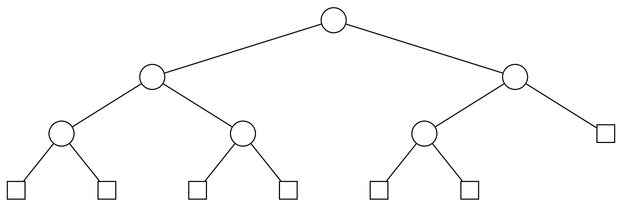
You will quickly see that there is no way to efficiently combine two lists without adding new sequential operations. This is because the `cons` constructor is of type `(All A (->A (Listof A)))`. It does not know how to combine two lists. Therefore, we cannot write a parallel version of `seq-map` on lists.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

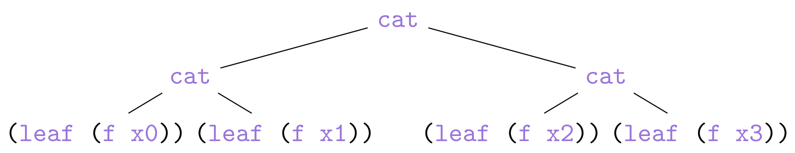
`(leaf a)` produces a singleton `(CatListof A)`. `(cat l r)` concatenates two `(CatListof A)` instances and produces a new instance of `(CatListof A)`. Here is an illustration of an instance of `(CatListof A)` where circles are `cat` instances and rectangles are `leaf` instances:



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 futures 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:

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