

Programming Strategies for Parallel Haskell



Applicant : Andrea Senese
Supervisor : Ugo de' Liguoro

University of Turin
Mathematical, Physical and Natural Sciences School
Computer Science Department
Bachelor's Degree in Computer Science - Languages and Systems

- 1 *What is Haskell?*
- 2 *Determinism in Haskell*
- 3 *Monad for Non-Deterministic Action*
- 4 *Parallel Haskell : Eval Monad and Strategies*
- 5 *Par Monad*
- 6 *Parallel Type Inference Algorithm*

- *Haskell* is a **purely functional advanced programming language** (based on the **lambda calculus system**) that takes its name from the US mathematical-logical Haskell Curry;
- Suitable for **teaching, research, applications** and the **construction of large systems**;
- Fully described through the publication of a **syntax** and **formal semantics**;
- Haskell uses a lazy evaluation and it is a lazy language;
- So, without having **side effects** in **Haskell**, we can say that it is **deterministic** and therefore by its nature we can create **parallel** programs.



Figure 1: *Haskell logo*

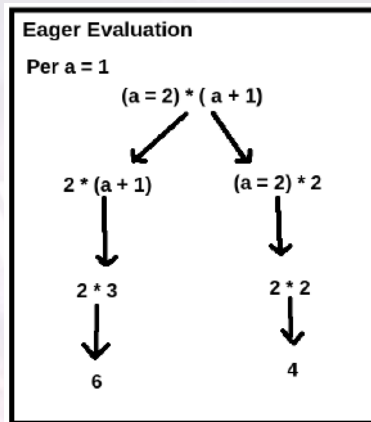
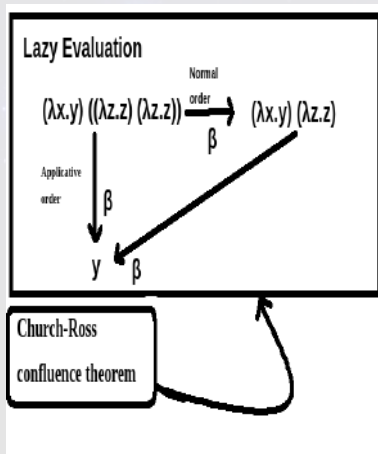


Figure 2: *Lazy Evaluation vs Eager Evaluation*

- Given a term n that can be reduced in one or more computation steps in n_1 , then there exists N such that N_1 is reduced to N (in one or more computation steps) and N_2 is reduced to N (in one or more steps). Formally:

Theorem (Church-Ross Confluence)

If $M \rightarrow N_1$ and $M \rightarrow N_2$ then $\exists N$ such that $N_1 \rightarrow N$ and $N_2 \rightarrow N$

Indirect Proof:

Supposed that N_1 and N_2 are two normal forms of M then by the confluence theorem since we know that N_1 is reduced to N (in 0 steps) and also N_2 then for transitivity:

$$(N_1 = N \wedge N_2 = N) \Rightarrow N_1 = N_2$$

- *Monads in Haskell* can be thought of as composable computation descriptions;
- The essence of *Monad* is thus separation of composition timeline from the composed computation's execution timeline, as well as the ability of computation to implicitly carry extra data, as pertaining to the computation itself, in addition to its one (hence the name) output, that it will produce when run (or queried, or called upon). This allows monads supplementing pure calculations with features like *I/O*, *common environment*, *updatable state*, etc..;
- *Monads* are composed by three features:
 - ▶ a *type constructor* *M*;
 - ▶ A *bind* operator (*>>=*);
 - ▶ a *return* operator.

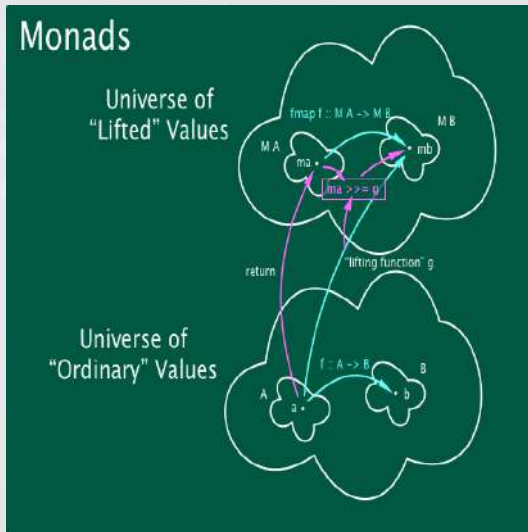


Figure 3: *Monads*



Figure 4: *Parallelism* \neq *Concurrency*

- *Parallelism* and *Concurrency* are distinct concepts:
 - ▶ *Parallelism* means that several threads work on the same task to reduce computation calculus and response times \Rightarrow *Determinism*;
 - ▶ *Concurrency* consists of performing different tasks from different threads and sharing the CPU resource \Rightarrow *Non-Determinism*.
- *Haskell* offers the following tools to the programmer to parallelize their programs:
 - ▶ *Eval Monad* and *Strategies*;
 - ▶ *Par Monad*.

- *Eval Monad* is the first approach to express *Parallelism* in *Haskell* from *Control.Parallel.Strategies*, using the *rpar* and *rseq* functions:
 - ▶ *runEval* → compute the operation in the *Eval Monad* and *return* the result;
 - ▶ *rpar* → the argument could be evaluated in parallel;
 - ▶ *rseq* → forces the sequential evaluation and waits for the result.
- For example :

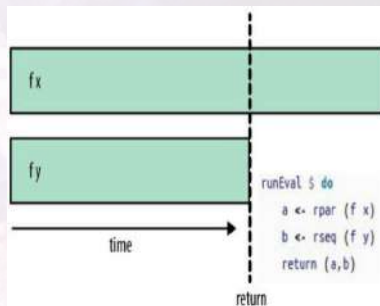
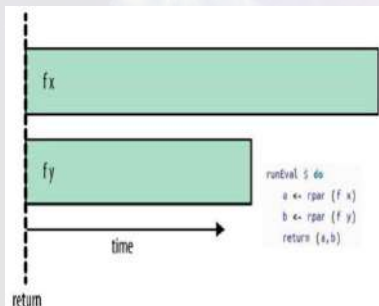


Figure 5: *rpar/rpar* and *rpar/rseq*

- The solution is:

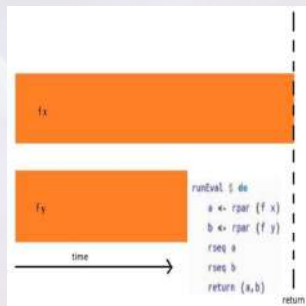


Figure 6: *rpar/rpar/rseq/rseq*

- *Eval Monad* can be used to diagnose programs with *Threadscope*.

- **Threadscope** was originally created to help programmers visualize the parallel execution of **Haskell** programs compiled with **GHC**. The idea is that when running a parallel **Haskell** program, the **Haskell** runtime system writes date and time reports of significant events to a log file. The **Threadscope** tool then reads this log file, and graphically shows the user what each **CPU** was doing in that time interval. The diagrams it shows reveal to the programmer where the program is using parallelism, and where it is not.

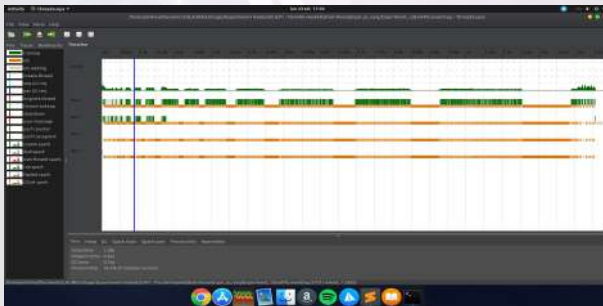


Figure 7: Threadscope tool

- There's a close relationship between the *Eval Monad* and *Strategies* as we can see in the *Haddock* documentation. *Strategies* step in as a higherlevel abstraction to ease many of the problems associated with regulating granularity and forcing evaluation manually;
- *Evaluation Strategies*, or simply *Strategies*, are ways used to modularize parallel code by separating the algorithm from the parallelism and provide ways to express parallel computations. *Strategies* have the following key features:
 - ▶ *Strategies* express deterministic parallelism: the result of the program is not affected by evaluating in parallel. The parallel tasks evaluated by a *Strategy* may have no side effects;
 - ▶ *Strategies* allow to separate the description of the parallelism from the logic of the program, enabling modular parallelism. The basic idea is to build a lazy data structure representing the computation, and then write a *Strategy* that describes how to traverse the data structure and evaluate components of it sequentially or in parallel;
 - ▶ *Strategies* are compositional: larger strategies can be built by gluing together smaller ones.

- *For example : for evaluating list in parallel*

ParList

```
parMap :: (a -> b) -> [a] -> [b]
parMap f xs = using (map f xs) (parList rseq)
```

```
evalList :: Strategy a -> Strategy [a]
evalList strat [] = return []
evalList strat (x:xs) = do
  y <- strat x
  ys <- evalList strat xs
  return (y : ys)
```

```
parList :: Strategy a -> Strategy [a]
parList strat = evalList (rparWith strat)
```

- The Tools mentioned so far lead to high failure rates because *rpar* requires the programmer to understand the operational properties of the program that reside, in most cases, in the implementation;
- Another practice used to correct the main weaknesses is the *Monad Eval*, which unfortunately is still limited (and does not go very far);
- The *Monad Par* offers three basic operations:
 - ▶ *runPar* → compute the operation in the *Par Monad* and *return* the result;
 - ▶ *fork* → create new *Thread*;
 - ▶ *new* → create new *IVar*;
 - ▶ *get* → get the result written in the *IVar*;
 - ▶ *put* → writes the result in the *IVar*.

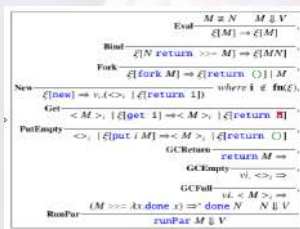


Figure 8: Par Monad Semantics

Warning

*The **IVar** to ensure determinism can only be written once.*

- For example : To parallelize a quicksort

ParQuick

```
parQuickSort :: (Ord a , NFData a) => [a] -> Par [a]
parQuickSort [] = return []
parQuickSort (x : xs) = do
    p1 <- spawn(parQuickSort(filter(< x) xs))
    p2 <- spawn(parQuickSort(filter(>= x) xs))
    left <- get p1
    right <- get p2
    return $ left ++ (x : right)

parQuickSortLimit :: (Ord a , NFData a) => Int -> [a] -> Par [a]
parQuickSortLimit 0 xs = return $ quickSortSeq xs
parQuickSortLimit d [] = return []
parQuickSortLimit d (x : xs) = do
    p1 <- spawn(parQuickSortLimit(d - 1)(filter(< x) xs))
    p2 <- spawn(parQuickSortLimit(d - 1)(filter(>= x) xs))
    left <- get p1
    right <- get p2
    return $ left ++ (x : right)
```


- This graph illustrates the speedup of the parallel quicksort when applying varying thresholds. The results were achieved on a quad-core CPU running 8 threads:

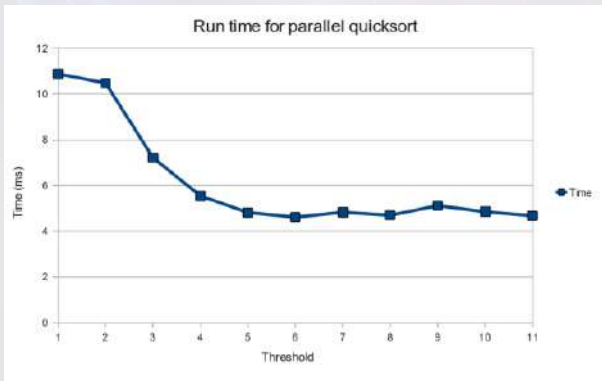


Figure 9: *Timeline ParQuickSortLimit*

- In *Threlescope*:

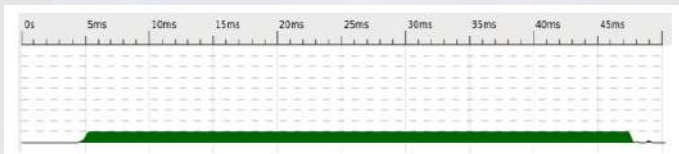


Figure 10: *Sequential quicksort as seen in Threlescope on quad-core CPU with 8-threads.*

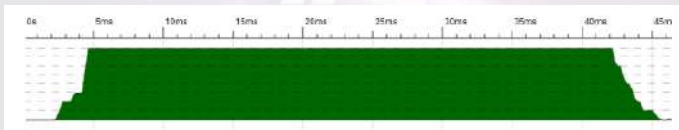
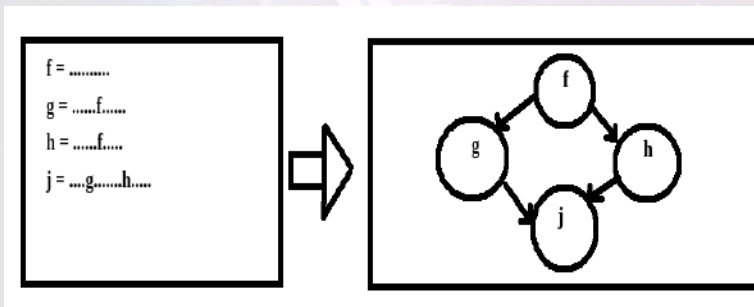


Figure 11: *Parallel quicksort as seen in Threlescope on quad-core CPU with 8-threads.*

Warning

The input data is not necessarily split evenly in each recursion in some runs there can be bad pivot elements which infers that the work is divided very unevenly between the cores. This will slow down the execution!

- An example that naturally fits the dataflow model is program analysis, in which information is typically propagated from definition sites to usage sites in a program;
- Type inference gives rise to a dataflow graph; each binding is a node in the graph with inputs corresponding to the free variables of the binding, and a single output represents the derived type for the binding. For example, the following set of bindings can be represented by the dataflow graph:



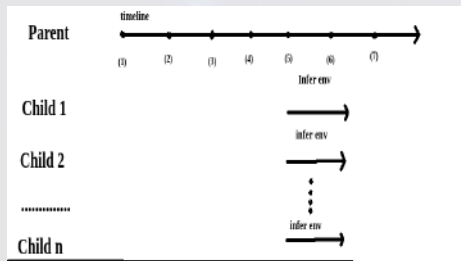
We define the function *infer* as follows:

parInfer

```
--(1)
parInfer :: [(Var, Expr)] -> [(Var, Type)]
parInfer bindings = runPar $ do
    let binders = map fst bindings --(2)
    ivars <- replicateM (length binders) new --(3)
    let Env = Map.fromList (zip binders ivars) --(4)
    mapM_ (fork . infer env) bindings --(5)
    types <- mapM_ get ivars --(6)
    return (zip binders types) --(7)
```

Parallel Type Inference Algorithm

A concrete execution :



- (1) call `parInfer` \rightarrow `parInfer [(var1,exp1),(var2,exp1),...,(var_n,exp_n)]`
- (2) let `binders = map fst [(var1,exp1),(var2,exp1),...,(var_n,exp_n)] = [var1,var2,...,var_n]` (forall elem in list apply `fst(elem)`)
- (3) `ivars = replicateM (length [(var1,exp1),(var2,exp1),...,(var_n,exp_n)]) new = replicateM n new = [Ivar1,Ivar2,...,Ivar_n]`
(create a list of length `n` with `n` `Ivar`)
- (4) `env = Map.fromList (zip [var1,var2,...,var_n] [Ivar1,Ivar2,...,Ivar_n]) = Map.fromList [(var1,Ivar1),(var2,Ivar2),...,(var_n,Ivar_n)] =`

Key	Value
1	var1
...	...
n	var_n

- (5) Infer calls `put` for each binding in the list for write the type in the `Ivar` mapped by that variable in `env`.
- (6) `type = mapM_get [Ivar1,...,Ivar_n] = [type1,type2,...,type_n]`
(wait the result computed by the childs).
- (7) return `(zip [var1,var2,...,var_n] [type1,type2,...,type_n]) = [(var1,type1),...,(var_n,type_n)]` (return the result)

- *This implementation extracts the maximum parallelism inherent in the dependency structure of the given set of bindings, with very little effort on the part of the programmer;*
- *The same trick can be pulled using lazy evaluation of course, and indeed we could achieve the same parallel structure using lazy evaluation together with Strategies, but the advantage of the Par monad version is that the structure is programmed explicitly and the runtime behaviour is not tied to one particular evaluation strategy (or compiler).*

GRAZIE A TUTTI PER L'ATTENZIONE

