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

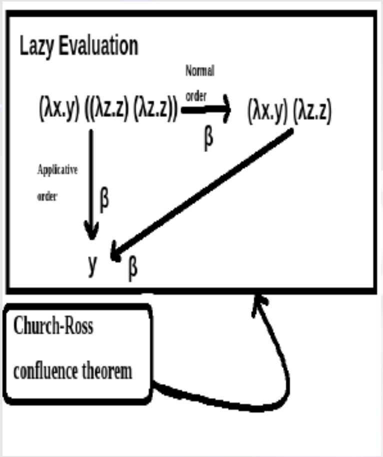
Table of Contents

What is Haskell?

- *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.*

Facebook Engineering
Fighting spam with Haskell
one heavyweight request was enabled
allocation limits enforced

Determinism



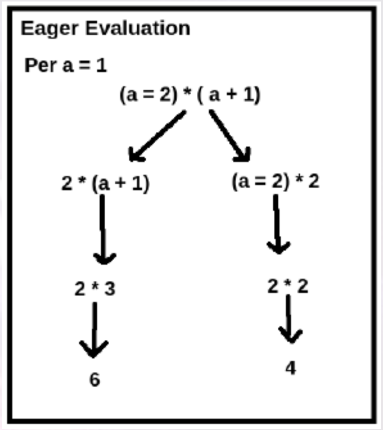


Figure 1: *Lazy Evaluation vs Eager Evaluation*

Determinism

- *Given a term M that can be reduced in one or more computation steps in N_1 and N_2, 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 $M \to N_1$ and $M \to N_2$ then $\exists N$ such that $N_1 \to N$ and $N_2 \to N$

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:

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

Monad for Non-Deterministic Action(side effect)

- *Monads in Haskell can be thoughts 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 allow monads supplementing pure calculations with features like I/O, common environment, updatable state, etc..;*
- *Monads are composed by three features:*
  - *a type costructor M;*
  - *A bind operator (>>=);*
  - *a return operator.*

Monad

```
class Monad m where
        (>>=)  :: m a -> (a -> m b) -> m b
        (>>)   :: m a ->  m b          -> m b
        return ::   a                  -> m a
        fail   :: String -> m a
```

Monad for Non-Deterministic Action(side effect)

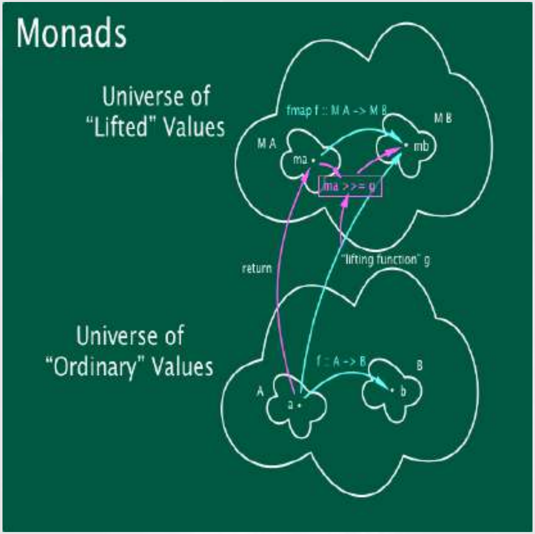


Figure 2: *Monads*

Parallel Haskell

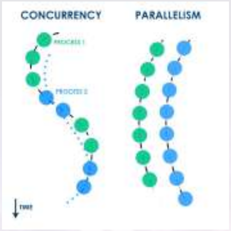
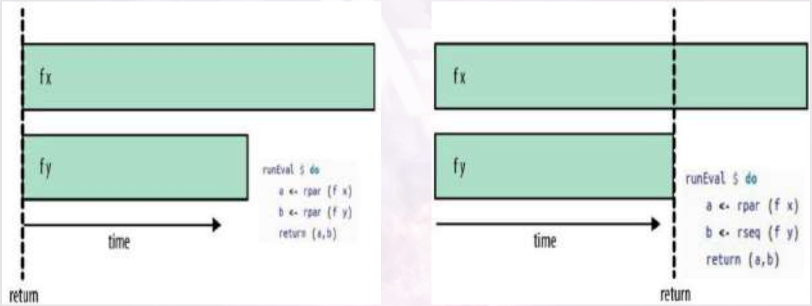


Figure 3: *Parallelism ≠ Concurrency*

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

Eval Monad

- *Eval Monad is the first approach to express Parallelism in Haskell from Control.Parallel.Strategies, refine use of the function rpar and rseq functions. There are tre basic operator in Eval Monad:*
  - *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 :*



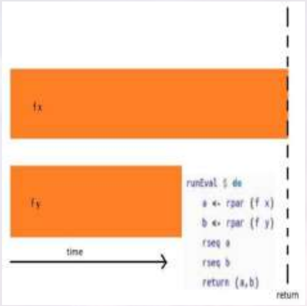
```
runEval $ do
    a <- rpar (f x)
    b <- rpar (f y)
    return (a,b)
```

```
runEval $ do
    a <- rpar (f x)
    b <- rseq (f y)
    return (a,b)
```

Figure 4: *rpar/rpar and rpar/rseq*

Eval Monad

- *The solution is:*



```
runEval $ do
    a <- rpar (f x)
    b <- rpar (f y)
    rseq a
    rseq b
    return (a,b)
```

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

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

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.*

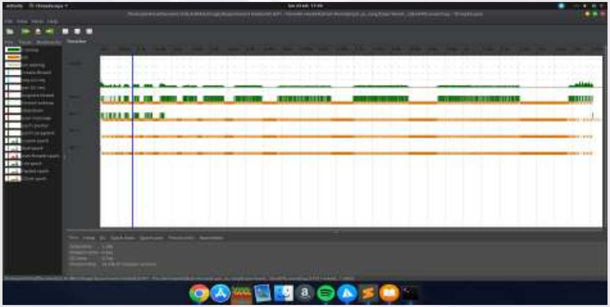


Figure 6: Threadscope tool

Strategies

- *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.*

Strategies

- *For example : for evaluating list in parallel*

ParList

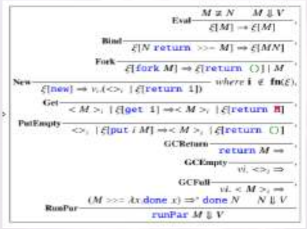
```haskell
parMap :: (a -> b) -> [a] -> [b]
parMap f xs = using (map f xs) (parList rseq)

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)
```

Par Monad

- *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 Monade Par offers three basic operations:*
  - *runPar* $\rightarrow$ *compute the operation in the Par Monad and return the result;*
  - *fork* $\rightarrow$ *create new Thread;*
  - *new* $\rightarrow$ *create new IVar;*
  - *get* $\rightarrow$ *get the result written in the IVar;*
  - *put* $\rightarrow$ *writes the result in the IVar.*



$$\textbf{Eval}\ \frac{M \not\equiv N \quad M \Downarrow V}{\mathcal{E}[M] \rightarrow \mathcal{E}[M]},$$

$$\textbf{Bind}\ \frac{}{\mathcal{E}[N\ \texttt{return} >\!\!>\!= M] \rightarrow \mathcal{E}[M\,N]},$$

$$\textbf{Fork}\ \frac{}{\mathcal{E}[\texttt{fork}\ M] \rightarrow \mathcal{E}[\texttt{return}\ ()] \mid M},$$

$$\textbf{New}\ \frac{}{\mathcal{E}[\texttt{new}] \rightarrow \nu_i.(<>_i \mid \mathcal{E}[\texttt{return}\ \texttt{i}])}\ \textit{where}\ \mathbf{i} \notin \mathbf{fn}(\mathcal{E}),$$

$$\textbf{Get}\ \frac{}{< M >_i \mid \mathcal{E}[\texttt{get}\ \texttt{i}] \rightarrow < M >_i \mid \mathcal{E}[\texttt{return}\ M]},$$

$$\textbf{PutEmpty}\ \frac{}{<>_i \mid \mathcal{E}[\texttt{put}\ i\ M] \rightarrow < M >_i \mid \mathcal{E}[\texttt{return}\ ()]},$$

$$\textbf{GCReturn}\ \frac{}{\texttt{return}\ M \rightarrow},$$

$$\textbf{GCEmpty}\ \frac{}{\nu i.\ <>_i \Rightarrow},$$

$$\textbf{GCFull}\ \frac{}{\nu i.\ < M >_i \rightarrow},$$

$$\textbf{RunPar}\ \frac{(M >\!\!>\!= \lambda x.\texttt{done}\ x) \Rightarrow^* \texttt{done}\ N \quad N \Downarrow V}{\texttt{runPar}\ M \Downarrow V}$$

Figure 7: Par Monad Semantics

Par Monad

Warning ☠

The IVar to ensure determinism can only be written once.

Determinism of Par Monad

We assume that runPar $M \Downarrow N$ and runPar $M \Downarrow N'$, then $N = N'$, and furthermore that if runPar $M \Downarrow N$ then there is no sequence of transitions starting from runPar M that reaches a state in which no reduction is possible. Informally, runPar M should either produce the same value consistently, or produce no value consistently (which is semantically equivalent to $\perp$). First, observe that the system contains no transitions that take a full IVar to an empty IVar, or that change the contents of a full IVar. A non-deterministic result can only arise due to a race condition: two different orderings of reductions that lead to a different result. To observe non-determinism, there must be multiple reduction sequences leading to applications of the rule (putEmpty) for the same IVar with different values. There is nothing in the semantics that prevents this from happening, but our determinism argument rests on the (runPar) rule, which requires that the state at the end of the reduction consists of only done M for some M. That is, the rest of the state has completed or deadlocked and been garbage collected by rules (GCReturn), (GCEmpty), (GCFull), and (GCDeadlock). In the case where there is a choice between multiple puts, one of them will not be able to complete and will remain in the state, and thus the runPar transition will never be applicable.

Soundness

If some subterm evaluates to $\perp$ during execution of runPar, then the value of the whole runPar should be $\perp$. The sequential scheduler implements this, but the parallel scheduler given above does not. In practice, however, it will make little difference: if one of the workers encounters a $\perp$, then the likely result is a deadlock, because the thread that the worker was executing will probably have outstanding puts. Since deadlock is semantically equivalent to $\perp$,the result in this case is equivalent. Nevertheless, modifying the implementation to respect the semantics is not difficult, and we plan to modify our real implementation to do this in due course.

Par Monad

- *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)
```

Par Monad

- *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:*

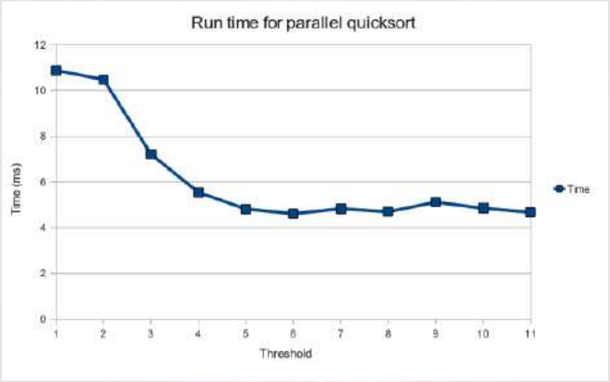


Figure 8: *Timeline ParQuickSortLimit*

Par Monad

- *In Threascope:*

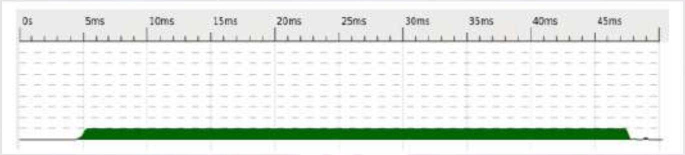


Figure 9: *Sequential quicksort as seen in Threadscope on quad-core CPU with 8-threads.*

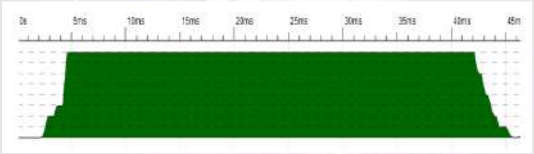


Figure 10: *Parallel quicksort as seen in Threadscope on quad-core CPU with 8-threads.*

Par Monad

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!

Parallel Type Inference Algorithm

- *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:*

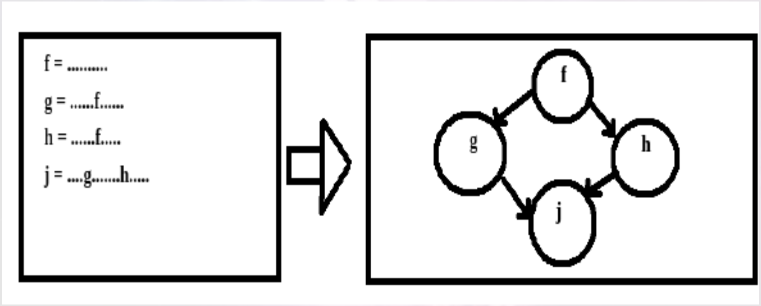


Figure 11: *Dataflow graph of bindings*

Parallel Type Inference Algorithm

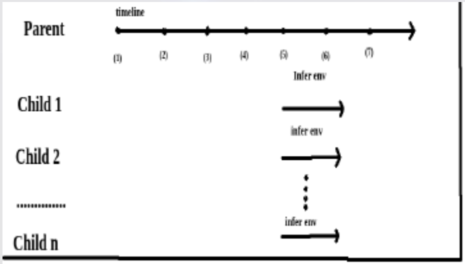
We define the function infer as follows:

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 -> parInfer [(var1,exp1),(var2,exp1),...,(var_n,exp_n)]

(2) let binders = map fst [(var1,exp1),(var2,exp1),...,(var_n,exp_1)] = [var1,var2,...,var_n] (forall elem in list apply fst(elem))

(3) ivars = replicateM (length [(var1,exp1),(var2,exp1),...,(varn,exp1)]) 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	Ivar1
...	...	...
n	var_n	Ivar_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)

Parallel Type Inference Algorithm

- *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).*

Parallel Type Inference Algorithm

- *In Threascope:*

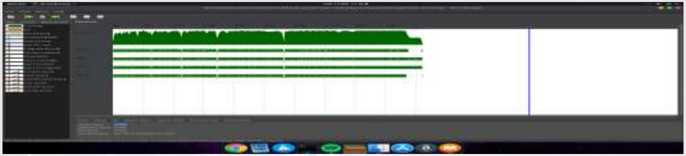


Figure 12: *ParInfer as seen in Threadscope on quad-core CPU with 8-threads.*

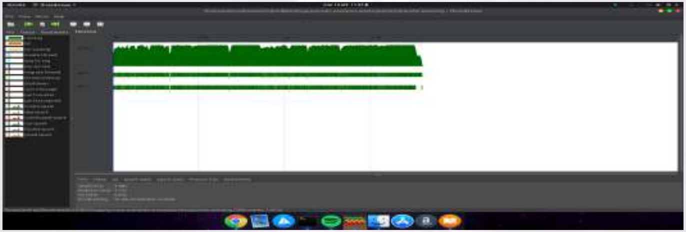


Figure 13: *ParInfer as seen in Threadscope on quad-core CPU limiting it to 2 cores*

Fine

GRAZIE A TUTTI PER L'ATTENZIONE