Università degli Studi di Torino

Dipartimento di Informatica

Scuola di Scienze Matematiche,
Fisiche e Naturali

Corso di Laurea Triennale in Informatica
Curriculum: Linguaggi e Sistemi

Strategie di Programmazione Parallela in Haskell

Relatore:
Ugo de'Liguoro

Candidato:
Andrea Senese

Anno Accademico 2019/2020

La scienza dei computer
non riguarda i computer
più di quanto l'astronomia
riguardi i telescopi.

Cit. *Edsger Wybe Dijkstra*

Abstract

Il seguente documento vuole
illustrare un'introduzione
alle monadi e illustrarne
i vari principi e tecniche
di programmazione parallela
in un linguaggio funzionale
lazy, facendo uso della Monade
Par e illustrando uno studio di
un caso per mostrarne l'applicazione
in Haskell. Al fine di comprendere il
contenuto di questo documento è richiesta
una buona conoscenza delle basi di Haskell,
accenni sulle forme normali in
lambda calcolo e semantica
operazionale one-step
e big-step.

Indice

Introduzione

Il seguente documento riporta inizialmente un piccolo ripasso di classi di tipo come Funtori, Applicativi e Monade di I/O. Questo perché propedeutici per la parte sul parallelismo. Dal secondo capitolo in poi vuole illustrare come in un linguaggio funzionale lazy è possibile realizzarlo attraverso l'uso delle Monadi Eval e Par. Tutto ciò per poter mostrare le principali tecniche/strategie per la buona programmazione parallela in Haskell ed eventuale analisi attraverso il tool Threadscope illustrato nella Sezione 2.2 del Capitolo 2. La fonte principale su cui si basa il documento è il libro [1].

Capitolo 1

Introduzione ai Funtori, Applicativi e Monade di I/O

1.1 Funtori

1.1.1 Funtori: definizione e uso

Un funtore in **Haskell** è una type class (classe di tipo) che può essere vista come un contenitore contenente una struttura dati su cui è possibile applicare una funzione senza modificare la struttura stessa. Quindi si può dire che in **Haskell** è definita come segue:

```haskell
map :: (a -> b) -> [a] -> [b] -- Restrittiva al solo tipo
 ↪  di dato []
-- Estendo map a qualsiasi struttura dati.
-- Definisco quindi la Functor type class.
class Functor f where
    fmap : (a -> b) -> f a -> f b
```

Dove definisco:

- f è il tipo di dati che dovrà essere incapsulato;
- `fmap` dovrà essere definito su quel tipo di dato.

Diciamo che `fmap` rispetta le seguenti leggi dei Funtori:

```haskell
  -- 1) Legge dell'Identità
  fmap id = id
  -- 2) Legge della Composizione
  fmap (f . g) = fmap f . fmap g
```

Questo permette di estendere `map` su un tipo di dato generico. Possiamo quindi utilizzare `map` come implementazione di `fmap` per liste :

```
class Functor [] where
    fmap = map
```

Esiste anche la notazione infissa per `fmap` che è identificata come <$>. Il modulo **Prelude** definisce per il tipo di dato **Functor** diverse istanze di tipi, ad esempio **Maybe** e **Either**. Entrambe sono considerate istanze di **Funtor** e definiti come segue:

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

Ora dato che **Maybe** e **Either** sono funtori significa che `fmap` è definito anche su di loro:

```
istance Functor Maybe where
    fmap f (Just x) = Just(f x)
    fmap f (Nothing) = Nothing

istance Functor Either where
    fmap f (Left a) = Left (f a)
    fmap f (Right b) = Right(f b)
```

Ora supponiamo di avere due funzioni. Cosa succede se applico una funzione ad un'altra funzione? Semplicemente ottengo una'altra funzione. Quindi:

```
istance Functor ((->) r) where
    fmap f g = f . g    -- (.) f g = f(g x)
```

Come si può vedere anche **IO** è un'istanza di **Functor**:

```
instance Functor IO where
fmap f action = do
    result <- action
    return (f result)
```

Ciò significa che un'azione di **IO** se eseguita da qualche esecutore, fa qualcosa e stampa `()` sulla bash. Si vedrà, più avanti, che **IO** è un'istanza di **Monad** nella sezione 1.3. Ora mostriamo un possibile esempio di applicazione di **Functor**.

1.1.2 Esempio di Applicazione di Funtori

Supponiamo di avere una lista di numeri interi e di volerli sommare fra di loro verificando che l'intero prodotto sia pari. Possiamo pensarla nel seguente modo:

```haskell
sumFunctors :: Integral a => Maybe [a] -> Maybe Bool
sumFunctors Nothing = Maybe True
                      -- La lista vuota produce somma 0
                      -- che è pari.
sumFunctors (Just xs) = fmap (even . sum) (Just xs)
                        -- fmap applica sum su xs
-- main
main :: IO ()
main = do
    (print . sumFunctors) (Just [1..10])
```

Ora considerando **Either** possiamo osservare la sua struttura come **Either** a b. Tutto ciò significa che **Haskell** sarà in grado di capire se dovrà tornare un tipo **a** o un tipo **b**:

```haskell
safeDivision :: Integral a => a -> a -> Either String
 ↪ String
safeDivision 0 _ = Left "Division by zero: error !!!!"
safeDivision n m | n < m = Left "The dividend is lower
                            than the divisor: error!"
                 | otherwise = Right (aux n m 0)
    where
        aux n m div | n < m = "Quotient: "
                              ++ show (fromIntegral n)
                              ++ " Division: "
                              ++ show (fromIntegral div)
                    | otherwise = aux (n - m) m (div + 1)

main :: IO ()
main = do
    (print . safeDivision 14) 3
```

Quindi **Either** può essere utilizzato per catturare eccezioni.

1.2 Applicativi

1.2.1 Applicativi: definizione e uso

Gli **Applicative** vengono considerati come un'astrazione in più rispetto a **Functor**. Possiamo vederli come una via di mezzo tra un **Functor** e **Monad**, quindi è una classe intermedia fra loro due. Fornisce un modo conveniente di strutturare i calcoli funzionali e fornisce anche mezzi per esprimere una serie di modelli importanti. Come abbiamo menzionato la classe **Functor** nella sezione 1.1, ora pensiamo al caso in cui vorremmo applicare una funzione di due argomenti tramite `fmap` su un tipo **Maybe**. Un modo possibile è forzare il tipo di dato **Maybe** estraendone il valore al suo interno e questo significherebbe fare dei controlli per verificare che non ci sia **Nothing** al suo interno. Questa però non è la strategia giusta, quindi si ha bisogno di un operatore del tipo `<*> :: f (a -> b) -> f a -> f b`. Questo operatore è parte della classe di tipo **Applicative**, definita come segue:

```
class Applicative where
  pure :: a -> f a
  <*> :: f (a -> b) -> a -> b

instance Applicative Maybe where
pure = Just
Nothing <*> _ = Nothing
(Just f) <*> something = fmap f something

instance Applicative (Either e) where
pure             =  Right
Left  e  <*>  _  =  Left e
Right f  <*>  r  =  fmap f r
```

Inoltre come si vede, nella funzione `pure` se applicatogli un argomento restituirà un **Functor** contenente quell'argomento. Come possiamo vedere anche **Maybe** e **Either** sono **Applicative**. Inoltre valgono le seguenti leggi su:

```
-- Legge dell'Identità:
pure id <*> v = v
-- Legge dell'Omomorfismo:
pure f <*> pure x = pure (f x)
-- Legge di Interscambio:
u <*> pure y = pure ($ y) <*> u
-- Legge di Composizione:
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```

Dove il significato delle seguenti leggi è il seguente:

- **La legge di Identità** afferma che il morfismo `pure id` non fà nulla, proprio come id.

- **La legge sull'Omomorfismo** afferma che applicare la funzione `pure` a un valore puro equivale ad applicare la funzione sul valore normalmente e applicarne `pure` sul valore risultante.
- **La legge di Interscambio** afferma che applicare un morfismo a un valore puro (`pure y`) è lo stesso che applicare `pure ($ y)` al morfismo.
- **La legge di Composizione** dice che la composizione `pure (.)` compone morfismi in modo simile a come compone le funzioni `(.)`: ossia applicare il morfismo composto `(pure (.) <*> u <*> v)` a **w** dà lo stesso risultato di applicare il risultato dell'applicazione di **u** al risultato dell'applicazione di **v** a **w**.

Esiste un'ulteriore legge che mette in relazione `fmap` e `<*>`:

```
1     map f x = pure f <*> x --fmap
```

La legge per `fmap` ci dice semplicemente che mappare la funzione `f` su **x** è la stessa cosa di applicare `pure f` che ci darà un `Functor f` di cui la funzione `<*>` permetterà di applicare `f` su **x**.

Inoltre diciamo che anche `IO` e le funzioni sono `Applicative`.

```
1     --Funzioni
2     instance Applicative ((->) r) where
3     pure x = (\_ -> x)
4     f <*> g = \x -> f x (g x)
5
6     --IO
7     instance Applicative IO where
8     pure = return
9     a <*> b = do
10        f <- a
11        x <- b
12        return (f x)
```

Ora consideriamo dei possibili esempi di applicazione degli `Applicative`.

1.2.2 Esempio di applicazione di Applicativi

Ora vedremo come usare gli **Applicative**:

```haskell
    -- Per ogni elem di lista1 effettua la moltiplicazione
    -- con lista2
    app = (*) <$> [1..10] <*> [2..20]

    -- Per ogni elem di lista1 effettua la divisione con lista2
    -- dopodichè filtra tutti i numeri positivi
    app1 = filter (> 0) $ div <$> [1..10] <*> [2..10]

    -- Quindi si può scrivere una funzione che
    -- moltiplica ogni elemento di xs con con tutti gli
    -- elementi in ys
    -- in questo modo :
    multListApp :: Integral a => [a] -> [a] -> Either String
     ↪  [a]
    multListApp [] _ = Left "Nothing"
    multListApp _ [] = Left "Nothing"
    multListApp xs ys = Right ((*) <$> xs <*> ys)

    --Il seguente invece è un esempio per IO:
    action :: IO String
    action = (++) <$> getLine <*> getLine

    main :: IO ()
    main = do
        print app
        print app1
        (print . multListApp [1..3]) [1..3]
```

Come possiamo vedere gli **Applicative** non bastano, in quanto per effettuare operazioni di I/O in un linguaggio funzionale lazy si ha bisogno della **Monade I/O** perchè in **Haskell** quando si scrive una funzione, questa non ha nessun side effect e quindi si ha bisogno di effettuare quest'operazione nella parte "impura" di Haskell.

1.3 Monade I/O

1.3.1 Input e Output in Haskell

Iniziamo con un problema davvero fondamentale. Un programma puramente funzionale implementa funzioni; ed esso stesso non ha effetti collaterali. Tuttavia, lo scopo finale di un programma è sempre quello di causare effetti collaterali: un file modificato, alcuni nuovi pixel sullo schermo, un messaggio inviato o altro. L'I/O è una funzionalità fondamentale offerta dai programmi; un programma senza I/O non sarebbe molto utile. Se l'effetto collaterale non può essere causato nel programma funzionale, allora dovrà essere causato al di fuori di esso. Con ulteriori studi sull'uso delle monadi si è arrivati alla Monade I/O. L'idea è la seguente: "un valore di tipo `IO a` è un azione che se eseguita da un qualche esecutore produce un'azione di I/O e fornisce un valore di tipo "**a**". Un modo più concreto per vedere queste "azioni" è :

```
1   type IO a = World -> (a, World)
```

Questa definizione di tipo dice che un valore di tipo `IO` a è una funzione che, quando applicata a un argomento di tipo `World`, offre un nuovo `World` insieme a un risultato di tipo **a**. L'idea è la seguente: "Il programma prende come input lo stato di tutto il mondo e, di conseguenza, offre un mondo modificato dagli effetti dell'esecuzione del programma". Alcune operazioni familiari di I/O possono essere le seguenti:

```
1   getChar :: IO Char
2   putChar :: Char -> IO ()
```

dove `getChar` è un'azione I/O che, quando eseguita, legge un carattere dallo standard input (avendo quindi un effetto sul mondo esterno al programma) e lo restituisce al programma come risultato dell'azione `getChar`. Mentre `putChar` è una funzione che accetta un carattere e restituisce un'azione che, quando eseguita, stampa il carattere sull'output standard (ossia è il suo effetto sul mondo esterno) e restituisce il valore `()`. Per concatenare più azioni usiamo l'operatore **bind** denotato nel seguente modo: `(>>=) :: IO a -> (a -> IO b) -> IO b`. Più precisamente, il combinatore `(>>=)` implementa la composizione sequenziale: passa il risultato dell'esecuzione della prima azione alla seconda azione (come parametro). Quindi data un'azione **a** `>>=` **f** significa che, eseguita l'operazione **a** il suo output sarà l'input dell'operazione **f**. Supponiamo di voler eseguire due azioni sequenziali dove per eseguire la seconda operazione non si ha bisogno del risultato della prima azione, allora definiamo un secondo combinatore nel seguente modo:

```
1       (>>) :: IO a -> IO b -> IO b
2       (>>) a b = a >>= (\x -> b)
```

che chiameremo **"then"** che, dopo aver consumato il primo argomento **a**, restituirà il secondo argomento **b**.
Definiamo un ultimo combinatore : `return :: a -> IO a`, detto **"iniettore"**, che non è un azione di I/O ossia restituisce il parametro applicatogli senza produrre alcun side effect.

Bisogna anche notare che (>>=) e (>>) non sono commutativi, ossia eseguire `q >>= p` è diverso da eseguire prima `p` e poi `q`. Un modo più concreto di vedere la Monade IO è il seguente:

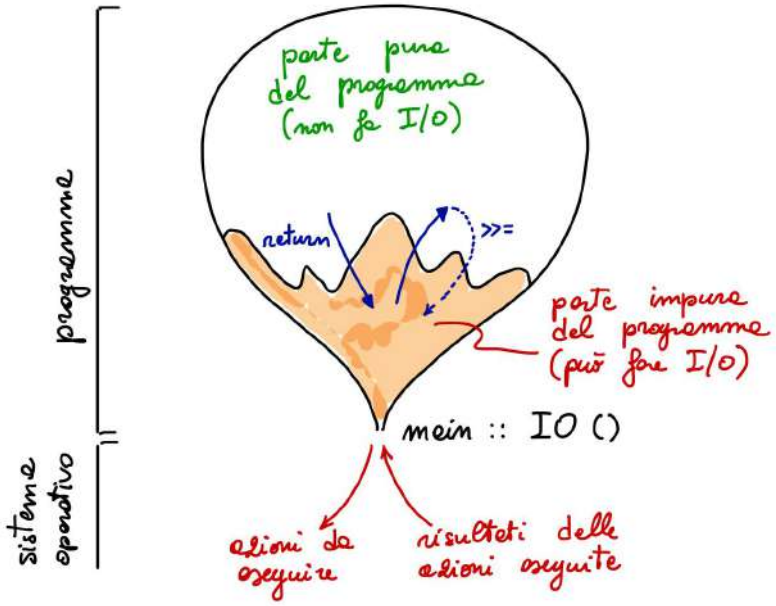

Figura 1.1: Monade IO

Come possiamo vedere nella Figura 1.1, possiamo distinguere il programma in due parti:

- Parte pura: dove eseguiamo funzioni e quindi non c'è nessun side effect (non fa I/O);
- Parte impura: dove eseguiamo tutte le azioni che producono side effect e quindi I/O. Inoltre come possiamo vedere, le operazioni da eseguire non sono eseguite da Haskell stesso ma dal Sistema Operativo.

1.3.2 Leggi Equazionali delle Monadi

Tutte le istanze della classe **Monad** sono soggette alle seguenti leggi:

```
1    return a >>= f  = f a -- legge di identità a sinistra
2    a >>= return f  = f a -- legge di identità a destra
3    (m >>= f) >>= g  =  m >>= (f >>= g) -- legge di
      ↪ associatività
```

Dove $\alpha = \beta$ significa semplicemente che si può sostituire α con β e viceversa, e il comportamento del programma non cambierà: α e β sono equivalenti.

1.3.3 Esempio di applicazione della Monade I/O

Un possibile esempio è il seguente:

```
1    printListInteger :: Integral a => [a] -> IO ()
2    printListInteger [] = return ()
3    printListInteger (x : xs) = printAction [1..10]
4      where
5         printAction :: (Show a, Integral a) => [a] -> IO ()
6         printAction [] = return ()
7         printAction (x : xs) = print x >> printAction xs
```

Come possiamo vedere, per stampare una lista sulla bash non si ha bisogno di ricordare l'azione precedente e quindi è conveniente usare `(>>)`. Per approfondire la tematica consultare l'articolo [2].

Capitolo 2

Parallelismo in Haskell

2.1 Parallelismo

2.1.1 Breve Introduzione al Parallelismo

Spesso si sente parlare di **Parallelismo** e **Concorrenza** perché in molti campi sono sinonimi, ma non in programmazione. Un **programma parallelo** è un programma che utilizza una molteplicità di hardware computazionali (ad esempio: diversi core del processore) per eseguire più rapidamente un calcolo, delegando ad ogni **CPU** una diversa parte del calcolo, in modo che il compito venga compiuto da diversi **CPU** contemporaneamente. Un **programma concorrente** invece, è una tecnica di strutturazione del programma in cui i **thread** vengono eseguiti attraverso un **interleaving** delle azioni, cioè viene eseguito un **thread** alla volta a cui verrà assegnata ad ogni **thread** la **CPU** per eseguire la propria computazione secondo delle politiche scelte dallo **scheduler** (qui nasce il non determinismo). A differenza di quanto accade nella **programmazione parallela** però, nella **programmazione concorrente** i **thread** eseguono compiti differenti, in quanto la **concorrenza** si occupa di strutturare il programma in modo da poter interagire con più fattori esterni indipendenti (ad esempio un server di un Database, soddisfare le richieste dei clienti e così via). La nozione di **thread** non ha più senso in **programmazione puramente funzionale**, in quanto non si ha il concetto di **side effect** e **l'ordine di valutazione** è irrilevante. Per riassumere, si può dire che la **concorrenza** è un modello di programmazione **non deterministico**, ovvero ammette risultati diversi ad ogni esecuzione, mentre il **parallelismo** è un modello di **programmazione deterministico**, ovvero ammette un solo risultato. I modelli di **programmazione concorrente** devono necessariamente essere **non deterministici**, poiché devono interagire con agenti esterni che possono causare eventi in momenti imprevedibili. La **programmazione deterministica parallela** è ottima perché **test**, **debug** e **ragionamenti** possono essere eseguiti direttamente sul programma sequenziale in modo che dopo si aggiungano solamente più **CPU** per rendere il programma più veloce. **Concorrenza** e **Parallelismo** possono anche essere messi in relazione: ad esempio, la **concorrenza** si può usare per mantenere un'**interfaccia utente reattiva** mentre le attività ad alta intensità di calcolo vengono eseguite in **background**.

2.2 Threadscope Tool: Parallel Program Analysis

2.2.1 Threadscope Tool e funzionalità offerte

ThreadScope è stato originariamente creato per aiutare i programmatori a visualizzare l'esecuzione parallela dei programmi **Haskell** compilati con il **Glasgow Haskell Compiler** o **GHC**. L'idea è che durante l'esecuzione di un programma **Haskell** parallelo, il sistema di runtime **Haskell** scrive rapporti con data e ora relativi a eventi significativi in un file di log. Lo strumento **ThreadScope** legge successivamente questo file di log, e mostra graficamente all'utente cosa stava facendo ogni **CPU** in quell'intervallo di tempo. I diagrammi che mostra rivelano al programmatore i punti in cui il programma sta utilizzando del **parallelismo**, nonché i luoghi in cui non lo sta usando. **ThreadScope** è estendibile anche ad altri linguaggi come ad esempio **Mercury**: si legga l'articolo [3]. La maggior parte degli eventi nel file di registro sono associati a un **motore**. Diciamo che un **motore** è un costrutto basato sulla **continuazione** (è una rappresentazione astratta dello stato di controllo di un programma per computer, che fornisce prelazione temporizzata). Per evitare di dover includere un **ID** del motore con ogni evento nel file di log, il formato del file di log raggruppa sequenze di eventi tutti uguali che vengono tutti dallo stesso **blocco motore** e scrive l'**ID** del motore solo una volta, in un **blocco di header di pseudo-eventi**. Dal momento in cui il **tool** leggerà i file di log ricorderà banalmente l'**ID** del motore nell'ultimo **blocco header** che ha letto, questo rende l'attuale **ID** del motore (l'**ID** del motore nel cui blocco appare un evento) un parametro implicito della maggior parte dei tipi di eventi. Questo **tool** cattura **eventi** di diversi tipi, i fondamentali sono:

- **STARTUP**: marca l'inizio dell'esecuzione del programma e registra il numero di motori che il programma utilizzerà;
- **SHUTDOWN**: marca la fine dell'esecuzione del programma;
- **CREATE THREAD**: registra l'atto del motore corrente creando un contesto e fornisce l'**ID** del contesto che viene creato. Gli ID di contesto vengono allocati in sequenza, ma il parametro non può essere omesso, poiché il sistema di runtime potrà riutilizzare la memorizzazione di un contesto dopo la fine della computazione che ha effettuato in precedenza;
- **RUN THREAD**: registra gli eventi di commutazione di scheduling del motore corrente per eseguire un contesto e fornisce l'**ID** del contesto in cui passerà.
- **STOP THREAD**: registra l'evento di pianificazione del motore corrente che passa dall'esecuzione di un contesto. Fornisce l'**ID** del contesto da cui si passa, nonché il motivo del passaggio. le possibili ragioni includono:
  a. **L'heap** è pieno e quindi il motore deve invocare il **garbage collector**;
  b. il contesto è bloccato;
  c. il contesto è terminato.
- **THREAD RUNNABLE**: registra che il motore corrente ha reso eseguibile un contesto bloccato, e restituisce l'**ID** del contesto nuovamente eseguibile;

- **RUN SPARK**: registra che il motore corrente sta iniziando a eseguire uno spark che ha recuperato dalla propria coda locale di spark. Dopodiché fornirà l'**ID** del contesto che eseguirà lo spark, sebbene ciò potrà essere dedotto dal contesto;
- **STEAL SPARK**: registra che il motore corrente eseguirà uno spark che ha rubato dalla coda degli spark di un altro motore. Fornirà poi l'**ID** del contesto che eseguirà lo spark (anche se, questo potrà essere nuovamente dedotto dal contesto). Fornisce anche l'**ID** del motore da cui è stata rubato lo spark;
- **CREATE SPARK THREAD**: registra la creazione di un nuovo contesto e il riuso di un vecchio contesto per eseguire uno **spark**. Fornirà poi l'**ID** del contesto. Però non segnalerà se il contesto è nuovo o riutilizzato. Nella maggior parte dei casi, tali informazioni non sono necessarie, ma se lo sono, possono essere dedotte dal contesto;
- **GC START**: il motore corrente ha avviato il **Garbage Collector**; il controllo del motore è stato preso dal **mutatore**;
- **GC END**: il **Garbage Collector** ha terminato sul motore corrente; il controllo del motore è stato restituito al **mutatore**.

In questo documento verrà usato il seguente tool per la visualizzazione dell'esecuzione di programmi **Haskell** e per ottenere informazioni dettagliate sul comportamento del nostro codice parallelo per poter mostrare quando una determinata tecnica di programmazione parallela è migliore di un altra. Perchè dal tool possiamo vedere realmente cosa accade durante l'esecuzione e potrebbe mostrare cosa succede in un determinato punto del programma che potrebbe suscitare dubbi. La seguente immagine sotto riportata, riporta l'interfaccia di **Threadscope**:

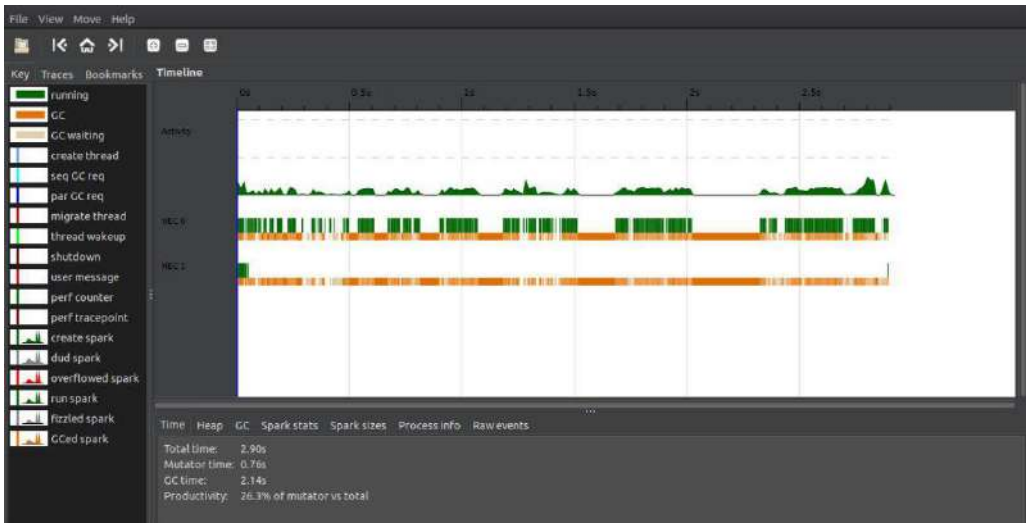

Figura 2.1: ThreadScope Tool

Come si può vedere nella Figura 2.1, l'asse delle x identifica il tempo e si hanno tre barre orizzontali che mostrano come viene eseguito il programma nel tempo. La barra più in alto è conosciuta come "profilo delle attività" e mostra su quanti core è eseguito il codice **Haskell**

in un determinato istante. Sotto il profilo delle attività è presente una barra per ogni core, che mostra cosa stava facendo quel core in ogni punto dell'esecuzione. Ogni barra ha due parti:

- La parte superiore, molto spesso, è verde quando quel core esegue codice **Haskell**;
- La parte inferiore (la barra più stretta) è verde o arancione. Identifichiamo il colore arancione per segnalare l'entrata in scena del garbage collector, che permette di liberare porzioni di memoria non più utilizzata dall'applicazione.

Come si può vedere, il lavoro non è distribuito uniformemente su entrambe i core perché **HEC1** per la maggior parte dell'esecuzione è inattivo è c'è solo pulizia del **Garbage Collector**. Quindi questo significa che le nostre due attività parallele non sono uniformi: un operazione richiede molto più tempo dell'altra. Quindi non si stanno sfruttando a pieno i due core e questo si traduce in uno speedup tutt'altro che perfetto. Nella Sottosezione 3.1.3 mostreremo il perchè e come risolvere questa situazione con l'applicazione di questi strumenti.

2.3 Valutazione lazy

Haskell è un linguaggio pigro (lazy), e questo significa che le espressioni non verranno valutate se non espressamente richiesto. Quindi si illustrerà brevemente con un esempio come funziona la valutazione pigra per mostrare come trarre vantaggi per il parallelismo.

2.3.1 Normal Form

Si consideri la seguente espressione:

```
C:\Users\andysinx>ghci
    GHCi, version 8.6.3: http://www.haskell.org/ghc/ :? for help
    Prelude> let x = 3 * 2 + 5 :: Int
    Prelude> :sprint x
    x = _
    Prelude> seq x "valutazione forzata"
    "valutazione forzata"
    Prelude> :sprint x
    x = 11
```

Quando in **Haskell** si memorizza un'espressione non si ha modo di sapere se sia stata valutata del tutto, quindi introduciamo il comando `sprint` che permette di sapere se il comando è stato valutato. Come si può vedere ci viene detto `x = _`, che significa che l'espressione al suo interno non è stata valutata, onde **ghci** identifica l'argomento non valutato (l'entità non valutata in memoria) con il simbolo _. Chiameremo questa entità **thunk**. Dopodiché verrà detto ad **Haskell** di valutare il suo argomento tramite la funzione `seq x y`, onde gli verrà applicato l'argomento da valutare e stamperà sulla bash y. Ora se proviamo a chiedere se l'argomento è stato valutato, **ghci** ne darà il valore perchè gli è stato richiesto di valutare quell'espressione. Si così arriva al punto di definire le forme normali in lambda calcolo, in quanto in Haskell sono molto rilevanti:

Definizione 2.1 (Normal Form) *Diciamo che M è in forma normale se non esiste un N tale per cui $M \to N$.*

Definizione 2.2 (Head Normal Form) *Un'espressione in HNF è la stessa della NF, ossia non ha thunk al suo interno.*

Definizione 2.3 (Weak Head Normal Form) *Un'espressione è in WHNF quando la parte più esterna è stata valutata. Però le sottoespressioni potrebbero non essere state valutate. Se tutte le sottoespressioni fossero valutate, l'espressione sarebbe in forma normale. La conclusione è: ogni espressione in forma normale è anche in WNF. Il contrario non è sempre vero.*

Definizione 2.4 (β-Normal Form) *Un'espressione è in β-NF quando non è possibile applicare riduzioni beta.*

Inoltre si può dire che la forma normale è unica, in quanto **Haskell** è un **linguaggio funzionale** e quindi implica **determinismo**, ossia l'assenza di **side effect**. Il seguente esempio mostra **l'unicità** della **Normal Form**:

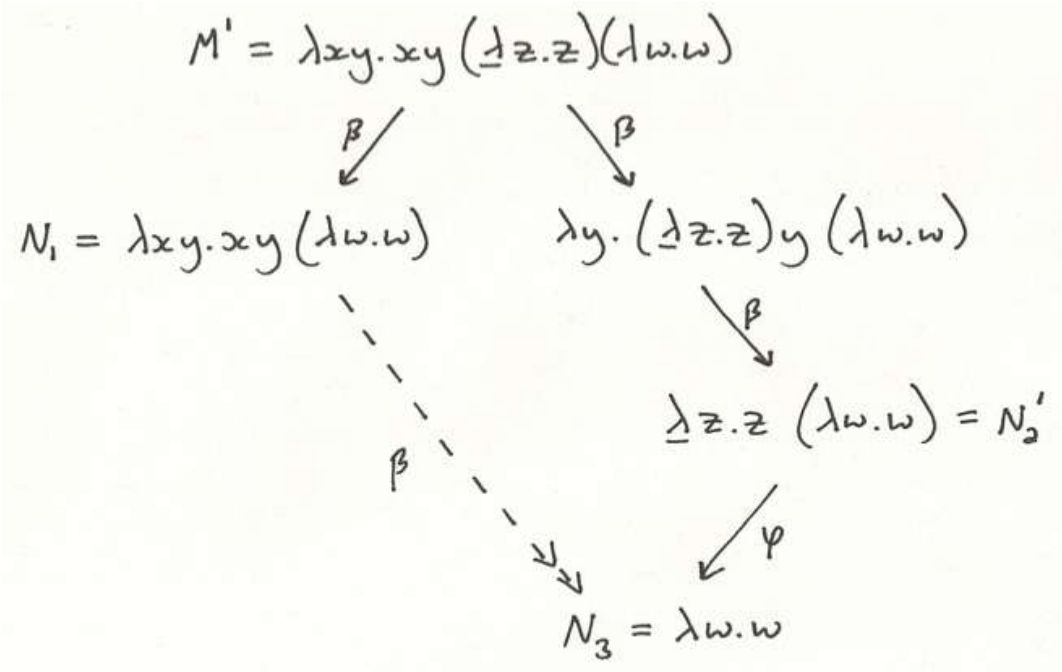

Figura 2.2: Unicità della Normal Form

2.3.2 Esempi di Normal Form

Ad esempio sulle seguenti espressioni si può affermare che:

```haskell
 -- Esempi di NF:
 x = 7 -- è in NF, ossia non si riduce più.
-- In generale Haskell non fa uso della HNF,
-- a causa della sua pigrizia. Ma HNF
-- può essere ottenuto con un po' di trasformazioni
-- all'interno del corpo della funzione.

 --Esempi di WHNF:
 fun x = x + 1
 -- è in WNF perché non ha thunk al suo interno.
 3 : 4 : [] = [3,4]
 -- è in NF, perché non ha thunk al suo interno.

 fun = let x = map (+1) [1..10] :: [Int] -- x = _
 --se forzo la valutazione di x con seq x y
 tmp = seq x "valutazione forzata"
 -- :sprint p |->  x = [_]
 -- (lo valuta solo fino al primo costruttore)
 -- posso usare una length che permette
 -- tramite la ricorsione di coda di
 -- determinarne la lunghezza
 -- e quindi :sprint x == [_,_,_,_,_,_,_,_,_,_]
 -- ora utilizzando sum potrò sapere per
 -- ogni thunk nella lista il proprio valore,
 -- perchè la somma  di tutti gli elementi della lista
 -- richiede di saperne il loro valore
 -- e quindi :sprint x == [1,2,3,4,5,6,7,8,9,10].

 --Esempio di Beta-NF:
 f z = (\y -> y + x) z -- è in beta-NF
 -- perchè con un passo di beta reduction
 -- si ha che z + x dove z € N e x non si
 -- riduce più (non ha nessun binder che
 -- glielo permette).

```

Capitolo 3

Monade Eval

3.1 Funzionalità di base della Monade Eval

3.1.1 rseq e rpar

Ora verranno introdotte le funzionalità per creare il parallelismo fornite dal modulo `Control.Parallel.Strategies`:

```
1      data Eval a
2      instance Monad Eval
3      runEval :: Eval a -> a
4      rpar:: a -> Eval a
5      rseq :: a -> Eval a
```

il parallelismo viene espresso usando la `Monad Eval` che fornisce tre operazioni di base:

- `runEval`: esegue una computazione Eval e ne restituisce il risultato;
- `rpar`: viene usato per creare il parallelismo;
- `rseq`: viene usato per forzare la valutazione in modo sequenziale e ne attende il risultato della computazione.

Ora si mostrerà l'analisi degli effetti di `rpar` e `rseq`. Si supponga di voler calcolare in parallelo l'applicazione di una data funzione a due determinati argomenti per poter analizzare il tempo di calcolo:

```
1          fun :: (a -> b) -> a -> a -> (b, b)
2          fun f x y = runPar $ do
3             a <- (rpar . f) x
4             b <- (rpar . f) y
5             return (a,b)
```

Come si può vedere l'applicazione di f su x impiega molto più tempo nella computazione del risultato rispetto all'applicazione di f su y:

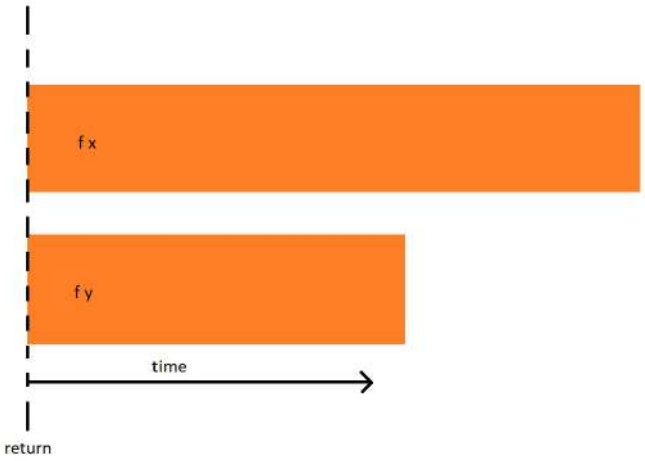

Figura 3.1: rpar/rpar execution

Come si può vedere in Figura 3.1, inizia la valutazione in parallelo e la `return` avviene immediatamente perché come si può vedere `rpar` non si mette in attesa del risultato. Si provi ora a sostituire la seconda `rpar` con `rseq`:

```
1          fun :: (a -> b) -> a -> a -> (b, b)
2          fun f x y = runPar $ do
3             a <- (rpar . f) x
4             b <- (rseq . f) y
5             return (a,b)
```

L'esecuzione è riportata nella Figura sottostante:

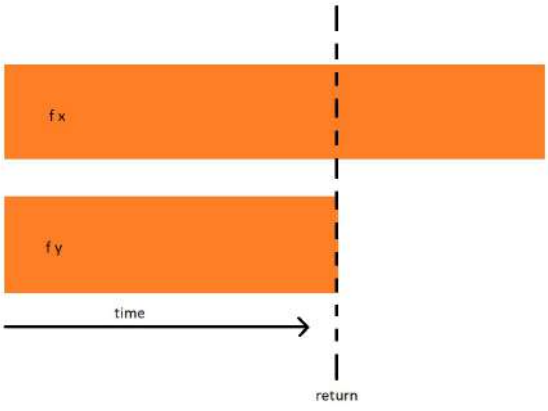

Figura 3.2: rpar/rseq execution

Come si può vedere in Figura 3.2, inizia la valutazione in parallelo di f su x e la valutazione sequenziale di f su y . Come abbiamo detto in precedenza, `rseq` attende la valutazione totale del suo argomento quindi la `return` verrà eseguita dopo che sarà terminata la valutazione sequenziale di f su y. Quindi ora si può dedurre che se vorremmo che la funzione `return` venga eseguita alla fine della valutazione di entrambe le operazioni, bisognerà segnalare attraverso `rseq` che sarà necessario attendere anche la terminazione della valutazione su x:

```
1          fun :: (a -> b) -> a -> a -> (b, b)
2          fun f x y = runPar $ do
3             a <- (rpar . f) x
4             b <- (rseq . f) y
5             rseq a
6             return (a,b)
```

Quindi come si può vedere, in Figura 3.3:

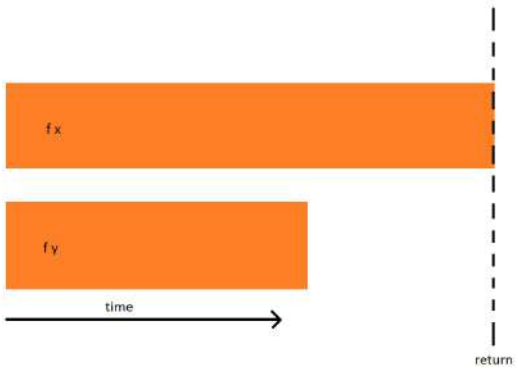

Figura 3.3: rpar/rseq/rseq execution

Ora potrete chiedervi: "quale di questi schemi è il migliore da utilizzare?"

- Figura 3.1: si può dire che non è molto utile, perché aspettare arbitrariamente una delle due operazioni è insensato. Non possiamo sapere a priori quale delle due impiega più tempo.
- Figura 3.2: utile nel momento in cui bisogna generare più parallelismo possibile senza dipendere dal risultato finale.
- Figura 3.3: utile nel momento in cui abbiamo generato tutto il parallelismo possibile e abbiamo bisogno del risultato di una delle due operazioni.

Infine si definisce un'ultima variante, che risulta essere quella più lunga, ma essendo più simmetrica delle altre risulta la scelta più accurata:

```
1      fun :: (a -> b) -> a -> a -> (b, b)
2      fun f x y = runEval $ do
3       a <- rpar (f x)
4       b <- rpar (f y)
5       rseq a
6       rseq b
7       return (a,b)
```

che risulta avere lo stesso comportamento di rpar/rseq/rseq in Figura 3.3, ossia prima di ritornare il risultato si è in attesa della terminazione di entrambe le operazioni. Quindi diciamo che l'argomento applicato a `rpar` è chiamato **spark**. Si può immaginare che esista uno **spark core** che è un "motore di calcolo" responsabile della pianificazione, distribuzione e monitoraggio di applicazioni costituite da molte attività di calcolo su molte macchine di lavoro o un cluster di elaborazione e tramite la `rpar` possiamo assegnargli uno **spark** (cioè una data attività da eseguire).

3.1.2 Deepseq

Come si può vedere, `rseq` valuta in WHNF e si vuole definire una versione che valuta in NF. Definiamo la funzione `force :: NFData a => a -> a` che permette di valutare il suo argomento in NF e successivamente ritornarlo. Il suo comportamento è definito per ciascun tipo di dato tramite la classe `NFData` che sta per "tipo di dati in NF". La classe `NFData` è definita come:

```
1        class NFData a where
2          rnf :: a -> ()
3          rnf a = seq a ()
```

dove `rnf` sta per "reduce to normal form" che valuta il primo argomento e stampa `()` sulla bash. Quindi ora si piò definire `deepseq` come segue:

```
1        deepseq :: NFData a => a -> b -> b
2        deepseq x y = (rnf . seq) x y
```

ed ora si ha tutto il necessario per definire la funzione `force`:

```
1        force :: NFData a => a -> a
2        force x  = deepseq x x
```

Quindi si può immaginare la funzione `force` come una trasformazione da WHNF a NF. Questo significa che valutare una struttura di taglia n implica che si avrà un costo computazionale di $O(n)$ mentre con `seq` si avrebbe un costo di $O(1)$.

3.1.3 Esempio: Fibonacci Stream lazy

Ora verrà mostrato l'uso di `rpar` e `rseq` su due versioni alternative della funzione che calcola i numeri di **Fibonacci**. La prima versione utilizza uno `Stream` definito tramite l'uso della lazyness di **Haskell** mentre l'altra è la funzione tradizionale per il calcolo della sequenza di Fibonacci. Si supponga quindi che il seguente codice calcoli i numeri di fibonacci tradizionalmente:

```
1     import Control.Parallel
2     import Control.Parallel.Strategies
3     import Control.Exception
4     import Data.Time.Clock
5     import Text.Printf
6     import System.Environment
7
8     fib :: Integer -> Integer
9     fib 0 = 1
10    fib 1 = 1
11    fib n = fib (n-1) + fib (n-2)
12
13    fibStream :: Integer -> Integer
14    fibStream x | x < 0 = error (show x ++ " is not valid") |
15                | x == 0 = 1
16                | otherwise =  (last . fibAux . (-) x)  1
17    where
18          fibAux x | x == 0 = []
19                   | otherwise = 0 : 1 : zipWith (+) ((fibAux
                      ↪  . (-) x) 1) ((tail . fibAux . (-) x) 1)
20    main :: IO ()
21    main = do
22        start <- getCurrentTime
23        [n] <- getArgs
24        let test = [test1,test2,test3,test4] !! (read n - 1)
25        startTest <- getCurrentTime
26        r <- evaluate (runEval test)
27        endTimeTest <- getCurrentTime
28        print "result: "
29        print r
30        t2 <- getCurrentTime
31        printf "Elapsed time : %.6fs\n" (realToFrac
           ↪  (diffUTCTime endTimeTest startTest) :: Double)
32        end <- getCurrentTime
33        printf "Total time: %.6fs\n" (realToFrac (diffUTCTime
           ↪  end start) :: Double)
```

Lo scopo è analizzare l'uso dei seguenti strumenti in un esempio concreto con l'aiuto del tool **Threadscope**. Si vuole mostrare dove il parallelismo garantisce di ottenere uno speedup migliore:

```
1     -- << test1
2     test1 = do
3           a <- rpar(fib 20)
4           b <- rpar(fib 30)
5           rseq a
6           rseq b
7           return (a,b)
8     -- >>
9
10    -- << test2
11    test2 = do
12          a <- rpar(fibStream 25)
13          b <- rpar(fibStream 30)
14          rseq a
15          rseq b
16          return (a,b)
17    -- >>
```

Come possiamo vedere, la lazyness permette di aumentare le prestazioni del programma parallelo e si ha la seguente situazione (su 2 core):

```
1 root@andysinx-dellOptiplex980:/home/andrea/Documenti/Par_Haskell
2     > ghc -O2 EvalFib.hs -threaded -rtsopts -eventlog
3     [1 of 1] Compiling Main             ( EvalFib.hs, EvalFib.o )
4     Linking EvalFib ...
5     root@andysinx-dellOptiplex980:/home/andrea
6     /Documenti/Par_Haskell> ./EvalFib 1 +RTS -N2 -l
7     "result:" (46368,514229)
8     Elapsed time : 1.315589s
9     Total time: 1.315983s
```

Come possiamo vedere si ha uno **speedup** all'incirca di:

$$Speedup = \frac{time_{total}}{time_{elapsed}} \tag{3.1}$$

$$= \frac{1,315983s}{1,315589s} \tag{3.2}$$

$$\simeq 1s \tag{3.3}$$

Inoltre il sistema di runtime **GHC** segnala anche quanti **spark** sono stati creati:

- **Sparks**: n (m converted, i overflowed, a dud, j GC'd, z fizzled)
  dove $n, m, i, a, j, z \in \mathbb{N}$;

onde definiamo:

- **Overflowed**: l'insieme degli **spark** ha dimensioni fisse e se proviamo a creare altri **spark** quando l'insieme è pieno, vengono rilasciati come **overflowed**;
- **Dud**: Quando `rpar` viene applicato a un espressione già valutata questa viene valutata come " a dud" e viene ignorata;
- **GC'd**: l'espressione generata è inutilizzata dal programma, quindi GHC rimuove lo **spark**;
- **Fizzled**: l'espressione non è valutata al momento dell'applicazione alla `rpar`, quindi i fizzled spark vengono rimossi dall'insieme degli spark.

Il seguente **eventlog** file mostra l'esempio riportato su:

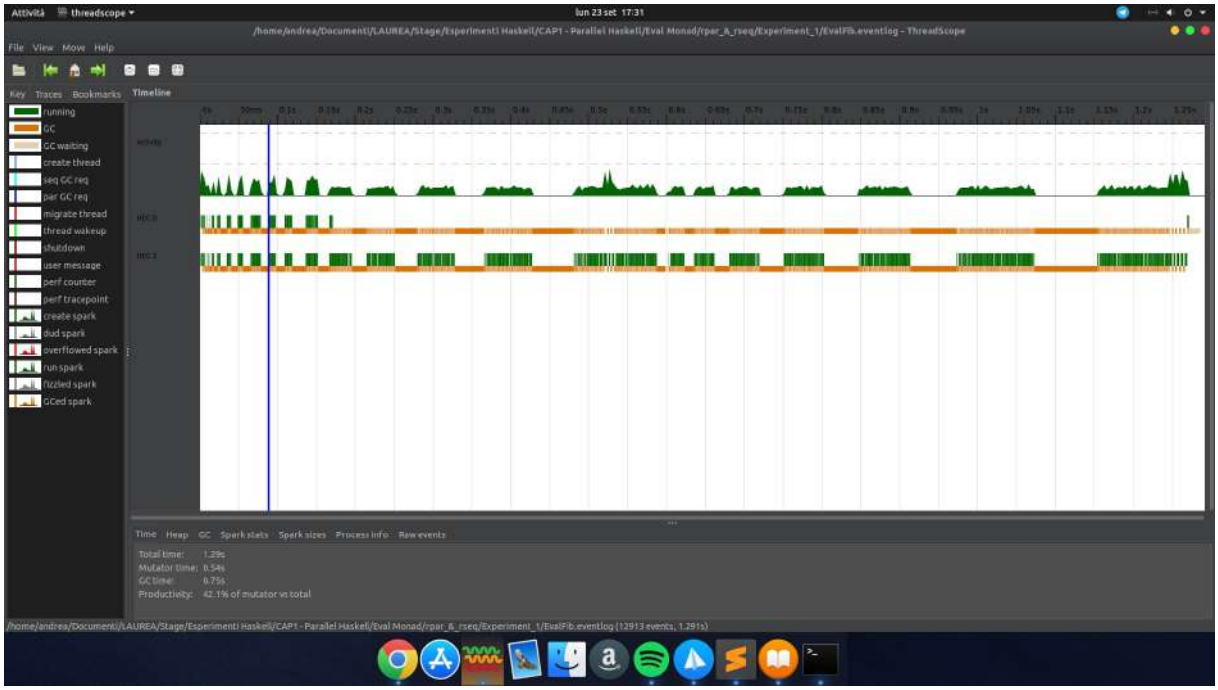

Figura 3.4: Test 1: Fibonacci rpar/rpar/rseq/rseq execution su 2 core

Come si può vedere in Figura 3.4, inizialmente il programma è parallelo dopodiché c'è un punto in cui lavora un solo processore fino al termine del programma. Da 0.09ms a 17.1s, il processore HEC0 smette di lavorare e il programma diventa sequenziale. Allora ora ci chiediamo: "Come è possibile calcolare il massimo speedup raggiungibile con un

determinato numero di processori?". Questo è possibile applicando la **legge di Amdahl** che ci darà il massimo speedup definito come il rapporto:

$$Amdahl_{law} = \frac{1}{\frac{(1-P)+P}{N}} \tag{3.4}$$

$$= (\frac{(1-P)}{N})^{-1} \tag{3.5}$$

$$= (\frac{(1-(1,315589s-1.20s))}{2})^{-1} \tag{3.6}$$

$$= 2.2613920451s \tag{3.7}$$

$$\simeq 2s \tag{3.8}$$

dove P è il tempo che impiega la porzione di codice che può essere parallelizzata, ed N è il numero dei processori disponibili. Il tempo della porzione eseguita sequenzialmente qui è abbastanza piccolo quindi ha un impatto significativo sulla velocità massima teorica con due soli processori. Però si può dire che la **legge di Amdahl** non ci dice solo che all'aumentare del numero di processori aumenta la complessità dello speedup in parallelo ma anche che la maggior parte dei programmi hanno un massimo teorico di parallelismo raggiungibile.

Ora si consideri la variante di **Fibonacci** che utilizza la **lazyness** di **Haskell** tramite la struttura dati `Stream` (su 4 core):

```
root@andysinx-dellOptiplex980:/home/andrea
    /Documenti/Par_Haskell> ./EvalFib 2 +RTS -N4 -l
    "result:" (46368,514229)
    Elapsed time : 1.267529s
    Total time: 1.267874s
```

Ora si apra l'eventlog con **Threadscope**:

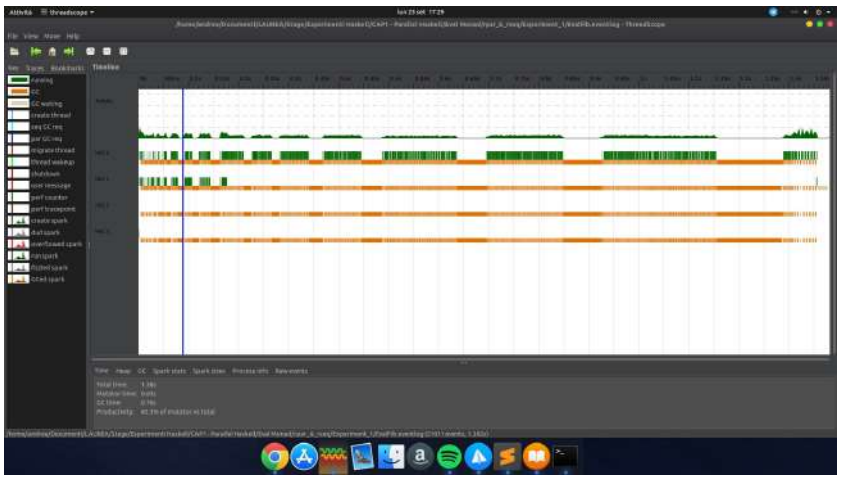

Figura 3.5: Test 2: FibStream rpar/rpar/rseq/rseq execution su 4 core

Ora si calcoli il massimo speedup con la **legge di Amdahl**:

$$Amdahl_{law} = \frac{1}{\frac{(1-P)+P}{N}} \tag{3.9}$$

$$= (\frac{(1-P)}{N})^{-1} \tag{3.10}$$

$$= (\frac{(1-0.15s)}{4})^{-1} \tag{3.11}$$

$$= 4.70588235294s \tag{3.12}$$

$$\simeq 5s \tag{3.13}$$

Si conclude affermando che in pratica si può raggiungere un massimo **speedup** teorico di 5s per quest'applicazione con 4 processori. Come si può vedere è preferibile alla versione tradizionale, con supporto della **lazyness** riesce ad aumentare lo **speedup** della computazione. Lo **speedup** che si può raggiungere è più del doppio, quindi garantisce delle prestazioni migliori.

3.2 Strategie di valutazione in parallelo

Definiamo un nuovo mezzo per garantire la modularità del codice parallelo per permettere di separare l'algoritmo dal parallelismo. Chiamiamo questi strumenti le **Strategie**. Concretamente, una `Strategy` è una funzione che inietta un dato nella `Monad Eval` ed effettua una qualche computazione definita nel seguente modo:

```
type  Strategy a = a -> Eval a
```

quindi si può dire che tramite una struttura dati crea del parallelismo e restituisce la struttura stessa. Un esempio è `parPair`:

```
parPair :: Strategy (a,b)
parPair (a,b)= do
  c <- rpar a
  d <- rpar b
  return (c,d)
```

definita come una funzione che dato un tipo di dato `Pair` restituisce quel dato applicatogli, che verrà iniettato nella `Monad Eval`. Quindi ora possiamo procedere nel seguente modo:

```
C:\Users\andysinx>ghci
   GHCi, version 8.6.3: http://www.haskell.org/ghc/ :? for help
   Prelude> import Control.Parallel.Strategies
   Prelude Control.Parallel.Strategies
      >  (runEval . parPair) (evalFib 24,evalFib 29)
      ...
```

ma come si può notare è meglio impacchettare tutta la `Strategy` in una funzione chiamata `using` definita come segue:

```
using :: Strategy a -> a -> a
using f x = (runEval . f) x
```

però come è possibile notare, possiamo enunciare la seguente affermazione:

```
using parPair (evalFib 24, evalFib 29)
== (runEval . parPair) (evalFib 24, evalFib 29)
```

tutto ciò che è stato illustrato, è il modo per separare il parallelismo dall'algoritmo che useremo in seguito.

3.2.1 Strategie parametrizzate

Il problema nasce dal fatto che `parPair` impone di applicargli un tipo di dato `Pair` che verrà sempre valutato in WHNF. Se si volesse fare qualcosa di diverso bisognerebbe ritoccare le `Strategy` definite nella Sezione 3.2. Quindi definiamo una strategia parametrizzata per le coppie:

```
evalPair :: Strategy a -> Strategy b -> Strategy (a,b)
evalPair f g (x,y) = do
                i <- f x
                j <- g y
                return (i,j)
```

così in questo modo non si ha più il parallelismo all'interno, ma si procede nel definire nuovamente `parPair` nel seguente modo:

```
parPair ::  Strategy (a,b)
parPair = evalPair rpar rpar
```

onde è possibile notare banalmente che si sta usando `rpar` stesso, questo significa che `rpar :: Strategy a`. Da questo fatto, deduciamo che `rpar` è una `Strategy` che effettua la valutazione del suo argomento mentre il calcolo che racchiude `Eval` procede in parallelo. Questo implica che anche `rseq` è una Strategia, ma `parPair` continua ancora ad essere restrittiva, in quanto le componenti della coppia continuano ad essere valutate in WHNF. Per valutarle in NF definiamo:

```
rdeepseq :: NFData a => a -> Strategy a
rdeepseq x = (rseq . force) x
```

e combiniamo nel seguente modo `rdeepseq` per ottenere una `Strategy` che valuta a pieno il suo argomento:

```
1   rparWith :: Strategy a -> Strategy a
```

e adesso si può fornire la versione parametrizzata di `parPair`:

```
1   parPair :: Strategy a -> Strategy b -> Strategy (a,b)
2   parPair f g  = evalPair (rparWith f) (rparWith g)
```

3.2.2 Strategie di valutazione di liste in parallelo

In questa sezione illustriamo come è possibile parallelizzare la funzione `map`, ossia gli ingredienti principali sono:

- L'algoritmo `map`;
- Il parallelismo: permette la valutazione degli elementi di una lista in parallelo.

Tutto ciò è realizzabile tramite le **Strategy** nel seguente modo:

```
1   evalList :: Strategy a -> Strategy [a]
2   evalList s [] = return []
3   evalList s (x : xs) = do
4           y <- s x
5           ys <- evalList s xs
6           return (y : ys)
7
8   parList :: Strategy a -> Strategy [a]
9   parList s = (evalList . rparWith) s
10
11  parMap :: (a -> b) -> [a] -> [b]
12  parMap f xs = (runEval . parList rseq . map f) xs
```

onde definiamo:

- `evalList`: percorre la lista ricorsivamente applicando il parametro s a tutta la lista per poter costruire la lista dei risultati;
- `parList`: è una **Strategy** parametrizzata su liste, che applicatogli una **Strategy** s restituisce una **Strategy** per la lista (ossia valuta gli elementi della lista in parallelo);
- `parMap`: contiene la definizione dell'algoritmo `map` isolato dal parallelismo.

Capitolo 4

Monade Par

Come è possibile vedere, gli strumenti di cui si è parlato fin'ora nascondono degli inconvenienti. La funzione `rpar` richiede che il programmatore sappia comprendere le proprietà operative del suo linguaggio che sono, nella migliore delle ipotesi, definite nell'implementazione. Questo rende `rpar` uno strumento complesso da utilizzare, in quanto gli utenti hanno un alto tasso di fallimento. Un'altra pratica utilizzata per correggere le debolezze principali di `rpar` è la `Monad Eval`, che purtroppo è ancora limitata (e non va molto lontano). Questo perchè il programmatore dovrà ancora riuscire a capire dov'è che viene utilizzata la **lazyness**. Viene proposto così la definizione di un nuovo modello che permette la programmazione parallela deterministica in Haskell anche ad utenti più inesperti. Questo modello è basato su una **monade** chiamata `Monad Par`.

4.1 Implementazione della Monade Par e funzionalità

Tale **monade** presenta granularità esplicita e assomiglia molto alla **concorrenza** con alcune differenze. Come prima cosa, la computazione nella `Monad Par` viene estratta usando `runPar :: Par a -> a`, dove si può notare che il tipo di questa funzione indica come il risultato non ha nessun **side-effect** e non è un operazione di **I/O**. Quindi tale operazione garantisce la produzione di un risultato deterministico per ogni computazione nella `Monad Par`. Lo scopo di `runPar` è introdurre il parallelismo quindi definiamo un modo per creare attività parallele tramite la funzione `fork :: Par() -> Par()`, dove la computazione applicatogli come argomento viene eseguito contemporaneamente(dal **figlio**) al calcolo corrente(dal **genitore**). In generale la `fork` permette di creare un albero di computazioni dove ogni **nodo** è un **thread** e ogni **arco** se presenta una biforcazione è un figlio. Inoltre si ha bisogno di un modo per comunicarsi i risultati (dal figlio al genitore) e quindi definiamo un tipo di strutture chiamata **I-structure** (concetto derivante da pH una variante parallela e valutata in modo eager di Haskell) o anche `IVar`-structure per permettere la comunicazione tra thread. Diciamo anche che `IVar` è parente stretto con `MVar`, dove entrambe sono un astrazione che permette di raffigurare una cella di memoria in cui un thread dove ogni **produttore** scriverà il proprio risultato tramite l'operazione di base `put` e il thread dove ogni **consumatore** preleverà il risultato prodotto tramite l'operazione di base `get`. La differenza sta nel fatto che un `MVar` può essere modificato (riscrivibile) più volte, quindi implica non determinismo mentre un `IVar` può essere scritto una sola volta. Definiamo le operazioni di base nel seguente modo:

```
newtype Par a
instance Functor Par
instance Applicative Par
instance Monad Par

data IVar a --istance Eq
new :: Par (IVar a)
get :: IVar a -> Par a
put :: NFData a => IVar a -> Par ()


```

dove l'uso ripetuto della `put` sullo stesso `IVar` consecutivamente è un errore, infatti è molto rigorosa ossia colloca il valore nell'`IVar` in NF (`NFData`) che viene richiesto come prerequisito per la massima rigidità. Questo significa che `put` causa una visita dell'`IVar` che può essere costosa se risulta essere una struttura dati molto grande. Esiste una versione meno restrittiva `put_` che valuta in WHNF, però come standard viene usata quella tradizionale perchè evita di memorizzare calcoli lazy nell'`IVar` in modo che il **programmatore** ottenga un quadro chiaro di quali lavori verranno svolti e su quali **thread**. La `get` attende fino al momento in cui viene scritto un valore nell'`IVar`. Un modello molto comune è quello in cui un **thread** crei molti figli e ne raccolga i suoi risultati. L'ingrediente principale per implementare tale idea è quello di usare le **primitive di sincronizzazione**, definendo inizialmente un astrazione che raffiguri la computazione effettuata da un singolo figlio che restituirà un risultato:

```
    spawn :: NFData a => Par a -> Par (IVar a)
    spawn p = do
        i <- new
        fork (do x <- p; put i x)
        return i
```

L'`IVar` in questo contesto è chiamato "future" perchè rappresenta il valore che verrà completato successivamente. Un modello più utile è combinare una `spawn` con una `map` in modo che per ogni elemento della lista, crea un figlio per calcolare una qualche funzione su quell'elemento. Inoltre viene incorpato anche l'attesa del termine della computazione di tutti i figli. Definiamo quindi questa funzione chiamata `parMapM`:

```
    parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
    spawn f as = do
        ibs <- mapM (spawn . f) as
        mapM get ibs
```

che è possibilmente estendibile a qualsiasi tipo di dato (questa versione è solo per liste). Possiamo mostrare nella seguente immagine un esempio della situazione considerata:

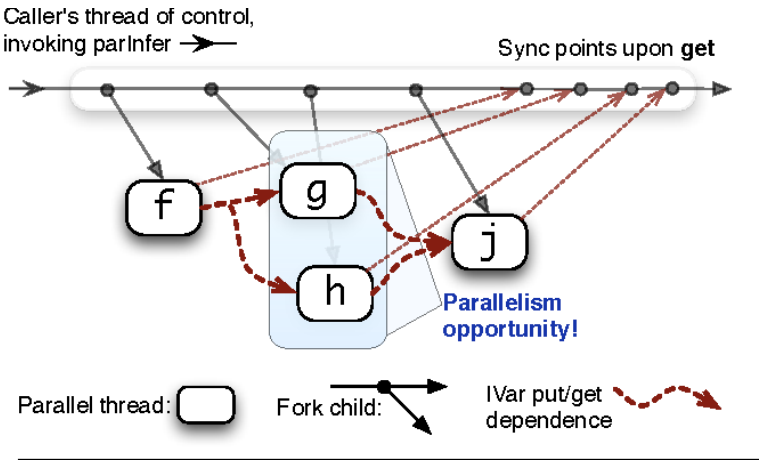

Figura 4.1: Fork effects.

Come si può vedere in Figura 4.1 viene prodotto un modello di flusso di dati a grafo in cui ogni `fork` crea un nuovo nodo di calcolo nel grafo e ogni **`IVar`** genera archi dal produttore per ciascun consumatore. La computazione nella **`Monad Par`** produce un **`Trace`**:

```
1    data Trace = Fork Trace Trace
2    | Done
3    | forall a . Get (IVar a) (a -> Trace)
4    | forall a . Put (IVar a) a Trace
5    | forall a . New (IVar a -> Trace)
```

le operazioni `get` e `put` richiedono di saper sospendere un calcolo e riprenderlo in un secondo momento. La tecnica standard per implementare la sospensione e la ripresa è usare la continuation-passing style che è esattamente ciò che si sta facendo sotto forma di una continuation-passing monad. Il risultato della continuazione è fissato per essere di tipo **`Trace`**, ma per il resto la monade è completamente standard:

```
1    newtype Par a = Par {
2      unPar :: (a -> Trace) -> Trace
3    }instance Monad Par where
4    return a = Par $ \c -> c a
5    m >>= k = Par $ \c -> unPar m (\a -> unPar (k a) c)
```

Le operazioni di base di Par semplicemente ritornano il valore **`Trace`** appropriato. In primo luogo, `fork`:

```
1    fork :: Par () -> Par ()
2    fork p = Par $ \c -> Fork (unPar p (\ _ -> Done)) (c ())
```

bisogna ricordare che `fork` ha due argomenti di tipo **`Trace`**, questi rappresentano le traces per figlio e genitore rispettivamente. Il figlio è la **`Trace`** costruita applicando l'argomento p alla continuazione (`\ _ -> Fine`), mentre la **`Trace`** genitore risulta dall'applicazione della

continuazione da c a `()` (il valore dell'unità poiché `fork` restituisce `Par()`). Le operazioni `IVar` restituiscono tutti i valori di `Trace` in modo semplice:

```
1     new :: Par (IVar a)
2     new = Par $ New
3     get :: IVar a -> Par a
4     get v = Par $ \c -> Get v c
5     put :: NFData a => IVar a -> a -> Par ()
6     put v a = deepseq a (Par $ \c -> Put v a (c ()))
```

Si noti come `put` valuta completamente l'argomento utilizzando `deepseq` prima che venga ritornato il `Trace` inserito. Quindi come possiamo vedere, un esempio può essere il seguente:

```
1      fibPar :: (NFData a ,Integral a) => a -> a -> a
2      fibPar n k = runPar $ do
3             fst_thr_comp <- new -- (1)
4             snd_thr_comp <- new -- (2)
5             fork (put fst_thr_comp (fib n)) -- (3)
6             fork (put snd_thr_comp (fib k)) -- (4)
7             res_fst <- get fst_thr_comp -- (5)
8             res_snd <- get snd_thr_comp -- (6)
9             return (res_fst + res_snd) -- (7)
10
```

onde:

1. Creo un `IVar` per le prima computazione;
2. Creo un `IVar` per le seconda computazione;
3. Forko due computazioni indipendenti di cui la prima calcola l'n-esimo numero di fibonacci.
4. La seconda calcola il k-esimo numero di fibonacci;
5. Il genitore della prima `fork` si mette in attesa del risultato;
6. Il genitore della seconda `fork` si mette in attesa del risultato;
7. Torno un valore iniettato nella `Monad Par` e il `runPar` mi andrà ad estrarre il valore dalla Monade.

Un altro possibile esempio è il seguente : `https://github.com/AndreaSenese/Infer_Algorithm/blob/master/inferAlg.hs`. Vuole mostrare come due thread figli si dividano la lista dei termini per inferirne il tipo in parallelo, in modo che il thread genitore concateni i due risultati. Il seguente è solo un possibile modellino per ragionare sul funzionamento. Nella sezione 5.1.2 verrà introdotto l'algoritmo di inferenza parallelo utilizzando un versione più specifica per l'esperimento.

4.2 Semantica operazionale della Monade Par

4.2.1 Semantica Operazionale

Lo scopo ora è di precisare la descrizione informale di modello di programmazione tramite **semantica operazionale onestep/bigstep**:

$Syntax \Downarrow$

$$
\begin{array}{lll}
x, y \in Variable & & \\
i \in \texttt{IVar} & & \\
Variable : V & ::= & x \\
 & | & i \\
 & | & \lambda x.M \\
 & | & \texttt{return}\ M \\
 & | & M >>= N \\
 & | & \texttt{runPar}\ M \\
 & | & \texttt{fork}\ M \\
 & | & \texttt{new} \\
 & | & \texttt{put}\ \texttt{i}\ M \\
 & | & \texttt{get}\ \texttt{i} \\
 & | & \texttt{done}\ M \\
Terms : M, N & ::= & V \\
 & | & MN \\
 & | & \ldots \\
States : P, Q & ::= & M\ (thread) \\
 & | & <>_i\ \ (empty\ \texttt{IVar}) \\
 & | & < M >_i\ \ (full\ \texttt{IVar}\ \texttt{i}\ di\ M) \\
 & | & vi.P\ (restriction) \\
 & | & P \mid Q\ (parallel\ composition)
\end{array}
\tag{4.1}
$$

$$StructuralRule_{transition} \Rightarrow \boxed{\begin{array}{rr} P \mid Q \equiv Q \mid P, & (4.2)\\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R, & (4.3)\\ vx.vy.P \equiv vy.vx.P, & (4.4)\\ vx.(P \mid Q) \equiv (vx.P) \mid Q, & (4.5)\\ where\ i \notin fn(Q), & (4.6)\\ \dfrac{P \Rightarrow Q}{P \mid R \Rightarrow Q \mid R}, & (4.7)\\ \dfrac{P \Rightarrow Q}{vx.P \Rightarrow vx.Q}, & (4.8)\\ \dfrac{P \equiv P' \quad P' \Rightarrow Q' \quad Q' \equiv Q}{P \Rightarrow Q} & (4.9) \end{array}}$$

La seguente figura 4.2.1 fornisce la sintassi di valori e termini del linguaggio. L'unica forma insolita qui è `done M`, che è uno strumento interno che verrà usata nella semantica per `runPar`; non è un termine a disposizione del programmatore nel linguaggio reale. La semantica principale per il linguaggio è una semantica operazionale big-step scritta con la seguente notazione: $M \Downarrow V$, che significa che il termine M si riduce al valore V in zero o più passi. Esso è del tutto convenzionale, quindi omettiamo tutte le sue regole tranne una, vale a dire `runPar` nella figura in basso. Ma il punto importante per ora è che a sua volta dipende da una semantica operazionale one-step (per la **`Monad Par`**) scritta $P \Rightarrow Q$. Qui P e Q sono stati, la cui sintassi è data nella figura della sintassi. Uno stato è un insieme di termini M (i suoi "thread" attivi), e **`IVars`** `i` che sono pieni, $< M >_i$, oppure vuoti, $<>_i$. In uno stato, $i.P$ serve (per convenzione) per limitare la portata di i in P. La notazione $P_0 \Rightarrow^* P_i$ è un abbreviazione per la sequenza $P_0 \Rightarrow ... \Rightarrow P_i$ dove $i >= 0$. Gli stati rispettano una relazione di equivalenza strutturale: $\equiv$, che specifica che la composizione parallela è associativa e commutativa e la restriction scope può essere ampliata o ridotta purché nessun nome esca dallo scope(campo di applicazione). Le tre regole infondo dichiarano che le transizioni possono aver luogo su qualsiasi sottostato. Quindi la relazione $\Rightarrow$ è intrinsecamente non deterministica. Le transizioni di $\Rightarrow$ sono riportate nella figura in basso usando una valutazione contesto ξ:

$$\boxed{\xi \quad ::= [.] \mid \xi >>= M}$$

Quindi il termine che determina una transizione sarà trovato guardando a sinistra di >>=.

- La regola **`Eval`**: consente la riduzione tramite big-step semantics di $M \Downarrow V$ per ridurre il termine in un contesto di valutazione se non è già un valore;
- La regola **`Bind`** (>>=): è la semantica definita per il bind monadico standard;
- La regola **`Fork`**: crea un nuovo Thread;
- La regola **`New`**: crea un nuovo **`IVar`** vuoto il cui nome non non entra in contrasto con un altro **`IVar`** in scope;

- La regola **Get**: restituisce il valore di un **IVar** completo;
- La regola **Put**: crea un IVar completo da un **IVar** vuoto. Inoltre bisogna notare che non c'è transizione per `put` quando l'**IVar** è pieno: nell'implementazione verrà segnalato l'errore mentre nella semantica no, perché la put anche essendo full-strict, non ha nessun effetto a livello semantico;
- La regola **GCReturn**: consente a un thread che ha completato la sua computazione di essere spazzato via dal garbage collector;
- La regola **GCEmpty** e **GCFull**: consentono di far spazzare via dal garbage collector un **IVar** pieno o vuoto a condizione che l'**IVar** non faccia riferimento a nessun altro in quello stato;
- La regola **GCDeadlock**: consente di impostare una serie di thread con deadlock da far spazzare via dal garbage collector: la sintassi $\xi[\texttt{get i}]^*$ significa uno o più thread del modulo indicato. Dal momento che non ci possono essere altri thread che fanno riferimento a "i", nessuno dei `get` potrà mai fare progressi. Quindi è possibile rimuovere l'intero set di thread in deadlock insieme al **IVar** vuoto dallo stato;
- La regola **RunPar**: fornisce la semantica di `runPar` e collega la semantica di riduzione **Par** $\Rightarrow$ con la semantica di riduzione funzionale $\Downarrow$. Informalmente si può affermare così: se l'argomento M di `runPar` viene eseguito nella semantica di **Par** ottenendo un risultato N e N si riduce a V, allora si dice che `runPar M` si riduca a V. Sul basso possiamo vedere le regole one-step per la **Monad Par**:

$$
Rule_{transition} \Rightarrow
\begin{cases}
\textbf{Eval}\ \dfrac{M \not\equiv N \qquad M \Downarrow V}{\xi[M] \rightarrow \xi[M]}, & (4.10)\\[2ex]
\textbf{Bind}\ \dfrac{}{\xi[N\ \texttt{return}\ >\!\!>\!\!=\ M] \Rightarrow \xi[MN]}, & (4.11)\\[2ex]
\textbf{Fork}\ \dfrac{}{\xi[\texttt{fork}\ M] \Rightarrow \xi[\texttt{return}\ ()] \mid M}, & (4.12)\\[2ex]
\textbf{New}\ \dfrac{}{\xi[\texttt{new}] \Rightarrow \nu_i.(<>_i \mid \xi[\texttt{return i}])}\ \textit{where}\ \mathbf{i} \notin \mathbf{fn}(\xi), & (4.13)\\[2ex]
\textbf{Get}\ \dfrac{}{< M >_i \mid \xi[\texttt{get i}] \Rightarrow < M >_i \mid \xi[\texttt{return M}]}, & (4.14)\\[2ex]
\textbf{PutEmpty}\ \dfrac{}{<>_i \mid \xi[\texttt{put}\ i\ M] \Rightarrow < M >_i \mid \xi[\texttt{return}\ ()]}, & (4.15)\\[2ex]
\textbf{GCReturn}\ \dfrac{}{\texttt{return}\ M \Rightarrow}, & (4.16)\\[2ex]
\textbf{GCEmpty}\ \dfrac{}{\nu i.\ <>_i \Rightarrow}, & (4.17)\\[2ex]
\textbf{GCFull}\ \dfrac{}{\nu i.\ < M >_i \Rightarrow}, & (4.18)\\[2ex]
\textbf{RunPar}\ \dfrac{(M >\!\!>\!\!= \lambda x.\texttt{done}\ x) \Rightarrow^* \texttt{done}\ N \qquad N \Downarrow V}{\texttt{runPar}\ M \Downarrow V} & (4.19)
\end{cases}
$$

Per approfondimenti si legga l'articolo [4].

4.2.2 Determinismo

Ora verrà richiesto che se `runPar` $M \Downarrow N$ e `runPar` $M \Downarrow N'$, allora $N = N'$. Inoltre se `runPar` $M \Downarrow N$ non c'è una sequenza di passi a partire da `runPar` M che raggiunge uno stato in cui nessuna riduzione è possibile. Informalmente, `runPar` M dovrebbe ridurre lo stesso valore in modo coerente o non produrre alcun valore (che è semanticamente equivalente a $\bot$). Un risultato non deterministico può sorgere solo a causa di una condizione di competizione: due diversi ordini di riduzioni che portano a un diverso risultato. Per ottenere il non determinismo, ci devono essere due ordini di riduzione che portano ad applicazioni della regola (**`PutEmpty`**) sullo stesso **`IVar`** con valori diversi. Non c'è nulla nella semantica che impedisca che ciò accada, ma l'argomento deterministico si basa sulla regola (**`RunPar`**), che richiede che lo stato alla fine della riduzione è costituito solo da `done` M per qualche M. Cioè il resto dello stato è completato o in deadlock ed è stato spazzato via dalle regole (**`GCReturn`**), (**`GCEmpty`**), (**`GCFull`**), e (**`GCDeadlock`**). Nel caso in cui vi sia una scelta tra più `put`, una di queste non sarà in grado di completare e lo farà rimanere nello stato attuale, e quindi la transizione `runPar` non sarà mai applicabile.

4.2.3 Soundness

Se alcuni sottotermini restituiscono $\bot$ durante l'esecuzione di `runPar`, allora il valore dell'intero `runPar` dovrebbe essere $\bot$. Lo scheduler sequenziale lo implementa, ma lo scheduler parallelo no. In pratica, tuttavia, farà poca differenza: se uno dei scheduler incontra un $\bot$, il risultato più probabile è un deadlock, perché probabilmente il thread che lo scheduler stava eseguendo avrà la `put` in sospeso. Poiché il **deadlock** è semanticamente equivalente a $\bot$, il risultato in questo caso è equivalente. Tuttavia, modificare l'implementazione per rispettare la semantica non è difficile e si potrebbe fare direttamente nell'implementazione.

4.2.4 Overhead

In questa sezione si vogliono analizzare le prestazioni della **`Par`** `monad` in due modi: in primo luogo misuriamo l'overhead dato dalla **`Par`** `monad`, e in secondo luogo consideriamo un esempio di benchmark parallelo esistente quando utilizziamo la **`Par`** `monad` e l'astrazione **`Strategies`**. In effetti, utilizzare questi strumenti può ridurre il tempo di esecuzione di `fibonacci` (Figura 3.4) del 40%. Per valutare il sovraccarico della **`Par`** `monad` rispetto alle operazioni `rpar` e `rseq` utilizzate nella libreria **`Control.Parallel.Strategies`**, si vuole utilizzare la funzione di `fibonacci`:

```haskell
fib :: Int -> Int
fib x | x < 1 = 1
fib x = fib (x - 2) + fib (x - 1)

pfib :: Int -> Par Int
pfib n | n < 25 = return (fib n)
pfib n = do
  av <- spawn (pfib (n - 1))
  b  <- pfib (n - 2)
  a  <- get av
  return (a + b)

test1 = (runPar . pfib) 30

main = do
        startApp <- getCurrentTime
        [n] <- getArgs
        let test = [test1] !! (read n - 1)
        startTest <- getCurrentTime
        res <- evaluate(test1)
        endTimeTest <- getCurrentTime
        print "result :"
        print res
        t2 <- getCurrentTime
        printf "Elapsed time : %.4fs\n" (realToFrac
         ↪  (diffUTCTime endTimeTest startTest) ::
         ↪  Double)
        endApp <- getCurrentTime
        printf "Total time: %.4fs\n" (realToFrac
         ↪  (diffUTCTime endApp startApp) :: Double)
```

```
root@andysinx-dellOptiplex980:/home/andrea/Documenti
    /Par_Haskell
    > ghc -O2 ParFib.hs -threaded -rtsopts -eventlog
    [1 of 1] Compiling Main         (EvalFib.hs, EvalFib.o)
    Linking EvalFib ...
    root@andysinx-dellOptiplex980:/home/andrea
    /Documenti/Par_Haskell> ./ParFib 1 +RTS -N4
    "result:" 2178309
    Elapsed time : 0.0179s
    Totale time : 0.0184s
```

Come si può vedere:

Version	Time Elapsed
pseq/rseq	0.000052s
Par monad	0.017927s

Come possiamo vedere la `Par monad` ha overhead. Tutte le versioni che sono state riportate nel documento non si parallelizzano bene perchè la granularità è molto fine. Come tanti algoritmi divide et impera, bisogna definire un punto di interruzione al di sotto del quale verrà utilizzata la versione sequenziale dell'algoritmo riducendo il sovraccarico che la `Par Monad` crea. Per esempio, si potrebbe eseguire la funzione `pFib 43` impostando un taglio a 24.

4.3 Stream processing pipeline

In questa sezione si vuole mostrare come utilizzare il parallelismo con l'uso delle pipeline parallele. In **Haskell** è possibile utilizzare il parallelismo in un applicazione in cui si producono e consumano i dati in modo **lazy**, anche se in quel caso avviene un parallelismo dei dati, che è un parallelismo tra elementi della lista. Invece in tale sezione si vuole mostrare l'utilizzo del parallelismo tra le diverse fasi di una pipeline. Data una pipeline del seguente tipo:

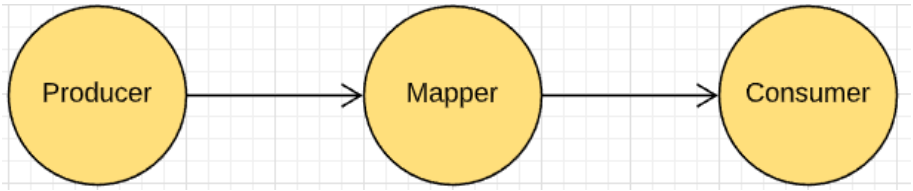

Figura 4.2: Structure of pipeline

Ogni fase della pipeline esegue una computazione sugli elementi del flusso di dati e lo farà mantenendone lo stato. Quando una fase della pipeline mantiene un certo stato, non possiamo più sfruttare il parallelismo tra gli elementi del flusso ossia come invece è possibile farlo su `Stream`. Lo scopo è quello di eseguire le diverse fasi della pipeline separate su più core, permettendone lo streaming di dati tra di loro. Definiamo lo `Stream` come una lista che verrà valutata in modo **lazy**, basandoci su `Stream`:

```
1    data IList a = Nil | Cons a (IVar(IList a))
2    type Stream a = IVar(IList a)
```

Un `IList` è una lista con un `IVar` come coda. Ciò consente al produttore di generare la lista in modo incrementale, mentre un consumatore eseguendo in parallelo consuma tutti gli elementi che vengono prodotti man mano dal produttore. Un flusso è un `IVar` contenente un `IList`. Definiamo allora un produttore generico che trasforma una lista in uno `Stream`:

```
1     streamFromList :: NFData a => [a] -> Par(Stream a)
2     streamFromList xs = do
3              var <- new --(1)
4              fork $ loop xs var --(2)
5              return var         --(3)
6         where
7             loop [] var = put var Nil  -- (4)
8             loop (x : xs) var = do   -- (5)
9                 tail <- new  --(6)
10                put var (Cons x tail) --(7)
11                loop xs tail --(8)
```

onde:

1. Crea L'`IVar` che sarà lo stream stesso;
2. Esegue una `fork` su `loop` che avrà il compito di creare i contenuti dello `Stream` in parallelo;
3. Restituisce lo `Stream` al chiamante;
4. Caso in cui la lista sia vuota: si memorizza semplicemente un `IList` vuota nell'`IVar`;
5. Lo scopo di `loop` è di attraversare la lista di input, producendo una `IList` mentre la percorre. Come si può vedere, il primo argomento è la lista e il secondo argomento è l'`IVar` in cui incapsulare l'`IList`;
6. Crea un `IVar` per la coda;
7. Viene memorizzato in un `IVar` un `Cons`, che rappresenta il nodo della `IList` non vuoto. Si può notare che la lista verrà valutata in NF, in quanto come citato nella Sezione 4.1, `put` è molto rigorosa.
8. Effettua una chiamata ricorsiva per creare il resto dello `Stream`.

Ora definiamo il consumatore di **Stream**:

```
1     streamFold :: (a -> b - a) -> a -> Stream b -> Par a
2     streamFold fn acc instrm = do
3          ilst <- get instrm
4          case ilst of
5            Nil -> return acc
6            Cons h t <- streamFold fn (fn acc h) t
```

onde si può notare che è una `foldl` sullo **Stream** ed è definita tramite una chiamata ricorsiva su **IList** che accumula il risultato fino al raggiungimento della fine del flusso. Se `streamFold` consuma tutti gli elementi del flusso disponibili andrà in attesa che venga prodotto un prossimo elemento. Ora definiamo lo schema produttore-consumatore:

```
1     streamMap :: NFData a => (a -> b) -> Stream a ->
      ↪ Par(Stream b)
2     streamMap fn instrm = do
3             outstrm <- new
4             fork $ loop instrm outstrm
5             return outstrm
6         where
7             loop instrm outstrm = do
8                 ilst <- get instrm
9                 case ilst of
10                    Nil -> put outstrm Nil
11                    Cons h t -> do
12                        newtl <- new
13                        put outstrm (Cons (fn h) newtl)
14                        loop t newtl
```

4.3.1 Limitazioni

Il parallelismo con pipeline è limitato in quanto possiamo produrre tanto parallelismo quante sono le fasi della pipeline. Tende quindi a essere meno efficace del dataflow parallelism con **Monad Par** che può produrre molto più parallelismo. Tuttavia è uno strumento utile a cui tenere conto. Infatti non è possibile produrre un lazy **Stream** da `runPar` stesso. La chiamata a `streamFold` accumula l'intera lista prima di ritornarla quindi non si può restituire un **IList** da `runPar` e consumarla in un altra chiamata a `runPar`, perche non è possibile restituire un **IVar** da `runPar` perche è illegale e causerebbe un errore di tipo. Inolte `runPar` esegue tutte le `fork` computazioni fino al completamente, perche è ciò per poter garantire risultati deterministici. Esiste una

versione di `Monad Par` per cui il determinismo non è garantito, ed è una versione `IO` della `Monad Par` che è possibile utilizzarla per lo `Stream` lazy.

Capitolo 5

Strategie di applicazione del parallelismo

5.1 Studio di un caso

In questa sezione si vuole illustrare come parallelizzare un algoritmo di inferenza di tipo, come si può vedere in qualsiasi compilatore per linguaggi puramente funzionali. Iniziamo definendo il funzionamento dell'algoritmo per determinare prima il funzionamento e poi applicare il pattern per aggiungere il parallelismo.

5.1.1 Type Inference Algorithm

Il termine "**Type Inference**" rappresenta un sistema che permette di rilevare il tipo di un dato attraverso conclusioni logiche con il ragionamento deduttivo. Un **Type Inference Algorithm** è un **algoritmo** che simula le modalità con cui la mente umana trae delle conclusioni logiche attraverso il ragionamento. L'algoritmo trae conclusioni logiche tramite il seguente **sistema** (**Type Inference System**):

$$TypeInf_{System} \Rightarrow \boxed{\begin{array}{rl} \textbf{T-Var}\ \dfrac{}{\Gamma, x : \alpha \vdash x : \alpha}, & (5.1) \\ \textbf{T-Bool}\ \dfrac{}{\Gamma \vdash c : \texttt{Bool}}, & (5.2) \\ \textbf{T-Lam}\ \dfrac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x.M : \alpha \to \beta}, & (5.3) \\ \textbf{T-If}\ \dfrac{\Gamma \vdash M : \texttt{Bool} \quad \Gamma \vdash N_1 : \alpha \quad \Gamma \vdash N_2 : \alpha}{\Gamma \vdash \texttt{if}\ M\ N_1\ N_2 : \alpha}, & (5.4) \\ \textbf{T-App}\ \dfrac{\Gamma \vdash M : \alpha \to \beta \quad \Gamma \vdash N : \alpha}{\Gamma \vdash M\ N : \beta} & (5.5) \end{array}}$$

Tale algoritmo è composto da 3 fasi:

1. Riscrittura dei λ-**termini** su una struttura ad **albero**;
2. Annotazione dell'**albero** e generazione dei **vincoli**;
3. Risoluzione dei **vincoli**.

Tale **algoritmo** fa uso di **espressioni di tipo**, che sono delle espressioni di tipi incomplete in cui di alcune parte dell'espressione non siamo ancora a conoscenza del **tipo** esatto. Quindi verranno annotate con i seguenti simboli finchè non si arriverà al tipo più generale:

- Insieme infinito di **variabili di tipo:** $TVar = \{\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta, \theta, \iota, \kappa, \lambda, \mu, \nu, \xi, \emptyset, \pi, \rho, \sigma, \tau, \upsilon, \varphi, \chi, \psi, \omega\}$;
- Espressioni di **tipo**: $\sigma, \tau := \alpha \mid \texttt{Bool} \mid \sigma \to \tau$ per denotare *costanti* | *booleani* | *funzioni*

Ora costruiamo l'albero e lo annotiamo tramite le seguenti regole:

Albero	Note
$x : \alpha$	α è nuova, avendo cura di usare la stessa α per tutte le occorrenze della stessa x
$c : \texttt{Bool}$	Altre costanti richiederanno tipi diversi
$\lambda x : \alpha \to \tau$ | $T : \tau$	α è la variabile di tipo usata per annotare x in T Se x non è usata in T scegliere una α nuova
$@ : \alpha$ $T_1 : \tau$ $\quad$ $T_2 : \sigma$	α è nuova Generare il vincolo $\tau = \sigma \to \alpha$
$@ : \tau_2$ $T_1 : \tau_1 \to \tau_2$ $\quad$ $T_2 : \sigma$	Ottimizzazione facoltativa del caso precedente che evita l'introduzione di una nuova α Generare il vincolo $\tau_1 = \sigma$
$\texttt{if} : \tau_2$ $T_1 : \tau_1$ $\quad$ $T_2 : \tau_2$ $\quad$ $T_3 : \tau_3$	Generare il vincolo $\tau_1 = \texttt{Bool}$ Generare il vincolo $\tau_2 = \tau_3$

Figura 5.1: Tree of type annotation

Ora la fase di **generazione dei vincoli** produce un sistema $\{\tau_i = \sigma_i\}_{1 \le i \le n}$ e bisogna determinare se questo sistema ammetta una **soluzione**: Se tale **sistema** ammette una **soluzione** vorremmo saperne la più generale. Definisco matematicamente il termine "**sostituzione**" e "**soluzione**" nel seguente modo:

Definizione 5.1 (Sostituzione) *Una **sostituzione** θ è una **funzione** da **variabili di tipo** a **espressioni di tipo**. Scriviamo $\theta(\tau)$ per l'espressione ottenuta da τ sostituendo ogni α con $\theta(\alpha)$.*

Definizione 5.2 (Soluzione) *Dato un **sistema di vincoli** $\{\tau_i = \sigma_i\}_{1 \leq i \leq n}$ e una **sostituzione** θ diciamo che θ è **soluzione** (o unificatore) del **sistema** se $\theta(\tau_i) = (\sigma_i)$ per ogni $1 \leq i \leq n$. Diciamo inoltre che θ è **l'unificatore** più generale del **sistema** se ogni **soluzione del sistema** è ottenibile componendo θ con un'altra **sostituzione**.*

onde bisogna rispettare le seguenti regole sui **vincoli**:

Se c'è un vincolo	e	allora
$\tau = \tau$	–	eliminare il vincolo
$\tau = \alpha$	τ non è una variabile	rimpiazzare il vincolo con $\alpha = \tau$
$\tau \to \tau' = \sigma \to \sigma'$	–	rimpiazzare il vincolo con $\tau = \sigma$ e $\tau' = \sigma'$
$\tau \to \sigma =$ `Bool` o `Bool` $= \tau \to \sigma$	–	☠ l'algoritmo fallisce (type error)
$\alpha = \tau$	$\alpha \neq \tau$ ma α compare in τ	☠ l'algoritmo fallisce (occur check)
$\alpha = \tau$	α non compare in τ α compare altrove	sostituire α con τ in tutti gli altri vincoli ($\alpha = \tau$ rimane)

Figura 5.2: type constraints

Applicando l'algoritmo ad un sistema

$$\Sigma = \{\sigma_i = \tau_i\}_{i \in I}$$

si ha che:

- L'algoritmo termina sempre con:
  1. fallimento;
  2. il sistema raggiunto non può essere trasformato.
- Se l'algoritmo fallisce in Σ allora è insoddisfacibile;
- Il sistema risultante ha la forma $\{\alpha_i = \rho_i\}_{i \in J}$ in cui ogni α_i compare una volta sola e $\theta = \{\alpha_i \to \rho_i\}_{i \in I}$ è soluzione in Σ (non solo), ma è anche la soluzione più generale (ovvero, ogni soluzione di Σ è ottenibile componendo θ con un altra sostituzione).

5.1.2 Parallel Type Inference Algorithm

Considerando la sezione 5.1.1, lo scopo è di dimostrare due cose essenziali:

- Come il parallelismo può essere applicato facilmente a problemi di analisi del programma;
- Come il modello di dataflow funziona molto bene anche quando la struttura del parallelismo dipende interamente dall'input e non può essere previsto in anticipo.

Procediamo nel definire lo schema del problema nel seguente modo: data una lista di bindings del formato

$$x = e$$

dove "x" è una variabile ed "e" un espressione. L'algoritmo inferisce il tipo di ciascuna variabile. Le espressioni consistono di numeri interi, variabili, applicazioni, espressioni lambda e operatori aritmetici.La situazione è la seguente:

$$\begin{array}{lll} Variable : X & ::= & n \\ & | & x \\ & | & \lambda x.M \\ & | & f\ arg \\ & | & \text{n op m} \end{array} \tag{5.6}$$

Ora dato l'algoritmo di type inference parallelo, si vogliono eseguire dei test chiamando solamente il type inferencer:

```
root@andysinx-dellOptiplex980:/home/andrea
/Documenti/Par_Haskell/Par_Inferencer> ghci
GHCi, version 8.4.4: http://www.haskell.org/ghc/  :? for help
Prelude> :load parinfer.hs
[1 of 14] Compiling FiniteMap (FiniteMap.hs, interpreted)
[2 of 14] Compiling Lex (Lex.hs, interpreted)
[3 of 14] Compiling MaybeM (MaybeM.hs, interpreted)
[4 of 14] Compiling MyList (MyList.hs, interpreted)
[5 of 14] Compiling Shows  (Shows.hs, interpreted)
[6 of 14] Compiling StateX (StateX.hs, interpreted)
[7 of 14] Compiling Term   (Term.hs, interpreted)
[8 of 14] Compiling Parse  (Parse.hs, interpreted)
[9 of 14] Compiling Type   (Type.hs, interpreted)
[10 of 14] Compiling Substitution(Substitution.hs,interpreted)
[11 of 14] Compiling InferMonad (InferMonad.hs, interpreted)
[12 of 14] Compiling Environment (Environment.hs, interpreted)
[13 of 14] Compiling Infer  (Infer.hs, interpreted)
[14 of 14] Compiling Main   (parinfer.hs, interpreted)
Ok, 14 modules loaded.
*Main> test "1 + 2"
Int
*Main> test "\\x -> ((x+1) * (3-2))/5"
Int -> Int
*Main> test "\\x -> x"
a0 -> a0
*Main> test "\\x -> \\f -> f x"
a0 -> (a0 -> a2) -> a2
```

Quando il type inferencer viene eseguito autonomamente andrà a controllare un file di bindings e ne fornisce un tipo per ognuno. Supponiamo, per semplicità, che la lista dei bindings sia ordinato e non sia ricorsivo, ossia qualsiasi variabile utilizzata in un espressione deve essere definita in precedenza nella lista. I binding successivi possono anche coprire quelli precedenti. Per esempio, se consideriamo il seguente insieme di binding per il quale vogliamo dedurne il tipo:

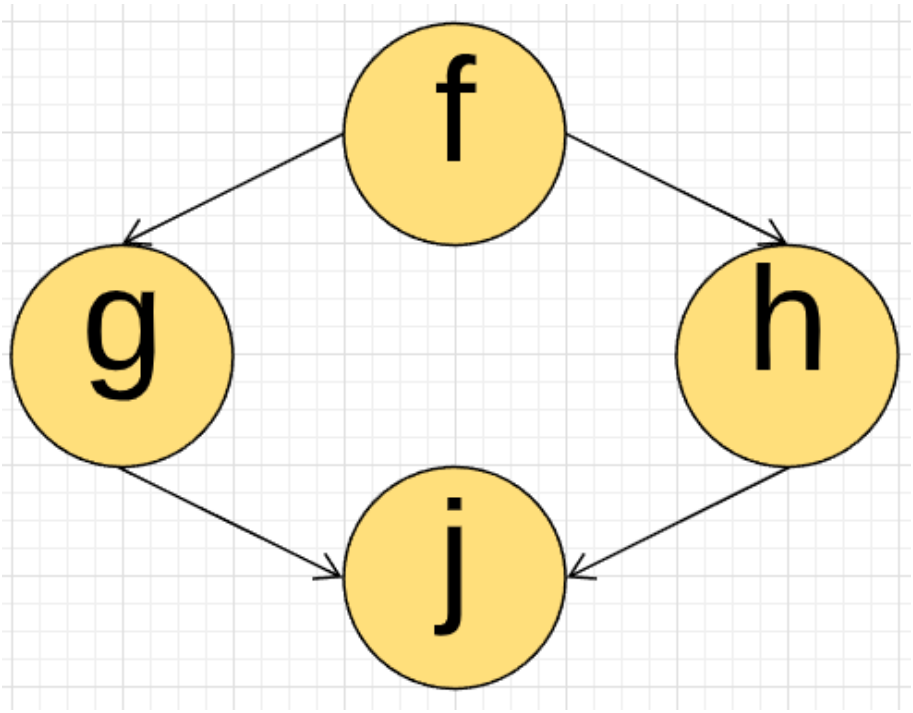

Figura 5.3: dataflow graph: parallel type dependencies

Possiamo procedere in modo lineare tramite la lista delle variabili. Iniziamo inferendo prima il tipo di f, poi il tipo di g, poi il tipo di h, e cosi via. Tuttavia se guardiamo il dataflow graph per questo insieme di variabili, si può vedere come esistano dipendenze fra di essi e quindi possiamo assumere che ci sia parallelismo. Chiaramente è possibile inferire il tipo di g e h in parallelo, dopo aver determinato il tipo di f. Come possiamo vedere, l'inferenza di tipo è un adattamento naturale per il dataflow model. Possiamo considerare ogni associazione come un nodo del grafo e gli archi del grafo forniscono tipi dedotti dai collegamenti ai bindings utilizzati nel programma. La creazione di un dataflow graph per il problema di inferenza di tipo consente di estrarre automaticamente il parallelismo dal processo di inferenza di tipo. La quantità effettiva di parallelismo dipende interamente dalla struttura dell'input di programma. Un input di programma in cui ogni variabile è dipendente dalla precedente in lista non avrebbe alcun parallelismo da estrarre. Tuttavia, il guadagno maggiore è portato dal parallelizzare il livello esterno. Quindi il motore di inferenza di tipo avrà i seguenti tipi:

1. `type VarId = String --Variabili`;
2. `data Term --Terms possibili da applicare all'input del programma`;
3. `data Env --Ambiente, mappa VarId a PolyType`;
4. `data PolyType --Tipi Polimorfi`.

Riguardo ai linguaggi di programmazione, si definisce ambiente una mappatura che assegna qualche significato per le variabili di un espressione. Un motore di inferenza di tipo utilizza un ambiente per assegnare tipi alle variabili; questo è lo scopo di `Env`. Quando selezioniamo un espressione, dobbiamo specificare un `Env` che fornisca i tipi delle variabili che è possibile creare tramite la funzione `makeEnv` definita come segue:

```
makeEnv :: [(VarId,PolyType)] -> Env
```

Per determinare di quale variabili si ha bisogno per popolare **Env**, abbiamo bisogno di un modo per estrarre le variabili libere (non associate) di un'espressione; questo è ciò che fà la funzione `freeVars`:

```
freeVars :: Term -> [VarId]
```

il motore di inferenza di tipo per le espressioni accetta un **Term** ed un **Env** e fornisce i tipi per le variabili libere del termine , fornendone anche un **PolyType**:

```
inferTopRhs :: Env -> Term -> PolyType
```

Mentre la parte sequenziale del motore di inferenza utilizza **Env** che mappa `varId` a **IVar PolyType**, in modo che possiamo effettuare una `fork` sul motore di inferenza per una data variabile, e quindi attenderne il risultato. L'ambinte per il parallel inferencer è chiamato **TopEnv**:

```
type TopEnv :: Map VarId (IVar PolyType)
```

Ora resta solo il codice del livello superiore: viene diviso in due parti. La prima parte è una funzione per inferire il tipo di una singola variabile:

```
inferBind :: TopEnv -> (VarId, Term) -> Par TopEnv
inferBind topenv (x,u) = do
     vu <- new -- (1)
     fork $ do -- (2)
       let fu = Set.toList(freVars u) --(3)
       tfu <- mapM (get . fromJust . flip Map.lookup
        ↪  topenv) fu --(4)
       let aa = makeEnv (zip fu tfu) --(5)
       put vu (inferTopRhs aa u) --(6)
       return (Map.insert x vu topenv) --(7)
```

che effettua le seguenti operazioni:

- Crea un **IVar**, chiamato vu, per contenere il tipo per quel binding;
- Effettua una `fork` per fare inferenza di tipo;
- Gli input per fare inferenza di tipo sono i tipi delle variabili menzionate nell'espressione u. Quindi si va a chiamare `freeVars` per ottenere queste variabili;

- Per ciascun delle variabili libere, cerca il suo `IVar` nel `TopEnv`, quindi chiama `get` su di esso. Questo passaggio termina quando saranno disponibili i tipi di tutte le variabili libere;
- Costruisce un `Env` dalle variabili libere e dai loro tipi;
- Inferisce il tipo dell'espessione u, e inserisce il risultato nell'`IVar` creato inizialmente;
- Torna nel padre, ritorna `topenv` esteso con x mappato sul nuovo `IVar` vu.

Ora usiamo `inferBind` per definire `inferTop` che fornisce i tipi per una lista di binding:

```
1  inferTop :: TopEnv -> [(VarId, Term)] ->
    ↪  Par[(VarId,PolyType)]
2  inferTop topenv binds = do
3      topenv1 <- foldM inferBind topenv0 binds --(1)
4      mapM (\(v,i) -> do
5                  t <- get i;
6                  return (v,t)) (Map.toList topenv1) --(2)
```

che effettua le seguenti operazioni:

- Usa `foldM` (da `import Control.Monad`) per eseguire `inferBind` su ogni binding, accumulando un `TopEnv` che conterrà un mapping per ciascuna variabile;
- Attende che si completi tutta l'inferenza di tipo per poterne raccogliere i risultati. Quindi ritrasforma il `TopEnv` in una lista e chiama `get` su tutti gli `IVar`.

Quindi la situazione è la seguente:

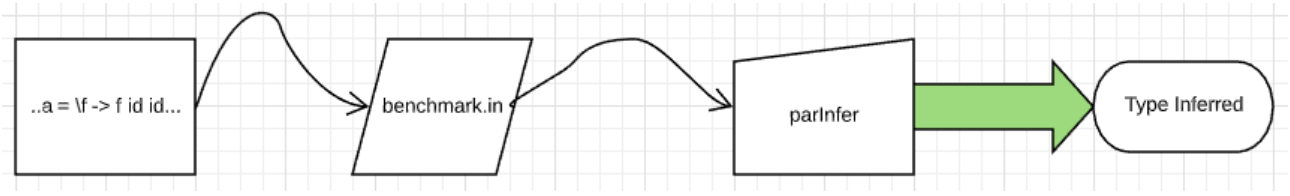

Figura 5.4: Input file for parInfer

Esistono quattro sequenze di bindings che possono essere dedotte in parallelo. La prima sequenza è l'insieme di associazioni per a (ogni associazione successiva per coprire la precedente), quindi sequenze identiche chiamate b, c e d. Ogni binding in una sequenza dipende dal precedente, ma le sequenze sono indipendenti l'una dall'altra. Questo significa che il nostro algoritmo di typechecking dovrebbe dedurre automaticamente i tipi per a, b, c e d bindati in parallelo, dando uno speedup massimo di 4. Il risultato con due processori è il seguente:

```
root@andysinx-dellOptiplex980:/home/andrea
/Documenti/Par_Haskell/Par_Inferencer>#
> ./parinfer <benchmark.in +RTS -s -N2
[("*",All . Int -> Int -> Int),
("+",All . Int -> Int -> Int),
("-",All . Int -> Int -> Int),
("/",All . Int -> Int -> Int),
("a",All a4095. a4095 -> a4095),
("b",All a4095. a4095 -> a4095),
("c",All a4095. a4095 -> a4095),
("d",All a4095. a4095 -> a4095),
("id",All a0. a0 -> a0)]
     366,007,320 bytes allocated in the heap
      56,148,888 bytes copied during GC
       3,570,456 bytes maximum residency (14 sample(s))
          59,432 bytes maximum slop
              13 MB total memory in use (0 MB lost due
              to fragmentation)
                     Tot time (elapsed) Avg pause Max pause
Gen 0 228 colls,228 par 0.402s   0.045s    0.0002s   0.0007s
Gen 1 14 colls,13 par   0.064s   0.023s    0.0017s   0.0035s
Parallel GC work balance: 69.17% (serial 0%, perfect 100%)
TASKS: 6 (1 bound, 5 peak workers (5 total), using -N2)
SPARKS: 0 (0 converted,0 overflowed,0 dud,0 GC'd,0 fizzled)
INIT    time    0.002s  (0.001s elapsed)
MUT     time    3.018s  (1.718s elapsed)
GC      time    0.467s  (0.069s elapsed)
EXIT    time    0.001s  (0.004s elapsed)
Total   time    3.487s  (1.791s elapsed)
Alloc rate    121,286,650 bytes per MUT second
Productivity  86.6% of total user, 96.1% of total elapsed
```

mentre su quattro processori:

```
root@andysinx-dellOptiplex980:/home/andrea
/Documenti/Par_Haskell/Par_Inferencer>#
>./parinfer <benchmark.in  +RTS -s -N4
[("*",All . Int -> Int -> Int),
("+",All . Int -> Int -> Int),
("-",All . Int -> Int -> Int),
("/",All . Int -> Int -> Int),
("a",All a4095. a4095 -> a4095),
("b",All a4095. a4095 -> a4095),
("c",All a4095. a4095 -> a4095),
("d",All a4095. a4095 -> a4095),
("id",All a0. a0 -> a0)]
     366,116,224 bytes allocated in the heap
      57,103,184 bytes copied during GC
       6,062,560 bytes maximum residency (10 sample(s))
          93,976 bytes maximum slop
              22 MB total memory in use (0 MB lost due
              to fragmentation)
                     Tot time (elapsed) Avg pause Max pause
Gen 0 131 colls,131 par 0.433s  0.031s    0.0002s   0.0005s
Gen 1 10 colls,9 par  0.095s    0.017s    0.0017s   0.0036s
Parallel GC work balance: 78.66% (serial 0%, perfect 100%)
TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)
SPARKS: 0 (0 converted,0 overflowed,0 dud,0 GC'd,0 fizzled)
INIT    time    0.003s  (0.002s elapsed)
MUT     time    3.323s  (0.937s elapsed)
GC      time    0.528s  (0.048s elapsed)
EXIT    time    0.001s  (0.005s elapsed)
Total   time    3.855s  (0.992s elapsed)
Alloc rate    110,179,780 bytes per MUT second
Productivity  86.2% of total user, 95.0% of total elapsed
```

Come si può vedere, su quattro core è quasi lo stesso di due core. Ma questo è prevedibile: si hanno solo quattro problemi indipendenti, quindi il meglio che si può fare è sovrapporre i primi tre e quindi eseguire quello finale. Pertanto, il programma impiegherà lo stesso tempo impiegato con due core, in cui potrà sovrapporre due problemi alla volta. L'aggiunta del quarto core consente di sovrapporre tutti e quattro i problemi, con un conseguente aumento della velocità a 3,85s. Come si può vedere in **Threadscope**, la situazione è la seguente:

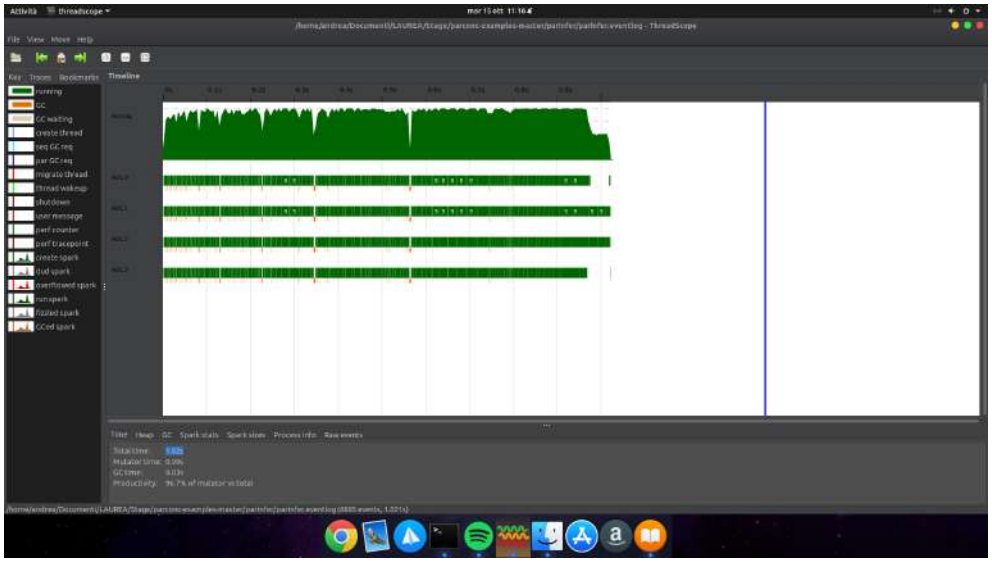

Figura 5.5: Threadscope profile per ParInfer su 4 core

Come si può vedere, l'attività parallela è molto concentrata ossia si ha una quantità molto alta di parallelismo, in quanto tutti e quattro i core lavorano in parallelo. Infatti è possibile vedere la differenza usando un solo core. Anche se l'attività è molto concentrata anche su singolo core, i tempi tendono ad aumentare. Quindi si conclude affermando che su 4 core lo speedup è migliorato. In totale su 4 core ha impiegato 1.02s mentre su 1 core ha impiegato 3.41s. Quello che è molto importante è il fatto che questo esempio si adatta molto bene con la `Monad Par`, infatti lo stesso trucco può essere fatto con le `Strategies`. Il problema delle `Strategies` è che utilizzano un certo criterio di valutazione mentre la `Monad Par` no. Infatti si parallelizza meglio con la `Monad Par`, ossia è più naturale e meno complesso. L'esecuzione su quattro core è stata ritentata, non è la stessa di quella a Pagina 53.

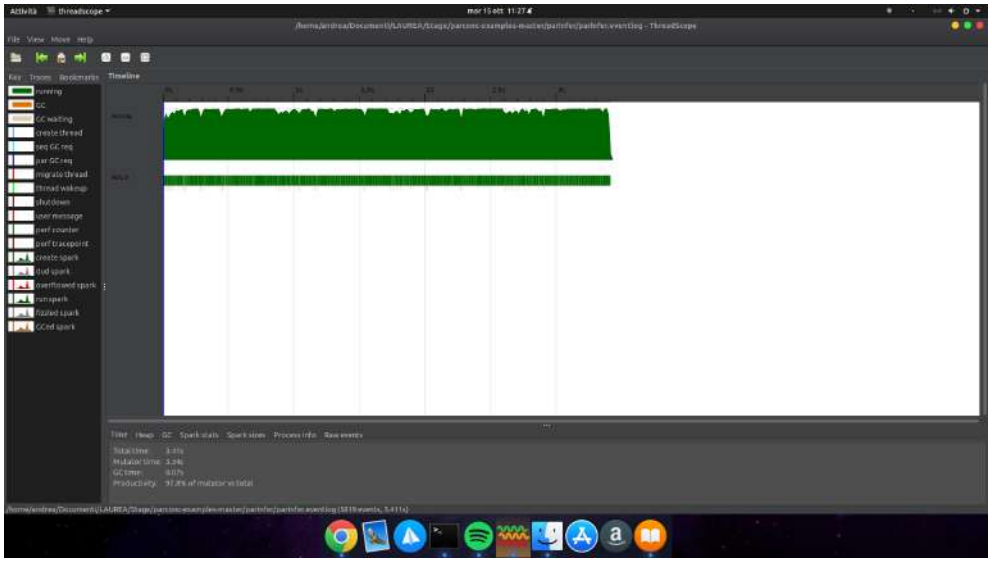

Figura 5.6: Threadscope profile per ParInfer su 1 core

Però come si può vedere, eseguire l'applicazione su 4 core è quasi lo stesso di 2 core (1.80s). I tempi totali di esecuzione sono molto vicini.

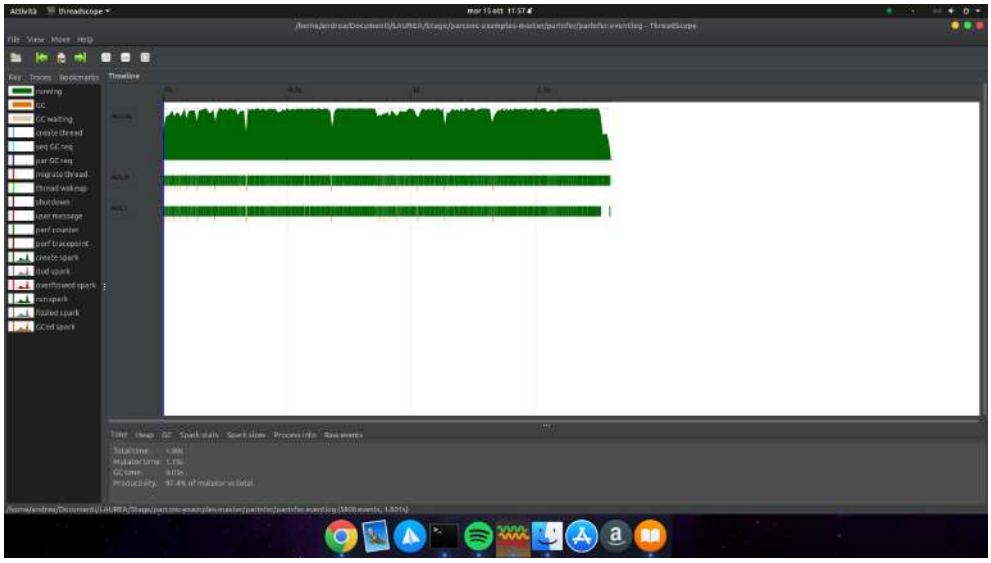

Figura 5.7: Threadscope profile per ParInfer su 2 core

Il fatto che i tempi distino molto dai tempi dell'esecuzione a Pagina 53, è dato dal fatto che la `Monad Par` non è attualmente integrata in **Threadscope**. L'importante è la barra delle attività e non tanto il tempo segnalato dal tool. Il codice è disponibile all'indirizzo: `https://github.com/simonmar/parconc-examples/tree/master/parinfer`.

5.2 Conclusioni

5.2.1 Confronto tra Par Monad e Strategies

Riassumendo, i modelli di programmazione parallela in **Haskell** presentati sono due, ognuno con vantaggi e svantaggi. La maggior parte dei benchmark di **Parallel Haskell** ottengono risultati simili sia se codificati con `Strategy` sia se codificati con `Monad Par`.

Nella Sezione 4.2.4 abbiamo delineato alcune difficoltà nell'uso dell'operatore par e `Strategy`.

La `Monad Par` è più facile da usare rispetto alla `rpar`?

- Ogni `fork` crea esattamente un'attività parallela e le dipendenze tra le attività sono esplicitamente rappresentate da `IVar`. La struttura parallela programmata nella `Monade Par` è ben definita: abbiamo fornito una semantica operativa nel la Sezione 4.2 in termini di operazioni della stessa monade.

- Il programmatore non ha bisogno di ragionare sulla lazyness. Infatti abbiamo deliberatamente reso l'operazione `put` molto rigorosa di default, per evitare qualsiasi confusione che potrebbe derivare dalla comunicazione di valori lazy attraverso un `IVar`. Quindi, se il programma è scritto in modo tale che ogni attività parallela legge solo input da `IVar` e produce output `IVar`, il programmatore può ragionare sul costo di ogni attività parallela e sul parallel task che può essere raggiunto.

Tuttavia, la `Monad Par` non impedisce che le computazioni lazy vengano condivise tra thread e il programmatore spericolato potrebbe persino essere in grado di catturare del parallelismo (inaffidabile) usando solo `fork` e lazyness. Probabilmente questo non è un bene: lo scheduler della `Monad Par` non può rilevare un thread bloccato su un calcolo lazy e può pianificare invece un altro thread. La condivisione di calcoli lazy tra thread nella `Monad Par` deve quindi essere evitata, ma sfortunatamente ci manca un modo per impedirlo staticamente. Un vantaggio chiave dell'approccio `Strategy` è che consente di separare l'algoritmo dal coordinamento parallelo, facendo sì che l'algoritmo produca una struttura di dati pigra che viene consumata dalla `Strategy` . La `Monad Par` non fornisce questa forma di modularità. Tuttavia, molti algoritmi non si prestano a essere decomposti in questo modo anche con `Strategy`, perché spesso è scomodo produrre una struttura dati lazy come output. Si ritiene che gli higher-order skeleton (modellini: ad esempio produttore-consumatore) siano più efficienti nel fornire parallelismo modulare. `Strategy` supporta la speculazione in un senso strettamente più forte di `Par`. In `Strategies`, il parallelismo speculativo può essere eliminato dal **Garbage Collector** quando risulta non referenziato, mentre in `Par`, tutto il parallelismo speculativo deve essere eseguito. Lo scheduler per `Strategies` è integrato nel sistema di runtime, mentre lo scheduler `Monad Par` è scritto in **Haskell**, che ci consente di utilizzare facilmente diverse politiche di pianificazione. `Strategies` include `parBuffer` che può essere utilizzato per valutare in parallelo elementi di una lista lazy. Non c'è niente di esattamente equivalente in `Par`: tutto il parallelismo deve essere completato nel momento in cui `runPar` dovrà ritornare il valore, quindi non possiamo restituire un elenco lazy da `runPar` con elementi valutati in parallelo. È possibile programmare qualcosa equivalente a `parBuffer` all'interno della `Monad Par`. Infine, `Par` e `Strategy` possono essere combinati in modo sicuro. Il sistema di runtime GHC darà la priorità ai calcoli del `Par` piuttosto che alle `Strategy`, perché gli sparks vengono valutati solo quando non c'è altro lavoro. Inoltre la `Monad Par` non è attualmente integrata in **Threadscope** mentre la `Monad Eval` si, quindi in programmi in cui viene usata `Monad Eval` si può effettuare diagnostica mentre con la `Monad Par` no.

Si legga l'articolo [5].

Ringraziamenti

Desidero ringraziare in primo luogo il prof. Ugo de' Liguoro per l'impagabile aiuto fornito nella redazione di questo lavoro, le spiegazioni dettagliate, la pazienza e la precisione nei suggerimenti, le soluzioni fornite, la competenza, la disponibilità, e per tutto ciò che mi ha insegnato durante questo percorso. Rivolgo un ringraziamento particolare alla mia famiglia che ha potuto sostenermi economicamente e moralmente durante questi anni di studio. Un grazie anche alla mia ragazza Federica per essermi stata vicina nei momenti più difficili e ai miei amici e colleghi universitari per avermi fornito supporto durante questi anni.

Bibliografia

[1] S. Marlow, *Parallel and Concurrent Programming in Haskell*. Gravenstein Highway North, Sebastopol: O'Reilly-Media, first ed., 2013. Info at https://www.oreilly.com/library/view/parallel-and-concurrent/9781449335939/, (07/10/2013).

[2] S. P. Jones, "Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell," Sept. 2009. Available at https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/mark.pdf, (7/03/2010).

[3] P. Bone and Z. Somogyi, "Profiling parallel mercury programs with threadscope," Available at https://www.mercurylang.org/documentation/papers/threadscope.pdf.

[4] S. P. J. Simon Marlow, Ryan Newton, "A monad for deterministic parallelism," Sept. 2009. Available at https://www.microsoft.com/en-us/research/wp-content/uploads/2011/01/monad-par.pdf, (1/11/09).

[5] H. W. L. M. K. A. P. T. Simon Marlow, Patrick Maier, "Seq no more : Better strategies for parallel haskell," Jan. 2009. Available at https://simonmar.github.io/bib/papers/strategies.pdf, (04/01/2009).