UNIVERSITÀ DEGLI STUDI DI PADOVA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA TRIENNALE IN INGEGNERIA INFORMATICA

CONCORRENZA E PARALLELISMO IN HASKELL

Relatore: Prof. Michele Moro

Laureando: Luca Bertipaglia

ANNO ACCADEMICO 2012 – 2013

INDICE

Introduzione

Sebbene molti linguaggi di programmazione al giorno d'oggi forniscano alcune strutture per la programmazione parallela e concorrente, molti pochi ne offrono cosi tante come Haskell. Il linguaggio di programmazione Haskell è un terreno fertile dove costruire astrazioni; la concorrenza e il parallelismo non fanno eccezione. Naturalmente il mondo della concorrenza e del parallelismo è cosi vasto che risulta difficile trovare un linguaggio che si addica perfettamente a tutti i compiti, e focalizzarsi su un particolare paradigma potrebbe portare il linguaggio a favorire certi tipi di problemi. In ogni caso Haskell ci fornisce una cosi vasta quantità di librerie e astrazioni da permetterci di disporre degli strumenti per risolvere praticamente ogni compito che ci venga richiesto.
Introdurremo i principali modelli di programmazione concorrente e parallela che Haskell ci fornisce.

Strumenti e risorse

Per provare Parallel e Concurrent Haskell si deve installare la piattaforma Haskell che include il compilatore GHC e tutte le più importanti librerie comprese le librerie parallele e concorrenti che tratteremo in questo elaborato.
Per installare la piattaforma Haskell seguire il link http://www.haskell.org/platform e seguire le istruzioni riportate.
Consigliamo inoltre di installare ThreadScope che è uno strumento per visualizzare l'esecuzione dei programmi Haskell e risulta essere particolarmente utile per approfondire la conoscenza del comportamento del codice Haskell parallelo e concorrente. È possibile trovare ThreadScope al seguente link: http://www.haskell.org/haskellwiki/ThreadScope

Presentazione del linguaggio Haskell

Haskell è un linguaggio di programmazione per computer polimorfico, statically typed (a tipizzazione statica), lazy e puramente funzionale molto diverso dalla maggior parte degli altri linguaggi di programmazione imperativi e object oriented. Il linguaggio prende il nome da Haskell Brooks Curry il cui lavoro in logica matematica serve come base per i linguaggi funzionali. Haskell è basato sul calcolo lambda che ne determina il logo.

Perché usare Haskell

Scrivere sistemi software di grandi dimensioni è un lavoro molto difficile e costoso. I linguaggi di programmazione funzionali, come per esempio Haskell, possono semplificare il lavoro e ridurne i costi.
Utilizzare Haskell risulta semplice perché offre:

- Un aumento sostanziale di produttività per il programmatore (Ericsson ha misurato un fattore di miglioramento tra 9 e 25 utilizzando Erlang, un linguaggio di programmazione funzionale simile a Haskell, in una serie di esperimenti su software di telefonia);
- Codice più conciso, chiaro e gestibile;
- Meno errori e maggiore affidabilità;
- Un minore gap semantico tra il programmatore e il linguaggio;
- Tempi di consegna più brevi.

Haskell è un linguaggio ad ampio spettro, adatto ad una varietà di applicazioni; si addice particolarmente a programmi che dovranno essere altamente modificabili e gestibili.
Gran parte della vita di un prodotto software in sé viene spesa nella specifica, progettazione e manutenzione e non nella programmazione. I linguaggi di programmazione funzionali sono eccezionali per scrivere le specifiche che possono essere effettivamente eseguite e quindi testate e corrette. Tale specifica è quindi il primo prototipo del programma definitivo.
I programmi funzionali sono anche relativamente facili da mantenere perché il codice è più breve e più chiaro, ed il controllo rigoroso degli effetti collaterali elimina una classe enorme di interazioni impreviste.

Che cos'è un programma funzionale?

C, Java, Pascal e così via sono tutti linguaggi imperativi; sono imperativi nel senso che consistono in sequenze di comandi che vengono eseguiti rigorosamente uno dopo

l’altro. Haskell invece è un linguaggio funzionale ed in un programma funzionale è la singola espressione che viene eseguita valutando l’espressione.
Chiunque abbia utilizzato un foglio di calcolo ha esperienza di calcolo funzionale; in un foglio elettronico si specifica il valore di una cella in base al valore delle altre celle. L’attenzione si focalizza su ciò che deve essere effettivamente calcolato e non sul come dovrà essere fatto. Per esempio non si specifica l’ordine in cui le celle devono essere calcolate, invece diamo per scontato che il foglio di calcolo calcolerà le celle in un ordine che rispetti le loro dipendenze. Non diciamo al foglio di calcolo come allocare la memoria, anzi ci aspettiamo che si presenti come uno spazio apparentemente infinito di celle e che allochi memoria solo per quelle celle che effettivamente sono in uso. Si specifica il valore di una cella da un'espressione (le cui parti possono essere valutati in qualsiasi ordine), piuttosto che da una sequenza di comandi per calcolarne il valore. Una conseguenza interessante del non avere un ordine specificato di calcolo del foglio di lavoro è che la nozione di assegnazione non è molto utile; nel senso che non viene usata una variabile nel senso tradizionale Dopo tutto, se non si sa esattamente quando avverrà un’assegnazione, non si dovrà né potrà occuparsi di essa. Questo contrasta fortemente con i programmi in linguaggi tradizionali come C, che consistono essenzialmente in una sequenza attentamente specificata di istruzioni, o Java, in cui l'ordinamento di chiamate di metodo è fondamentale per il significato di un programma. Questo concentrarsi ad alto livello sul "che cosa" piuttosto che a basso livello sul "come" è una caratteristica distintiva di linguaggi di programmazione funzionali.

Facili da capire

Programmi funzionali sono spesso più facili da capire; di solito è possibile comprenderne il significato a colpo d'occhio. Lo stesso non si può certo dire dei programmi C. Ci vuole un bel po' per capirli, e anche quando vengono compresi, è estremamente facile fare un piccolo errore e ritrovarsi con un programma incorretto.

Per esemplificare, ecco una spiegazione dettagliata del quicksort Haskell:

```
quicksort [] = []
```

La prima clausola recita: "Il risultato dell’ordinare una lista vuota ([]) è solo una lista vuota ([])".

```
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
where
lesser  = filter (< p) xs
greater = filter (>= p) xs
```

La seconda clausola recita: "Il risultato dell'ordinamento di una lista non vuota il cui primo elemento sarà d'ora in poi indicato come p e il resto che verrà indicato come xs, è il risultato del concatenamento di tre sottoliste: la prima è il risultato dell'ordinamento gli elementi di xs che sono minori di p, poi p stesso, e quindi il risultato dell'ordinamento elementi di xs che sono maggiori o uguali a p. "

Brevità

Programmi funzionali tendono ad essere molto più concisi, più brevi di un fattore di 2-10 in genere, rispetto alle loro controparti imperative.
Il codice qui sopra può essere riscritto anche in modo più conciso:

qsort (p: xs) = qsort [x | x <-xs, x <p] + + [p] + + qsort [x | x <-xs, x> = p]

La prima espressione significa, per x tratti da xs in ordine, tali che x <p, raccogliere xs in un elenco e chiamare la funzione qsort su di esso, in modo ricorsivo. Allo stesso modo per l'ultimo.

No core dumps

La maggior parte dei linguaggi funzionali e Haskell, in particolare, sono fortemente tipizzati, eliminando una classe enorme di errori comuni in fase di compilazione. In particolare, la tipizzazione forte significa no core dump! Semplicemente non c'è alcuna possibilità di trattare un intero come un puntatore, o puntare a un puntatore nullo.

Riutilizzo del codice

Naturalmente, la tipizzazione forte è disponibile in molti linguaggi imperativi, come Ada o Pascal. Tuttavia, il sistema di tipo Haskell è molto meno restrittivo rispetto, ad

esempio, a Pascal, perché utilizza il polimorfismo. Ad esempio, il programma qsort visto qui sopra non solo ordina gli elenchi di numeri interi, ma anche elenchi di numeri in virgola mobile, liste di caratteri, liste di liste, anzi serve ad ordinare liste di qualsiasi cosa per cui ha senso avere operazioni di "minore di" e "maggiore di". Al contrario, la versione C può solo ordinare un array di interi, e nient'altro. Il polimorfismo migliora la riutilizzabilità.

Strong glue

"Non rigidi" linguaggi funzionali, come ad esempio Haskell, hanno un'altra caratteristica potente: valutano solamente la parte del programma necessaria per ottenere una risposta; questo viene chiamato valutazione pigra. Non rigidi linguaggi forniscono esattamente questo tipo di valutazione on demand. Le strutture dati sono valutate solamente quanto basta per fornire la risposta, e parti di esse possono anche non essere valutate affatto**.** Questo significa che è possibile il riutilizzo dei programmi, o pezzi di programmi, molto più spesso di quanto si possa fare in un ambiente imperativo. La valutazione pigra ci permette di scrivere programmi più modulari.

Astrazioni potenti

In generale, i linguaggi funzionali offrono nuovi modi efficaci per incapsulare astrazioni. Un’astrazione consente di definire un oggetto il cui funzionamento interno è nascosto; una procedura C, per esempio, è un'astrazione. Le astrazioni sono la chiave per la realizzazione di programmi modulari e gestibili per cui è importante valutare i meccanismi di astrazione che un linguaggio fornisce. Un meccanismo potente di astrazione disponibile nei linguaggi funzionali è la funzione di ordine superiore. Una funzione Haskell può liberamente essere passata ad altre funzioni, venir restituita come risultato di una funzione, venir memorizzata in una struttura dati, e così via. Si scopre che l'uso giudizioso delle funzioni di ordine superiore può migliorare sostanzialmente la struttura e la modularità di molti programmi.

Gestione della memoria integrata

Molti sofisticati programmi hanno bisogno di allocare memoria dinamica in un heap; alcuni forniscono anche strumenti di gestione integrata ma risultano talmente onerosi che per sistemi di grandi dimensioni è preferibile una gestione manuale.
Invece ogni linguaggio funzionale solleva il programmatore dall'onere della gestione della memoria. La memoria viene allocata e inizializzata implicitamente, e recuperata automaticamente dal Garbage collector. La tecnologia di allocazione della memoria e garbage collection è ormai ben sviluppata, e l'incidenza sui costi è piuttosto lieve.

Quando C è meglio

Non è tutto rose e fiori, ovviamente. Il C utilizza una tecnica estremamente ingegnosa, inventata da Hoare, per cui viene ordinato l'array in posizione, cioè, senza l'utilizzo di spazio di archiviazione aggiuntivo. Come risultato, viene eseguito rapidamente, e in una piccola quantità di memoria. Invece, Haskell alloca un sacco di memoria extra dietro le quinte, e esegue più lento del programma C. Nelle applicazioni in cui sono richieste prestazioni ad ogni costo, o quando l'obiettivo è un algoritmo di basso livello, un linguaggio imperativo come il C sarebbe probabilmente una scelta migliore di Haskell, proprio perché fornisce un maggior controllo sul modo esatto in cui il calcolo viene effettuato.

Funzionale vs imperativo.

Tuttavia non tutti i programmi richiedono prestazioni ad ogni costo. Dopo tutto, abbiamo tutti smesso di scrivere programmi in linguaggio assembly, se non forse per i principali cicli interni, molto tempo fa. I vantaggi di avere un modello di programmazione più solidale (un numero arbitrario di variabili locali invece di un numero fisso di registri, per esempio) superano di gran lunga i modesti costi di runtime. Allo stesso modo, si accettano di buon grado i costi di un sistema di paging della memoria virtuale, in cambio di un modello di programmazione più favorevole con uno spazio infinito di indirizzi virtuali ponendo fine alla rilocazione esplicita della memoria. I linguaggi funzionali sono un passo avanti rispetto agli altri linguaggi di programmazione di alto livello perché sono più facili da progettare, scrivere e gestire

sebbene offrano al programmatore un minor controllo sulla macchina. Per la maggior parte dei programmi il risultato è perfettamente accettabile.

Cos'è Haskell?

Haskell è un moderno, standardizzato linguaggio di programmazione puramente funzionale. Esso fornisce tutte le funzionalità presentate precedentemente, comprendente la tipizzazione polimorfica, la valutazione pigra e funzioni di ordine superiore. È' progettato appositamente per gestire una vasta gamma di applicazioni, da quelle numeriche a quelle simboliche. A tal fine, Haskell ha una sintassi espressiva, e una ricca varietà di tipologie di dati integrati, tra cui i convenzionali numeri interi, razionali, in virgola mobile e a precisione arbitraria ed anche i booleani.

Monadi

Una monade è una struttura dati con uno stato associato. Tipicamente è utilizzata per modellare un aspetto del mondo esterno al programma e permette ai linguaggi funzionali di rimanere puri, cioè senza effetti collaterali. Una monade incapsula lo stato dell'ambiente esterno e viene passata come parametro alle operazioni che hanno bisogno di conservare uno stato (un analizzatore sintattico, per esempio).
Le monadi sono notoriamente utilizzate nel linguaggio di programmazione Haskell per le operazioni di input/output. In Haskell, la funzione principale, il `main`, è una monade:

```
*Main> :t main
main :: IO ()
```

Quindi il suo tipo è `IO`. Allo stesso modo, tutte le funzioni che fanno input/output sono monadi:

```
*Main> :t putStr
putStr :: String -> IO ()
```

`putStr` prende una stringa come parametro e restituisce una monade, essa è quindi anche una funzione di ordine superiore (cioè una funzione che prende come input o restituisce come output un'altra funzione).

Concorrenza e parallelismo

Mentre la maggior parte dei linguaggi di programmazione al giorno d'oggi fornisce una qualche forma di struttura per la programmazione parallela e concorrente, molti pochi offrono un'ampia gamma come Haskell. Il linguaggio Haskell è un terreno fertile su cui costruire astrazioni, e la concorrenza e il parallelismo non fanno eccezione qui. Ci sono buone ragioni per credere che nessuno offra proprio tutti i modelli di programmazione per la concorrenza e il parallelismo e quindi focalizzarsi su un particolare paradigma di programmazione porterebbe il linguaggio a favorire certi tipi di problemi. Quindi in Haskell ci concentriamo nel fornire una vasta gamma di astrazioni e librerie, in modo che per ogni problema dato sia possibile trovare uno strumento che si adatti al compito assegnato.
Introdurremo i principali modelli di programmazione disponibili per la programmazione concorrente e parallela in Haskell. Naturalmente il campo da coprire è troppo vasto e ci occuperemo solamente degli aspetti fondamentali.

Strumenti e risorse

Per provare Haskell parallelo e concorrente è necessario installare la piattaforma Haskell. La piattaforma Haskell comprende il compilatore GHC e tutte le librerie importanti, tra cui le librerie parallele e concorrenti che utilizzeremo.
Inoltre, si consiglia di installare ThreadScope 2. ThreadScope è uno strumento per visualizzare l'esecuzione dei programmi Haskell, ed è particolarmente utile per visualizzare le informazioni sul comportamento del codice Haskell parallelo e concorrente.

Terminologia: Parallelismo e concorrenza

In molti campi le parole *parallelismo* e *concorrenza* sono sinonimi, non è così per la programmazione, in cui vengono utilizzate per descrivere concetti fondamentalmente differenti. Un *programma parallelo* utilizza una molteplicità di calcolo hardware (ad esempio vari core) per eseguire i calcoli più rapidamente. Parti differenti del calcolo sono delegate a diversi processori che eseguono contemporaneamente (in parallelo), in

modo tale che i risultati possano essere consegnati prima che se il calcolo fosse stato eseguito sequenzialmente.

La *concorrenza*, invece, è una tecnica di programmazione in cui vi sono più thread di controllo. Teoricamente i thread di controllo vengono eseguiti in contemporanea, anche se l'utente vede i loro effetti susseguirsi**.** Se effettivamente vengono eseguiti nello stesso momento o no è dovuto all'implementazione; un programma concorrente può essere eseguito su un singolo processore attraverso l'esecuzione stratificata, o su più processori fisici.

Mentre la programmazione parallela riguarda soltanto l'efficienza, la programmazione concorrente si occupa di strutturare un programma che ha bisogno di interagire con vari agenti esterni indipendenti (ad esempio l'utente, un database, e alcuni agenti esterni). La concorrenza permette a questi programmi di essere modulari, il thread che interagisce con l'utente è distinto da quello che interroga il database. In assenza di concorrenza, tali programmi devono essere scritti con cicli di eventi e callback; infatti, cicli di eventi e callback sono spesso utilizzati anche quando la concorrenza è disponibile perché in molti linguaggi la concorrenza è troppo onerosa, o troppo difficile da utilizzare. La nozione di thread di controllo perde di significato in un programma puramente funzionale, perché non ci sono effetti da osservare, e l'ordine di valutazione è irrilevante.

Specifichiamo ora la differenza tra i modelli di programmazione *deterministici* e *non deterministici*.

Un modello di programmazione *deterministico* è quello in cui ogni programma può dare un solo risultato, mentre uno *non deterministico* ammette che i programmi possano ottenere risultati differenti, a seconda dei vari aspetti dell'esecuzione. I modelli di programmazione concorrenti sono necessariamente non deterministici perché devono interagire con gli agenti esterni che causano eventi a volte imprevedibili. Il determinismo ha alcuni svantaggi notevoli però, i programmi diventano significativamente più difficili da testare e realizzare. Vorremmo che i nostri programmi paralleli fossero deterministici per quanto possibile. Dal momento che l'obiettivo è solo quello di arrivare alla risposta più rapidamente preferiremmo non rendere il nostro programma più difficile da ricontrollare. Programmi paralleli deterministici sono i migliori perché il debug, il test e il ragionamento per svilupparli possono essere fatti sequenzialmente nonostante si possa eseguire il programma più rapidamente aggiungendo più processori. In effetti, la maggior parte dei processori per computer attuano loro stessi un parallelismo deterministico in forma di pipelining e unità di esecuzione multiple. Inoltre è possibile creare programmi paralleli usando la concorrenza, ma spesso si rivela una scelta sbagliata perché la concorrenza sacrifica il

determinismo. In Haskell, i modelli di programmazione parallela sono deterministici. Tuttavia, è importante notare che i modelli deterministici di programmazione non sono sufficienti per esprimere tutti i tipi di algoritmi paralleli; ci sono algoritmi che dipendono da un non determinismo interno, in particolare i problemi che si occupano della ricerca di uno spazio delle soluzioni. In Haskell, questa classe di algoritmi è esprimibile solo con la concorrenza. Infine, è del tutto ragionevole voler mescolare parallelismo e concorrenza nello stesso programma. La maggior parte dei programmi interattivi necessita dell'utilizzo della concorrenza per mantenere un'interfaccia utente reattiva, mentre i compiti di elaborazione più onerosi vengono eseguiti.

Parallel Haskell

Parallel Haskell consiste nel far eseguire i programmi Haskell più velocemente dividendo il lavoro da fare tra più processori. Ora che i produttori di processori hanno in gran parte rinunciato a cercare di spremere maggiori prestazioni da singoli processori e hanno concentrato la loro attenzione nel fornirci più processori, i maggiori guadagni in termini di prestazioni si possono avere utilizzando tecniche parallele nei nostri programmi in modo da sfruttare pienamente questi ulteriori core. Ci si potrebbe anche chiedere se il compilatore potrebbe automaticamente parallelizzare i programmi per noi. Dopo tutto, dovrebbe essere più facile farlo in un linguaggio funzionale puro in cui le dipendenze trai vari calcoli sono date solamente dalle dipendenze tra dati che, essendo più chiare, risultano quindi facilmente analizzabili. Al contrario, quando gli effetti sono senza restrizione, l'analisi delle dipendenze tende ad essere molto più difficile, portando ad una maggiore approssimazione e un grado elevato di false dipendenze. Tuttavia, anche in un linguaggio con solo le dipendenze tra i dati, la parallelizzazione automatica soffre ancora del vecchio problema della gestione di processi in parallelo che necessita di un certo controllo relativo all'esecuzione sequenziale e quindi ha un overhead intrinseco, quindi la dimensione dei task in parallelo deve essere grande sufficiente a superare l'overhead. L'analisi dei costi in fase di compilazione è difficile, quindi un approccio è quello di utilizzare un Runtime profiler per individuare i compiti che sono abbastanza costosi e possono essere eseguiti in parallelo, e fornire questa informazione al compilatore. Anche questo, tuttavia, non porta grossi miglioramenti nella pratica e la parallelizzazione completamente automatica risulta essere ancora un sogno irrealizzabile. Tuttavia i modelli di programmazione in parallelo forniti da Haskell

riescono ad eliminare alcuni aspetti noti o soggetti a errori tradizionalmente associati alla programmazione parallela:

- La programmazione parallela in Haskell è deterministica: il programma parallelo produce sempre la stessa risposta, indipendentemente dal numero di processori che vengono utilizzati per eseguirlo; è inoltre possibile eseguire il debug dei programmi paralleli senza peraltro doverli effettivamente eseguire in parallelo.

- I programmi paralleli in Haskell non hanno a che fare in modo esplicito con la sincronizzazione o la comunicazione. La *sincronizzazione* è l'atto di aspettare che altri compiti vengano completati, probabilmente a causa di dipendenze tra dati. La *comunicazione* comporta la trasmissione di risultati tra i processi in esecuzione su differenti processori. La sincronizzazione è gestita automaticamente dal sistema runtime GHC e/o dalle librerie parallele. La comunicazione è implicita in GHC in quanto tutti i processi condividono lo stesso heap, e possono condividere gli oggetti senza restrizioni. In questo contesto, anche se non c'è una comunicazione esplicita a livello di programma o anche a livello di esecuzione, a livello hardware la comunicazione riemerge per la trasmissione di dati tra le cache dei diversi processori. Un eccessiva comunicazione può causare un conflitto per il bus di memoria principale, e tali overhead possono essere difficili da diagnosticare.

Parallel Haskell impone al programmatore di occuparsi del *partizionamento*. Il lavoro del programmatore è quello di suddividere il lavoro in attività che possono essere eseguiti in parallelo. Idealmente, vogliamo un numero sufficiente di task da mantenere tutti i processori impegnati per l'intera fase di esecuzione. Tuttavia, i nostri sforzi potrebbero risultare vani a causa di due importanti aspetti:

- La *granularità*. Se facciamo i nostri task troppo piccoli il sovraccarico di gestione delle attività potrebbe superare ogni vantaggio che avremmo dall'esecuzione in parallelo. Quindi la granularità deve essere abbastanza grande da contrastare l'overhead, ma non troppo grande, perché allora rischiamo di non avere abbastanza lavoro per mantenere tutti i processori occupati, soprattutto verso la fine dell'esecuzione, quando ci sono meno compiti rimasti.

- Le *dipendenze tra dati* tra le attività rafforzano la sequenzializzazione. I due modelli di programmazione parallela di GHC adottano approcci differenti riguardo alle dipendenze di dati: con le Strategie, le dipendenze tra dati sono del tutto implicite, mentre con la monade Par, sono esplicite. Questo rende la programmazione con le strategie un po' più concisa, a scapito della possibilità che le dipendenze nascoste possano causare una sequenzializzazione in fase di runtime.

Descriveremo successivamente due modelli di programmazione parallela forniti dal GHC. Il primo, Strategie di valutazione (Strategie in breve), è ben consolidata e ci sono molti buoni esempi dell'utilizzo di Strategie per scrivere programmi paralleli Haskell. Il secondo è un modello di flusso di dati di programmazione basato sulla monade Par. Questo è un modello di programmazione più recente in cui è possibile esprimere il coordinamento parallelo più esplicitamente che con le Strategie, anche se a scapito della stringatezza e modularità delle strategie.

Specifiche GHC per la concorrenza

È possibile ottenere l'accesso alle operazioni di concorrenza importando la libreria Control.Concurrent.
Dal 2004, GHC supporta l'esecuzione di programmi in parallelo su una macchina SMP o multi-core. Per fare questo si deve:

1. Scaricare l'ultima versione di GHC

2. Compilare il proprio programma con l'opzione `-threaded` .

3. Eseguire il programma con l'opzione `+RTS -N2` per usare 2 thread, per esempio (RTS indica il runtime system). SI usi un valore `-N` pari al numeri dei processor della propria macchina (senza contare gli Hyper-threading cores). Dalla versione v6.12 di GHC si può omettere il numero di processori e tutti i processori disponibili verranno utilizzati (è comunque necessario inserire l'opzione -N tuttavia, in questo modo: + RTS-N).

4. Thread concorrenti (`forkIO`) eseguiranno in parallelo, ed è anche possibile utilizzare il combinatore par e le strategie dal modulo Control.Parallel.Strategies per creare parallelismo.

5. Usare `+RTS -sstderr` per le statistiche temporali.

6. Per eseguire il debug delle prestazioni parallele del programma, utilizzare ThreadScope.

Parallelismo Base: la monade Eval

L'argomento + RTS-s indica al sistema runtime GHC di emettere delle statistiche che risulteranno particolarmente utili come primo passo per l'analisi delle prestazioni in parallelo. L'output è spiegato in dettaglio nella Guida per l'utente GHC, ma per i nostri scopi siamo interessati in particolare ad una voce cioè il Tempo totale. Questo è dato in due forme: il primo è il tempo totale della CPU usato dal programma, e il secondo è il tempo trascorso per l'esecuzione, o il wall-clock time. Se il programma viene eseguito su un singolo processore, questi tempi sono identici (a volte il tempo trascorso potrebbe essere leggermente maggiore a causa di altre attività del sistema).

Abbiamo bisogno di un po' di funzionalità di base per esprimere parallelismo, che è fornito dal modulo Control.Parallel.Strategies:

```
data Eval a
instance Monad Eval
runEval :: Eval a -> a
rpar :: a -> Eval a
rseq :: a -> Eval a
```

Il coordinamento parallelo sarà effettuato in una monade, vale a dire la monade Eval. La ragione di questo è che la programmazione parallela comporta fondamentalmente l'ordinare gli oggetti, iniziare a valutare a in parallelo, e quindi valutare b. Le monadi sono ottime per esprimere relazioni di ordinamento in modo composto. La monade Eval fornisce una operazione runEval che ci permette di estrarre il valore da Eval. Si noti che runEval è completamente puro, non c'è bisogno di essere nella monade IO qui. La

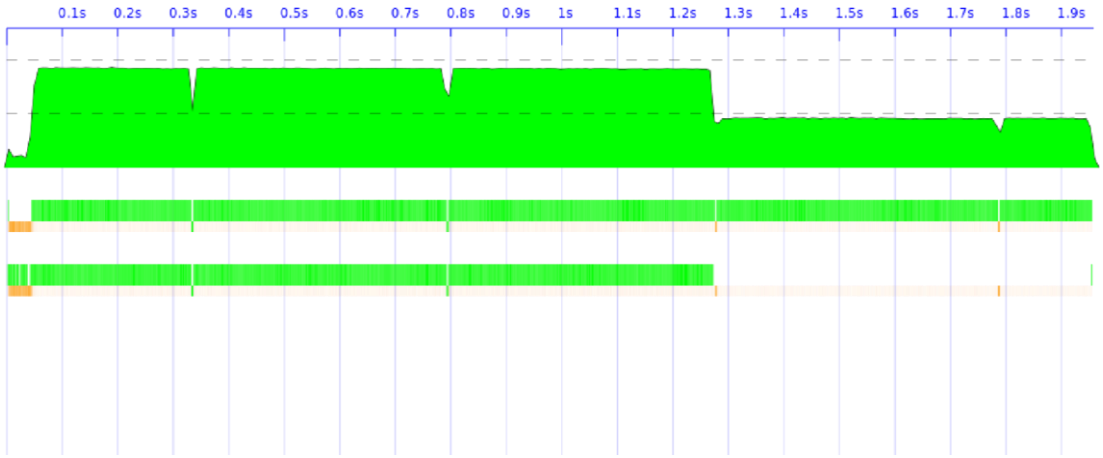
monade Eval è dotata di due operazioni di base, *rpar* e *rseq*. Il combinatore rpar viene utilizzato per la creazione del parallelismo, cioè dice “il mio argomento potrebbe essere valutato in parallelo", mentre rseq viene utilizzato per forzare la valutazione sequenziale: cioè dice “valuta il mio argomento ora "(in weak-head normal form). Queste due operazioni vengono utilizzate insieme solitamente; per esempio, per valutare A e B in parallelo, potremmo applicare rpar su A, seguita da rseq su B.
Per valutare pienamente un intera lista, compresi gli elementi dobbiamo aggiungere la funzione *deep*, che è definita come segue:

```
deep :: NFData a => a -> a
deep a = deepseq a a
```

deep valuta in profondità l'intera struttura dei suoi argomenti(riducendola in forma normale), prima di restituire l'argomento stesso. È definita come la funzione *deepseq*, che è disponibile nel modulo Control.DeepSeq. Non valutare abbastanza profondamente è un errore comune quando si utilizza la monade rpar, quindi è una buona idea mantenere l'abitudine di pensare, per ogni rpar, “quanto di questa struttura voglio valutare nel task parallelo? "(in effetti, è un problema comune nella monade Par che introdurremo in seguito, siamo arrivati al punto di rendere il deepseq comportamento predefinito).

Uno strumento utile per diagnosticare il parallelismo ottenuto nei nostri programmi è ThreadScope. Per controllare il programma utilizzando ThreadScope dobbiamo ricompilarlo con l'opzione -eventlog ed eseguirlo con +RTS –ls e quindi richiamare ThreadScope sul programma generato

$ rm esempio; ghc -O2 esempio.hs -threaded -rtsopts -eventlog
[2 of 2] Compiling Main (esempio.hs, esempio.o)
Linking esempio ...
$ ./esempio +RTS -N2 –ls
$ threadscope esempio.eventlog



Possiamo generare un grafico selezionando esporta in PNG "da ThreadScope, in modo che includa solamente la timeline del grafico, e non il resto della GUI ThreadScope. L'asse x del grafico è il tempo, e ci sono tre barre orizzontali che mostrano come il programma venga eseguito nel tempo. La barra più in alto è conosciuta come *barra delle attività*, e mostra quanti processori stanno eseguendo il codice Haskell (o un periodo di inattività o di garbage collection) in un determinato momento. Sotto la barra delle attività vi è una barra per processore che mostra ciò che il processore faceva in ogni punto nell'esecuzione. Ogni barra ha due parti, quella superiore e più spessa è verde quando il processore sta eseguendo codice Haskell, e quella inferiore e più stretta è di colore arancione o verde quando il processore esegue la raccolta di rifiuti.
Se nel grafico ci dovesse essere un punto alla fine dell'esecuzione dove un solo processore sta eseguendo, e l'altro è inattivo (eccetto per le operazioni ordinarie di garbage collection regolari, che sono necessarie) questo indicherebbe che le nostre attività parallele sono irregolari: una ci metterebbe molto più tempo ad eseguire rispetto alle altre, e quindi non stiamo sfruttando appieno i nostri processori, ciò si traduce in aumento di velocità tutt'altro che ottimale.
Questo illustra una distinzione importante tra due strategie di partizionamento:

- *Partizionamento statico* consiste nel dividere il lavoro secondo una certa politica predefinita (per esempio, suddividendo una lista in modo uguale in due parti).

- *Partizionamento dinamico* cerca invece di distribuire il lavoro in modo più uniforme, dividendo il lavoro in compiti più piccoli e assegnandoli ai processori solamente quando sono inattivi. Il sistema runtime GHC

supporta la distribuzione automatica dei task paralleli, tutto quello che dobbiamo fare per ottenere il partizionamento dinamico è dividere il problema in compiti abbastanza piccoli e il runtime farà il resto per noi. L'argomento di rpar viene chiamato *spark* (scintilla)**.** Il runtime raccoglie le spark in un pool **e** utilizza questo come fonte di lavoro da fare quando ci sono i processori disponibili, utilizzando una tecnica chiamata work stealing .

Le spark possono essere valutate ad un certo punto nel futuro, o potrebbero non esserlo affatto**,** dipende dalla disponibilità di processori liberi. Le spark sono molto semplici da creare (Rpar essenzialmente aggiunge solo un riferimento all'espressione di un array).

Ora proviamo a utilizzare il partizionamento dinamico**.** Prima definiamo un'astrazione che ci permetterà di applicare una funzione a un elenco in parallelo, *parMap*:

```
1 parMap :: (a -> b) -> [a] -> Eval [b]
2 parMap f [] = return []
3 parMap f (a:as) = do
4      b <- rpar (f a)
5     bs <- parMap f as
6     return (b:bs)
```

Questo è un po' come una versione monadica di map, se non che abbiamo usato rpar per portare l'applicazione della funzione f all'elemento a nella monade Eval. Quindi, parMap esegue lungo l'intero elenco, creando spark per l'applicazione di f per ogni elemento, e restituisce infine la nuova lista. Quando parMap termina, avrà creato una scintilla per ogni elemento della lista. Abbiamo ancora bisogno di valutare l'elenco dei risultati in sé, che è semplice con deep:

Inoltre, il sistema di runtime GHC ci dice quante spark sono state create, quante sono state convertite (cioè, trasformate in parallelismo reale in fase di esecuzione) e quante vengono invece potate (pruned) cioè rimosse dal pool delle spark dal sistema runtime o perché erano già stati valutate, o perché sono risultate non essere collegate al resto del programma, e così sono considerati inutili. Inoltre, visto che il runtime gestisce la distribuzione del lavoro per noi, il programma può automaticamente scalare per più processori.

Alcune operazioni preliminari del programma sono per forza sequenziali, per esempio la lettura di un file. Si potrebbe volere che la parte preliminare venisse fatta tutto in una volta, prima di iniziare il calcolo principale, in modo tale da dividere la parte sequenziale iniziale da quella parallela. Questo è possibile aggiungendo un evaluate che

non modifica il tempo di esecuzione globale ma ci permette di dividere la parte sequenziale da quella parallela rendendola visibile e analizzabile anche in Threadscope. Quando abbiamo una parte sequenziale di programma, questa incide sul massimo speedup parallelo che si può ottenere, che si può calcolare con *la legge di Amdahl*. La legge di Amdahl calcola l'aumento di velocità massima raggiungibile come il rapporto

$$\frac{1}{(1-P)+\frac{P}{N}}$$

dove P è la porzione del runtime che può essere parallelizzata, e N è il numero di processori disponibili. La legge di Amdahl ci dice non solo che l'incremento di velocità in parallelo diventa più difficile da raggiungere aggiungendo più processori, ma anche che in pratica la maggior parte dei programmi ha un massimo parallelismo teorico.

Strategie di valutazione

Le Strategie di valutazione sono un livello di astrazione costruito sulla monade Eval che permette che grandi specifiche parallele possano essere costruite in modo composito e permettono inoltre di descrivere in maniera modulare il coordinamento parallelo separando il parallelismo dall'algoritmo da parallelizzare. Una strategia è semplicemente una funzione nella monade Eval che accetta un valore di tipo A e restituisce lo stesso valore:

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

Le strategie sono funzioni di identità, cioè, il valore restituito da una strategia è equivalente al valore che è stato passato. Purtroppo la libreria non può garantire questa proprietà per le funzioni con Strategie definite dall'utente, ma vale anche per le funzioni di strategie forniti dal modulo Control.Parallel.Strategies. Abbiamo già visto alcune semplici strategie, rpar e rseq, anche se siamo in grado di dare i loro tipi, in termini di strategia:

```
rseq :: Strategy a
rpar :: Strategy a
```

Ci sono altri due membri di questa famiglia:

```
r0 :: Strategy a
r0 x = return x
rdeepseq :: NFData a => Strategy a
rdeepseq x = rseq ( deep x)
```

r0 è la strategia che non valuta nulla, e rdeepseq è la strategia che valuta l'intera struttura del suo argomento, che può essere definito in termini di deep che abbiamo visto in precedenza. Si noti che è necessario rseq qui; sostituendo rseq con return non verrà effettuata la valutazione immediatamente, ma sarebbe rinviata fino a quando il valore restituito da rdeepseq non viene richiesto (che potrebbe anche essere mai). Ci sono alcuni semplici modi per costruire strategie, ma come viene effettivamente utilizzata una strategia? Una strategia è solo una funzione che comporta un calcolo nella monade Eval, in modo da poter usare runEval. Ad esempio, applicare la strategia s per un valore x sarebbe semplicemente runEval (s x).
Questa è una struttura comune a cui la libreria Strategies dà il nome using:

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

using prende un valore di tipo a e una strategia per a, e applica la strategia al valore.
La proprietà di identità per le Strategie ci dà

```
x `using` s == x
```

che è una proprietà significativa delle strategie: ogni occorrenza di `using` s può essere cancellata senza affliggere la semantica. A rigor di termini ci sono due puntualizzazioni a questa proprietà. In primo luogo, come accennato in precedenza, funzioni di strategia definite dall' utente potrebbero non soddisfare la proprietà di identità. In secondo luogo, l'espressione x `using` s potrebbe essere meno definita di x, perché valuta più la struttura di x rispetto al contesto quindi l'eliminazione di `using` s potrebbe avere l'effetto di far terminare il programma con un risultato quando prima avrebbe generato un'eccezione o non sarebbe riuscito a terminare.
Rendere i programmi più definiti è generalmente considerato come un cambiamento positivo della semantica (anzi, l'ottimizzatore GHC può anche rendere i programmi più

definiti sotto determinate condizioni), ma in ogni caso si tratta solamente di un cambiamento di semantica.

Una strategia per la valutazione di un elenco in parallelo

Precedentemente abbiamo definito una funzione parMap che mappa una funzione su una lista in parallelo. Possiamo pensare parMap come composizione di due parti:

1. L'algoritmo: *map*

2. Il parallelismo: valutare gli elementi di una lista in parallelo

e in effetti con le strategie possiamo esprimere esattamente in questo modo:

parMap f xs = map f xs `using ` parList rseq

I vantaggi di questo approccio sono due; non solo si separa l'algoritmo dal parallelismo, ma si riutilizza anche map, piuttosto che implementarne una nuova versione parallela. La funzione parList è una Strategia sulle liste, definita come segue:

```
parList :: Strategy a -> Strategy [a]
parList strat [] = return []
parList strat (x:xs) = do
x' <- rpar (x `using ` strat )
xs ' <- parList strat xs
return (x ':xs ')
```

Infatti, parList viene già fornito da Control.Parallel.Strategies così non c'è bisogno di ridefinirlo da soli, stiamo usando la sua implementazione qui come esempio. La funzione parList è una strategia parametrizzata che prende come argomento una Strategia su valori di tipo a, e restituisce una strategia per la lista a. Questo illustra un altro aspetto importante delle Strategie: sono componibili nel senso che siamo in grado di costruire grandi strategie componendo componenti più piccole riutilizzabili. Qui, parList descrive una famiglia di strategie per le liste, che valutano gli elementi della lista in parallelo.

Dataflow parallelism: la monade Par

A volte vi è la necessità di essere più espliciti sulle dipendenze e sui limiti dei task di quello che ci è possibile fare con Eval e le Strategie. In questi casi si ricorre di solito al Concurrent Haskell, dove possiamo creare i vari task ed esplicitare quale thread fa il lavoro. Tuttavia, tale approccio causa una perdita di determinismo. Il modello di programmazione che introduciamo in questa sezione colma le lacune di Strategie e del Concurrent Haskell: questo risulta essere esplicito sulle dipendenze e i limiti delle attività senza sacrificare il determinismo. Inoltre, questo modello di programmazione ha qualche altro interessante beneficio; ad esempio, è implementato interamente come una libreria Haskell e l'applicazione è prontamente modificata per permettere strategie di programmazione alternative.
Come al solito, l'interfaccia si basa su una monade, questa volta chiamata Par:

```
newtype Par a
instance Functor Par
instance Applicative Par
instance Monad Par
runPar :: Par a -> a
```

Come con la monade Eval, la monade Par restituisce un risultato puro. Tuttavia, runPar va usato con attenzione; internamente è molto più dispendioso di runEval, perché (almeno nell'implementazione corrente) da inizio a una nuova istanza di scheduler che consiste in un thread di lavoro per processore. In generale si dovrebbe usare runPar per programmare attività parallele su larga scala.
Lo scopo del Par è quello di introdurre il parallelismo, quindi abbiamo bisogno di un modo per creare attività parallele:

```
fork :: Par () -> Par ()
```

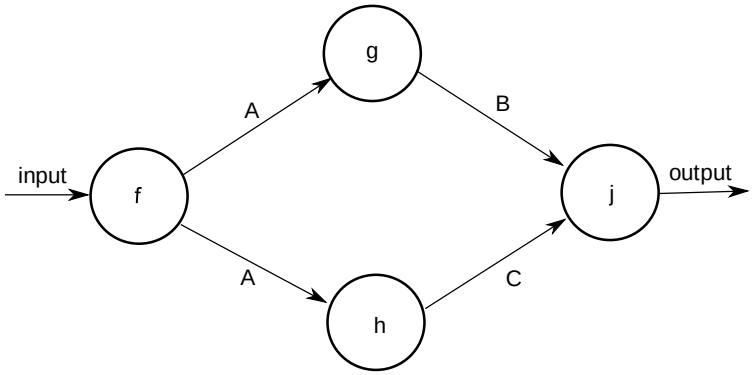
fork fa esattamente quello che ci si aspetta: il calcolo passato come argomento al fork (“il figlio”) viene eseguito in concomitanza con l'attuale calcolo (“il padre").
Naturalmente, fork di per sé non è molto utile, abbiamo bisogno di un modo di comunicare i risultati dal figlio del fork al padre, o in generale tra due calcoli Par in parallelo. La comunicazione è fornita da Ivar e le sue operazioni:

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

IVar è così chiamata perché è una implementazione di I-Structure, un concetto dalla variante del Parallel Haskell pH.
New crea una nuova IVar, che è inizialmente vuota, *put* riempie l'IVar con un valore, e *get* recupera il valore di una IVar (aspettando che un valore sia effettivamente inserito con *put* se necessario). Multipli put sulla stessa Ivar comporteranno un errore. Il tipo IVar è un parente del tipo MVAR che vedremo più avanti parlando di Concurrent Haskell (la principale differenza è che un'IVar può essere scritta una sola volta, cioè si può eseguire il comando put una sola volta. Un IVar è anche come un futuro o una promessa, concetti che possono essere familiari da altri linguaggi paralleli o concorrenti. Insieme, fork e IVars consentono la costruzione di reti di flussi di dati. I nodi della rete sono creati con fork, e le frecce collegano un put con ogni get su quella IVar. Per esempio, supponiamo di avere le seguenti quattro funzioni:

```
f :: In -> A
g :: A -> B
h :: A -> C
j :: (B,C) -> Out
```

Risulta il seguente grafo:



Non ci sono dipendenze sequenziali tra g e h, quindi possono essere eseguite in parallelo. Per ottenere un vantaggio dal parallelismo qui non ci resta che esprimere il grafo nella monade Par :

```
do
[ia ,ib ,ic] <- replicateM 4 new
fork $ do x <- get input
put ia (f x)
fork $ do a <- get ia
put ib (g a)
fork $ do a <- get ia
put ic (h a)
fork $ do b <- get ib
c <- get ic
put output (j b c)
```

Per ogni collegamento nel grafico creiamo una IVar (ia, ib e così via) Per ogni nodo del grafo chiamo fork e il codice per ogni nodo chiama get su ciascun input e put su ogni output del nodo. L'ordine delle chiamate a fork è irrilevante, la monade Par eseguirà il grafo risolvendo le dipendenze durante l'esecuzione.
Mentre la monade Par è particolarmente adatta ad esprimere le reti di flussi di dati, può anche essere utilizzata per altre strutture comuni. Per esempio, possiamo costruire un equivalente del combinatore parMap che abbiamo visto precedentemente. Prima costruiamo una semplice astrazione per il calcolo in parallelo che ci dia un risultato :

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

La funzione spawn crea un calcolo in parallelo e restituisce un'IVar che può essere utilizzata per attendere il risultato. Ora il map in parallelo consiste nel chiamare spawn per applicare la funzione ad ogni elemento della lista e quindi attendere tutti i risultati:

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

Notiamo che ci sono un paio di differenze tra questo parMap e la versione con la monade Eval. In primo luogo, l'argomento della funzione restituisce il risultato nella monade Par; naturalmente è facile portare una funzione arbitraria pura a questo tipo, ma la versione monadica consente il calcolo su ogni elemento per produrre più task paralleli ed inoltre parMapM attende tutti i risultati. A seconda del contesto, questo può o non può essere il comportamento più utile, ma ovviamente è semplice definire la versione modificata se necessario.

A parallel type inferencer

In questa sezione verrà parallelizzato un motore di inferenza dei tipi con la monade Par. L'inferenza dei tipi è insita nei modelli per flussi di dati perché si può considerare ogni associazione come un nodo nel grafo, e i collegamenti del grafo trasmettere i tipi derivati dal programma stesso.
Ad esempio, si consideri il seguente insieme di associazioni dalle quali vogliamo dedurre i tipi per:

```
f = ...
g = ... f ...
h = ... f ...
j = ... g ... h ...
```

Questo modello dà luogo a un grafo con esattamente la stessa forma del grafo a 4 nodi visto nella sezione precedente: dopo aver dedotto un tipo per f, possiamo usare questo tipo per dedurre i tipi di g e h (in parallelo), e una volta che abbiamo i tipi per g ed h si può dedurre un tipo per j.
Costruire un grafo per il problema di inferenza dei tipi consente di estrarre il massimo parallelismo dal processo di inferenza di tipo. La quantità reale di parallelismo presente dipende dalla struttura del programma di ingresso, in ogni caso.

La monade Par rispetto alle strategie

Abbiamo presentato due differenti modelli di programmazione parallela, ciascuno con vantaggi e svantaggi. Di seguito li riassumeremo in modo che si possa prendere una decisione ponderata scegliendo il più adatto per un determinato compito

- L'utilizzo delle strategie e della Monade Eval richiede una certa comprensione del funzionamento della valutazione pigra. Per gli utenti ancora inesperti questo potrebbe rivelarsi difficoltoso, specialmente diagnosticare i problemi. La monade Par rende tutte le dipendenze esplicite, effettivamente sostituisce la valutazione pigra con una chiamata esplicita a put/get sulle IVars. Questo è certamente ridondante ma risulta meno fragile e più facile da realizzare.
  Programmare con rpar richiede di stare attenti a mantenere i collegamenti alle spark per evitare loro di essere eliminate dal garbage collector, questo può essere delicato e difficile da ottenere in alcuni casi. La monade Par non ha nessuno di tali requisiti, sebbene non supporti il parallelismo speculativo come fa rpar: il parallelismo speculativo nella monade Par viene sempre eseguito.

- Le Strategie permettono una separazione tra l'algoritmo e il parallelismo che permette un riutilizzo maggiore del codice in molti casi.

- La monade Par ha più overhead della monade Eval anche se non si deve ricreare le strutture dati come per Eval. Attualmente, Eval tende a funzionare meglio a granularità più fine a causa del supporto diretto del runtime per le scintille. A granularità più grossa Par e Eval si comportano all'incirca allo stesso modo.

- La monade Par è implementata interamente nelle librerie Haskell (il pacchetto monad-par) ed è quindi facilmente modificabile in caso di necessità.

Concurrent Haskell

Concurrent Haskell è una estensione di Haskell 2010 con l'aggiunta del supporto per la programmazione concorrente esplicita dei thread. L'interfaccia di base rimane sostanzialmente invariata nella sua implementazione attuale, anche se un certo numero di arricchimenti da allora sono stati aggiunti:

- Le eccezioni asincrone sono state aggiunte come mezzo per la cancellazione asincrona dei thread

- La Memoria Software Transazionale (Software Transactional Memory) è stata aggiunta permettendo la composizione sicura delle astrazioni concorrenti, e rendendo possibile la costruzione in sicurezza di sistemi concorrenti più grandi.

- È stato specificato il comportamento di Concurrent Haskell in presenza di chiamate a e da linguaggi esterni.

Creazione di Thread

Il requisito base della concorrenza è la possibilità di creare un nuovo thread di controllo. In Concurrent Haskell questo si ottiene con l'operazione forkIO:

```
forkIO :: IO () -> IO ThreadId
```

forkIO prende un'operazione di tipo IO () come suo argomento, cioè un'operazione nella monade IO che restituisce alla fine un valore di tipo (). Il calcolo passato a forkIO viene eseguito in un nuovo thread che esegue concorrentemente con gli altri thread del sistema. Se il thread genera degli effetti, questi effetti saranno intervallati in modo indeterminato agli effetti degli altri thread.
Per mostrare come gli effetti risultino intervallati in modo indeterminato proviamo questo semplice esempio in cui due thread vengono creati, uno che stampi in continuazione la lettera A e l'altro allo stesso modo la lettera B :

```
1 import Control . Concurrent
2 import Control . Monad
3 import System .IO
5 main = do
6 hSetBuffering stdout NoBuffering
7 forkIO ( forever ( putChar 'A '))
8 forkIO ( forever ( putChar 'B '))
9 threadDelay (10^6)
```

La riga 6 mette l'output in modalità senza buffer, in modo che possiamo vedere l'interleaving più chiaramente. Linee 7 e 8 creano i due thread, e la linea 9 dice al thread principale di aspettare per un secondo (10^6 microsecondi) e poi terminare.
Quando viene eseguito, il programma produce un output simile a questo:

```
AAAAAAAAABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABABABABABABABAB
ABABABABABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABABAB
```

Si noti che l'intervallamento dei caratteri non è deterministico; qualche volta otteniamo stringhe di una singola lettera ma spesso l'output si alterna regolarmente tra i due thread. Perché si alterna cosi regolarmente e perché ogni thread ha la possibilità di mettere in output una sola lettera prima di alternarsi con l'altro? I thread in questo esempio si contendono una sola risorsa, il stdout Handle, quindi lo scheduling è influenzato da come la contesa della risorsa viene gestita. Nel caso di GHC (Glasgow Haskell Compiler) un Handle è protetto da un lock implementato da una Mvars. Presto vedremo come l'implementazione delle Mvars causi il comportamento ABABABA.
Abbiamo sottolineato in precedenza che la concorrenza è una tecnica di strutturazione del programma o un'astrazione. Le astrazioni sono pratiche quando sono efficienti ed è proprio qui che l'implementazione di GHC dei thread entra in merito. I thread sono estremamente leggeri in GHC: un thread costa in genere meno di un centinaio di byte più lo spazio per il suo stack, in modo che il runtime possa supportare letteralmente milioni di loro, limitato solo dalla memoria disponibile. A differenza dei Thread del sistema operativo, la memoria utilizzata dai thread Haskell è mobile, quindi il garbage collector è in grado di raggruppare i thread insieme vicini in memoria ed eliminare la frammentazione. I thread possono anche espandere e ridurre le dimensioni a richiesta, secondo le esigenze del programma. Quando si utilizzano più processori, il sistema runtime GHC sposta automaticamente i thread tra i vari core, al fine di bilanciare il carico di lavoro.

Questo metodo non è una prerogativa di Haskell, anzi molti altri linguaggi, tra cui le prime implementazioni Java, hanno avuto il supporto per i thread utente, talvolta chiamati green thread. Si è spesso pensato che questo sistema ostacoli l'interoperabilità con codice estraneo e le librerie che utilizzano thread del sistema operativo, e questo è uno dei motivi per cui i thread del sistema operativo tendono a essere preferiti; tuttavia, con un'attenta progettazione è possibile superare queste difficoltà, come vedremo in seguito.

Comunicazione: Mvars

L'astrazione di comunicazione di più basso livello in Concurrent Haskell sono le Mvar, la cui interfaccia è data qui sotto :

```
data MVar a -- abstract
newEmptyMVar :: IO ( MVar a)
newMVar :: a -> IO ( MVar a)
takeMVar :: MVar a -> IO a
putMVar :: MVar a -> a -> IO ()
```

Una Mvar può essere vista come un contenitore che può essere o pieno o vuoto; newEmptyMVar crea un nuovo contenitore vuoto, e newMVar crea un nuovo contenitore pieno contenente il valore passato come argomento. L'operazione putMVar mette un valore nel contenitore ma si blocca (waits) se il contenitore è già pieno. Simmetricamente, l'operazione takeMVar rimuove il valore da un contenitore pieno ma si blocca se il contenitore è vuoto.
Mvars generalizza semplici astrazioni concorrenti:

- Mvar() è un lock; takeMVar acquisisce il lock e putMVar lo rilascia. Una Mvar viene usata in questo modo per proteggere stati mutabili condivisi o sezioni critiche.

- Una Mvar è un canale unilaterale che può essere usato per comunicazioni asincrone tra due thread.

- Una Mvar è un contenitore utile per stati mutabili condivisi. Per esempio un modello di progettazione comune in Concurrent Haskell, quando

diversi thread necessitano di accesso in lettura e scrittura a qualche stato, è quello di rappresentare il valore dello stato come un normale immutabile struttura di dati Haskell memorizzati in una MVAR. Modificare lo stato consiste nel prendere il valore corrente con takeMVar (che acquista implicitamente un blocco), e poi mettere successivamente un nuovo valore nel MVAR con putMVar (che rilascia implicitamente il blocco di nuovo). Si possono anche usare le MVars per fare qualche semplice operazione asincrona di I/O.

I canali

Uno dei punti di forza di MVars è che sono un punto di partenza interessante per costruire grandi astrazioni. Qui useremo Mvars per costruire un canale illimitato e bufferizzato (unbounded buffered channel), secondo la seguente interfaccia base:

```
data Chan a
newChan :: IO ( Chan a)
readChan :: Chan a -> IO a
writeChan :: Chan a -> a -> IO ()
```

Questa interfaccia è disponibile nel modulo Haskell Control.Concurrent.Chan. I contenuti correnti del canale sono rappresentati come un flusso, definito in questo modo:

```
type Stream a = MVar ( Item a)
data Item a = Item a ( Stream a)
```

La fine del flusso è rappresentato da una MVAR vuoto, che noi chiamiamo "*hole*", perché sarà riempita quando un nuovo elemento viene aggiunto. Il canale stesso è una coppia di MVars, una che indica il primo elemento del flusso (la posizione di lettura), e l'altra punta all' Mvar vuota in fondo (la posizione di scrittura):

```
data Chan a
= Chan ( MVar ( Stream a))
       ( MVar ( Stream a))
```

Per costruire un nuovo canale dobbiamo prima creare un flusso vuoto che consiste in una singola Mvar vuota e poi il costruttore Chan con le sue Mvar per la lettura e la fine della scrittura, entrambe che puntino a un flusso vuoto:

```
newChan :: IO ( Chan a)
newChan = do
        hole <- newEmptyMVar
        readVar <- newMVar hole
        writeVar <- newMVar hole
        return ( Chan readVar writeVar )
```

Per aggiungere un nuovo elemento al canale dobbiamo creare un oggetto (item) con un nuovo hole, riempire l'hole corrente facendolo puntare ad un nuovo oggetto e regolare il fine scrittura del canale in modo che punti a un nuovo hole:

```
writeChan :: Chan a -> a -> IO ()
writeChan ( Chan _ writeVar ) val = do
      new_hole <- newEmptyMVar
      old_hole <- takeMVar writeVar
      putMVar writeVar new_hole
      putMVar old_hole ( Item val new_hole )
```

Per rimuovere un valore dal canale, dobbiamo seguire la fine della lettura Chan per la prima MVAR del canale (Chan), prendere quel MVAR per ottenere l'item, regolare la fine-lettura per puntare al Mvar successivo nel flusso, e infine restituire il valore memorizzato nell' item:

```
1 readChan :: Chan a -> IO a
2 readChan ( Chan readVar _) = do
3      stream <- takeMVar readVar
4      Item val new <- takeMVar stream
5      putMVar readVar new
6      return val
```

Pensate a cosa succederebbe se il canale fosse vuoto. Il primo takeMVar (linea 3) avrà successo, ma il secondo takeMVar (linea 4) troverà un hole vuoto, e così bloccherà. Quando un altro thread chiama writeChan riempirà l'hole permettendo al primo thread

di completare il suo takeMVar, aggiornare la fine-scrittura (riga 5) e finalmente ritornare.
Se thread multipli concorrentemente chiamassero readChan, il primo chiamerebbe con successo takeMVar sul fine-lettura ma i thread seguenti si bloccherebbero tutti in quel punto fino a che il primo thread non termini le sue operazioni e aggiorni il fine-scrittura.
Se thread multipli chiamassero writeChan una cosa simile accadrebbe: il fine-scrittura del canale è il punto di sincronizzazione che permette ad un solo thread alla volta di aggiungere un oggetto al canale. Comunque il fine scrittura e il fine-lettura, essendo differenti Mvars, permettono a operazioni di readChan e writeChan concorrenti di procedere senza interferenze.
Questa implementazione permette una buona generalizzazione su canali multicast senza cambiare la struttura che sta alla base. L'idea è di aggiungere un'ulteriore operazione:

```
dupChan :: Chan a -> IO ( Chan a)
```

che crea un Chan duplicato con la seguente semantica:

- Il nuovo Chan inizialmente è vuoto
- Successive scritture su entrambi i Chan sono lette da entrambi, quindi leggere un oggetto da un canale non lo rimuove dall'altro

```
dupChan :: Chan a -> IO ( Chan a)
dupChan ( Chan _ writeVar ) = do
      hole <- takeMVar writeVar
      putMVar writeVar hole
      newReadVar <- newMVar hole
      return ( Chan newReadVar writeVar )
```

Entrambi i canali condividono un unico fine-scrittura, ma hanno indipendenti fine-lettura. La fine lettura del nuovo canale viene inizializzato per puntare all'hole alla fine del contenuto corrente. Purtroppo, questa implementazione di dupChan non funziona! La definizione di dupChan in sé non è errata, ma combinata con la definizione di readChan data in precedenza non implementa la necessaria semantica. Il problema è che readChan non sostituisce il contenuto di un hole dopo averlo letto, quindi se readChan è chiamato a leggere i valori sia del canale restituito dal dupChan e il canale originale, la

seconda chiamata verrà bloccata. La soluzione è quella di cambiare un takeMVar con un readMVar nell' implementazione di readChan:

```
1 readChan :: Chan a -> IO a
2 readChan ( Chan readVar _) = do
3          stream <- takeMVar readVar
4          Item val new <- readMVar stream -- modified
5          putMVar readVar new
6          return val
```

Nella riga 4 si restituisce l'Item a Stream, che può essere letto da qualsiasi canale duplicato creato da dupChan.
La programmazione di grandi strutture con MVAR può essere molto più difficile di quanto sembri. Come vedremo tra poco, si complica ulteriormente quando consideriamo le eccezioni ma per fortuna c'è una soluzione, che presenteremo dopo. Nonostante le difficoltà di scalare le MVars a grandi astrazioni, queste dispongono di alcune proprietà interessanti, come vedremo nel prossimo paragrafo.

Equità

Equità è un argomento ben studiato e molto tecnico, che non proveremo a rivedere qui. Tuttavia, vogliamo evidenziare una particolare importante garanzia fornita da MVars per quanto riguarda l'equità:

> Nessun thread può essere bloccato indefinitamente su una MVAR a meno che un'altro thread trattenga indefinitamente un'MVAR.

In altre parole, se un thread T è bloccato da takeMVar, e ci sono operazioni regolari di putMVar sulla stessa MVAR, vi è la garanzia che ad un certo punto il takeMVar del thread T termini. In GHC questa garanzia viene implementata mantenendo i thread bloccati da una coda FIFO collegata alla MVAR, così alla fine ogni thread in coda arrivi a completare le proprie operazione fintanto che non ci sono altri thread che svolgono regolare operazioni di putMVar (una garanzia equivalente è applicabile a thread bloccati da putMVar quando ci sono takeMVars regolari). Si noti che non è sufficiente semplicemente risvegliare il thread bloccato, perché un altro thread potrebbe eseguire prima e prendere (rispettivamente put) la MVAR portando il thread appena svegliato ad andare al termine della coda di nuovo, questo invaliderebbe la garanzia di equità.

L'implementazione deve quindi atomicamente svegliare il thread bloccato ed eseguire l'operazione di blocco, che è esattamente ciò che fa GHC.

Equità in pratica

Ricordiamo il nostro esempio**,** dove avevamo due thread, uno che stampava A e l'altro B, e l'uscita era spesso la perfetta alternanza tra i due: ABABABABABABABAB. Questo è un esempio della garanzia di equità nella pratica. Il stdout Handle è rappresentato da un MVAR, in modo che quando entrambi i thread tentano la chiamata a takeMVar per operare sull'handle, uno di essi vince e l'altro viene bloccato. Quando il thread vincente completa il suo funzionamento e chiama putMVar, lo scheduler risveglia il thread bloccato e completa il suo takeMVar bloccato, in modo che il thread originale vincente venga immediatamente bloccato quando cercherà di riacquisire l'handle. Quindi questo porta ad un'alternanza perfetta tra i due thread. L'unico modo per interrompere il modello di alternanza è quando un thread è anticipato mentre non è ancora in possesso della MVAR, anzi questo non capita di volta in volta, come si vede la lunga stringa occasionale di una singolo lettera in uscita. Una conseguenza dell'implementazione dell'equità è che quando più thread sono bloccati, abbiamo solo bisogno di svegliarsi un singolo thread. Questa singola proprietà di risveglio è una caratteristica di rendimento particolarmente importante quando un gran numero di thread si contendono un'unica MVAR. Come potremmo vedere in seguito, è la garanzia di equità insieme con la sola proprietà del risveglio che significa che MVars non sono completamente assorbite dalla Memoria Software Transazionale. (Software Transactional Memory.)

La cancellazione: eccezioni asincrone

In un'applicazione interattiva, è spesso importante per un thread poter interrompere (interrupt) l'esecuzione di un altro thread, quando qualche condizione particolare si verifica. Alcuni esempi di questo tipo di comportamento nella pratica sono:

- In un browser Web, il thread che sta scaricando la pagina Web e il thread che sta facendo il rendering della pagina deve essere interrotto quando l'utente preme il pulsante stop.

- Un'applicazione server vuole tipicamente dare al client una certa quantità di tempo per eseguire una richiesta prima di chiudere la connessione, in modo da evitare connessioni dormienti che utilizzino risorse.

- Un'applicazione in cui un thread di calcolo intensivo sta lavorando (ad esempio, la visualizzazione di alcuni dati), ed i dati di ingresso cambiano a causa di qualche input dell'utente.

La decisione di progettazione cruciale nel supportare la cancellazione è se la vittima designata possa incidere sulla condizione di cancellazione, o se il thread viene immediatamente annullata in qualche modo.

1. Se il thread ha diritto di voto, vi è il pericolo che il programmatore si possa dimenticare della votazione abbastanza facilmente, e il thread risulta non rispondente, forse in modo permanente. I thread non rispondenti causano sospensioni e stalli, particolarmente sgradevoli per l'utente.

2. Se la cancellazione avviene in maniera asincrona, sezioni critiche che modificano lo stato devono essere protette dalla cancellazione, altrimenti la cancellazione potrebbe avvenire durante l'aggiornamento lasciando alcuni dati in uno stato inconsistente.

In realtà, la scelta è tra fare solo (1), o fare entrambe (1) e (2). Nella maggior parte dei linguaggi imperativi è impensabile utilizzare il metodo (2) come predefinito, perché la maggior parte del codice modifica gli stati. Haskell ha un netto vantaggio in questo ambito, perché la maggior parte del codice è puramente funzionale, quindi può essere tranquillamente interrotto o sospeso, e poi riprendere, senza incidere sulla correttezza. Inoltre non abbiamo alternative perché il codice puramente funzionale non può per definizione decidere la condizione di annullamento, quindi deve essere cancellabile di default.
Pertanto, la cancellazione completamente asincrona è l'unica impostazione predefinita sensata in Haskell, e il problema di progettazione si riduce a decidere come la cancellazione venga rappresentata nella monade IO. È logico pensare alla cancellazione come ad una eccezione, in quanto le eccezioni sono già presenti nella monade IO, e gli idiomi usuali per scrittura del codice nella monade IO includono gestori di eccezioni per liberare risorse e ripulire in caso di errore. Ad esempio, per eseguire un'operazione che richiede un file temporaneo, si può usare il combinatore *bracket* per garantire che il file temporaneo venga sempre rimosso, anche se l'operazione solleva un'eccezione:

```
bracket ( newTempFile " temp ")
(\ file -> removeFile file )
(\ file -> ...)
```

Dove *bracket* è cosi definito:

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket before after during = do
a <- before
c <- during a `onException ` after a
after a
return c
```

onException esegue il suo primo argomento e se viene lanciata un'eccezione, esegue il secondo argomento prima di lanciare nuovamente l'eccezione.

```
onException :: IO a -> IO b -> IO a
```

Vogliamo che i gestori di eccezioni intervengano in caso di annullamento, così la cancellazione può essere considerata come un'eccezione. Tuttavia c'è una fondamentale differenza tra il tipo di eccezione generata dal openFile quando il file non esiste, per esempio, e eccezioni che possono insorgere in qualsiasi momento perché l'utente ha premuto il "pulsante di stop"; chiamiamo quest'ultimo tipo eccezioni asincrone, per ovvi motivi. Per avviare un'eccezione asincrona, Haskell fornisce la primitiva throwTo che genera un'eccezione da un thread ad un altro:

```
throwTo :: Exception e => ThreadId -> e -> IO ()
```

Il vincolo Exception richiede che il valore generato dall'eccezione sia un'istanza della classe Exception, che implementa una semplice gerarchia. Il ThreadId è un valore precedentemente restituito dal forkIO, e può fare riferimento ad un thread in qualsiasi stato: in esecuzione, bloccato o terminato (in quest'ultimo caso, throwTo non fa nulla).

Il supporto Haskell per la cancellazione è modulare e la maggior parte delle librerie di codice non devono fare nulla per supportarlo, anche se ci sono alcune opportune e semplici regole che devono essere seguite quando si ha a che fare con gli stati, come vedremo nel prossimo paragrafo.

Mask e le eccezioni asincrone

Come abbiamo accennato in precedenza, il pericolo con le eccezioni completamente asincrone è che potrebbero terminare qualche thread mentre siamo nel bel mezzo dell'aggiornamento di qualche stato condiviso, lasciando i dati in uno stato incoerente, causando con molta probabilità confusione in seguito.
Quindi abbiamo certamente bisogno di un modo per controllare la consegna di eccezioni asincroni durante le sezioni critiche. Ma dobbiamo procedere con cautela; sarebbe facile fornire al programmatore un modo per spegnere temporaneamente la consegna di eccezioni asincrone ma questo non è ciò di cui abbiamo realmente bisogno.
Si consideri il seguente problema: un thread vuole chiamare takeMVar, eseguire un'operazione a seconda del valore della MVAR, e infine mettere il risultato dell'operazione nell'MVAR. Il codice deve essere capace di gestire le eccezioni asincrone, ma dovrebbe essere sicuro: se un'eccezione asincrona arriva dopo il takeMVar, ma prima della putMVar finale, la MVAR non dovrebbe poter essere lasciata vuota, invece il valore originario deve essere sostituito.
Il codice per risolvere questo problema usando gli strumenti appena visti dovrebbe portare a qualcosa del genere:

```
1problem m f = do
2a <- takeMVar m
3r <- f a `catch ` \e -> do putMVar m a; throw e
4putMVar m r
```

Ci sono almeno due punti in cui, se dovesse esserci un'eccezione asincrona, l'invarianza sarebbe violata. Se un'eccezione capitasse tra le linee 2 e 3, o tra linee 3 e 4, l'MVAR rimarrebbe vuota. In realtà, non c'è modo di risistemare il gestori di eccezioni per garantire che la MVAR sia sempre lasciata non vuota. Per risolvere questo problema, Haskell fornisce il combinatore mask:

```
mask :: (( IO a -> IO a) -> IO b) -> IO b
```

Il tipo sembra un po' confuso, ma non è come sembra. L'operazione mask rinvia la consegna di eccezioni asincrone per la durata del suo argomento, ed è usato in questo modo:

```
problem m f = mask $ \ restore -> do
a <- takeMVar m
r <- restore (f a) `catch ` \e -> do putMVar m a; throw e
putMVar m r
```

mask viene applicato ad una funzione, che prende come argomento una funzione restore, che può essere utilizzata per ripristinare il lancio di eccezioni asincrone alla sua stato attuale. Se immaginiamo di proteggere l'intero argomento di mask tranne per l'espressione (f a), le eccezioni asincrone non possono essere sollevate in porzioni protette. Questo risolve il problema che abbiamo avuto in precedenza, dato che ora un'eccezione può essere sollevata mentre (f a) sta lavorando, e abbiamo un gestore di eccezioni per catturare eventuali eccezioni in questo caso. Ma un nuovo problema è stato introdotto: takeMVar potrebbe bloccarsi per lungo tempo, ma è all'interno del mask e così il thread non risponderebbe per quel periodo di tempo. Inoltre, non c'è un buono motivo per mascherare le eccezioni durante takeMVar. Quindi, questo è esattamente il comportamento che Haskell definisce per takeMVar: si designa un numero limitato di operazioni, tra cui takeMVar, interrompibili. Le operazioni interrompibili possono ricevere eccezioni asincrone anche all'interno del mask.
Che cosa giustifica questa scelta? Pensate a mask come al “passaggio alla modalità di voto" per le eccezioni asincrone. All'interno dell’operazione mask, le eccezioni asincrone non risultano più asincrone, ma possono ancora essere sollevate da alcune operazioni. In altre parole, le eccezioni asincrone diventano sincrone all'interno del mask.
Tutte le operazioni che possono bloccare indefinitamente vengono progettate come interrompibili. Questo risulta essere il comportamento ideale in molte situazioni, come per il problema qui sopra.
In realtà, siamo in grado di fornire combinatori di livello superiore per tutelare i programmatori dalla necessità di utilizzare direttamente mask. Ad esempio, la funzione problem è generalmente utile quando si lavora con le MVars, ed è fornita sotto il nome modifyMVar_ nella libreria Control.Concurrent.MVar.

Sicurezza delle eccezioni asincrone

Tutto ciò che è necessario perché la maggior parte del codice sia sicuro in presenza di eccezioni asincrone è quello di utilizzare le operazioni come modifyMVar_ invece di takeMVar e putMVar direttamente. Ad esempio, si considerino i canali con buffer che

abbiamo definito in precedenza. Come definito, le operazioni non sono al sicuro dalle eccezioni asincrone; per esempio, writeChan era stata definita in questo modo:

```
1writeChan :: Chan a -> a -> IO ()
2writeChan ( Chan _ writeVar ) val = do
3new_hole <- newEmptyMVar
4old_hole <- takeMVar writeVar
5putMVar writeVar new_hole
6putMVar old_hole ( Item val new_hole )
```

Ci sono diversi momenti in cui qui se venisse lanciata un'eccezione asincrona una MVar verrebbe lasciata vuota e l'utilizzatore successivo del canale causerebbe uno stallo.
Per renderlo sicuro usiamo modifyMVar_:

```
1 writeChan ( Chan _ writeVar ) val = do
2 new_hole <- newEmptyMVar
3 modifyMVar_ writeVar $ \ old_hole -> do
4 putMVar old_hole ( Item val new_hole )
5 return new_hole
```

Abbiamo visto l'utilizzo della funzione bracket in precedenza; bracket è definita con mask per rendere le eccezioni asincrone sicure:

```
1 bracket before after during =
2 mask $ \ restore -> do
3 a <- before
4 r <- restore ( during a) `catch ` \e -> after a; throw e
5 _ <- after a
6 return r
```

Timeouts

Un buon esempio di programmazione con le eccezioni asincrone è quello di scrivere una funzione che può imporre un limite di tempo per una data azione. Vogliamo fornire l'involucro timeout come un combinatore del seguente tipo:

```
timeout :: Integer -> IO a -> IO ( Maybe a)
```

1. timeout t m si comporta esattamente come FMAP just m se m restituisce un risultato o solleva un'eccezione (compresa un'eccezione asincrona), entro t microsecondi.

2. in caso contrario, m viene inviata un'eccezione asincrona nella forma Timeout u.
   Timeout è un nuovo tipo di dati che definiamo, ed u è un valore univoco di tipo Unique, distinguendo questa istanza particolare di timeout dalle altre. La chiamata a timeout restituisce Nothing.

L'implementazione non dovrebbe attuare una semantica in tempo reale, quindi in pratica il timeout sarà solo di circa t microsecondi. Si noti che (1) richiede che m venga eseguito nel contesto del thread corrente, dal momento che m potrebbe chiamare myThreadId, per esempio. Inoltre, un altro thread che lanci un'eccezione al thread corrente con throwTo si aspetta di interrompere m.
Il codice per timeout è mostrato qui di seguito, questa implementazione è stata presa dalla libreria System.Timeout. L'implementazione è difficile da ottenere. L'idea di base è quella di creare un nuovo thread che attende t microsecondi e poi chiamare throwTo per restituire l'eccezione Timeout al thread originale; che sembra abbastanza semplice. Tuttavia, dobbiamo fare in modo che questo thread non possa generare l'eccezione Timeout dopo che la chiamata a timeout sia ritornata, in caso contrario l'eccezione Timeout uscirebbe della chiamata, quindi timeout deve uccidere il thread prima di tornare.

```
1 timeout n m
2 | n < 0 = fmap Just m
3 | n == 0 = return Nothing
4 | otherwise = do
5 pid <- myThreadId
6 u <- newUnique
7 let ex = Timeout u
8 handleJust
9 (\e -> if e == ex then Just () else Nothing )
10 (\_ -> return Nothing )
```

```
11 ( bracket ( forkIO $ do threadDelay n
12 throwTo pid ex)
13 (\t -> throwTo t ThreadKilled )
14 (\_ -> fmap Just m))
```

Ecco come funziona l'applicazione, riga per riga:
1-2 gestisce i casi semplici, in cui il timeout è negativo o pari a zero.
5 trova il threadid del thread corrente
6-7 crea una nuova eccezione Timeout, generando un valore univoco con newUnique
8-14 handleJust è un gestore di eccezioni, con il seguente tipo:

```
handleJust :: Exception e
=> (e -> Maybe b) -> (b -> IO a) -> IO a
-> IO a
```

Il suo primo argomento (linea 9), seleziona quali eccezioni catturare; in questo caso, solo l'eccezione Timeout definita nella linea 7. Il secondo argomento (linea 10) è il gestore di eccezioni, che in questo caso restituisce solamente Nothing, dal momento che il timeout è trascorso.
Righe 11 14 sono il calcolo da eseguire nel gestore di eccezioni. bracket è usato qui per la creazione del thread figlio, e assicura che venga ucciso prima di tornare.
Righe 11-12 crea il thread figlio. Nel thread figlio aspettiamo n microsecondi con threadDelay, e quindi generiamo l'eccezione Timeout al thread genitore con throwTo.
Riga 13 uccide sempre il thread figlio prima di tornare.
Riga 14 il corpo della di bracket: esegue il calcolo m passato come secondo argomento al timeout, e racchiude il risultato in just.

Pensiamo a due casi limite: o m completa e restituisce Just x nella linea 14, o, il thread figlio lancia la sua eccezione, mentre m è ancora funzionante.
Vi è un caso difficile da prendere in considerazione: cosa succede se sia il thread figlio sia il thread genitore tentano di chiamare throwTo contemporaneamente (linee 12 e 13 rispettivamente)? Chi vince?
La risposta dipende dalla semantica di throwTo. Affinché questo implementazione di timeout funzioni correttamente, non deve essere possibile per la chiamata a bracket nella riga 11 di tornare mentre l'eccezione di timeout può essere ancora lanciata, in caso contrario l'eccezione può perdere. Da qui, la chiamata a throwTo che uccide il thread figlio alla riga 13 deve essere sincrona: una volta che questa chiamata sia ritornata, il

thread figlio non può più lanciare la sua eccezione. In effetti, questa garanzia è fornita dalla semantica di throwTo: una chiamata a throwTo restituisce solo dopo che l'eccezione è stata sollevata nel target thread. Quindi, throwTo può bloccare se il thread figlio sta mascherando eccezioni asincrone con mask, e siccome throwTo può bloccare, è quindi interrompibile e può esso stesso ricevere eccezioni asincrone.
Tornando alla nostra domanda “chi vince”, la risposta è “esattamente uno di loro", e questo è esattamente ciò di cui abbiamo bisogno per garantire il corretto comportamento di timeout.

Riflessioni sulle eccezioni asincrone

Astrazioni come timeout sono certamente difficili da ottenere corrette, ma per fortuna devono essere scritte una sola volta. Abbiamo scoperto che in pratica trattare eccezioni asincrone è abbastanza semplice, seguendo alcune semplici regole:

- Utilizzare bracket quando si acquisiscono risorse che devono essere successivamente rilasciate

- usare modifyMVar_ piuttosto che takeMVar e putMVar, perché hanno integrata la sicurezza per le eccezioni asincrone.

Se la gestione degli stati comincia a diventare complicata con più livelli di gestori di eccezioni ci sono due approcci per semplificare le cose:

- Il passaggio alla modalità di polling (voto) con mask può aiutare a gestire la complessità. La libreria GHC di I/O, per esempio, viene eseguita interamente all'interno di mask. Si noti che all'interno di mask è importante ricordare che le eccezioni asincrone possono anche verificarsi con operazioni interrompibili; la documentazione contiene un elenco di operazioni che sono garantite non essere interrompibili.

- Usare la memoria transazionale software (STM) invece delle Mvars o altre rappresentazioni di stati può spazzare via tutta la complessità in un solo colpo. Descriveremo STM in seguito.

Queste semplici regole di solito non sono cosi onerose: ricordiamo che questo vale solo per il codice nella monade IO, così le vaste librerie di codice puramente funzionali disponibili per Haskell risultano sicure per costruzione. Troveremo che la maggior parte del codice nella monade IO è semplice da rendere sicuro, e se le cose si complicano tornare ad utilizzare mask o STM è una soluzione soddisfacente.
In cambio del rispetto delle regole, tuttavia, l'approccio di Haskell per le eccezioni asincrone conferisce molti benefici.

- Molte condizioni eccezionali si mappano naturalmente sulle eccezioni asincrone. Ad esempio, stack overflow e interrupt dell'utente (ad esempio control-C alla console) sono mappati nelle eccezioni asincrone in Haskell. Di conseguenza, il control-C non solo interrompe il programma, ma lo fa in modo pulito, eseguendo tutti i gestori di eccezioni. I programmatori Haskell non devono fare nulla per attivare questo comportamento.

- Costrutti come timeout funzionano sempre, anche con codici di terze parti

- I thread non muoiono mai semplicemente in Haskell, viene garantito che un thread abbia la possibilità di pulire ed eseguire la sua gestione per le eccezioni.

Software Transactional Memory

Software Transactional Memory (STM) è una tecnica per la semplificazione della programmazione concorrente, che consente che molteplici operazioni di cambio di stato vengano raggruppate insieme ed eseguite come una singola operazione atomica. STM è una tecnica di implementazione dove il costrutto di linguaggio che ci interessa è un "blocco atomico". Purtroppo il vecchio termine è rimasto, e tale capacità a livello di linguaggio viene chiamata STM.
STM risolve una serie di problemi che si verificano con le tradizionali astrazioni concorrenti, che descriviamo qui attraverso una serie di esempi. Immaginate il seguente scenario: un window manager che gestisce molteplici desktop. L'utente può spostare le finestre da un desktop ad un altro, al tempo stesso, un programma può richiedere che la propria finestra si sposti dal desktop corrente ad un altro desktop. Il window manager

usa più thread: uno per ascoltare gli input da parte dell'utente, uno per ogni desktop esistente per ascoltare le richieste di tali programmi, e un thread che rende il display per l'utente.

```
1 data STM a -- abstract
2 instance Monad STM -- amongst other things
4 atomically :: STM a -> IO a
6 data TVar a -- abstract
7 newTVar :: a -> STM ( TVar a)
8 readTVar :: TVar a -> STM a
9 writeTVar :: TVar a -> a -> STM ()
11 retry :: STM a
12 orElse :: STM a -> STM a -> STM a
14 throwSTM :: Exception e => e -> STM a
15 catchSTM :: Exception e => STM a -> (e -> STM a) -> STM a
```

Come dovrebbe il programma rappresentare lo stato del display? Una soluzione sarebbe mettere tutto in una singola Mvar:

```
type Display = MVar ( Map Desktop ( Set Window ))
```

questo funzionerebbe ma la MVar sarebbe un unico elemento di contesa. Per esempio il thread di rendering che deve solamente vedere il desktop visualizzato al momento, potrebbe venir bloccato da una finestra di un altro desktop che si sta spostando.
Quindi potremmo consentire una maggiore concorrenza avendo una diversa Mvar per ogni desktop

```
type Display = Map Desktop ( MVar ( Set Window ))
```

Purtroppo questo approccio crea subito dei problemi. Si consideri un'operazione per spostare una finestra da un desktop ad un altro:

```
moveWindow :: Display -> Window -> Desktop -> Desktop -> IO ()
moveWindow disp win a b = do
wa <- takeMVar ma
wb <- takeMVar mb
putMVar ma (Set . delete win wa)
```

```
putMVar mb (Set . insert win wb)
where
ma = fromJust (Map. lookup disp a)
mb = fromJust (Map. lookup disp b)
```

Si noti che dobbiamo prendere entrambe le MVars prima di poter inserire i risultati, in caso contrario un altro thread potrebbe osservare il display in uno stato in cui la finestra che stiamo spostando non esiste. Ma questo solleva un problema: che cosa accadrebbe se ci fosse una chiamata concorrente di MoveWindow che cerchi di spostare una finestra nella direzione opposta? Entrambe le chiamate riuscirebbero a fare il primo takeMVar, ma si bloccherebbero al secondo, ed il risultato è una situazione di stallo. Questo è un esempio del classico problema della cena dei filosofi.
Una soluzione è quella di imporre un ordinamento delle MVars, e richiedere che tutti gli agenti prendano le MVars nell'ordine corretto e le rilascino in ordine inverso. Questo è scomodo e soggetto ad errori, ed inoltre dovremmo estendere il nostro ordinamento a qualsiasi altro stato a cui potremmo voler accedere concorrentemente. Sistemi di grandi dimensioni con molti blocchi (ad esempio i sistemi operativi) sono spesso afflitti da questo problema, e la gestione della complessità richiede costruzione di infrastrutture elaborate per rilevare le violazioni di ordinamento. La memoria transazionale fornisce un modo per evitare questo problema di stallo senza imporre un requisito di ordinamento al programmatore. Per risolvere il problema con STM, sostituiamo MVAR con TVAR:

```
type Display = Map Desktop ( TVar ( Set Window ))
```

TVAR sta per “variabile transazionale", ed è una variabile mutevole che può solo essere letta o scritta all'interno di una transazione. Per implementare MoveWindow, abbiamo semplicemente eseguito le operazioni necessarie sul TVars nella monade STM, e avvolto l'intera sequenza in modo atomico:

```
moveWindow :: Display -> Window -> Desktop -> Desktop -> IO ()
moveWindow disp win a b = atomically $ do
wa <- readTVar ma
wb <- readTVar mb
writeTVar ma (Set. delete win wa)
writeTVar mb (Set. insert win wb)
where
```

```
ma = fromJust (Map. lookup a disp )
mb = fromJust (Map. lookup b disp )
```

Il codice è quasi identico alla versione con le MVAR, ma il comportamento è abbastanza differente; la sequenza di operazioni all'interno di atomically vengono eseguite in maniera indivisibile dal punto di vista del resto del programma. Nessun altro thread può osservare uno stato intermedio, l'operazione potrebbe essere stata completata o non essere ancora iniziata. Inoltre non è necessario aver letto entrambe leTvars prima di scriverle, questo andrebbe bene lo stesso:

```
moveWindow :: Display -> Window -> Desktop -> Desktop -> IO ()
moveWindow disp win a b = atomically $ do
wa <- readTVar ma
writeTVar ma (Set. delete win wa)
wb <- readTVar mb
writeTVar mb (Set. insert win wb)
where
ma = fromJust (Map. lookup disp a)
mb = fromJust (Map. lookup disp b)
```

Quindi STM è molto meno soggetto ad errori qui. L'approccio scala inoltre a qualsiasi numero di TVars, così abbiamo potuto facilmente scrivere un'operazione che sposta le finestre da tutti gli altri desktop al desktop corrente, per esempio. Supponiamo ora di voler scambiare due finestre, spostando la finestra W dal desktop A a B, e, contemporaneamente, V da B ad A. Utilizzando le MVAR si sarebbe dovuto scrivere un'applicazione specifica per fare questo, perché deve prendere le MVars per A e B (nel giusto ordine), e poi rimettere entrambe le MVars a posto con i nuovi contenuti. Con STM, tuttavia, possiamo esprimere questo molto più ordinatamente come una composizione. In primo luogo abbiamo bisogno di esporre una versione di MoveWindow senza il incapsulatore atomically:

```
moveWindowSTM :: Display -> Window -> Desktop -> Desktop
-> STM ()
moveWindowSTM disp win a b = do ...
```

e quindi possiamo definire swapWindows componendo due chiamate di moveWindowsSTM :

```
swapWindows :: Display
-> Window -> Desktop
-> Window -> Desktop
-> IO ()
swapWindows disp w a v b = atomically $ do
moveWindowSTM disp w a b
moveWindowSTM disp v b a
```

Ciò dimostra la componibilità delle operazioni STM: qualunque operazione di tipo STM può essere composta con altre per formare una più grande transazione atomica. Per questo motivo, le operazioni STM sono generalmente fornite senza il wrapper atomically, in modo che i gli utenti possano comporle, se necessario, prima di avvolgere l'intera operazione con atomically.
Finora abbiamo coperto le strutture di base della STM, e ha dimostrato che STM può essere usato per far scalare l'atomicità in modo componibile. STM conferisce un miglioramento qualitativo in espressività e robustezza durante la scrittura di programmi concorrenti. I vantaggi di STM in Haskell tuttavia vanno ben oltre; nelle sezioni seguenti si mostra come STM può essere usato per comporre blocchi di astrazioni, e come può essere usato per gestire la complessità in presenza di failure e interruzioni.

Blocking

Una parte importante della programmazione concorrente si occupa di blocco; quando abbiamo bisogno di aspettare una condizione si avveri, o per acquisire una particolare risorsa. STM fornisce un modo ingegnoso per fare questo, con un'unica operazione:

```
retry :: STM a
```

Il significato di retry è semplicemente "esegui la transazione corrente di nuovo". Sembra strano perché dovremmo voler eseguire la transazione corrente di nuovo? Per prima cosa, il contenuto di alcuni TVars che abbiamo letto potrebbe essere stato modificato da un altro thread, quindi rieseguire l'operazione può fornire risultati differenti. Infatti, non ha senso rieseguire la transazione a meno che non è possibile che qualcosa di diverso sia accaduto, e il sistema runtime questo lo sa, quindi retry attende fino a che una TVAR che è stato letta nella transazione corrente venga scritta e quindi

avvia un retry della corrente transazione. Fino a che questo non succede il thread corrente viene bloccato.

Operazioni possibili con STM

Ci siamo occupati dell'operazione unGetChan che non poteva essere implementata con la semantica desiderata usando le Mvar. Ecco qui l'implementazione con STM:

```
unGetTChan :: TChan a -> a -> STM ()
unGetTChan ( TChan read _write ) a = do
listhead <- readTVar read
newhead <- newTVar ( TCons a listhead )
writeTVar read newhead
```

L'implementazione più ovvia qui fa la cosa giusta. Le altre operazioni che non erano possibili con le Mvars sono semplici con STM, per esempio isEmptyChan, che nella versione con le Mvar soffriva dello stesso problema di unGetChan

```
isEmptyTChan :: TChan a -> STM Bool
isEmptyTChan ( TChan read _write ) = do
listhead <- readTVar read
head <- readTVar listhead
case head of
TNil -> return True
TCons _ _ -> return False
```

Composizione delle operazioni di blocco. Supponiamo di voler implementare un'operazione readEitherTChan che può leggere un elemento da uno dei due canali. Se entrambi i canali sono vuoti si blocca, se un canale è non vuoto legge il valore di quel canale, e se entrambi i canali sono non vuoti dia la possibilità di scegliere il canale da leggere. Il suo tipo è

```
readEitherTChan :: TChan a -> TChan b -> STM ( Either a b)
```

Non possiamo realizzare questa funzione con le operazioni introdotta finora, ma STM fornisce un'operazione ancora più importante che consente di comporre il blocco delle transazioni. L'operazione è orElse:

```
orElse :: STM a -> STM a -> STM a
```

L'operazione OrElse ab ha il seguente comportamento:

- Prima viene eseguito a. Se a restituisce un risultato, tale risultato è immediatamente restituito dalla chiamata OrElse.

- Se invece a chiama retry, allora gli effetti di a vengono scartati, e b viene eseguito(al suo posto)

Possiamo usare OrElse per comporre operazioni di blocco atomico. Ritornando al nostro esempio, readEitherTChan potrebbe essere implementata come segue:

```
readEitherTChan :: TChan a -> TChan b -> STM ( Either a b)
readEitherTChan a b =
fmap Left ( readTChan a)
`orElse `
fmap Right ( readTChan b)
```

Questa è una semplice composizione delle due chiamate readTChan, l'unica complicazione sta in come contrassegnare il risultato con Destra o Sinistra in base a quale dei due rami ha successo.
Nell'attuazione MVAR di Chan non c'è modo per attuare l'operazione readEitherChan senza elaborare la rappresentazione di Chan per supportare il protocollo di sincronizzazione che sarebbe richiesto.
Una cosa da notare è che OrElse è prevenuto (ha un ordine preferenziale), se entrambi i Tchan non sono vuoti, allora readEitherChan restituirà sempre un elemento del primo. Se questo è un problema o no dipende dall'applicazione: bisogna considerare che la natura preferenziale di OrElse può creare iniquità in alcune situazioni.

Sicurezza delle eccezioni asincrone.

Fino ad ora non abbiamo detto nulla su come si comportano le eccezioni in STM. La monade STM supporta le eccezioni un po' come la monade IO, con due operazioni:

```
throwSTM :: Exception e => e -> STM a
catchSTM :: Exception e => STM a -> (e -> STM a) -> STM a
```

throwSTM genera un'eccezione, e catchSTM le cattura e richiama un gestore, proprio come catch nella monade IO. Tuttavia, le eccezioni in STM sono differenti in un modo fondamentale:

> In catchSTM m h, se m solleva un'eccezione, allora tutti i suoi effetti sono scartati, quindi il gestore h viene richiamato. Come caso degenere, se non c'è proprio catchSTM, allora tutti gli effetti della transazione vengono scartati e l'eccezione si propaga fuori dall'atomically.

Questo comportamento di catchSTM è stato introdotto in una successiva modifica di Harris, il comportamento originale in cui gli effetti non venivano scartati viene generalmente considerato molto meno utile. Un esempio aiuta a dimostrare la motivazione:

```
readCheck :: TChan a -> STM a
readCheck chan = do
a <- readTChan chan
checkValue a
```

checkValue impone alcuni vincoli ulteriori sul valore letto dal canale. Tuttavia, supponiamo che checkValue sollevi un'eccezione (forse accidentalmente, ad esempio la divisione per zero). Si preferirebbe che la readTChan non aveva successo, poiché un elemento del canale sarebbe perduto. Inoltre, vorremmo che readCheck avesse questo comportamento, indipendentemente dal fatto che vi sia un gestore di eccezioni o meno. Quindi catchSTM scarta gli effetti del suo primo argomento in caso di un'eccezione.
Il metodo di rigetto degli effetti è ancora più utile in caso di eccezioni asincrone. Se un'eccezione asincrona si verifica durante un transazione STM, l'intera transazione viene interrotta (a meno che l'eccezione non venga presa e gestita, ma gestire eccezioni asincrone in STM non è qualcosa che di solito si vuole fare). Quindi in molti casi, la sicurezza delle eccezioni asincrone in STM è data dal non fare assolutamente nulla. Non

ci sono lock da sostituire, quindi nessuna necessità di gestori di eccezioni o bracket, né bisogno di preoccuparsi di sezioni critiche da proteggere con mask.

L'attuazione di Tchan dato in precedenza è del tutto sicura rispetto alle eccezioni asincrone cosi come è, e inoltre la composizione di queste operazioni è anch'essa sicura.

STM fornisce un bel modo di scrivere codice che è automaticamente sicuro rispetto alle eccezioni asincrone, in modo da poter essere utile anche per stati non condivisi tra i thread. L'unico problema è che dobbiamo usare STM costantemente per ogni stato, ma dopo aver fatto questo passo la sicurezza delle eccezioni asincrone è automatica.

Prestazioni

Come con la maggior parte delle astrazioni, STM comporta un costo di runtime. Se si comprende il modello dei costi, allora possiamo evitare di scrivere codice che comporti costi eccessivi nei casi peggiori. Quindi in questa sezione diamo una descrizione informale dell'implementazione di STM (almeno in GHC), con dettagli sufficienti che permettano al lettore di poter comprendere il modello dei costi.
Una transazione STM funziona accumulando un registro (log) di operazioni readTVar e writeTVar che sono accadute finora durante la transazione.
Il registro è utilizzato in tre modi:

- Memorizzando operazioni writeTVar nel registro piuttosto che applicandole immediatamente alla memoria principale, scartare gli effetti di una transazione è facile, dobbiamo solo eliminarle dal registro. Di conseguenza, l'interruzione di una transazione ha un costo prestabilito piccolo.

- Ogni readTVar deve scorrere il registro per verificare se la TVAR era stata scritta da un writeTVar precedente. Quindi, readTVar è un'operazione di grado O (n) con n lunghezza del log.

- Poiché il registro contiene un record di tutte le operazioni readTVar, può essere usato per scoprire l'insieme completo di TVars lette durante la transazione, questo è necessario per eseguire retry.

Quando una transazione raggiunge la fine, l'attuazione STM confronta il registro e il contenuto di memoria utilizzando un protocollo a due fasi di bloccaggio. Se il contenuto

corrente della memoria corrisponde ai valori letti da readTVar, gli effetti della transazione sono applicati alla memoria atomicamente, in caso contrario, il registro viene eliminato e la transazione viene eseguita nuovamente dall'inizio. L'implementazione STM in GHC non usa lock globali; solo le TVars coinvolte nella transazione vengono bloccate durante l'assegnazione, quindi le operazioni che operano su insiemi disgiunti di TVars possono procedere senza interferenze.
La regola generale da seguire quando si utilizza STM non è mai di leggere un numero illimitato di TVars in una singola transazione, perché la prestazione di grado O (n) di readTVar poi dà O (n2) per l'intera transazione. Inoltre, transazioni lunghe sono molto più propense a fallire nell'applicazione, perché un'altra transazione avrà probabilmente modificato una o più delle stesse Tvars nel frattempo, per cui vi è un'alta probabilità di riesecuzione. È possibile che una implementazione STM futura possa utilizzare una differente struttura per memorizzare il log, riducendo l'overhead di readTVar a O (log n) o meglio (in media), ma la probabilità che una transazione lunga fallisca sarebbe ancora un problema. Per evitare questo problema è richiesto un metodo per la gestione delle contese intelligente, che è un'area di ricerca ancora attiva.

Riassumendo, STM garantisce diversi benefici per la programmazione concorrente

1. Atomicità componibile. Possiamo costruire arbitrariamente grandi operazioni atomiche su stati condivisi, in grado di semplificare l'attuazione delle simultanee strutture dati con un bloccaggio a grana sottile.

2. Blocco componibile. Siamo in grado di costruire le operazioni che fanno una scelta tra più operazioni di blocco, qualcosa che è molto difficile con MVars e altre astrazioni per la concorrenza di basso livello.

3. Robustezza in presenza di failure e cancellazione. L'operazione in corso viene interrotta se si verifica un'eccezione, così STM rende l'invarianza degli stati facile da mantenere in presenza di eccezioni.

Concorrenza e la Foreign Function Interface

Haskell ha una F*oreign Function Interface* (FFI) cioè un'interfaccia per le funzioni esterne che permette al codice Haskell di utilizzare ed essere utilizzato da codice di altri

linguaggi (soprattutto C). Linguaggi diversi hanno anche i loro propri modelli di threading (in C c'è POSIX o Win32 Thread, ad esempio) quindi abbiamo bisogno di specificare come Concurrent Haskell interagisce con modelli di threading di codici diversi.
D'ora in poi presumiamo che l'opzione GHC -thread sia in uso. Senza -threaded, il processo di Haskell utilizza un thread unico del sistema operativo, e chiamate multi-thread non sarebbero supportate.

Foreign out-call

Un out-call è una chiamata effettuata da Haskell a un altro linguaggio. Al momento FFI supporta solo le chiamate a C, quindi d'ora in poi ci riferiremo ai thread in C con OS threads.
Ad esempio consideriamo come rendere la funzione POSIX read() utilizzabile da Haskell:

```
foreign import ccall " read "
c_read :: CInt -- file descriptor
-> Ptr Word8 -- buffer for data
-> CSize -- size of buffer
-> CSSize -- bytes read , or -1 on error
```

Dichiara una funzione Haskell c_read che può essere utilizzato per chiamare la funzione C read ().
Proprio come i thread Haskell eseguono contemporaneamente l'uno con l'altro, quando un thread Haskell effettua una chiamata all'esterno, quella chiamata esterna ha luogo in concomitanza con gli altri thread Haskell, e quindi con qualsiasi altra chiamata esterna attiva. Chiaramente l'unico modo che due chiamate C possano essere eseguite contemporaneamente è che siano in esecuzione in due thread OS separati, questo è esattamente ciò che accade; se più thread Haskell chiamano c_read e tutti sono bloccati in attesa dei dati da leggere, ci sarà un thread OS per chiamata bloccato in read ().

Questo deve lavorare nonostante i thread Haskell non sono normalmente mappati uno ad uno con i thread del sistema operativo, come abbiamo già detto in precedenza, in GHC, i thread Haskell sono leggeri e gestiti nello spazio utente dal sistema di runtime. Quindi, per gestire le chiamate concorrenti esterne, il sistema runtime deve creare più thread OS, e di fatto lo fa su richiesta. Quando un thread

effettua una chiamata Haskell esterna, un altro thread OS viene creato (se necessario), e la responsabilità per l'esecuzione dei thread rimanenti Haskell è consegnato al nuovo thread del sistema operativo, nel frattempo il thread corrente del sistema operativo fa la chiamata esterna.

L'implicazione di questa progettazione è che una chiamata esterna può essere eseguita in qualsiasi thread del sistema operativo, e successive chiamate possono anche essere eseguite in differenti OS thread. Nella maggior parte dei casi questo non è importante, ma a volte lo è: del codice esterno deve essere chiamato da un particolare thread del sistema operativo.
Ci sono due istanze di questo requisito:

- Le librerie che permettono solo ad un thread OS di utilizzare il loro API. Le librerie GUI spesso rientrano in questa categoria: non solo la libreria deve essere chiamata solamente dal thread OS, spesso deve essere un thread particolare (ad esempio il thread principale). L'API Win32 GUI sono un esempio di questo.

- Le API che usano stati interni del thread locale. Il più noto esempio di questo è OpenGL, che supporta l'utilizzo multi-threaded, ma memorizza gli stati tra le chiamate API in nella memoria locale del thread. Da qui, le successive chiamate devono essere effettuate nello stesso thread OS, altrimenti la chiamata successiva vedrà stati errati.

Per questo motivo, il concetto di *bound thread* è stata introdotta. Un bound thread è una coppia formata da thread Haskell e thread del sistema operativo, in modo tale che le chiamate esterne da parte del thread Haskell avvengono sempre nel relativo thread del sistema operativo. Un bound thread è creato da forkOS:

```
forkOS :: IO () -> IO ThreadId
```

Bisogna fare attenzione quando si utilizza forkOS perché crea un thread OS completamente nuovo e può risultare molto costoso.

Foreign in-calls

Le in-calls sono chiamate a funzioni Haskell che sono state esposte a codice esterno usando foreign export. Per esempio se abbiamo una funzione f del tipo Int -> IO Int :

```
foreign export ccall "f" f :: Int -> IO Int
```

Questo creerà una funzione C con la seguente firma

```
HsInt f( HsInt );
```

HsInt qui è il tipo C corrispondente al tipo di Int. Haskell. In un programma multithread è del tutto possibile che f potrebbe essere chiamato da più thread OS contemporaneamente. Il sistema runtime GHC supporta questo (almeno con l'opzione -threaded), con il seguente comportamento: ogni chiamata diventa un nuovo bounded-thread. Cioè, un nuovo thread Haskell viene creato per ogni chiamata, e il thread Haskell è collegato al thread OS che ha fatto la chiamata. Pertanto, ogni ulteriore out-call effettuate dal thread Haskell si svolgerà nello stesso thread OS che ha fatto l'originale in-call. Questo si rivela essere importante per trattare con callback GUI: GUI vuole eseguire solamente nel principale thread del sistema operativo, in modo che quando fa un callback in Haskell, dobbiamo garantire che le chiamate GUI effettuate dal callback si svolgano nello stesso thread OS che ha richiamato il callback.

Conclusioni

Haskell fornisce diversi di modelli di programmazione per la multiprogrammazione suddivisi in due classi: i modelli di programmazione parallela il cui l'obiettivo è quello di scrivere programmi che fanno uso di più processori per incrementare le prestazioni, e la concorrenza in cui l'obiettivo è quello di scrivere programmi che interagiscono con più agenti esterni indipendenti. I modelli di programmazione parallela in Haskell sono deterministici, cioè, questi modelli di programmazione sono definiti per dare gli stessi risultati indipendentemente da quanti processori sono utilizzati per farli funzionare. Ci sono due approcci principali: Strategie, che si basa su lazy evalutation (valutazione pigra) per ottenere il parallelismo, e le Monade Par che utilizza un più esplicito dataflow-graph per esprimere calcoli paralleli.

Dal punto di vista della concorrenza è stato introdotto il modello di base di programmazione che comprende thread e MVars per la comunicazione, e quindi abbiamo descritto il supporto Haskell per la cancellazione in forma di eccezioni asincrone. Infine si è mostrato come la Memoria Software Transazionale consenta di costruire astrazioni concorrenti in modo composto, e rende molto più facile la programmazione con l'utilizzo delle eccezioni. Abbiamo accennato brevemente anche all'uso della concorrenza con Haskell Foreign Function Interface che permette ad Haskell di interagire con altri linguaggi.

Bibliografia

[1] Real World Haskell by Bryan O'Sullivan, Don Stewart, and John Goerzen
http://book.realworldhaskell.org/read/

[2] The Haskell Programming Language
http://www.haskell.org/haskellwiki/Haskell

[3] Hoogle is a Haskell API search engine
http://www.haskell.org/hoogle/

[4] A Beginner's Guide by Miran Lipovača
http://learnyouahaskell.com/

[5] GHCi commands
http://www.haskell.org/ghc/docs/latest/html/users_guide/ghci-commands.html

[6] Parallel = Functional, the way of future by Simon Peyton Jones
http://research.microsoft.com/en-us/um/people/simonpj/papers/parallel/Parallel-Haskell.pdf

[7] GHC/Data Parallel Haskell
http://www.haskell.org/haskellwiki/GHC/Data_Parallel_Haskell

[8] Control.Parallel.Strategies
http://hackage.haskell.org/packages/archive/parallel/3.2.0.3/doc/html/Control-Parallel-Strategies.html

[9] Parallel Processing with Haskell
http://www.yellosoft.us/parallel-processing-with-haskell

[10] Monad
http://www.haskell.org/haskellwiki/Monad

[11] Control.Concurrent
http://hackage.haskell.org/packages/archive/base/4.5.0.0/doc/html/Control-Concurrent.html