**Emerging Programming Paradigms**

[ lezioni 16-18 marzo ]

La receive è sempre bloccante in Erlang (come in ogni linguaggio asincrono), mentre la send non lo è mai.

Date le funzioni:

g(x) { f(x) {

int y = 4; int y = 3;

w = g(x + 2); w = g(x + 2);

return x + y + w; l: return x + y + w;

} }

Quando viene chiamata ad esempio f(2) viene creato un *record di attivazione*, che contiene:

---------------

y: 4

x: 4 g(4) record di attivazione

RA: l

---------------

w:

y: 3

x: 2 f(2) record di attivazione

RA: ...

---------------

Nei *linguaggi meno recenti*, tutti i processi quando vengono creati ricevono uno stack e un heap. Nel momento in cui viene terminato lo spazio dedicato allo stack il processo termina (con un’eccezione di *Stack Overflow*). Un problema importante è dato dai continui controlli se la memoria eccede stack o heap, per evitare questo overhead è presente un componente hardware che controlla quando lo stack e/o l’heap eccedono la propria memoria e lancia un’eccezione.

Per le chiamate ricorsive si utilizza una sola pagina, per cui terminato lo spazio verrà lanciata un eccezione.

Nei *linguaggi più recenti* non viene più assegnata una pagina fisica per ogni attore. Lo stack viene implementato come una struttura dati nello heap, precisamente un *array estensibile*. Un array in memoria con una certa dimensione che viene gestito internamente come uno stack. Quando questo eccede la memoria dedicatagli viene copiato interamente da un’altra parte raddoppiandogli la memoria a disposizione. L’overhead è dato dal controllo che lo stack non ecceda il proprio spazio e dalla copia dei dati in una nuova porzione di heap. Quando l’heap finirà, il programma avrà terminato la RAM e sarà lanciata un’eccezione.

Affinchè Erlang e gli altri linguaggi funzionali possano essere puri (ovvero usare solo funzioni ricorsive) devono garantire che le funzioni ricorsive vengano compilate senza far crescere lo stack. Questa ottimizzazione è chiamata **ottimizzazione di tail recursion**.

La *tail recursion* può essere implementata in tutti i linguaggi. Se un linguaggio dice di avere feature funzionali ma non implementa la tail recursion allora vuol dire lo stack può terminare. Java è uno di questi casi.

Una chiamata di coda (*tail call*) si ha quando il chiamante:

- Non compie altre operazioni dopo la chiamata.

- Restituisce come valore il valore di ritorno della chiamata.

Un esempio è dato dalla seguente funzione:

main(L) ->

case L of

[] -> io:format("Fine lista~n");

io:format("Passo e chiudo");

[\_|TL] -> main(TL).

In questo caso sono chiamate di coda la seconda chiamata ad io:format e quella a main.

Una funzione ricorsiva in cui tutte le chiamate sono di coda si chiama funzione ricorsiva di coda (*tail recursion*). È garantita utilizzare spazio costante se invoca solo altre funzioni con la stessa caratteristica.

Nel caso di chiamata *tail-ricorsiva*, il record del chiamante viene rimosso (pop) e sostituito dal record del chiamato (push), senza far crescere la dimensione della pila. Differenza importante è però che come indirizzo di ritorno del chiamato si usa quello del chiamante.

L’ottimizzazione non è solo per chiamate ricorsive ma in generale per chiamate funzioni.

Erlang deve implementare la tail recursion optimization perchè altrimenti non potrebbe garantire sempre il servizio dei server.

**Tail Recursion Optimization**

In caso di ricorsioni multiple risulta inutile passare l’indirizzo di ritorno alla chiamata precedente, la quale farà lo stesso con la precedente e così via fino alla prima chiamata. Un’ottimizzazione consiste nel passare direttamente dall’ultima chiamata ricorsiva alla prima.

Ad esempio date le funzioni:

f(X, Y) -> X + Y

g(A) -> f(A, A)

In un certo momento si mantiene l’indirizzo di ritorno di g e si esegue f(A, A) a questo punto si rilasciano i parametri di g e si riallocano i parametri di f nella stessa area di memoria:

Y: 13

A: 13 diventa X: 13

RA: … RA: ...

Il compilatore controlla che non ci siano operazioni dopo la chiamata ricorsiva e nel caso ottimizza. In termini di occupazione di memoria una funzione tail recursive o solo ricorsiva hanno un comportamento molto differente.

Problema: sommare tutti i numeri in una lista:

% Soluzione NON tail ricorsiva [occupazione memoria O(len(L))]

sum([]) -> 0;

sum([H | T]) -> H + sum(T).

% Soluzione tail ricorsiva [occupazione memoria O(1)]

sum([], A) -> A;

sum([H | T], A) -> sum(T, H + A).

Per le funzioni tail recursive si utilizza un accumulatore (nell’esempio la variabile A).

**Trasformazioni meccaniche**

1. Si implementa “a mano” uno *stack delle chiamate.*

È uno stack di funzioni anonime non ricorsive.

- Non si utilizza più lo stack di sistema ma il programmatore gestisce uno stack nello heap.

- Questo è utile solo quando il linguaggio gestisce un heap di dimensione piccola e fissata.

Si passa come parametro uno stack che simula quello delle chiamate e si fanno pop/push di queste funzioni in questo stack.

somma([], S) -> pop(0, S);

somma([H | T], S) -> somma(T, push(S, fun (R) -> H + R end)).

push(S, F) -> [F | S].

pop(R, []) -> R;

pop(R, [F | S]) -> pop(F(R), S).

ESEMPIO

somma([1, 2, 3], [])

-> somma([2, 3], [fun (R) -> 1 + R])

-> somma([3], [fun (R) -> 2 + R end,

fun (R) -> 1 + R end ])

-> somma([], [ fun (R) -> 3 + R end,

fun (R) -> 2 + R end,

fun (R) -> 1 + R end ])

-> pop(0, [ fun (R) -> 3 + R end,

fun (R) -> 2 + R end,

fun (R) -> 1 + R end ])

-> pop(3, [ fun (R) -> 2 + R end,

fun (R) -> 1 + R end ])

-> pop(5, [ fun (R) -> 1 + R end ])

-> pop(6, [])

-> 6

Riscritto con i while (nei linguaggi che li mettono a disposizione, tipo C/C++/Java/ecc):

L = lista iniziale da sommare

S = []

continue = true

while(continue) {

case L of

[] -> pop(0); continue <- false;

[H | T] -> push(fun (R) -> H + R end); L <- T

}

push(F) { S <- [F | S] }

pop(R) {

continue = true

while(continue)

case S of

[] -> return R

[F | T] -> R <- F(R); S <- T;

}

2. *CPS Translation* (Continuation Passing Style)

Ha varie proprietà tra cui dopo aver applicato la CPS, tutte le chiamate sono di coda.

L’idea è che ogni funzione prenda in input la funzione che esegue il lavoro che deve essere eseguito dopo che la funzione è ritornata. Si trasforma la ricorsione in una composizione di funzioni anonime. È una sequenza di applicazioni.

cpssum([], K) -> K(0);

cpssum([H | T], K) -> cpssum(T, fun (R) -> K(H + R) end).

cpssum(L) -> cpssum(L, fun (R) -> R end).

ESEMPIO

cpssum([1, 2, 3])

-> cpssump([1, 2, 3], fun (R) -> R end)

-> cpssum([2, 3], fun (R) -> (fun (R) -> R end)(1 + R) end)

-> cpssum([3], fun (R) -> (fun (R) -> (fun (R) -> R end)(1 + R) end)(2 + R) end)

-> cpssum([], fun (R) -> (fun (R) -> (fun (R) -> (fun (R) -> R end)(1 + R) end)(2 + R) end)(3 + R) end)

-> (fun (R) -> (fun (R) -> (fun (R) -> (fun (R) -> R end)(1 + R) end)(2 + R) end)(3 + R) end)(0)

-> (fun (R) -> (fun (R) -> (fun (R) -> R end)(1 + R) end)(2 + R) end)(3)

-> (fun (R) -> (fun (R) -> R end)(1 + R) end)(5)

-> (fun (R) -> R end)(6)

-> 6

3. Tecnica dei *trampolines*

Il problema di partenza è dato dal seguente codice dove la chiamata della g è in posizione di coda:

f() { ...if return g(3)

...else return 5

}

...

x = f() - prima push record di f

... - prima push record di g

La soluzione prevede che si trasformi il codice come segue:

f() { ...if return {call, fun () -> g(3) end}

...else return {value, 5}

}

...

x = call(f())

..

call(J) {

I <- J;

while(true) {

case I of

{value, X} -> return X ;

{call, F} -> I <- F()

}

}

Invece che fare la chiamata vera e propria, viene restituita la funzione da chiamare in seguito (con un atomo “call” che indica che la funzione va chiamata).

Ad esempio data la funzione:

fact(N) -> fact(N, 1).

fact(0, A) -> A;

fact(N, A) -> fact(N - 1, N \* A).

Viene trasformata in ricorsiva di coda tramite *trampolines*:

fact(N) -> call(fact(N, 1)).

fact(0, A) -> {value, A};

fact(N, A) -> {call, fun () -> fact(N - 1, N \* A)}

L’esecuzione a livello di *activation records* è la seguente:

fact(5)

| RA di fact(5) |

-----------------------------------------------

call(fact(5, 1))

| RA di call({call, fun () -> fact(4, 5)) |

-----------------------------------------------

| RA di fact(5) |

-----------------------------------------------

fact(4, 5)

| RA di fact(4, 5) |

-----------------------------------------------

| RA di call({call, fun () -> fact(4, 5)) |

-----------------------------------------------

| RA di fact(5) |

-----------------------------------------------

dopo la chiamata

| RA di call({call, fun () -> fact(4, 5)) |

-----------------------------------------------

| RA di fact(5) |

-----------------------------------------------

fact(3, 20)

| RA di fact(3, 20) |

-----------------------------------------------

| RA di call({call, fun () -> fact(4, 5)) |

-----------------------------------------------

| RA di fact(5) |

-----------------------------------------------

dopo la chiamata

| RA di call({call, fun () -> fact(4, 5)) |

-----------------------------------------------

| RA di fact(5) |

-----------------------------------------------

fact(2, 60)

| RA di fact(2, 60) |

-----------------------------------------------

| RA di call({call, fun () -> fact(4, 5)) |

-----------------------------------------------

| RA di fact(5) |

-----------------------------------------------

... eccetera ...

Questo è il modo in cui molti linguaggi funzionali implementati sulla JVM implementano la ricorsione di coda.

**Gestione delle eccezioni**

Il modo più semplice di implementare i blocchi try-catch consiste nel creare uno speciale stack frame che punta al codice del catch e poi eseguire la try. Se la try termina con un valore, il blocco try-catch viene rimosso dalla cima dello stack. La throw itera sullo stack facendo pop di tutti gli stack frame (ovvero terminando le varie chiamate annidate di funzioni) fino a quando non incontra uno stack frame speciale che rappresenti un try-catch che sia in grado di catturare l'eccezione. In tal caso il frame speciale viene rimosso e viene eseguito il codice associato alla cattura. I blocchi con after richiedono ulteriore codice per il loro trattamento (ad esempio: il codice dopo after va eseguito prima di eliminare lo speciale stack frame). La creazione di uno stack frame speciale impedisce di ottimizzare una chiamata "di coda" contenuta nel try.

In Erlang per lanciare un’eccezione di tipo throw si usa:

throw({foo, 3}).

Queste eccezioni non hanno la backtrace delle chiamate a differenza delle eccezioni di tipo error. Esistono anche eccezioni di tipo exit, lanciate quando un programma termina.

Le eccezioni di tipo error rappresentano un bug nel codice, un errore di programmazione. Può avere senso catturare un’eccezione error per abortire eventualmente l’intero programma in quanto il codice è stato effettivamente scritto male e non è solo un errore a run time.

Gli throw vengono utilizzati in Erlang non per gestire situazioni del tutto errate o anomalie di cui fare recovery ma situazioni ragionevoli, ovvero viene utilizzato come un operatore di *controllo del flusso*.

Per catturare un’eccezione si usa:

try

throw({error1, 17})

catch

{error1, X} -> X;

{error2, Y} -> h()

end.

Prendiamo il seguente esempio di funzione di ricerca di un valore all’interno di un albero:

search(\_, nil) -> not\_found;

search(K, {node, \_, K, V, \_}) -> V;

search(K, {node, L, \_, \_, R}) ->

case search(K, L) of

not\_found -> search(K, R);

V -> V

end.

Non è tanto efficiente perché quando arriva in fondo a tutti i rami e viene trovato il valore cercato poi si torna indietro restituendo il valore e facendo pattern matching per tutti i rami passati [complessità O(n)].

Un’altra implementazione può essere la seguente:

s(\_, nil) -> not\_found;

s(K, {node, \_, K, V, \_}) -> throw({found, V});

s(K, {node, L, \_, \_, R}) ->

s(K, L),

s(K, R).

search2(K, T) ->

try s(K,T)

catch {found, R} -> R

end.

Usando un throw quando viene trovato il risultato si evita di risalire l’albero per ritornare il risultato ma si esce direttamente dall’esecuzione.

Prendiamo il seguente esempio di codice che fa uso di try-catch:

h(N) -> throw({err1, N}).

f(N) -> h(N) + 1.

g() ->

try

f(3)

catch

{err1, X} -> X;

{err2, Y} -> g()

end.

Ogni blocco catch diventa un nuovo tipo di record di attivazione:

g()

| record di attivazione per g() |

-----------------------------------------------

eseguo il try-catch

| record di tipo catch che punta al |

| codice del pattern matching |

-----------------------------------------------

| record di attivazione per g() |

-----------------------------------------------

eseguo f(3)

| record per f(3) |

-----------------------------------------------

| record di tipo catch che punta |

| al codice del pattern matching |

-----------------------------------------------

| record di attivazione per g() |

-----------------------------------------------

eseguo h(3)

| record per h(3) |

-----------------------------------------------

| record per f(3) |

-----------------------------------------------

| record di tipo catch che punta |

| codice del pattern matching |

-----------------------------------------------

| record di attivazione per g() |

-----------------------------------------------

throw({err1, 3})

Implementato come:

trovato = false;

valore = unknown;

while (not trovato)

R = pop();

case R of

{record\_di\_attivazione, F} ->

case F() of

{gestito, X} -> trovato = true; valore = X;

non\_gestito -> ok

end

{record\_di\_funzione, \_} -> ok

end.

| record di attivazione per g() |

-----------------------------------------------

Nel caso di terminazione con successo del codice nel blocco try:

pop();

eseguito il codice per "of" se presente

ATTENZIONE: i rami dell'of e i rami del catch sono eseguiti **dopo** aver fatto pop() e quindi **non** sono protetti dal blocco.

update(\_State) -> ok. % qui posso sollevare un eccezione

test\_no\_tr(State) ->

io:format("memory used no\_tr: ~p~n", [erlang:memory(total)]),

try test\_no\_tr(update(State))

catch \_ -> ko end.

test\_no\_tr creerà ad ogni chiamata ricorsiva un record di attivazione per la catch e la memoria prima o poi finirà. Questo avviene perché la chiamata non è di coda.

test\_tr(State) ->

io:format("memory used tr: ~p~n", [erlang:memory(total)]),

test\_tr( try update(State)

catch \_ -> ko

end).

test\_tr invece esegue il pop di ogni record di attivazione per la catch prima di eseguire la chiamata (tail) ricorsiva, quindi la memoria non cresce. Qui la chiamata è di coda.

test\_tr2(State) ->

io:format("memory used tr: ~p~n", [erlang:memory(total)]),

try update(State) of

UpdatedState -> test\_tr2(UpdatedState)

catch \_ -> ko

end.

In generale il try-of-catch esegue il codice nel try, se viene lanciata un’eccezione viene eseguito il contenuto del catch, altrimenti si esegue ciò che sta dopo of, FUORI dal blocco try catch (ovvero dopo aver fatto pop del record di attivazione del catch).

test\_tr2 come codice è equivalente al precedente, ma esteticamente più piacevole. Dopo l'of si può descrivere (tramite pattern matching) cosa fare del risultato. In questo esempio si procede sempre comunque alla chiamata ricorsiva. Il codice nell'of NON è protetto dal catch (ovvero le eccezioni sollevate in tale ramo scappano). Pertanto prima di valutare l'of si può immediatamente fare pop del catch frame dallo stack. Il codice after (se presente) viene comunque eseguito.

test\_no\_tr2(State) ->

io:format("memory used tr: ~p~n", [erlang:memory(total)]),

try update(State) of

UpdatedState -> test\_no\_tr2(UpdatedState)

catch \_ -> ko

after ok

end.

La keyword after è l’equivalente del finally in Java, ovvero permette di eseguire codice dopo il try catch a prescindere dall’esito.

test\_no\_tr2 nonostante il codice usi l'of come nel caso test\_tr2, c'è una clausola after che deve essere eseguita dopo il ramo of. Pertanto la chiamata ricorsiva non è di coda e la memoria cresce (per via dello spazio occupato da ogni blocco relativo all’after ad ogni chiamata ricorsiva).

Un uso tipico dell’after è quando si apre un file e lo si chiude, appunto, dell’after per essere sicuri che venga chiuso anche se viene lanciata un’eccezione.

try

E

of

P' -> F'

catch

P -> F

after G

Viene compilata come:

R = (try

{ok, E}

catch

S -> {exc, S}),

case R of

{ok, E} ->

case E of

P' ->

try

Y = F',

G,

Y

catch

U -> G, throw(U)

end

end;

{exc, S} ->

case F of

P ->

try

X = F,

G,

X

catch

Y -> G, throw(Y)

end

end

end.