**Emerging Programming Paradigms**

[ lezioni 02-04-09-11 marzo ]

**Programmazione ad attori ed Erlang**

**Paradigma ad attori**

Questo paradigma prevede la presenza di **attori** identificati univocamente da un **PID** che comunicano tramite messaggi sulla propria **mailbox**. Ogni attore ha un **behaviour** che mappa ogni messaggio in una lista di azioni ed in un nuovo behaviour (computazione reattiva/event driven). Un’**azione** può essere banalmente una *computazione interna*, un invio asincrono di *messaggi verso un PID*, la *creazione* di nuovi *attori*, ecc.

Ogni **attore** viene eseguito su un **unico thread** di computazione che **non condivide** né **memoria** né stato con gli altri attori.

**Paradigma ad attori e quello ad agenti**

Spesso il paradigma ad agenti è visto come un caso particolare di quello ad attori. In generale due **agenti** non condividono stato, behaviour o memoria, possono **comunicare** solamente **modificando l’ambiente condiviso**. Un **attore** invece come visto prima, possiede un **identificativo**, e una **mailbox** a cui vengono recapitati direttamente i messaggi.

È possibile simulare la programmazione ad agenti con quella ad attori se si designa un attore ad essere l’ambiente che verrà letto e modificato dai vari agenti. Simulare la programmazione ad attori con quella ad agenti è possibile se si assegna un identificatore univoco ad ogni agente ed ognuno recupera tuple rappresentanti messaggi indirizzati a loro direttamente dall’ambiente condiviso.

**Punti di forza e di debolezza del paradigma ad attori**

Il paradigma ad attori è **consigliabile** quando è possibile **suddividere** il problema **in task indipendenti**. Inoltre l’indipendenza della memoria tra gli attori **evita** l’utilizzo di **lock** per coordinare l’accesso allo stato globale.

Di **contro** questo paradigma non fornisce certezze sull’**arrivo dei messaggi**, scenario che dovrà essere gestito con attenzione in fase di progettazione.

**Erlang**

Erlang fu sviluppato presso la **Ericsson** nel 1987 per implementare software di telecomunicazione. L’obiettivo era quello di creare software in grado di gestire una **concorrenza massiva** ed essere **disponibile il 99.9999%** del tempo. Dopo aver prima sperimentato vari altri linguaggi la Ericsson decide di implementare Erlang, prima in modo proprietario poi nel 1998 reso open-source.

Le caratteristiche principali sono l’**assenza** di **memoria condivisa**, lock e mutex, tipico del paradigma ad attori. Il **message passing** è **asincrono** e l’ordine dei messaggi casuale, la politica di **fault tolerance** è “**let it fail**”, ovvero in caso di errore gli attori coinvolti e quelli collegati ad essi vengono lasciati fallire per poi eventualmente essere rilanciati.

La macchina virtuale di Erlang, la **BEAM**, gira come **unico kernel process** multi-threaded (1 kernel thread per core) ed **ogni attore** rappresenta un **language thread** schedulato sui kernel thread con bilanciamento del carico automatico.

**Features di un linguaggio funzionale**

* **Higher-order functions**, ovvero le funzioni possono prendere in input e restituire altre funzioni. Più genericamente le funzioni sono oggetti di prima classe: possono essere usate ovunque posso usare un valore (prese in input, date in output, memorizzate in una struttura dati, etc.), questo perché a basso livello una funzione altro non è che una tabella (certi valori in input danno certi valori in output) e può tranquillamente essere trattata come un dato normale. Il lambda calcolo è il cuore di un linguaggio in cui si hanno solo funzioni anonime, le quali si possono passare in input e in output.
* Il linguaggio è **puro** quando *non sono presenti side-effect* (ciò che non ha a che fare con l’input-output di una funzione). In particolare tutte le variabili e le celle in memoria sono *immutabili*. Modificare una purely functional data structure significa crearne una nuova versione, che si affianca alla precedente, riusando le parti della precedente che non vengono modificate. Tale costruzione è corretta grazie all'assenza di mutabilità. Vedi per esempio il file bstrees.erl negli esempi di codice. L'assenza di mutabilità semplifica la programmazione concorrente (no race conditions), permette di evitare costose copie di strutture dati per evitare modifiche non volute e permette il backtracking for free.
* Hanno **sistemi di tipi molto complessi** che permettono di scrivere programmi molto generali.

I linguaggi funzionali sono spesso definiti HOT = Higher Order and Typed.

In un linguaggio di programmazione le funzioni possono essere **oggetti di prima classe**, ovvero si può fare tutto ciò che si fa banalmente con un intero (ad esempio creare un array).

**Erlang** non **utilizza** lo stack ma **solo** **heap** quindi è più veloce. Inoltre non c'è mutazione di variabili e l'unico modo per ripetere un'operazione è tramite la ricorsione. Il **λ-calcolo** è un requisito minimo per un linguaggio di ordine superiore.

**Sintassi Erlang**

* Uguaglianza: =:=
* Disuguaglianza: /=
* Atomo: parola
* Stringa: “parola”
* Concatenazione stringhe: “ciao” ++ “come”
* Tuple: {“ciao”, 2, {atom}}
* PID: self()
* Cookie: make\_ref() [ovvero un nonce, un numero generato una volta sola]
* Liste: [1, ciao, [], self()]
* Creazione nuova lista da precedente: [0, atom | L]
* Mappe: #{19 => 4, b => 5}
* BIFs: is\_atom(), is\_integer(), atom\_to\_list(), binary\_to\_term(), …

Nota: ci sono casi in Erlang in cui possono essere usate solo BIFs e non funzioni create dall’utente.

Tipi di dato atomici in Erlang: int, float, stringhe, atomi, PID, cookies, porte.

Un record (tupla) può tranquillamente contenere un numero ed una funzione anonima, ad esempio: {1, fun (X, Y) -> X + Y end} (fun definisce l’apertura di una funzione, e la keyword end la sua chiusura).

Una funzione che prende in input un valore e restituisce in output un’altra funzione:

adder = fun (I) -> fun (X) -> X + I end end

All’interno dei file .erl compilati vi è zucchero sintattico che permette di evitare “fun” e “end” quando si dichiara una funzione: H = fun (X) -> X + 1 end diventa H(X) -> X + 1.

Per poter fare chiamate ricorsive è possibile utilizzare un nome (maiuscolo) per la funzione:

F = fun G (X, Y) -> G(X + 1, Y)

**Pattern matching**

{X, Y} = {3, “ciao”} se la tupla a destra avrà la stessa forma di quella di sinistra, verrà accoppiato l’i-esimo elemento della tupla di destra con quello di sinistra. Questo pattern matching può essere effettuato su strutture di vario tipo, ad esempio una lista:

[A, B | L] = [1, 2, 3, 4]

In Erlang il pattern matching è gestito da 3 costrutti:

- Il simbolo uguale

- Le funzioni (con guardia gestita da when)

- La receive dei messaggi

- E ovviamente il costrutto case of

Si può utilizzare il placeholder \_ per dire “qualsiasi caso non matchato prima”.

Codice Erlang per BST: [bstree3.erl](https://iol.unibo.it/pluginfile.php/543966/mod_folder/content/0/bstree3.erl?forcedownload=1)

Costo spaziale dell’algoritmo è logaritmico. Perché deve mantenere in memoria i record di attivazione delle funzioni essendo un implementazione ricorsiva e non iterativa.

**Rappresentazione a run-time dei valori in memoria**

La funzione di swap è una funzione che prende in input una coppia e ne restituisce un’altra in cui gli elementi sono posti in ordine inverso.

In Erlang questa funzione è polimorfa in quanto può essere eseguita su tipi di dato differenti. In C in base al tipo di dato l’implementazione di swap sarebbe stata notevolmente differente.

Primo approccio (C++): **monomorfizzazione**

In C++ si potrebbe utilizzare un template in cui si generalizza il tipo degli elementi passati ad una struttura in modo che sia adattabile a vari contesti. In questo caso il codice verrebbe generato solo quando vengono passati gli elementi alla swap (che utilizza il template), a questo punto conoscendo il tipo è possibile sapere come implementare il codice.

struct {

T1 x;

T2 y;

} T<T1,T2>;

struct T<T2,T1> swap(struct T<T1,T2> p) {

struct T<T2,T1> o;

o.x = p.y;

o.y = p.x;

return o;

}

main() {

struct T<int,int[2]> i = {1, {2, 3}};

struct T<int[2],int> o = swap<int,int[2]>(i);

}

Il *vantaggio* è che il codice sarebbe super-ottimizzato in quanto ogni istanza viene ottimizzata separatamente. Inoltre il codice ottenuto post-monomorfizzazione è C standard e può essere invocato semplicemente da altri linguaggi.

Lo *svantaggio* della monomorfizzazione è che la quantità del codice prodotta è elevata dal momento che il codice verrà generato più e più volte (una versione della funzione specifica per ogni tipo con cui viene invocata), quindi gli eseguibili saranno più pesanti. Il tempo di caricamento iniziale aumenterebbe ma soprattutto l’utilizzo elevato della cache rischia di riempirla con una parte di codice e se la località non è particolarmente elevata potrebbero avvenire molti swap di codice all’interno della cache, la quale ha tempi di accesso molto più lenti della RAM portando ad un rallentamento generale dell’esecuzione.

Secondo approccio (C): **cicli per copiare manualmente i byte**

Questo approccio viene utilizzato dalla libreria standard C, la funzione swap non prenderebbe solo un puntatore a una generica area di memoria ma anche due interi che indicano la dimensione in byte del primo e secondo campo. Quindi non si comunica più il tipo e la dimensione con un solo parametro ma si separano le due cose. L’unica differenza presente è la dimensione del tipo di dato, non la struttura del tipo.

void \* swap(void \*p, int size\_x, int size\_y) {

void \*o = malloc(size\_x + size\_y);

// ciclo per copiare i primi size\_x byte di p in o a partire da size\_y

// ciclo per copiare i byte da size\_x a size\_x + size\_y byte di p in o

return o;

}

...

struct T i = { 5, 4 };

swap((void \*)i, size\_of(i.x), size\_of(i.y));

...

Nota: size\_of è un costrutto del linguaggio, *non* è una funzione. A tempo di compilazione determina il tipo di una variabile e di conseguenza la sua lunghezza.

Gli *svantaggi* di questo approccio sono che è applicabile solo se l'unica differenza a run-time è riconducibile alla taglia e che il codice non viene ottimizzato sulle singole istanze (es. ciclo anche su un dato che occupa un numero fissato di parole).

Terzo approccio (Erlang): **rappresentazione uniforme dei dati**

La filosofia è che tutti i dati occupano esattamente un parola. Un intero di fatto occupa già una parola, valori più “piccoli” come booleani, int8, 4 atomi che di base occuperebbero meno di una parola vengono “forzati” ad occuparne una comunque, sprecando dei bit. In questo caso parliamo di **dati unboxed**. Un tempo questo approccio non era efficiente in quanto la RAM era molto limitata.

Per quanto riguarda altri tipi di dati come gli array: il dato viene sempre allocato sullo heap, e viene sempre rappresentato con il suo puntatore che occupa una parola. In questo caso parliamo di dati **boxed**.

Esempio:

{a, 5, {2, 3}} rappresentato con il puntatore p0 e l'HEAP seguente

p0 HEAP

...

p0: a // un numero associato all'atomo a

5

p1

...

p1: 2

3

Con il pattern matching:

{X, Y, Z} = {a, 5, {2, 3}}

Viene compilato come:

match(void \*p0) {

X = \*(p0); // sull'input {a, 5, {2, 3}} X diventa l’atomo a

Y = \*(p0+1); // Y il numero 5

Z = \*(p0+2); // Z il puntatore p1; riusare Z significa solo copiare il

// puntatore e non il record nello heap condiviso

}

Un problema dei linguaggi non tipati è che banalmente 0 == false potrebbe restituire true, per questo è necessario un metodo che salvi in memoria anche il tipo di un dato. Una parola di 64 bit viene quindi rappresentata come:

MSB -------------------- | ---- LSB

payload tag

Dove il tag codifica il tipo del dato.

Un *problema* con questo approccio è che il **tag ha spazio limitato** quindi anche il numero di tipi sarà un numero fisso.

Stesso problema si presenta per il **payload**, quindi nel caso degli interi (che potrebbero usare tutta l’intera word) non verrebbe usata tutta la word. Per risolvere questo problema si passa a un tipo boxed con tante word quante sono necessarie per rappresentare l’intero in questione. Ci sono delle librerie che ti permettono di usare payload più grandi.

Un’ulteriore problema è che quando ci si interfaccia con il **mondo esterno**, in cui ad esempio gli interi sono rappresentati da 64bit, potrebbero non corrispondere con quelli rappresentabili nel linguaggio (a causa dello spazio ridotto dal tag). Gli interi che riceviamo potrebbero essere troppo grandi.

Ancora, le **operazioni aritmetico/logiche** della CPU non sono più utilizzabili. In quanto, in una banale somma ad esempio, la CPU andrebbe a sommare anche i tag in coda alle word, fornendo un tag diverso nel risultato (alterando il suo tipo). Per questo è necessario sottrarre al primo operando il proprio tag (ottenendo tag con tutti 0) e poi sommare il risultato al secondo valore (ottenendo tag come quello del secondo valore). Più complesso quando si parla di moltiplicazioni e divisioni.

Un ultimo problema riguarda i **puntatori**, in quanto anch’essi devono essere taggati e questo limita ulteriormente lo spazio.

Una possibile *soluzione* è utilizzare tag di lunghezza variabile. I puntatori puntano solo agli indirizzi pari che sono memory aligned, quindi l’accesso è efficiente e si accede a tutta la RAM al bisogno (semplicemente le strutture contigue in RAM tipo array devono iniziare in posizione pari).

I tag possono essere di lunghezza variabile (esempio):

0 puntatori ==> restano 63 bit di payload che indirizzano

solamente gli indirizzi di memoria pari così il

garbage collector può facilmente distinguere

puntatori da non-puntatori

01 interi

011 atomi

0111 pid

1111 ref

In Erlang c’è un garbage collector che si occupa di allocare e deallocare la memoria, queste decisioni le prende riconoscendo i puntatori (l’ultimo bit del tag è 0). Erlang non è particolarmente efficiente se un programma richiede molte operazioni su interi.

Un approccio alternativo usato da altri linguaggi è utilizzare due parole, la prima parola contiene il payload, ad esempio, e il secondo il puntatore.

Caso di esempio: Erlang vs Ocaml

- **Erlang** è un linguaggio **non tipato**. Pertanto a run-time il numero 0 e il booleano false

devono essere rappresentati diversamente come sequenze di bit per evitare che 0 = false

abbia successo. Pertanto Erlang usa un tag per ogni tipo (intero, atomo, PID, ...)

- **Ocaml** è un linguaggio **tipato**. Pertanto espressioni tipo 0 = false sono rigettate a compile

time e a run-time il linguaggio può usare la stessa sequenza di bit per 0 e per false.

Pertanto Ocaml usa solo due tag di lunghezza 1 bit: il tag 0 è per i puntatori (dati boxed)

e quello 1 per tutto il resto (dati unboxed). La distinzione è necessaria per implementare un

garbage collector mark & sweep.

**Garbage Collector**

Descrizione dettagliata: [Erlang Garbage Collector](https://www.erlang-solutions.com/blog/erlang-garbage-collector.html) [fino a “Sizing the heap”]

La **congestione della memoria** prevede di gestire l’allocazione, la deallocazione, evitare l’uso di memoria allocata e gestire gli accessi concorrenti alla memoria.

La gestione *manuale* della memoria la troviamo nei linguaggi più antiquati e/o quelli che nascono per essere a basso livello (es. utilizzo di primitive come malloc, delete, …). Il C++ idiomatico prevedeva di allocare e deallocare manualmente gli oggetti, ora non lo fa più, stessa cosa vale per Rust, che non gestisce più totalmente in modo manuale la memoria.

Questo approccio prevede che il programmatore si occupi di quale thread dovrà allocare la memoria e quale deallocarla.

*Smart pointers* (es. in C++ o Rust): un sistema di gestione della memoria semi-automatica in cui il programmatore fa allocazione esplicita, ma quello che viene restituito è un oggetto che gestisce l’accesso al puntato.

La gestione *automatica* della memoria avviene tramite un **garbage collector**, ovvero un meccanismo che tiene traccia dell’uso della memoria in modo che il programmatore non si occupi della memoria ma che venga deallocata, con una certa approssimazione, in modo automatico. E lo stesso che controlla l’accesso concorrente della memoria. Rust ha un sistema di tipi ispirato alla logica lineare.

> **Reference Counting** (C#, GTK, …)

Tutti i dati che vengono allocati sullo heap vengono preceduti sullo heap da una parola, un numero che conta il numero di puntatori entranti. Quanti puntatori puntano a quell’area di memoria.

Quando si **copia un puntatore**, ad esempio in un’altra struttura dati, si incrementa il reference counter dell’oggetto puntato.

Ogni qual volta si **cancella un puntatore**, si accede al puntatore, si legge la parola, si decrementa di uno, se arriva a 0 viene considerata garbage e si dealloca il dato memorizzato, ripetendo l’operazione ricorsivamente se il valore puntava a sua volta altri elementi.

La reference counting ha delle falle, tra cui situazioni di **memory leak**, ovvero memoria inutilizzata non riconosciuta come tale che rimane occupata fino alla fine dell’esecuzione del programma.

Ad esempio strutture di dati circolari (liste circolari, grafi, ...):



Nella situazione in cui abbiamo una lista circolare in cui ogni elemento punta al successivo e l’ultimo punta al primo, un registro punta al primo elemento della lista, nel momento in cui si cancella il primo puntatore del registro, si decrementa il counter del primo elemento della lista, e se non si vuole avere un blocco di memoria occupato e non utilizzato bisogna ripetere l’operazione ricorsivamente per tutti gli altri elementi. Arrivato in fondo libera la memoria occupata da ogni elemento della lista. Tuttavia questa operazione ha un costo di O(n), per cui il reference counting non permette di avere **tempi di attesa** “**bounded**”.

L’introduzione di due tipi diversi di puntatore può risolvere il problema dei puntatori circolari.

**Puntatori** **Strong**: ovvero un puntatore che incrementa o decrementa il counter.

**Puntatori** **Weak**: ovvero un puntatore che *non* incrementa o decrementa il counter.

Può succedere che un puntatore debole si trovi a puntare a un oggetto già deallocato. Avere solo puntatori weak renderebbe i counter sempre a 0 facendo si che la memoria venga reclamata.

> **Mark and sweep** (Erlang, Haskell, OCaml, ...)

Bisogna stare attenti a distinguere cosa è un puntatore e cosa non lo è. Invece che avere un heap (area di memoria continua, utilizzata andando ad allocare i dati da essa, solitamente frammentata), si spezza in due aree: “*from space*” e “*to space*”. In ogni momento dell’esecuzione, tranne durante il garbage collection, il programma usa solo una delle due parti, quindi il 50% della memoria è inutilizzato.

Per *allocare* i dati ad esempio nella “from space”, essendo contigua, può essere gestita come uno stack, quindi non si fanno ricerche di un “buco” di dimensione corretta, ma si aggiunge in cima allo stack e si restituisce il puntatore.

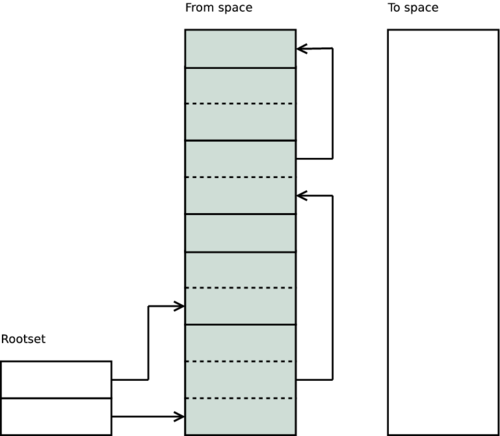
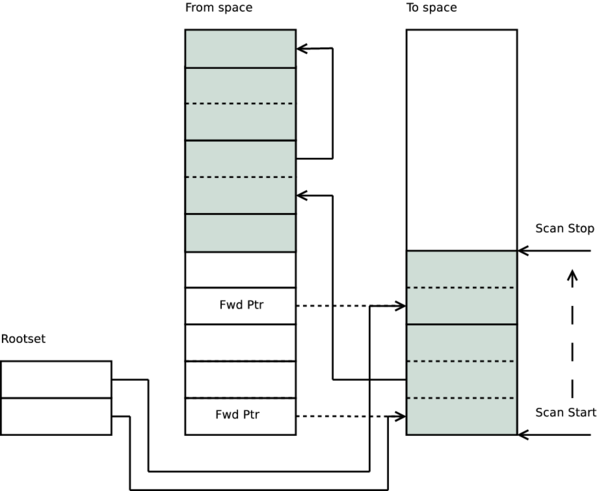
Non esiste alcuna operazione di *deallocazione*. Semplicemente se a un certo punto un dato è cancellato, viene cancellato il puntatore ma non il contenuto nell’area di memoria puntata.

L’algoritmo viene lanciato di tanto in tanto, una possibilità è eseguirlo a intervalli di tempo regolari, anche se solitamente si usa come criterio quanto from space rimane, quando lo spazio rimasto è troppo poco viene lanciato l’algoritmo.

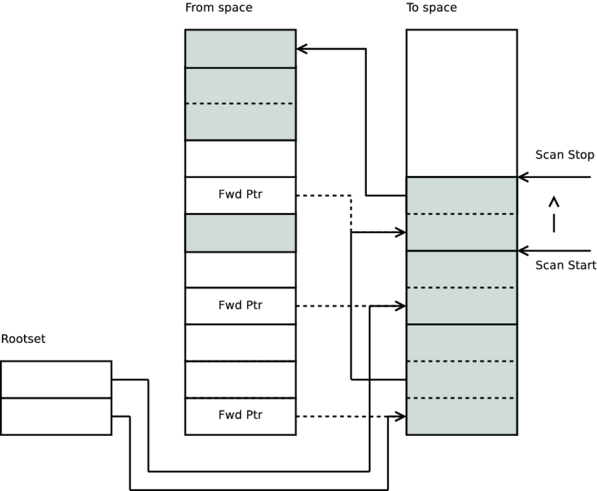
Una radice (**rootset**) è un dato non nell’heap che contiene un puntatore a un dato nello heap, Questi dati sono essenzialmente due: i *registri* e lo *stack delle chiamate*. Lo stack delle chiamate gestito solitamente in parte dal processore e in parte dal linguaggio, vengono messi record sullo stack con una politica FIFO. Il record contiene i parametri input, e le variabili locali della funzione e il return address della funzione. Quando viene eseguita la return della funzione, viene fatto pop della funzione e si torna ad eseguire la funzione precedente.

La fase di **mark** funziona che per ogni registro che contiene un puntatore e per ogni elemento dello stack che contiene un puntatore si segue il puntatore e si marca ogni dato come “vivo”, e questo si ripete ricorsivamente anche per gli elementi puntati a loro volta.

La fase di **sweep** si prendono le celle marcate come “vive” e trasferirle nel “to space” (solitamente viene fatto automaticamente, invece che marcare e poi spostare). In questo modo il “to space” conterrà dati consecutivi e utilizzati effettivamente. Nota: anche i puntatori del rootset vengono cambiati con le celle del “to space” e le celle del “from space” già spostate nel “to space” vengono marcate in modo che in presenza di puntatori all’interno del “from space” che puntano ancora alle celle già spostate “capiscano” che il valore è stato già trasferito.

Successivamente, dopo essersi occupati del rootset, si effettua un’iterazione da “scan start” a “scan stop” per determinare quali altre celle del “from space” sono vive perché puntate da qualcuno (già stato messo nel to space). Quando vengono trovate si spostano anch’esse nel “to space”. Questa operazione viene poi rieseguita fino a quando non si hanno più elementi puntati dal “to space” al “from space”.



La semantica dell’operatore *minore-uguale*, ad esempio, viene rotta se si usa un garbage collector mark and sweep. Ad esempio in un BST le *chiavi* vengono ordinate con una relazione d’ordine, tipicamente si usano come chiavi i puntatori alle stringhe e la ricerca di un dato si effettua con il minore uguale dei puntatori. Un altro esempio sono le *hashtable*, in cui se cambia l’indirizzo di un puntatore a run-time cambia il suo hash e quindi la tabella diventa inutilizzabile. In generale quindi tutti i linguaggi che implementano mark and sweep non hanno aritmetica dei puntatori.

La **complessità computazionale** dell’algoritmo di mark and sweep è **lineare** alla memoria utilizzata dal programma (numero di celle vive).

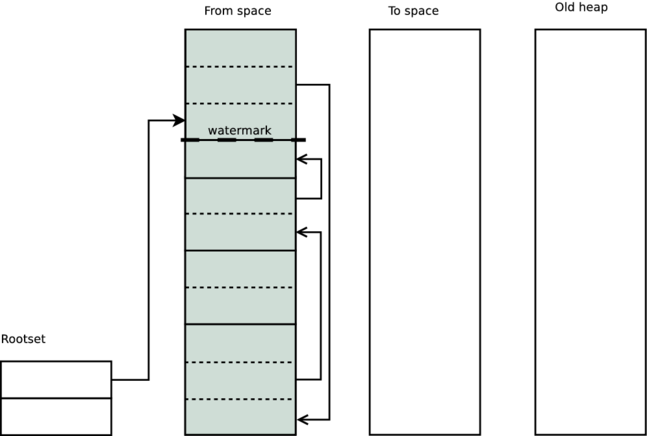
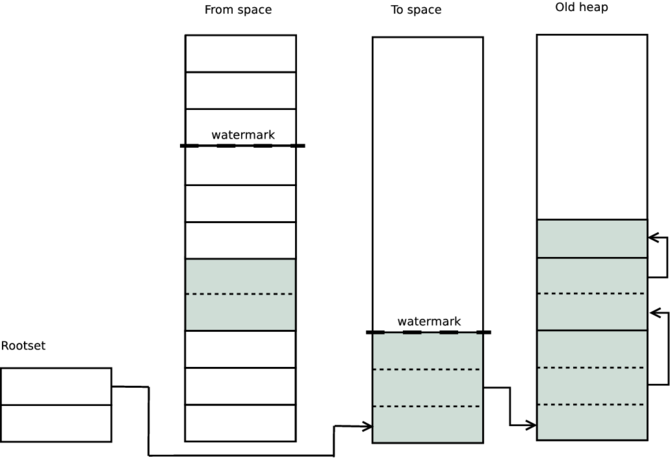
**Ottimizzazione dell’algoritmo di Mark and Sweep**

Empiricamente un dato di solito o viene allocato all’inizio e dura per tanto tempo, oppure viene allocato e rimosso immediatamente. **Ipotesi generazionale**, ho tantissimi dati che muoiono prestissimo o dopo molto molto tempo e pochi dati che invece rimangono vivi per un tempo intermedio.

Concretamente invece che dividere lo spazio di memoria in due si divide in tre: “from space”, “to space” e “old heap”. Globalmente verrà quindi “sprecata” più memoria. Grossomodo viene eseguito lo stesso algoritmo di prima muovendo i dati dal “from space” al “to space”, e in più viene mantenuto un indicatore chiamato “high-water mark” (intuitivamente il segno dove è arrivata l’alta marea), ovvero dove è arrivata l’ultima passata dell’algoritmo mark and sweep. Quando l’algoritmo viene rieseguito ci saranno nuovi dati da spostare dal from al to, quelli sotto all’high-watermark sono già “vivi” da abbastanza tempo (da più passate) e si assume che probabilmente questi continueranno ad essere vivi. Per questo questi dati vengono spostati non nel “to space” ma nell’”old heap”.

Nel old heap si mettono i dati che hanno lunga vita.

Anche l’old heap può riempirsi, in questo caso viene effettuato un mark and sweep anche dell’old heap cercando di recuperare spazio dai dati “morti”. Questa si chiama “major collection” a differenza di quella più frequente eseguita nel from space che chiamiamo “minor collection”.

La **complessità computazione** a questo punto continua ad essere **lineare** ma solo in base a quelli presenti al “**from space**” ovvero quelli giovani. Tendenzialmente questi dati sono molto meno frequenti.

Un *problema* può sorgere se ci fosse un puntatore dall’old heap al from space, quest’ultima cella non verrebbe riconosciuta come viva e non verrebbe copiata nel to heap. L’algoritmo parte quindi dal presupposto che non ci siano puntatori di questo tipo. Le caratteristiche di un linguaggio che mi garantiscono che non ci siano questo tipo di puntatori è l’**immutabilità dei dati**. In quanto i dati contenuti nel from space sono sempre più nuovi di quelli contenuti dall’old heap. Quindi un dato vecchio non poteva puntare a un dato nuovo (sopra al watermark) in quanto il nuovo non esisteva ancora quando il vecchio era nel from space, e non essendoci mutabilità non verrà mai rotta questa condizione.

In caso di linguaggi con mutabilità (esempio Ocaml) è possibile che dati nell’old heap puntino a dati nel from space, in questo caso il codice per mutare un dato prevede si controlli se il puntatore sta nell’old heap, in caso positivo si controlla se sta puntando a qualcosa nel from space, in tal caso il puntatore nell’old heap viene spostato all’interno del rootset (non spostato fisicamente, ma puntato dal rootset).

Se un dato del old heap punta a un dato del from\_space che a sua volta punta a un dato del old\_heap, il dato del from\_space viene sostituto con un *forward pointer*.

**Il costrutto “case”**

case List of

[X | Xs] when X >= 0 -> X + f(Xs);

[\_X | Xs] -> f(Xs);

[] -> 0;

\_ -> throw(error)

end

Switch booleano:

case Bool of

true -> …

false -> …

end