

Emerging Programming Paradigms

Michele Dinelli

michele.dinelli5@studio.unibo.it

Dept. of Computer Science and Engineering, University of Bologna

Indice

1.	Programmazione ad attori	3
2.	Erlang	3
2.1.	Design Decisions	3
2.2.	Tipi di Dato	5
2.3.	Rappresentazione dei Dati	8
2.3.1.	Monomorfizzazione	9
2.3.2.	Alla C	10
2.3.3.	Rappresentazione Uniforme dei Dati	10
2.4.	Gestione della memoria	12
2.5.	Implementazione del Pattern Matching	15
2.5.1.	Ottimizzazioni nella Compilazione del Pattern Matching	18
2.6.	Ricorsione	18
3.	The Next Big Emerging Constructs: Algebraic Effects	24
4.	Gestione della memoria	26
4.1.	Reference Counting	27
4.2.	Mark & Sweep	30
4.3.	Ipotesi Generazionale	33
4.4.	Mark & Sweep Generazionale	33
5.	Programmazione Parallela in Erlang	35
6.	Hot Code Swap in Erlang	40
7.	Failure Handling in Erlang	42
8.	Rust	47
8.1.	Gestione della memoria	48
8.1.1.	Ownership	48
8.1.2.	Borrowing	49
8.1.3.	Lifetime	50
8.2.	Slices	52
8.3.	Chiusure	53
8.4.	Smart Pointers	53
9.	Generalized Algebraic Data Type (GADT)	57
10.	Storia dei Linguaggi	63
11.	Entità di Prima Classe e Chiusure	66
11.1.	Sulla Laziness di Haskell	71
11.2.	Chiusure in Java	72
12.	Classi, Interfacce e Traits	72
12.1.	Linguaggi Object-Based	72
12.2.	Linguaggi Class-Based	73
12.3.	Ereditarietà e Polimorfismo	75
12.4.	Interfacce	77

12.5. Go	77
12.5.1. Interfacce in Go	79
12.5.2. Ereditarietà di Interfacce	80
12.5.3. Composizione di Strutture	81
12.6. Rust	81
12.6.1. Binding Statico e Dinamico	83
13. Type Classes	84
13.1. Classi su Costruttori di Tipo	89
13.2. Interpretazione delle Classi di Tipo	93
13.2.1. Overloading	93
13.3. Implementazione delle Classi di Tipo	93
14. Monadi	95
14.1. Le 3 Leggi di una Monade	101
14.2. La classe Monad	103
14.3. Monadi Composte	103
Bibliografia	108

1. Programmazione ad attori

La prima proposta dei linguaggi ad attori si ebbe con Hewitt, Bishop, Steiger, “A Universal Modular Actor Formalism for Artificial Intelligence” [1] nel 1973. Esistono numerosi linguaggi accademici che utilizzano il paradigma ad attori mentre nei linguaggi “mainstream” i più conosciuti sono Erlang e Elixir (implementato sulla Virtual Machine di Erlang (BEAM))

Un attore è composto da 3 elementi:

- Process IDentifier (PID)¹
- Mailbox: una coda di messaggi
- Behaviour: una mappa messaggi in lista di azioni + un nuovo behaviour
- Azione: computazione interna + invio asincrono di messaggi verso un PID + creazione di attori

Un attore ha associato un solo thread e diversi attori non condividono stato o memoria, sono isolati “if you were an actor in Erlang’s world, you would be a lonely person, sitting in a dark room with no window, waiting by your mailbox to get a message”. La programmazione ad attori può essere un esempio di programmazione event/driven ovvero computazione che avviene solamente in risposta ad un messaggio. La programmazione ad attori predilige situazioni con moltissimi attori ma dove pochi sono attivi simultaneamente (e.g thread system di Linux). Un sistema ad attori è un sistema composto da più attori in esecuzione. L’esecuzione è indipendente dalla locazione fisica dagli attori, più attori possono vivere sullo stesso nodo ad esempio utilizzando macchine virtuali.

2. Erlang

Erlang nasce come linguaggio di programmazione logico ispirandosi al Prolog e fu sviluppato dall’azienda Ericsson per implementare software di telecomunicazione [2]. Si tratta di un linguaggio di programmazione con costrutti sia sequenziali che distribuiti, ma il suo scopo principale è quello di essere un Concurrent Functional Programming Language, quindi un linguaggio iper-concorrente, funzionale e soprattutto adatto ai contesti distribuiti. Viene chiamato il linguaggio dei sei 9 infatti gli AXD301 della Ericsson (azienda che lo sviluppò) usavano Erlang e rimasero attivi per decenni garantendo availability per il 99.999999% del tempo. Erlang infatti è uno dei pochi linguaggi che permette l’hot swap del codice in produzione. Se consideriamo Erlang e solo i suoi costrutti sequenziali, siamo di fronte a un linguaggio di programmazione funzionale puro. Erlang, diversamente alla maggior parte dei linguaggi funzionali, non è tipato ed è più di basso livello, possiamo pensarlo come l’Assembly dei linguaggi di programmazione funzionali. Nasce già pensato per la concorrenza, infatti è l’unico linguaggio di programmazione funzionale che ha introdotto il supporto multicore senza modificare l’implementazione del linguaggio stesso. Tutte le variabili sono costanti ovvero variabili immutabili.

Elixir è un altro linguaggio ad attori implementato su BEAM. La sintassi di Elixir è ispirata da Ruby. Rispetto ad Erlang ha un livello di metaprogrammazione/macro igieniche e permette forme di poliformismo (simili alle interfacce).

2.1. Design Decisions

- Assenza di memoria condivisa, lock e mutexes. Questo perché la gestione sarebbe troppo complessa e non adatta a scenari distribuiti. Pessimo per fault tolerance (pensiamo al caso in cui un attore muoia mentre conserva il lock su una risorsa, questo causerebbe deadlock).

¹Il PID è un esempio di nome logico. Un nome fisico è ad esempio la coppia IP + Socket. L’uso di nomi fisici rappresenta un sistema rigido, infatti serve conoscere la topologia della rete. Erlang utilizza nomi logici, non serve conoscere la topologia della rete. Un nome logico permette di contattare un attore. La topologia della rete può modificarsi costantemente.

- Message passing asincrono con ricezione out-of-order. L'ordine nel quale i messaggi vengono processati è casuale in uno scenario distribuito. Su un messaggio viene estratto dalla mailbox e non può essere processato in un determinato istante, allora deve essere processato esplicitamente più tardi.
- Let it fail. Dato che la gestione degli errori è complessa in uno scenario distribuito (pensiamo ad esempio a processi remoti che possono essere irraggiungibili, o alle situazioni di deadlock se alcuni messaggi vengono persi) si preferisce uccidere direttamente un processo e tutti coloro ad esso connessi. Un supervisore si occupa di far ripartire i processi terminati. Erlang gestisce la soluzione dei problemi in maniera gerarchia. Un attore padre spawna dei figli che collaborano per computare. Il padre rimane in stato quiescente e osserva i figli. Se i figli falliscono li re-spawn. Dopo un certo numero di volte che i figli crashano il padre si suicida, e il nonno re-spawn il padre. Creare e distruggere attori deve essere un'operazione molto efficiente.

Erlang è open source (lo è dal 1998). Gira su una virtual machine chiamata BEAM che a sua volta gira sull'host come unico kernel process multi-threaded (quando la BEAM viene avviata, il sistema operativo vede un solo processo kernel process). La BEAM può avviare 1 Kernel Thread² per core. Gli attori (noti anche come language thread) vengono schedulati sui kernel thread con automatic load balancing. Il context switch per i language thread estremamente light-weight (context switch preemptive vs context switch collaborativo). Erlang è un linguaggio interpretato, infatti è l'interprete che ogni tanto cambia il controllo, quindi il context switch è collaborativo ma non è il programmatore che deve rilasciare il cpu-time, è direttamente l'interprete. Il memory footprint dei language thread è lightweight (300 word³). Gli attori Erlang partono senza uno stack pre-allocato, viene fatto crescere dinamicamente durante l'esecuzione. Copiare lo stack e raddoppiare l'allocazione risulta in un costo ammortizzato costante ($O(1)$).

In Erlang, una "word" si riferisce alla dimensione nativa della word dell'architettura CPU sottostante su cui è in esecuzione la BEAM (Erlang Virtual Machine). Su un sistema a 64 bit, una word corrisponde a 8 byte. Su un sistema a 32 bit, una word è di 4 byte. Poiché i sistemi moderni sono prevalentemente a 64 bit, assumeremo un'architettura a 64 bit per il calcolo, il che rende una word equivalente a 8 byte. Così, un processo Erlang appena creato occupa inizialmente circa 2.55 KB di memoria. Questa occupazione di memoria iniziale include:

- Process Control Block (PCB): Le strutture dati che la VM di Erlang utilizza per gestire il processo stesso, come il suo PID, lo stato, i link, i monitor, le informazioni sul timer, il nome registrato (se presente) e i puntatori alla coda dei messaggi.
- Stack Iniziale: Uno stack piccolo e pre-allocato per le chiamate di funzione.
- Heap Iniziale: Un piccolo heap pre-allocato per la memorizzazione dei dati iniziali. Anche un processo appena creato ha bisogno di spazio per contenere i suoi argomenti iniziali e l'indirizzo di ritorno per la prima chiamata di funzione.
- Mailbox: La struttura iniziale per la sua coda di messaggi, anche se vuota.


Nello snippet **Codice 1** viene mostrato un Hello World! in Erlang. Si vuole gestire un conto bancario mettendo a disposizione 4 operazioni attraverso pattern matching: la prima per effettuare una stampa di debug, la seconda per immettere denaro nel conto bancario, la terza per ottenere il saldo del conto bancario attuale e l'ultima per terminare. `print`, `put`, `get`, `exit`, `ok` sono atomi, ovvero nomi riservati che in memoria avranno una rappresentazione binaria. In Erlang gli

²A kernel thread is a kernel task running only in kernel mode; it usually has not been created by `fork()` or `clone()` system calls. An example is `kworker` or `kswapd`

³A word is any processor design's natural unit of data. A word is a fixed-sized datum handled as a unit by the instruction set or the hardware of the processor. The number of bits or digits[a] in a word (the word size, word width, or word length) is an important characteristic of any specific processor design or computer architecture.

atomi hanno una proprietà molto importante: dato un atomo esso sarà sempre uguale a sè stesso e sempre diverso da tutti gli altri atomi. Il metodo `cc(Bal)` deve essere richiamato ricorsivamente così da riassegnare un behavior all'attore. Quando la funzione `cc(Bal)` viene invocata si controlla l'input e quale pattern viene matchato, eseguendo il codice corrispettivo. Notare come l'istruzione `{get, PID} -> PID ! Bal, cc(Bal) ;` necessita del PID, questo perché gli attori non condividono memoria e le variabili sono immutabili quindi serve richiedere all'attore corrente di effettuare bang ! della variabile `Bal`.


```
1 -module(hello).
2 -export([cc/1]).
3
4 cc(Bal) ->
5   receive
6     print ->
7       io:format("Il balance è ~p ~n", [Bal]),
8       cc(Bal) ;
9     {put, N} -> cc(Bal+N) ;
10    {get, PID} -> PID ! Bal, cc(Bal) ;
11    exit -> ok
12  end.
```

 Erlang

Codice 1: Hello World! in Erlang

Per utilizzarlo possiamo inizializzare un attore con la funzione `spawn(Mod, Fun, Args)`, questo restituisce il suo PID. A questo punto possiamo interagire con l'attore creato utilizzando ! (bang) ovvero il simbolo per mandare un messaggio [\[3\]](#).

```
1 1> Actor = spawn(hello, cc, [0]).
2 <0.86.0>
3 2> Actor ! {put, 50}.
4 {put,50}
5 3> Actor ! print.
6 Il balance è 50
7 print
8 4> Actor ! {get, self()}.
9 5> receive Bal -> io:format("Received balance: ~p~n", [Bal]) end.
10 Received balance: 50
11 ok
```

 Bash

`self()` è una funzione che ritorna il PID dell'attore corrente.

2.2. Tipi di Dato


Tipi di dato possono essere atomici, non atomici, predefiniti o definiti dall'utente. Erlang non supporta tipi di dato user-defined essendo Erlang non tipato (tipato dinamicamente). L'utente può creare nuovi tipi componendo i termini di Erlang.

- Atomici ovvero tipi di dato che non ne contengono altri
 - o Numeri (Interi, Float) con operazioni (+, -, <, >, >=, <=)
 - o Booleani (:=, /=).

- PID con `self()`.
- Reference con `make_ref()`. Usata ad esempio per generare i nounces negli attacchi man-in-the-middle.
- Porte, come i PID per contattare attori.
- Atomi che vengono rappresentati a run-time in memoria con sequenze di bit. Atomi diversi avranno sequenze di bit differenti. È possibile definire atomi usando più parole racchiudendole in apici singoli `'ciao come stai'`.
- Non atomici
 - Tuple ad esempio `{4, {ciao, 2.0}, true}`. Esiste la tupla vuota `{}`.
 - Liste come `[1 | [2 | [3 | []]]]`. Le liste hanno una testa e una coda. La testa e la coda sono separate dal simbolo `|`. Possono esistere liste improprie, ovvero liste senza coda. Questo accade perché il linguaggio non è tipato staticamente e non è garantito che la coda di una lista sia una lista. È possibile usare la funzione `length` per ottenere la lunghezza. L'operatore `++` unisce due liste `[2,3] ++ [4, 5]`. mentre `--` rimuove gli elementi specificati dalla lista `[2,3,4,5] -- [4, 5]`. con attenzione perché `--` è associativo a destra.

Non esistono le stringhe in Erlang ma sono rappresentate come liste di caratteri. Ma non esistono neanche i caratteri e quindi?

```
1 [97,98,99].
2 % "abc"
3
4 [97,98,99,-3].
5 % [97,98,99,-3]
6
7 [97] == "a".
8 % true
```

 Erlang


Codice 2: Gestione delle stringhe in Erlang

Quando Erlang fa pretty printing di una lista controlla se tutti i valori sono nel range ASCII, in tal caso li converte... al contrario se un numero sfora il range ASCII torniamo a vedere una lista normale.

L'operatore `==` collassa due numeri alla stessa rappresentazione, quindi `4.0 == 4. %false` mentre `4.0 == 4. %true`.

In Erlang le funzione sono oggetti di prima classe, possiamo passarle in input ad altre funzioni, aggiungerle a delle lista ecc... sono a tutti gli effetti tipi di dato. La keyword `fun` definisce una funzione. Possiamo dichiarare delle funzioni anonime ma per utilizzarle in chiamate ricorsive possiamo definire un nome interno che non viene esportato come mostrato nell'esempio [Codice 3](#).


```
1 fun G (N) -> N * G(N) end.
```

 Erlang

Codice 3: Una funzione anonima con nome interno (G) usato per la ricorsione

Vedremo che una funzione che all'interno presenta un'altra funzione che usa la variabile esterna (definita però nella funzione esterna) a run-time ha una rappresentazione particolare, parleremo di chiusure.

```
1 1> G = fun(X) -> fun(Y) -> Y + 2 end end.
2 % #Fun<erl_eval.42.113135111>
3 2> H = G(3).
```

 Shell

```

4 % #Fun<erl_eval.42.113135111>
5 3> H(5).
6 % 7

```

È possibile utilizzare delle guardie con la keyword `when`. Erlang vuole assicurarsi assolutamente che la guardia che viene definita rispetti un insieme di funzioni predefinite, non è possibile inserire codice arbitrario all'interno di una guardia. Questo per prevenire che il codice di una guardia spawni nuovi attori o invii messaggi. Se la creazione delle guardie avesse side-effects renderebbe molto complesso costruire macchine a stati finiti per combinare i risultati del pattern matching.

```

1 F = fun ({N, 2}) when N >= 0 -> N ; ({X, Y}) -> X + Y.
    % F({-1, 2}) la prima guardia scatta ma non serve che il secondo pattern si
2 assicurati che l'input è una tupla di due elementi dato che lo abbiamo già
    controllato

```

Inserire side-effects nel codice delle guardie renderebbe complicate queste ottimizzazioni. In Erlang le guardie sono un linguaggio molto poco espressivo.

```

1 fun ({N, 2}) when N >= 0 -> N ;
2   ({ciao, N, M}) -> N + M ;
3   ([_, _, {X, Y}]) -> X + Y
4 end.

```

Erlang permette la list comprehension, ad esempio in teoria degli insiem scriviamo $\{x^2 \mid x \leq 10\}$. In Erlang possiamo scrivere

```

1 [ {X, Y + 1} || X <- [1,2,3], {Y, _} <- [{4,5}, {6,7}], X + Y <
2 6 ].
3 # [{1,5}]

```

Erlang permette anche di accedere direttamente alla rappresentazione binaria utilizzando `<< >>`. L'operatore base `#` number rappresenta un numero in una certa base.

```

1 N = 16#7A5.
2 %1957
3 <<16#7A5>> % voglio accedere alla rappresentazione in bit
4 << R:4, G:4, B:4 >> = <<N:12>> # R. 7, G. 10, B. 5


```

Nel mondo funzionale puro non abbiamo mutabilità, tutti gli algoritmi che conosciamo devono essere modificati. Ad esempio aggiungere un nodo ad un albero binario di ricerca richiede la copia dell'intero albero aggiungendo il nuovo nodo. Questo pare computazionalmente ingestibile ma i linguaggi funzionali permettono con tecniche ad-hoc di implementare tutti gli algoritmi che conosciamo. Con l'esempio presente in **Codice 4** introduciamo i tipi di dato algebrici. Un tipo di dato algebrico può avere un certo numero (finito) di forme e ogni forma è un tipo di dato composto. Definire un tipo di dato algebrico significa comunicare al compilatore le forme che vogliamo utilizzare. In Erlang questo è inutile dato che l'interprete ignora i tipi però è uno sforzo mentale che vale la pena fare.

```

1  -module(dict).
2  -export([search/2, insert/3]).
3
4  % Alberi binari di ricerca con chiavi nei nodi interni e coppie
5  % chiavi-payload nelle foglie
6  %
7  % Esempio: Algebraic Data Type
8  % Tipo tree:
9  %   Tree K V ::= { leaf, K, V } | { node, Tree K V, K, Tree K V }
10
11 % Esempio:
12 %   { node, { leaf, 2, anna }, 4, { leaf, 5, bruno } }
13 %           4
14 %         /  \
15 %   {2,anna} {5,bruno}
16
17 % Tipo option:
18 % Option V ::= not_found | { found, V }
19
20 % Cerco un valore associato a una chiave in un albero binario di ricerca
21 search(K, {leaf, K2, V}) when K == K2 -> { found, V } ;
22 search(_, {leaf, _, _}) -> not_found ;
23 search(K, {node, T1, K2, _}) when K <= K2 -> search(K, T1) ;
24 search(K, {node, _, _, T2}) -> search(K, T2).
25
26 insert(K, V, {leaf, K2, _}) when K == K2 -> { leaf, K, V } ;
27 insert(K, V, {leaf, K2, V2}) when K <= K2 ->
28   { node, { leaf, K, V }, K, { leaf, K2, V2 } } ;
29 insert(K, V, {leaf, K2, V2}) ->
30   { node, { leaf, K2, V2 }, K2, { leaf, K, V } } ;
31 insert(K, V, {node, T1, K2, T2}) when K <= K2 ->
32   { node, insert(K, V, T1), K2, T2 } ;
33 insert(K, V, {node, T1, K2, T2}) ->
34   { node, T1, K2, insert(K, V, T2)}.

```

 Erlang

Codice 4: Implementazione di un BST in Erlang

Ci chiediamo come viene gestita la memoria, come vengono gestite le rappresentazioni dei tipi a run-time, e qual è il costo computazionale in spazio e tempo del pattern matching e delle operazioni che abbiamo definito. Serve anche capire la gestione della ricorsione quanto costa in termini di spazio.

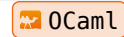
2.3. Rappresentazione dei Dati

Tutti i dati sono sequenzi di bit (bytes) o sequenze di word. La CPU manipola i bit ma non attribuisce un significato alle sequenze di bit, è il programmatore che quando manipola una sequenza

di bit attribuisce un significato alla sequenza. 97 può essere sia un numero intero che un carattere ASCII (come visto in [Codice 2](#)).

Il sistema di tipi è un'analisi modulare statica a compile-time che controlla che l'interpretazione dei bit in memoria sia coerente con quella che viene esplicitata dal programmatore. Le funzioni monomorfe sono funzioni che lavorano correttamente solo se il tipo in input è quello atteso dalla funzione.

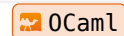
```
1 let f n s = n+4, s ^ "!";;  
2 (* val f : int -> string -> int * string = <fun> *)
```



Ad esempio in OCaml se scriviamo una funzione che accetta due input `n` e `s` dove al primo input sommiamo 4 e al secondo concateniamo `!"` otteniamo un controllo statico che garantisce che la funzione accetta due parametri dove il primo è un intero e il secondo è una stringa. Questo è un esempio di funzione monomorfa.

Molte operazioni hanno senso in base alla rappresentazione del dato, ci sono alcune operazioni le quali dal punto di vista logico possono essere implementate su qualunque tipo di dato. Ad esempio allocare o deallocare un dato, muovere un dato o copiare un dato. In un linguaggio tipato se implementiamo una funzione che esegue una di queste operazioni allora la funzione sarà polimorfa.

```
1 let swap (x, y) = (y, x);;  
2 (* val swap : 'a * 'b -> 'b * 'a = <fun> *)
```



Codice 5: Una funzione che scambia due variabili in OCaml

`'a` e `'b` sono elementi di una coppia, il linguaggio in questo caso non si esprime sul loro tipo e diviene polimorfa. Questa forma di polimorfismo prende il nome di polimorfismo generico, polimorfismo uniforme o template.

Se un linguaggio di programmazione è non tipato non abbiamo un controllo a priori delle rappresentazioni. Di fatto i linguaggi di programmazione non tipati si comportano come se tutti i costrutti fossero polimorfi, quindi per implementare un linguaggio che supporta polimorfismo ha le stesse problematiche dell'implementazione di un linguaggio non tipato. Per implementare le operazioni di cui sopra non serve sapere come interpretare i bit ma è necessario sapere quanto è lungo il dato per poter operare su esso (pensiamo a scambiare due dati, ci serve conoscere la loro dimensione per allocare lo spazio necessario). Tutte le operazioni aritmetiche/logiche necessitano la conoscenza della lunghezza del dato su cui operano. In origine le nozioni di tipi sono due: una fa riferimento a definire quanto sono grandi i dati, la seconda fa riferimento a proprietà di interpretazione dei dati, nei linguaggi moderni queste due nozioni sono confluite.

Come possiamo implementare funzioni polimorfe se i tipi di dato possono avere dimensioni differenti? vediamo tre tecniche.

2.3.1. Monomorfizzazione


Utilizzata da C++ e Rust. Presenta dei vincoli di implementazione:

- È a disposizione solo quando il linguaggio di programmazione è tipato.
- Dato un programma è possibile calcolare un insieme finito di tipi su cui ogni funzione lavorerà. Vediamo un programma che non rispetta b).

```

1 f(0, T) -> { leaf, T } ;
2 f(N, T) -> { node, f(N-1, { T, T }), T, f(N-1, { T, T }) } ;

```

 Erlang

Codice 6: Esempio di polymorphic recursion. Ad ogni chiamata ricorsiva il tipo di T cambia.

Nel [Codice 6](#) il tipo T cambia ad ogni chiamata ricorsiva quindi non riusciamo staticamente a definire un insieme finito di tipi per la funzione f. L'esempio è in Erlang non perché sia tipato ma solamente per fare un esempio di quando può accadere.

Supponiamo di avere un linguaggio tipato che non supporta polymorphic recursion, allora l'implementazione sarà molto semplice: la funzione polimorfa viene compilata una volta per ogni combinazione di tipi sulla quale verrà usata (C++, Rust...). Questo implica dei vantaggi:

- + Non pone vincoli sulla rappresentazione dei dati. Dato che il codice compilato viene compilato sempre su dati concreti. Ogni implementazione sarà nota per il caso specifico del dato.
- + Scattano ottimizzazioni ad-hoc sui dati. Se la funzione viene compilata più volte per dati diversi allora il compilatore attiva ottimizzazioni particolari caso per caso.
- + Complessivamente è una tecnica che porta efficienza.

Gli svantaggi sono:

- ! Implementazione limitata ai vincoli.
- ! Tempi di compilazione aumentati.
- ! Dimensione dell'eseguibile aumentata. Tipicamente dato che una funzione avrà più rappresentazioni in base ai tipi di dato che riceve in input ci saranno più funzioni. Per distinguerle il compilatore attua il **mangling** (`fun_name_string_int`) ovvero nominare la funzione a partire dai tipi. La tabella dei simboli può diventare molto grande. Questo è un problema per i sistemi di caching, potrebbe darsi che la cache di I e II livello non siano abbastanza grandi e si è costretti a caricare/scaricare codice continuamente.

2.3.2. Alla C

Rust fa la stessa cosa in casi residuali. L'idea è che l'unica cosa che ci importa per realizzare funzioni polimorfe è la dimensione del dato, conoscendola potremmo generare codice uniforme che alloca, dealloca, sposta, e copia il dato. Possiamo ottenere il tutto con un semplice loop. Questo è meno efficiente della monomorfizzazione. In C per dichiarare una funzione polimorfa basta che un certo dato in input/output viene acceduto tramite un puntatore. Il puntatore viene dichiarato `void*` che significa dichiarare un puntatore di cui si ignora la dimensione di ciò a cui sta puntando. La funzione riceve sempre in input coppie puntatore e dimensione del dato (`void*`, `size_t`). Un esempio delle funzioni polimorfe in C sono le funzioni di ordinamento (`qsort` di C). Questo approccio ha alcuni svantaggi:

- ! A run-time bisogna preservare e passare la dimensione dei dati.
- ! (In C) necessario aiuto da parte del programmatore (serve passare la dimensione dei dati).
- ! Inefficiente perché il codice conterrà cicli sulle dimensioni dei dati.

2.3.3. Rappresentazione Uniforme dei Dati

Questa tecnica è usata da tutti i linguaggi che non usano monomorfizzazione (Erlang, OCaml, Haskell, Java? ...). Il problema che impedisce di avere una sola variante del codice senza dover passare la dimensione del dato è il fatto che i dati possono avere dimensione diversa, se tutti i dati fossero lunghi uguali non ci sarebbero problemi. L'idea è di rappresentare tutti i dati con una word. La word è la dimensione necessaria per avere puntatori in memoria. In una word ci può stare sempre un puntatore. Se allora dobbiamo salvare tutti i dati in una word questo partiziona i dati in 3:

- Ci sono dati che già sono grandi una word.
- Ci sono dati più piccoli di una word, in quel caso si sprecano i rimanenti comunque allocando una word. In certi linguaggi sono chiamati value types, noi li chiameremo dati **unboxed**.
- Ci sono dati più grandi di una word, in quel caso si allocano sullo heap e si rappresentano con puntatori allo heap (che è esattamente una word). In certi linguaggi sono chiamati reference types, noi li chiameremo **boxed**.

I vantaggi sono:

+ Tempi di compilazione e dimensione dell'eseguibile ridotti.

Gli svantaggi sono:

! Introduce indirezioni e questo può portare a minore efficienza perché serve sempre passare attraverso a un puntatore. In C magari il dato sarebbe stato messo sullo Stack.

Se consideriamo **Codice 4** e il tipo di dato algebrico Tree K V come viene rappresentato?

Nell'heap ci metteremo una quadrupla (node, Tree, K, Tree),
 1 vorremmo rappresentarla con 4 word consecutive ma non è
 sempre possibile. Il dato T è rappresentato da
 2 T = { node, { leaf, 4, true }, 5, { leaf, 6, false } }.
 3
 4 Node è un atomo quindi sta in 2⁶⁴ bit (si spera)
 5 {leaf, 4, true} è un altro dato boxed
 6 leaf è un atomo quindi unboxed, così come 4 e true (1bit)
 7
 8 HEAP
 9 T -----> node [0]
 10 ----- [1]
 11 | 5 [2]
 12 | ---- [3]
 13 | | ...
 14 | | ...
 15 | --> leaf
 16 | 6
 17 | false
 18 | ...
 19 | ...
 20 -----> leaf
 21 4
 22 true

[Plain Text](#)

Vediamo un esempio per il tipo di dato lista.

1 % cons non vuoto, T testa e List T coda
 2 % List T ::= nil | { cons, T, List T }
 3 L = {cons, 0, {cons, 1, nil}} % [0 | [1 | []]]
 4
 5 HEAP
 6 L -----> cons [0]

[Plain Text](#)

7		0	[1]
8	-----		[2]
9			...
10			...
11	----->	cons	
12		1	
13		nil	

2.4. Gestione della memoria

La rappresentazione uniforme introduce un problema molto importante nei linguaggi con automatic Garbage Collection (GC). Serve essere in grado di distinguere puntatori e non puntatori, in modo tale da poter implementare tecniche di GC automatica che si basano sull'esplorazione di grafi (seguendo i puntatori). La stessa sequenza di bit potrebbe essere un puntatore oppure no. La soluzione naïve ma efficace è aggiungere un tag alle word, basta 1bit. Se una word è 64 bit è possibile usare 1bit come tag e i restanti 63 come payload (Erlang, OCaml, Haskell). Usiamo il bit più significativo (MSB) o il meno significativo (LSB)? se usiamo MSB perdiamo il 50% della memoria indirizzabile perché avremmo word del tipo $0xxx \dots xxx$ e non potremmo utilizzare metà della memoria indirizzabile per i puntatori (tutti gli indirizzi $1xxx \dots xxx$ non sarebbero validi). Viceversa, se utilizziamo LSB otteniamo indirizzi del tipo $xxx \dots xx0$, sprechiamo comunque alcune celle di memoria ma molte meno. Questo perché che tutti i puntatori devono essere indirizzi pari (il bit meno significativo = 0). Sprechiamo alcuni indirizzi per via dell'allineamento forzato.

Non è che si sprechino davvero celle di memoria fisica, ma piuttosto non si possono usare tutte le possibili word da 64 bit come indirizzi validi. In un sistema a 64 bit, gli oggetti in memoria (heap, stack, ecc...) sono tipicamente allineati ad almeno 8 byte questo significa che tutti gli indirizzi validi finiscono con almeno 3 zeri nei bit meno significativi ($2^3 = 8$). Ad esempio in OCaml si rappresenta 3 come $0b000 \dots 011$ (con $LSB = 1$). $0b000 \dots 011$ è un valore immediato e non un puntatore. Notare che per accedere alla terza word di un dato boxed puntato da p si accede come $*p+3$.

E se invece usassimo $xxx \dots xx1$, beh dato che il bus di sistema sposta i dati in blocchi allineati (2, 4 o 8 bytes) si accederebbe a dati non allineati in memoria. Gli indirizzi che terminano con 0 possono essere allineati mentre se terminano con 1 non possono essere allineati.

Ma i numeri come si rappresentano? dato che si usa il LSB per taggare le word il payload dei numeri sarà ad esempio $0000001 = 0$, $0000011 = 1$, $0000101 = 2$. Ma come si implementano le operazioni?

```
1 ; x + y = ADD 0000011 0000011 = 0000110
2 mov rax, 3    ; carica 3 nel registro RAX 0b00000011
3 mov rbx, 3    ; carica 3 nel registro RBX 0b00000011
4 add rax, rbx  ; RAX = RAX + RBX (3 + 3 = 6) 0b00000110
```

[Assembly \(x86_64\)](#)

Questo è 6 ma avevamo appena definito $0000011 = 1$ quindi ci aspettavamo 2.


Le moltiplicazioni saranno ancora più complicate e soprattutto più di quelle che sarebbero normalmente. Mai usare Erlang (o altri linguaggi con rappresentazione uniforme + GC) per calcoli high speed. Dunque gli svantaggi sono:

! Si sprecano delle word quando un dato allocato sullo heap termina in posizione pari pertanto il dato successivo deve iniziare due celle dopo.

! Implementazione costosa delle operazioni aritmetico/logiche. Non buono per number crunching.


! Interazione con altri linguaggi complessa (serve convertire sempre i numeri tra vari linguaggi). Al momento possiamo utilizzare le stesse sequenze di bit per rappresentare valori diversi di dati diversi (0 come numero o come il primo carattere ASCII ad esempio). Ci troviamo davanti però a un altro problema: se vogliamo distinguere valori diversi di tipi di dato diversi come fare? ossia domandare se 0 è uguale al primo carattere ASCII ha senso?

```
1 0 ::= []. % confrontiamo 0 e lista vuota
2 %false
```

 Erlang

Come visto in **Codice 2** Erlang fa fatica in questi casi. Se il linguaggio è tipato dipende dal linguaggio e dal sistema di tipi. In OCaml ad esempio il tipo dell'uguaglianza è - : 'a -> 'a -> bool = <fun> quindi la funzione si aspetta lo stesso tipo degli argomenti nonostante sia una funzione polimorfa.

```
1 type semi = Cuori | Quadri | Fiori | Picche;;
2 type stagioni = Autunno | Inverno;;
3
4 Cuori = Autunno;;
5 (* Error: qui OCaml dato che Autunno e Fiori sono due tipi diversi può
   usare la stessa rappresentazione di sequenze di bit i.e., 0000...0 *)
6
7 (* Se disabilitiamo il sistema di tipi di OCaml otteniamo: *)
8 Cuori = Obj.magic Autunno;;
9 (* true *)
10 (* Sotto Cuori c'è la stessa sequenza di bit di Autunno *)
11
12 Cuori = 0;;
13 (* true *)
```

 OCaml

Codice 7: Esempio di atomi associati a un tipo in OCaml e confronti di rappresentazione

Possiamo concludere che se un linguaggio (come Erlang ad esempio) ammette tipi di dato diversi e ammette confronti tra valori di tipi di dato diversi allora non è possibile utilizzare le stesse sequenze di bit per due tipi di dato diversi. OCaml (senza bypassare il sistema di tipi) non permette il confronto quindi è tutto ok. Questo significa che a run-time serve essere in grado di distinguere diversi tipi di dato.

- Dati boxed: questo è sempre un puntatore di dato nello heap e il dato finisce nello heap. Vogliamo distinguere i diversi tipi di dato, quindi serve ancora un tag. Il dato è formato da word consecutive sullo heap e semplicemente si aggiunge una prima word che contiene un TAG per distinguere i dati. Il puntatore ora punta alla word con il TAG. La parola con il TAG contiene anche la dimensione del dato sull'heap per la GC.
- Dati unboxed: come abbiamo usato il bit meno significativo per definire un puntatore oppure no, continuiamo con questo approccio. Ad esempio xxx...xxxTTT dove un possibile convenzione potrebbe essere
 - xxx...xxx000 = puntatori
 - xxx...xxx001 = atomi

Così facendo però i puntatori avranno a disposizione 1/8 degli indirizzi. I TAG di lunghezza fissa non vanno bene. Riduce troppo il payload dei puntatori. Possiamo utilizzare codifica a lunghezza variabile (Huffman Coding) producendo TAG che non sono rispettivamente suffissi di altri TAG.

Ma come viene rappresentato internamente il payload? I numeri interi, floating point ecc... usano la rappresentazione standard, ad eccezione per l'ultimo bit che usiamo per il tag. Per i tipi di dati molto specifici e.g., PID e porte il compilatore si occupa di scegliere una sequenza di bit ma non è molto interessante essendo tipi specifici del linguaggio (in questo caso Erlang). Gli atomi invece sono particolarmente interessanti. Vogliamo sempre che due atomi distinti abbiano sequenze di bit diverse, non ci interessa ad esempio l'ordinamento. Ci sono alcuni linguaggi dove gli atomi non sono liberi ma sono vincolati ad essere dichiarati all'interno ad un tipo (ad esempio Algebraic Data Types in OCaml [Codice 7-1](#)) e gli atomi di ciascun tipo hanno una rappresentazione sequenziale (xxx...000, xxx...001). In questi linguaggi l'insieme degli atomi è noto a compile-time e ogni atomo appartiene a un solo tipo. In altri linguaggi l'insieme degli atomi è aperto, quindi gli atomi possono comparire senza essere dichiarati in due momenti: al momento del linking o a run-time.

a) Linking, ad esempio due librerie usano il proprio insieme di atomi, al momento del linking serve unire i due insiemi di atomi. Questo può accadere in Erlang, Prolog e OCaml [Codice 8](#). Quando due attori o due librerie usano due atomi e se li scambiano, se hanno usato lo stesso atomo non è detto che abbiano la stessa rappresentazione. Atomi con lo stesso nome usati da attori, moduli o librerie distinte vanno identificati. Durante la compilazione/interpretazione serve associare a ogni atomo una sequenza di bit in maniera tale sia possibile il linking/message-passing rispettando l'identificazione degli atomi.

– Per ogni atomo si usa il suo hash (sha256(pippo)).

- Possono esserci collisioni. Se avviene compilazione locale e due atomi collidono basta segnalare errore di compilazione al programmatore che modifica uno dei due atomi. Ma se qualcuno scrive una libreria e ha usato un atomo, qualcun'altro scrive una libreria con un altro atomo e queste vengono compilate separatamente e c'è una collisione, beh è un problema simpatico. In questo caso nel file oggetto (l'output compilazione separata, o l'input del linking) si scrivono associazioni atomo/hash e si verifica se esistono collisioni notificando eventualmente il programmatore. Se capita bisogna modificare il codice.
- Si associano sequenze di bit completamente sparse quindi se implementiamo dei cases l'unica implementazione possibile sono degli if. Il che non è particolarmente efficiente. Se potessimo scegliere sequenze di bit consecutive potremmo usare il costrutto switch del C che usa tecniche di indirizzione ovvero salta (mette il PC) all'indirizzo giusto direttamente (jump table).

– Usare sequenze di bit consecutive a mano a mano che vengono incontrati atomi. Ma in Erlang un altro nodo potrebbe vedere gli atomi in ordine inverso. Cosa fare in caso di linking o message passing? l'unica soluzione è tradurli con una tabella di traduzione che mantiene le traduzioni di due . Ogni volta che un messaggio contiene un atomo nuovo e un attore lo riceve la BEAM se sa che è la prima volta che lo vede trasmette anche l'associazione atomo/sequenza di bit. Si crea così mano a mano la tabella di traduzione.

! Richiede di mantenere diverse tabelle di traduzione.

! Richiede di tradurre ogni messaggio scambiato fra nodi diversi.

+ Funziona sempre.

- b) Run-time, in Erlang ad esempio un altro attore invia un messaggio contenente un atomo che l'attore che riceve il messaggio non conosce. Ad ogni messaggio l'insieme degli atomi noto può crescere.

```
1 (* Dichiarazione di un atomo in OCaml *)
2 # `Ciao;;
3 - : [> `Ciao ] = `Ciao
4 (* Definizione di una funzione che accetta un atomo `A o un atomo `B
   associato a un intero *)
5 # function `A -> 0 | `B x -> x + 1;;
6 - : [< `A | `B of int ] -> int = <fun>
```

OCaml

Codice 8: Dichiarazione di atomi in OCaml

2.5. Implementazione del Pattern Matching

Cerchiamo di capire quanto costa in termini di tempo e memoria il codice definito in [Codice 4](#).

```
1 % Rappresentazione di un albero con un ADT
2 T = {node, {leaf, 4, true}, 5, {leaf, 6, false}}
3
4 % Consideriamo una funzione che usa pattern matching profondo
5 % _ ignora l'input, il pattern può essere composto infatti abbiamo un
   pattern che è una tupla di tuple
6 % questo pattern riusa parte dell'input in output
7 f({_, T1, 5, {leaf, K, V}}) -> {node, T1, 7, {leaf, K, V}}
```

Erlang

Codice 9: Esempio di pattern in Erlang

In memoria T è boxed sarà un puntatore dello heap. T punta alla cella che contiene il tag (tipo di dato e dimensione del dato) di seguito node e {leaf, 4, true} è ancora boxed quindi sarà rappresentato da un altro puntatore. 5 è unboxed mentre {leaf, 6, false} è boxed. Vediamo come compilare il pattern [Codice 9-7](#)

```
1 [| ... |] è una funzione a tempo di compilazione (crea
   codice)
2 [| pattern, p |] dove - pattern è il pattern
3                       - p è il dato boxed o unboxed (una word)
4                       - [| ... |] è una funzione a tempo di compilazione
5
6 Per compilare underscore non produciamo codice
7 [| _, p |] =
8
9 Per compilare un pattern che matchi la variabile X si dichiara una word X
   uguale a p, se assegniamo a X un intero albero stiamo assegnando il
   puntatore all'albero
10 [| X, p |] = word X = p;          * X variabile nuova (**)
11
12 Per compilare un pattern che matchi una costante (ad esempio 5) il pattern
   può avere successo o meno a seconda che p sia uguale a k. La variabile k è
```

Plain Text

```

    sempre unboxed, se fosse boxed punterebbe nello heap e non troveremmo un
    tipo atomico.
13 [| k, p |] = if (p != k) return -1;    * k costante di un tipo atomico
14
    Per avere sharing anche nei pattern profondi possiamo assegnare una
    variabile al pattern, in questo caso il pattern viene compilato contro la
15 variabile p per entrambi i pattern. Tipicamente P2 deve essere un nome e
    non una variabile.
16 [| P1 = P2, p |] =
17   [| P1, p |]
18   [| P2, p |]
19
    Il pattern matcha solo se p è unboxed (guardiamo ultimo bit). Se è un
    puntatore guardiamo la prima cella dell'array che contiene il tag e la
    lunghezza del dato, se il dato è lungo n (come il numero di elementi della
20 tupla) allora andiamo in ricorsione. Compiliamo il pattern 1 contro il
    pattern p1 e il pattern Pn contro p[n].
21 [| {P1, ..., Pn}, p |] =
22   if(is_unboxed(p)) return -1;
23   if((*p)[0].length != n) return -1;
24   [| P1, p[1] |]
25   ...
26   [| Pn, p[n] |]
27
28 Appliciamolo al nostro esempio
29
30 T = {node, {leaf, 4, true}, 5, {leaf, 6, false}}
31 f({_, T1, 5, {leaf, K, V}}) -> {node, T1, 7, {leaf, K, V}}
32
33 if (is_unboxed(T)) return -1;
34 if((*p)[0].length != 4) return -1;
35 word T1 = T[2]
36 if (5 != T[3]) return -1;
37 if (is_unboxed(T[4])) return -1;
38 if (T[4][0].length != 3) return -1;
39 if (leaf != T[4][1]) return -1;
40 word K = T[4][2];
41 word V = T[4][3];
42
43           HEAP                                Return
44 T -----> 4      [0]                        4
45           node  [1]                        node
46           ----- [2]                        ----
47           |      5      [3]                    |    7
48           | ---- [4]                        |    -----

```



```

49      | |      ...      ...      |      ...      |
50      | |      ...      |      3 <----
51      | --> 3      |      leaf
52      |      leaf      |      6
53      |      6      = K      |      false
54      |      false      = V      |
55      |      ...      |
56      |      ...      |
57      ---> 3 <----- T1
58      leaf
59      4
60      true

```

Siamo lineari in tempo nella dimensione del pattern e lineari in spazio nella dimensione del pattern. Allochiamo solo 3 variabili, quelle contenute nel pattern. Vediamo il costo per produrre l'output. Notiamo che la parte del sotto-albero T1 è stata condivisa (sharing). Si parla di sharing quando una parte di struttura dati viene condivisa da due istanze. Notiamo anche una cosa non ottimale ovvero che il sotto-albero {leaf, 6, false} non è riutilizzato ma viene ri-allocato. Il costo in tempo e spazio per produrre l'output è sempre lineare nella dimensione dell'output.

- Costo computazionale del pattern-matching: $O(n)$ dove n è la lunghezza del pattern.
- Costo computazionale dell'allocazione del risultato: $O(n)$ dove n è la lunghezza dell'espressione usata in output.

In entrambi i casi però di fatto sono costi $O(1)$ a run-time perché n non è un parametro dell'input a run-time. Notiamo che ogni variabile che ha catturato un tipo di dato boxed ha generato **sharing**. Questo non è problematico sse il linguaggio di programmazione non ammette mutabilità (dello heap). In C ad esempio che è un linguaggio di programmazione mutabile serve stare molto attenti se parti di strutture dati sono condivise (e.g., modificare una modifica anche l'altra). Lo sharing è la chiave dell'efficienza della programmazione funzionale.


Abbiamo visto che lo sharing si verifica solo per T1, per averlo anche nel sotto-albero {leaf, K, V} possiamo attribuire un nome al pattern $f(\{_, T1, 5, \{leaf, K, V\} = T2\}) \rightarrow \{node, T1, 7, T2\}$.

(**) Le variabili dei pattern sono sempre nuove variabili in tutti i linguaggi eccetto per Erlang/Elixir. Se usiamo una variabile X già dichiarata/assegnata in un pattern si intende il valore della variabile (scatta il codice della costante).

```

1 {X, Y} = {4, 5}.
2 % {4, 5}
3 {X, Y} = {2, 3}.
4 % exception error non match of right hand side value
5 {X, Z} = {4, 3}.
6 %{4, 3}

```


 Erlang

Questa non è una buona idea, la semantica del codice dipende da un contesto molto ampio, ovvero dal codice scritto in precedenza.

2.5.1. Ottimizzazioni nella Compilazione del Pattern Matching

Nei linguaggi che hanno nativamente pattern matching profondi (Erlang/OCaml/Haskell ma non Scala/Javascript/Python) quando si compilano dichiarazioni di funzioni non si compilano i pattern uno alla volta come detto fin'ora. Ad esempio

```
1 f({node, {node, T1, K1, T2}, K, {leaf, _, _}})
2   when K1 < K -> ... ;
3 f({leaf, _, 0}) -> ... ;
4 f({node, {node, T1, K1, T2}, K, {node, T3, K3, T4}}) -> ...;
```

 Erlang

La semantica del linguaggio dice che viene usato il primo pattern in ordine che fa match. Pensiamo all'invocazione di `f({node, {node, ..., 3, ...}, 5, {node, ..., ..., ...}})`. Il primo pattern fallisce, quindi quando viene valutato il terzo ripetiamo tantissimi controlli che in realtà avevamo già fatto. Mano a mano che processiamo dei pattern e scopriamo informazioni è possibile riutilizzarli. Non possiamo usare alberi di decisione perché dobbiamo rispettare l'ordine dei pattern ma allo stesso tempo è stupido effettuare tante volte gli stessi controlli. La procedura di compilazione dei pattern avviene producendo una sorta di automa a stati finiti modificato dove ogni stato dell'automa codifica l'informazione già scoperta sull'input (come anticipato nella [Sezione 2.2](#)). La complessità è sub-lineare sulla somma della lunghezza dei pattern.

2.6. Ricorsione

Analizziamo ora la complessità computazionale della funzione `insert()` definita in [Codice 4-26](#). Il primo caso è il caso della foglia che va in $O(1)$ dato che pattern-matching è a compile-time e la guardia costa $O(1)$. Notiamo che in realtà tutti i casi sono $O(1)$ in tempo e spazio. Per analizzare il costo computazionale dobbiamo analizzare il costo delle chiamate ricorsive. L'idea è che a runtime viene mantenuto uno stack di Record di Attivazione (RA) mantiene:

- Valore dei registri salvati da ripristinare quando si ritorna indietro
- Indirizzo del valore di ritorno (dove salvare l'output)
- Parametri di funzione
- Variabili locali
- Indirizzo di ritorno (dove restituire il controllo)

Lo stack per convenzione cresce verso il basso (parte da indirizzi di memoria alti e decrese).

Quanto costa in spazio e tempo una chiamata di funzione? in tempo $O(1)$ (basta prendere lo stack pointer ci sommiamo la dimensione del RA che è noto a tempo di compilazione e si copiano un paio di cose) e in spazio $O(1)$ (allochiamo il RA). Se abbiamo una sequenza di chiamate a funzione allora il costo sarà $O(n)$ dove n è il numero di chiamate di funzione. Se una funzione chiama se stessa allora allocherà ogni volta un RA ($O(1)$). L'uso della memoria aumenta fin quando la funzione termina, se non termina mai l'utilizzo dello spazio da parte della funzione tende a infinito.

Consideriamo l'attore definito in [Codice 1](#). Ogni qual volta riceve un messaggio assume un nuovo behavior con una chiamata ricorsiva. Ma quindi come fa a girare per sempre? non termina mai la memoria?

L'idea: stiamo eseguendo una funzione `f()` e si invoca una funzione `g()`. Il RA di `f()` deve contenere tutta e sola l'informazione che serve per terminare l'esecuzione di `f()` dopo che `g()` sarà terminata. Si osserva però che non tutta l'informazione contenuta nel RA di `f()` è necessaria dopo la chiamata di `g()`. Un esempio in C potrebbe essere

```

1 f(int x, int y, int p) {
2     int z, w;
3     ...
4     g();
5     p = y * z;
6     return p + p;
7 }

```



```

1 +-----+ <-- indirizzo alto
2 | Return Address | <- Indirizzo di ritorno
3 +-----+
4 | Return Value   | <- Valore restituito da f()
5 +-----+
6 | Saved Registers | <- Registri salvati
7 +-----+
8 | x (parametro)   | <- parametro passato (by value)
9 +-----+
10 | y (parametro)  |
11 +-----+
12 | p (parametro)  |
13 +-----+
14 | z (variabile loc.) | <- variabile locale
15 +-----+
16 | w (variabile loc.) |
17 +-----+ <-- indirizzo basso

```

Plain Text

Dopo avere chiamato `g()`, `w` e `x` non servono più. Possiamo quindi ottimizzare questa situazione

- Prima di invocare `g()` si decrementa lo stack-pointer per rilasciare `w` che non serve più.
- Attraverso un'analisi statica determinare che è più conveniente piazzare `x` subito dopo `w` nel RA in modo da rilasciare anche `x`. Per esempio Prolog implementa da sempre queste ottimizzazioni. Il caso limite sarebbero le “tail calls” ovvero dopo `g()` non bisogna più fare nulla.

Defnizione: Tail Call

- Una chiamata a `g()` dentro a `f()` è di coda se e solo se
- l'unica istruzione eseguita dopo `g()` è `return`.
 - il valore ritornato da `f()` è esattamente il valore ritornato da `g()`.

Questo è un caso limite perché sembra che tutte le variabili non servono più. Sembra che servano solo i registri, i valori di ritorno e il return address. Ma questo è proprio quello che fa la funzione `g()` prima di terminare. Praticamente possiamo fare `pop()` completo del RA di `f()`. Quando la chiamata è di coda la si compila come `pop()` di tutti i registri, variabili locali ecc., `push()` dei parametri attuali della `g()` e `jump()` al codice di `g()` (passiamo il controllo a `g()`). La prima parte del RA di `g()` coincide con la prima parte del RA di `f()`.

Una chiamata di coda quando è implementata la tail call optimization è:

- $O(1)$ in tempo

- “ $O(0)$ ” in spazio perché ricicliamo il record di attivazione

Definizione Tail Recursion

Una funzione $f()$ si dice tail recursive se e solo se tutte le sue chiamate ricorsive sono di coda.


In **Codice 1** tutte le chiamate `cc()` sono tail ricorsive quindi sono sostanzialmente un ciclo, non si spreca memoria per lo stack.

In **Codice 4** la `insert()` non è tail recursive quindi costa in tempo $O(h)$ dove h è l'altezza dell'albero. In spazio $O(h)$ dove h è l'altezza dell'albero dovuto interamente all'allocazione sullo stack (ogni chiamata ricorsiva è $O(1)$).

In **Codice 4** la `search()` è tail recursive quindi costa in tempo $O(h)$ dove h è l'altezza dell'albero. In spazio $O(1)$.

Aggiungiamo una funzione a **Codice 4** una funzione che asintoticamente sarà inefficiente `search_val()`, infatti quando troviamo un valore dobbiamo comunque risalire tutto l'albero prima di ritornare, questo è dovuto alla natura ricorsiva dell'implementazione. Per renderla più efficiente possiamo pensare di lanciare un'eccezione in modo tale da ritornare immediatamente interrompendo la naturale ricorsione.

```
1  % search_val(V, T) restituisce una tupla e ricicliamo il tipo optional
2  % search_val(V, T) ritorna {found, K} se K è associata a V in T, not_found
   altrimenti
3  search_val(V1, {leaf, K, V2}) when V1 == V2 -> {found, K} ;
4  search_val(V1, {leaf, _, _}) -> {not_found, K} ;
5
6  % Vogliamo comportarci diversamente in base al valore di ritorno della
   chiamata ricorsiva, se not_found andiamo nell'altro sotto-albero.
7  % Possiamo usare un costrutto e.g., switch/match/case
8  search_val(V, {node, T1, _, T2}) ->
9      case search_val(V, T1) of
10         {found, _} = Res -> Res ; % per sharing
11         not_found -> search_val(V, T2)
12     end.
```

 Erlang

Questo codice oltre ad essere non bello è poco efficiente, non in termini asintotici perché non possiamo fare meglio di visitare tutto l'albero nel caso pessimo, ma nel senso di costanti moltiplicative. Il valore deve risalire dalla foglia fino alla radice perché ad ogni step c'è un'analisi per casi che controlla se il valore è stato trovato o meno. Vorremmo un costrutto che interrompe la risalita del valore di ritorno al chiamante e restituire il controllo laddove abbiamo iniziato la ricerca. Ricordiamoci che lo stack rappresenta l'esecuzione futura (ogni chiamata ha una stack entry che descrive il codice che deve essere eseguito) quindi vorremmo eliminare tutto il lavoro sullo stack che non vogliamo più effettuare. Ci serve un operatore di controllo che alteri l'ordine futuro di esecuzione del codice (altera lo stack). Introduciamo le eccezioni.

```

% Le eccezioni sono valori che possono atomici/non atomici. Il
1 fatto che siano anche non atomi è molto utile perché possiamo
trasferire dati
2 2 * try 3 + (5 + throw({ecx1, 5})) catch {exc1, V} -> V ; exc2 -> 3 end.

```

Erlang

```

1 % search_val_aux(V, T) solleva {found, K} se K è associata a V
in T, ritorna not_found altrimenti
2 search_val_aux(V1, {leaf, K, V2}) when V1 == V2 -> throw({found, K}) ;
3 search_val_aux(V1, {leaf, _, _}) -> not_found ;
4 search_val_aux(V, {node, T1, _, T2}) ->
5     search_val_aux(V, T1),
6     search_val_aux(V, T2).
7
8 search_val2(K, T) ->
9     try
10         search_val_aux(K, T)
11     catch
12         {found, _} = R -> R
13     end.

```

Erlang

Questo è un uso interno delle eccezioni, non “scappano”. Le eccezioni selvagge che fuggono dal codice non è sempre una buona pratica. Vediamo ora come sono implementate. Supponiamo di avere uno stack:

```

1 +-----+
2 | Stack frame n+2 | RA di chiamate di funzione durante esecuzione di E
3 +-----+
4 | Stack frame n+1 | record try catch
5 +-----+
6 | Stack frame n   | fn() = ... try E catch ...
7 +-----+
8 | Stack frame n-1 | ...
9 +-----+
10 | Stack frame n-2 | f2() = ... f3() ...
11 +-----+
12 | Stack frame     | f1() = ... f2() ...
13 +-----+

```

Plain Text

Fin'ora la stack frame lo abbiamo sempre visto come un Record di Attivazione (RA). Allo stack frame n abbiamo una funzione che introduce un blocco try catch. L'espressione E potrebbe far crescere ulteriormente lo stack. La domanda è dove memorizziamo il codice da eseguire del blocco catch. Aggiungiamo uno stack frame che non è un RA, bensì un record try catch.

Un record try catch semplicemente mantiene un puntatore al codice catch non contiene tanto altro al suo interno. Ci serve anche distinguere i RA dai record try catch. Quindi avrà anche un TAG che lo differenzia dai RA. Anche i RA avranno un TAG a loro volta. Ci chiediamo ora cosa accade al momento dell'istruzione throw. Ci sarà un ciclo while che fa pop() dei RA e quando incontra un record try catch chiama la funzione usando l'indirizzo del blocco try catch.

```

1  throw(E) {
2      finished = false;
3      while(!finished) {
4          while(Stack[0].TAG != "record try catch") Stack.pop();
5          addr = Stack[0].indirizzo_codice_catch(E)
6          Stack.pop()
7          case CALL addr(E) of
8              {caught, V} -> finished = true; V
9              {not_catched, V} -> nothing
10     }
11 }

```


Dopo il catch si analizzano le varie eccezioni, se l'eccezione è gestita restituisce {caught, E} per un qualche E. Altrimenti ritorna {not_catched, V}. Potrebbe darsi che venga anche ritornata un'altra eccezione in quel caso si ripete il codice già scritto.

Consideriamo ora l'esempio in [Codice 1](#). Supponiamo di introdurre una nuova funzione tax() che divide il balance corrente per un certo N. Questo potrebbe causare una divisione per zero.

```

1  -module(hello).
2  -export([cc/1]).
3
4  cc(Bal) ->
5      try
6          receive
7              print ->
8                  io:format("Il balance è ~p ~n", [Bal]),
9                  cc(Bal) ;
10             {put, N} -> cc(Bal+N) ;
11             {tax, N} -> cc(Bal/N) ;
12             {get, PID} -> PID ! Bal, cc(Bal) ;
13             exit -> ok
14         catch
15             error:_ -> cc(Bal)
16     %
17     %
18 end.

```


 Erlang

Possiamo gestire la divisione /0 con un blocco try catch. Ma cc(Bal) è ancora tail ricorsiva? apparentemente sì. Ma non può scattare l'ottimizzazione perché lo stack sarà

```

1  +-----+
2  | TAG try catch |
3  | Addr catch code | <--- Record try catch
4  +-----+
5  | Bal |

```

 Plain Text

6		RA		<--- Record di attivazione per la funzione cc()
7		RV		
8		TAG RA		
9	+-----+			

Ma non può scattare l'ottimizzazione di coda (buttare via le variabili, riciclare RV e RA). C'è un record try-catch sopra il frame di `cc()` (più recente nello stack). Questo significa che il sistema di esecuzione deve ricordare cosa fare in caso di eccezione e deve mantenere informazioni nel frame, per sapere dove saltare (Addr catch code). Quindi il frame di `cc()` non può essere eliminato, perché il blocco try è ancora attivo e aspetta che qualcosa possa generare un'eccezione.

Definizione Tail Call (Updated)


Una chiamata a `g()` dentro a `f()` è di coda sse

- l'unica istruzione eseguita dopo `g()` è `return`.
- il valore ritornato da `f()` è esattamente il valore ritornato da `g()`.
- non è contenuta all'interno di un blocco try catch nella parte dell'espressione da valutare (non è protetta).

Cerchiamo una versione tail recursive. Tutte le chiamate di coda devono essere al di fuori del blocco try catch. Allo stesso tempo non ha senso proteggere la chiamata ricorsiva (è già protetta nella chiamata annidata). Il blocco try catch deve terminare facendo la chiamata ricorsiva praticamente.


```

1  -module(hello).
2  -export([cc/1]).
3
4  cc(Bal) ->
5      case
6          try
7              receive
8                  print ->
9                      io:format("Il balance è ~p ~n", [Bal]),
10                     {recur, Bal} ;
11                     {put, N} -> {recur, Bal+N} ;
12                     {tax, N} -> {recur, Bal/N} ;
13                     {get, PID} -> PID ! Bal, {recur, Bal} ;
14                     exit -> {result, ok}
15             end
16         catch
17             error:_ -> {recur, Bal}
18         end
19     of
20         {result, R} -> R ;
21         {recur, Bal} -> cc(Bal) ;
22     end.
```


 Erlang

Possiamo fare case analysis sul risultato del blocco try catch. Usiamo il risultato per chiamare ricorsivamente `cc(Bal)`. Prima se la chiamata ricorsiva sollevava un'eccezione veniva catturata dal blocco try catch del chiamante. Ora invece se l'eccezione viene sollevata esce dal controllo, "fugge". Poco alla volta i linguaggi aggiunsero zucchero sintattico, Erlang aggiunge:

```
1 try
2   E % codice protetto
3 of
4   p1 -> c1 ; % pattern 1
5   ...
6   pn -> cn ; % codice non protetto che matcha il risultato di E
7 catch
8   ... -> ... % codice non protetto
9 after
```


 Erlang

```
1 try io:format("try~n"), 4 after io:format("ok~n") end.
2 % try
3 % ok
4 % 4
```

 Erlang

Notare che il costrutto `after` esegue sempre il codice al suo interno anche in presenza di eccezioni sollevate dentro il blocco `try`. Con un side effect (stampa a video) si nota come il costrutto `after` esegua sempre il codice al suo interno. Lo zucchero sintattico viene tradotto con:

```
1 case
2   try
3     try {E} of F catch G
4     catch
5       Exc -> {exception, Exc}
6     end
7   of
8     {return, V} -> C, V ;
9     {exception, Exc} -> throw(Exc)
10 end.
```

 Erlang

L'utilizzo di `after` (sarebbe il `finally` di Java) rende tutte le chiamate protette e quindi perdiamo l'ottimizzazione delle chiamate di coda. In Erlang esistono le eccezioni di tre tipi:

- `throw()`: nate per essere catchate `throw(ciao)`
- `exit()`: non andrebbero catchate ma si può fare con `exit: _`. Può essere un'uscita normale o abnormale.
- `error`: non andrebbero catchate ma si può fare con `error: _`. Sarebbero errori unrecoverable. Vengono propagati, si abortisce la BEAM e si legge il backtrace.

3. The Next Big Emerging Constructs: Algebraic Effects

Un esempio di linguaggio che li supporta maggiormente è OCaml ma nascono in primis con il linguaggio accademico Eff. La prima idea che ci permette di comprendere gli effetti algebrici è nota come "eccezioni rientranti", in pratica una chiamata `throw(E)` passa remotamente il controllo

all'istruzione `catch` che gestisce l'errore `E`. Il codice remoto può restituire il controllo alla `throw` passando un valore `V` che diventa il risultato della `throw`.

```
1 2 * try 3 + (5 + throw 7)
2 catch
3 0 -> 22 % come un'eccezione normale
4 N -> resume N + 1 % resume nuova keyword che restituisce il controllo
5 end
6 % 32
```

Ma c'è un modo di implementare questo costrutto in maniera efficiente in spazio e tempo? nelle eccezioni l'istruzione `throw` effettua un ciclo `while` sullo stack e per ogni record di attivazione/record `try-catch` che non cattura l'eccezione effettua `pop` sullo stack. Negli algebraic effects (implementazione `naïve`) la `throw` anziché effettuare `pop` sullo stack effettua

```
1 RA = Stack.pop(); // stacciamo il Record di Attivazione
2 Detached.push(RA); // lo aggiungiamo in uno Stack Detached
```

`Detached` è uno stack ordinato in senso inverso dove memorizziamo temporaneamente gli stack frame staccati. La `resume` invece effettua

```
1 while(!Detached.isEmpty()) {
2   RA = Detached.pop();
3   Stack.push(RA);
4 }
5 // then we have to write the resume value as result value of the throw
  instruction
```

Se invece nel ramo `catch` non facciamo `resume`

```
1 while(!Detached.is_empty()) Detached.pop()
```

Notiamo però che per implementare il costrutto `resume` è necessario un pezzo di stack che chiamiamo `detached`, che contiene a sua volta stack-frame.

In un linguaggio con effetti algebrici si aggiunge un nuovo tipo di dato astratto fibra (`fiber`) che rappresenta un frammento di stack `detached`, il quale è un tipo di dato di prima classe (lo possiamo muovere, copiare, passare in giro, passarlo ad un altro processo, ...). Su di esso possiamo effettuare l'operazione `resume` per reinstallarlo in cima allo stack. Il ramo `catch` cattura la fibra. Poiché lo stack `detached` può essere passato, riutilizzato, duplicato, diventa un mezzo potente per realizzare control flow non lineari. Ma queste operazioni hanno senso? Pensiamo ad un pezzo di codice che gestisce strutture dati che descrivono l'esecuzione futura del codice, se la tiene da parte e ne ripristinavano l'attivazione più tardi. Beh uno scheduler! vediamo come implementarlo in user-space:

```
1 yield() -> throw(yield). % yield è un effetto (atomo)
2 fork(G) -> throw(fork). % fork è un effetto (atomo)
3
4 code_to_fiber(F) ->
5 try
```

```

6     throw(stop), F
7   catch
8     stop, K -> K
9   end.
10
11 % Main è il primo thread da eseguire
12 % Queue è la coda dei thread sospesi
13 scheduler(Main, Queue) ->
14   try
15     resume(Main, ok)
16   catch
17     % Catturiamo l'eccezione yield e K (il codice che il thread deve ancora
18     % eseguire dopo la yield)
19     case Queue of % della forma pattern_eccezione + pattern K
20       [] -> scheduler(K, []) ;
21       [F|L] -> scheduler(F, append(L, [K]))
22     fork(G), K ->
23       scheduler(K, append(queue, [code_to_fiber(G)]))
24   end.
25
26 scheduler(Main) ->
27   scheduler(code_to_fiber(Main), []).

```

Quindi gli effetti algebrici servono per gestire in maniera non locale ma come se fossero gestiti localmente, implementare scheduler, ecc... In generale il costrutto è molto espressivo. Tutto ciò che possiamo fare con le monadi in Haskell possiamo farlo con gli effetti algebrici.

Per un'implementazione più efficiente:

- Lo stack diventa stack di fibre
- Le fibre sono stack a loro volta
- + Staccare/riattaccare una fibra diventa $O(1)$
- ! Lo stack non è più consecutivo in memoria

4. Gestione della memoria

La memoria e la sua gestione presenta diverse problematiche

- Garbage ovvero memoria che non è utilizzata dal programma ma che non viene deallocata. Sapere se un programma utilizzerà più memoria o meno è un classico problema indecidibile. Solamente il programmatore (forse) può saperlo (se utilizza librerie diventa difficile) ma dipende dalle invarianti del linguaggio.
- Puntatori a memoria non allocata (in linguaggi che permettono il cast a puntatori)
- Riutilizzo di memoria deallocata
 - Accedo
 - Ri-alloco
- Frammentazione della memoria (best fit, first fit)
- Memoria non inizializzata (la memoria non è allocata ma proviamo comunque ad accedervi)

I punti elencati sopra sono concetti che avvengono in ambito sequenziale. Quando si passa in campo concorrente/parallelo emergono ulteriori problematiche:

- Race condition, accesso contemporaneo di un thread scrittore e n thread lettori/scrittori. Per risolvere esistono mutex, lock, regioni critiche ecc... ma i problemi che si portano dietro (deadlock, starvation, ecc...) non saranno trattati

In ambito distribuito non ci sono problemi di gestione della memoria.

```
1 T f(..., T x); // f riceve in input alcuni parametri e x di tipo T,  
   ritornando un tipo T
```

Vediamo gli scenari che possono verificarsi per quanto riguarda il dato in input/output. Simuliamo di essere un compilatore e staticamente cercare di capire quando allocare/deallocare la memoria.

- a) Il dato T in output è interamente allocato a ogni chiamata, non ci sono puntatori da input a output e viceversa, non ci sono puntatori al dato in variabili globali statiche (in C una variabile globale top-level ancora disponibile dopo la chiamata di funzione) (una variabile static ha scope solo all'interno della funzione). Questo significa che l'unico che ha in mano il dato è il chiamante. Per determinare se il dato è ancora utile allora guardiamo il chiamante.
- b) L'input è condiviso con l'output, ci sono puntatori dall'output all'input (ad esempio in [Codice 4](#) nella funzione `insert()` ricicliamo l'input con l'output attraverso lo sharing). In questo caso l'input non è garbage fino a quando l'output lo è.
- c) L'output è condiviso con l'input, ci sono puntatori dall'input all'output. Ad esempio una slice di un'array. L'output non è garbage fino a quando l'input lo è.
- d) 2) + 3). Restituire un output legato in maniera bi-direzionale con l'input. Non possiamo deallocare uno prima dell'altro.
- e) La funzione mantiene un puntatore all'output o a parte di esso (singleton o memoization). L'output non è garbage fino a quando non lo è per tutti gli output precedenti e quell'input non verrà mai più richiesto.

Al netto di questi possibili scenari vediamo quali sono le possibili tecniche di gestione della memoria:

- Nessuna (C).
- Garbage Collection Automatica (GCA). Il run-time del linguaggio approssima la nozione di “essere ancora utile” e questa approssimazione determina de deallocare o meno il dato.
 - o Ci sono diverse euristiche di garbage detection. Reference Counting, usato dai linguaggi mutabili e Mark and Sweep linguaggi non mutabili. Indipendentemente dall'euristica il programmatore deve informare il GC.
 - o Intervento del programmatore, attraverso weak pointers, che spiegano all'algoritmo che certi puntatori non contano più per determinare se qualcosa è garbage o meno.
- Gestione informata del programmatore (C++, RUST) dove non c'è euristica di garbage detection. Solo il programmatore informa il compilatore attraverso le invarianti che il programmatore sceglie.

4.1. Reference Counting

Utilizza l'euristica più grezza ovvero se un dato non ha puntatori entranti (a un dato boxed) implica che il dato è garbage. Come valutiamo questa euristica? serve sapere per ogni dato boxed quanti puntatori ci sono.

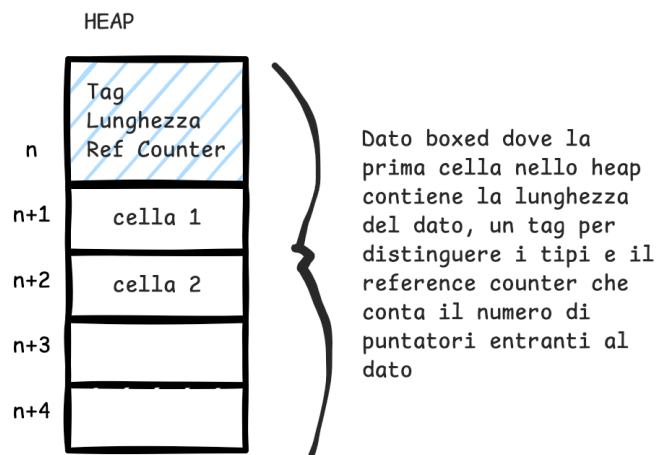


Figura 1: Rappresentazione dello Heap con aggiunta del reference counter nella prima cella di un dato boxed

Il reference counter sarà un intero che conta i puntatori entranti, sarà sempre > 0 , perché se è 0 viene deallocato il dato. Quanto è buona questa euristica?

Stack e Registri sono dette **radici**, ovvero aree di memoria sempre accessibili al programma. Anche se non funziona proprio così lo heap, se non abbiamo un puntatore non possiamo accedere allo heap. Esiste uno scenario dove ci sono celle che non sono accessibili al programma (non sono raggiungibili dalle radici), che non sono utili eppure hanno reference counter > 1 . Quando ci sono dei cicli! per esempio quando abbiamo una Double LinkedList appena eliminiamo il puntatore alla lista diventa tutto garbage ma data la natura della double linked list i reference counter rimangono > 0 e la memoria non viene reclamata.

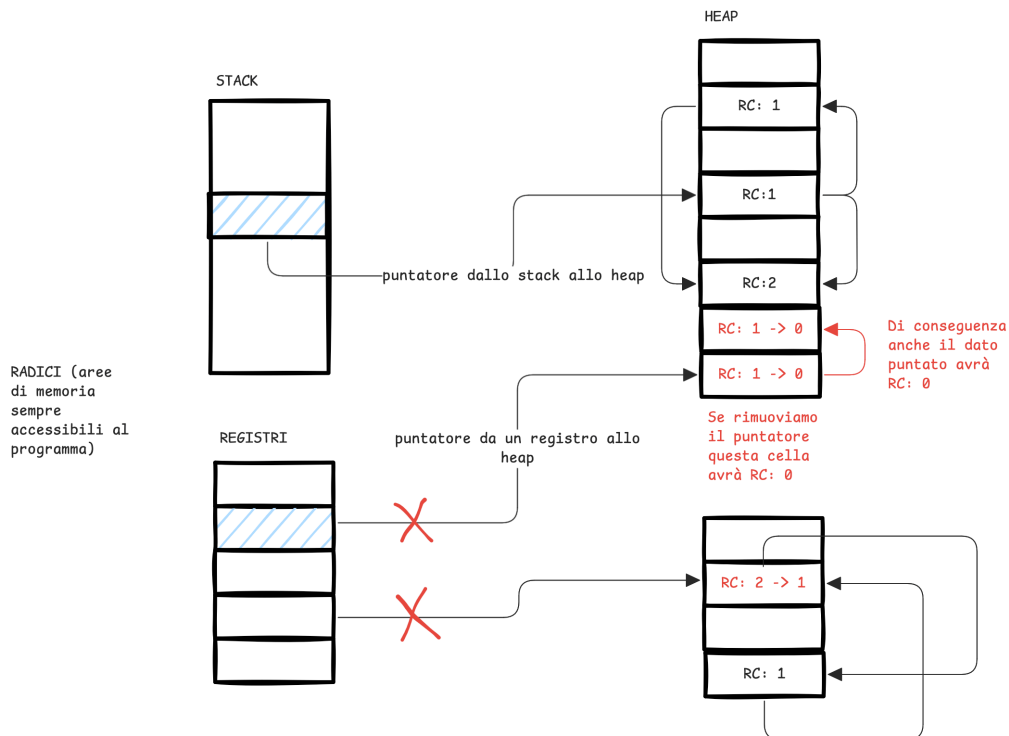


Figura 2: Rappresentazione dell'algoritmo di Reference Counting

Vediamo come implementare l'allocazione della memoria

- a) Se la memoria è frammentata serve un algoritmo di management del pool di aree di memoria libera (best fit o first fit) questo è costoso in spazio e in tempo. A basso livello allocare diventa diventa

```
1 p = alloc(size + 1);
2 (*p)[0] = 1; // incrementiamo il RC
```

- b) Copia di un puntatore (un nuovo puntatore al puntato di p). Ricordiamoci che q è un puntatore e puntava a qualcosa

```
1 if (q != null)
2   (*q)[0]--; // decrementiamo il RC di q
3
4 if ((*q[0]) == 0)
5   dealloc(q); // se il RC è 0 deallochiamo tutto q ricorsivamente
6
7 // copia effettiva
8 q = p;
9 (*p)[0]++; // incrementiamo il RC di p
```

Dove dealloc

```
1 dealloc(p) {
2   for i = 1 to *p[0].size {
3     if (boxed(*p[i]))
4       (*p)[i]--;
5     if (*p[0] == 0) // se RC == 0 deallochiamo ricorsivamente
6       dealloc((*p)[i])
7   }
8
9   free(p)
10 }
```

Quindi quanto costerebbe deallocare l'intera lista? il costo è unbounded in tempo. Sarebbe lineare nella dimensione della lista ma non è l'input dell'operazione di assegnamento (che è l'operazione che scatena la deallocazione). Nel caso pessimo quando assegniamo un valore ad una variabile scateniamo una ricorsione lunga quanto? quanto le operazioni svolte fin'ora dal programma (nel caso pessimo sono tutte allocazioni). Quindi $O(n)$ dove n c'è il numero di operazioni nella lista. Usare Reference Counting per scrivere un programma real-time che deve avere un tempo massimo di esecuzione delle operazioni non è il massimo. Come evitare questo comportamento? ad esempio possiamo utilizzare una sorta di skip-list, ovvero una seconda catena di puntatori dove anziché puntare uno al successivo puntano con un passo e.g., 50. In questo modo vengono deallocate al massimo 50 celle, e teniamo il puntatore alla 51-esima. Alla prossima assegnazione verrà scatenata la deallocazione delle successive 50 e così via.

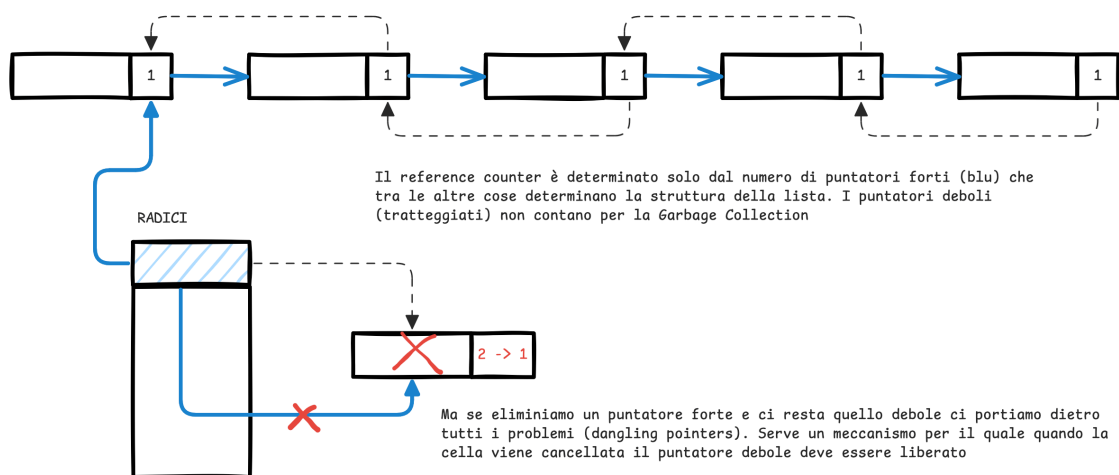


Figura 3: Sistema di weak/strong pointers per garantire il funzionamento dell'euristica di RC per strutture dati cicliche

Il Reference Counting (e la sua euristica) ha un costo unbounded sia quando si ha allocazione e deallocazione. Inoltre come abbiamo detto ci sono problemi nella gestione delle strutture dati cicliche (Figura 3). Viene introdotta la nozione di puntatore debole, ovvero un puntatore che non contribuisce al valore del reference counter. Il puntatore debole potrebbe essere una coppia che conserva informazione sull'esistenza del puntato però questo sicuramente introduce overhead, così come le molte soluzioni possibili.

Gli algoritmi di GC hanno un costo significativo. Alcune volte un programmatore potrebbe richiedere una grande quantità contigua di memoria e la utilizza in maniera autonoma, magari implementandosi autonomamente meccanismi di allocazione/deallocazione della memoria.

4.2. Mark & Sweep

L'euristica del Mark & Sweep è la seguente: se un dato non è raggiungibile dalle radici allora il dato è garbage. Non vale viceversa. Il nome dell'algoritmo si deve al fatto che l'implementazione naive è pensato per avvenire in due passate, in realtà nessuno lo implementerebbe in due step. L'idea è marcare le celle da spostare, e le spostiamo (deframmentando). In realtà si implementa con un'unica passata, anziché marcare le celle le si spostano direttamente. L'algoritmo inizialmente chiede una grande quantità di memoria al sistema operativo, divide a metà la memoria che ha a disposizione e alloca le variabili in una metà, quando si sveglia per fare garbage collection marca e deframmenta prendendo le celle vive spostandole nell'altra metà libera, segnando la metà corrente come libera. Si implementa in un'unica passata che direttamente sposta le celle anziché marcarle e poi spostarle.

Il run-time chiede un'enorme quantità di memoria al sistema operativo e lo divide in heap di lavoro (quello su cui vengono allocati i dati) e in heap inutilizzato. Si lavora sempre sullo heap di lavoro fino a quando non scatta la GC che scambia i due heap. In questo modo però il 50% della memoria è sprecata (vedremo come mitigare questo problema). Non è necessario allocare subito due heap molto grandi. Quando ci si accorge che è necessaria una dimensione maggiore per lo heap allora quando scatta il GC è possibile richiedere un heap più grande al sistema operativo.

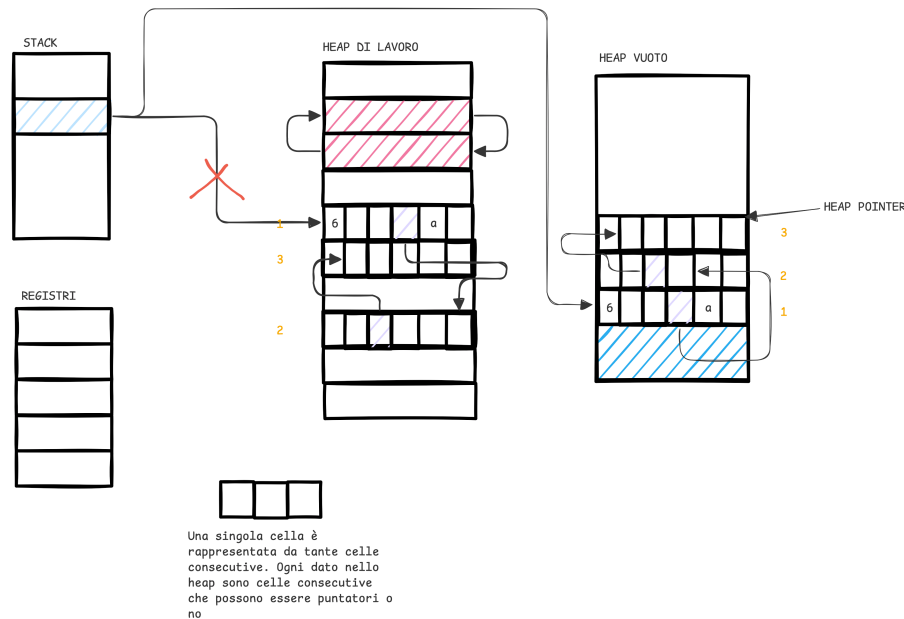


Figura 4: Algoritmo Mark & Sweep

L'algoritmo descritto in [Figura 4](#) itera su tutte le radici, i dati boxed li ignora, mentre segue i puntatori. Il puntatore porta a un dato nello heap unboxed, a questo punto si alloca il dato sul nuovo heap (heap inutilizzato) deframmentando, quindi utilizzandolo come se fosse uno stack. Avremo un Heap Pointer (HP) che indica a che punto siamo arrivati. Quando all'interno di un dato unboxed troviamo un puntatore allora si segue e si ripete l'operazione incrementando l'HP. Una volta terminata la visita del grafo radicato in una delle radici i dati non raggiunti vengono spostati da un'altra parte. Anche i puntatori (in viola) vengono copiati. Una volta terminato lo spostamento di tutti i puntatori, la memoria libera sarà sopra l'HP e avverrà incrementando l'HP.

Mark & Sweep rende iper-efficiente l'allocazione dei dati. Infatti è molto popolare per implementare GC nei linguaggi funzionali. Dato che ci sono i dati molto vicini sfruttiamo il principio di località potenzialmente con guadagno in termini di performance/cache-hit.

L'algoritmo naïve ha però diversi problemi, innanzitutto se ci sono cicli di puntatori si ha ricorsione infinita (possiamo inserire l'informazione visited per prevenire i cicli). Inoltre è molto importante mantenere lo sharing per ottenere efficienza in Erlang. L'algoritmo così presentato creerebbe nuove celle nello heap inutilizzato se ci fossero puntatori provenienti da radici differenti, (e.g., puntatore da un registro alla cella 1 in [Figura 4](#)). Preservare lo sharing significa inserire un puntatore a una cella già creata se esiste ([Figura 5](#)). Per risolvere questo problema potremmo utilizzare la prima cella dei dati unboxed (contenente lunghezza, tag) per salvare un puntatore che va dal dato nell'heap attivo al dato che abbiamo allocato nell'heap inutilizzato. Questo inoltre evita i cicli perché si mantiene informazioni sui dati che sono già stati inseriti nel nuovo heap.

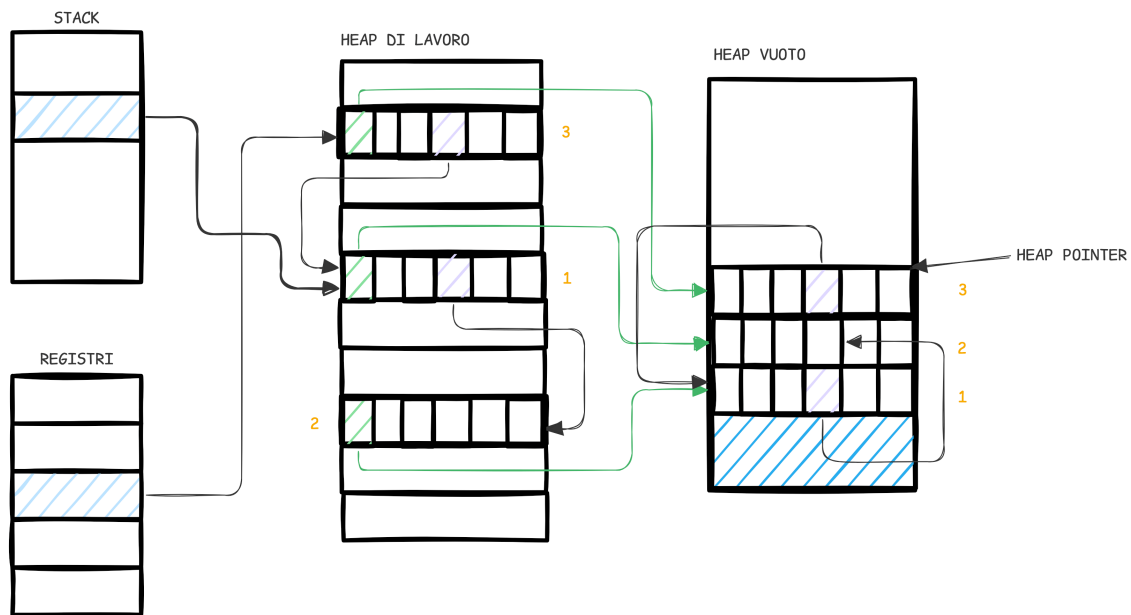


Figura 5: Implementazione di Mark & Sweep per mantenere lo sharing

Notiamo che dichiarare libero l'heap attivo significa semplicemente spostare l'heap pointer alla base dello heap (pulirlo costerebbe $O(n)$). Il costo dell'algoritmo è:

- In spazio è $O(0)$
- In tempo è lineare in funzione nel numero di dati vivi $O(\#radici + \#dati\ vivi)$. Non vengono visitati i dati irraggiungibili (garbage). Il numero di dati vivi potrebbe essere molto grande quindi non siamo contenti di questo costo.

Attenzione perché ogni passata di GC ha questo costo. Come decidere quando effettuare la “passata”? ci sono diverse strategie e.g., quando lo heap pointer sta per raggiungere la cima (e.g., 90% della capacità), oppure quando il programma sta utilizzando poco CPU time.

Dobbiamo inoltre assicurarci che questo algoritmo, dato che muove i dati in memoria, non rompa la semantica del linguaggio di programmazione. Pensiamo all'utilizzo di Mark & Sweep in C. I puntatori cambieranno valore a run-time. Le operazioni sui puntatori possono essere:

- a) Dereferencing (non ha problemi, troviamo sempre il dato in memoria se l'algoritmo muove anche il dato e ovviamente lo fa).
- b) Aritmetica dei puntatori (tutte le operazioni sui puntatori implementate come se fossero interi e non puntatori).
 - Somma scalare all'interno dei blocchi (non ha problemi). Rispetto allo standard del C, è lecito effettuare somme/sottrazioni quando allochiamo un'array per spostarci al suo interno. In realtà lo standard dice che se abbiamo allocato n celle di memoria possiamo spostarci di $n + 1$ celle (usciamo a destra di 1). Uscendo di 1 si può confrontare ma non dereferenziare il dato in posizione $n + 1$.
 - Sottrazione (per ottenere un offset) all'interno dei blocchi (non ha problemi)
 - Confronto tra puntatori (lecito anche tra blocchi diversi). $=$ e $!=$ (non danno problemi). Grazie al fatto che ci siamo assicurati di preservare lo sharing
 - $<$, $>$ e \leq , \geq (sono rotte). Se confrontiamo qualcosa fuori dallo stesso blocco dato che i dati si spostano potrebbe darsi che una volta vince vero e una volta vince falso. Quando capita di usare relazioni d'ordine sui puntatori? per le chiavi degli alberi, il puntatore non può essere chiave negli alberi!

c) Un puntatore non può essere chiave di una funzione hash

Non è possibile dunque utilizzare i puntatori come chiavi nei dizionari. L'idea sarebbe quella di aggiungere un id univoco al dato ma costa un'indirizzione.

Concludendo Mark & Sweep è troppo lento in tempo. Idealmente vorremmo ottimizzarlo sfruttando un concetto statistico noto come Ipotesi Generazionale.

4.3. Ipotesi Generazionale

Un'ipotesi statistica che potrebbe essere vera o meno, ma nella maggior parte dei casi si applica al codice. L'ottimizzazione del GC terrà conto di questa ipotesi e la sfrutterà. Un dato x è soggetto a due casistiche

$$x := \begin{cases} \text{è destinato a essere vivo molto a lungo} \\ \text{o è destinato a diventare unreachable velocemente} \end{cases} \quad (1)$$

Viene esclusa l'ipotesi intermedia, questo caso binario è il caso estremo. L'ipotesi potrebbe essere rilassata per avere casi intermedi. Molti linguaggi usano solo due generazioni ma è possibile appunto rilassare e avere algoritmi più fini utilizzando più generazioni. Vediamo un algoritmo Mark & Sweep generazionale. L'idea per l'ottimizzazione è la seguente: se avessimo un modo per capire un dato x a quale categoria appartiene potremmo allocare i dati della prima categoria in una terza regione che non modifichiamo, mentre i dati della seconda tipologia vengono gestiti normalmente con Mark & Sweep pagando il suo costo poche volte (i dati muoiono in fretta).

4.4. Mark & Sweep Generazionale

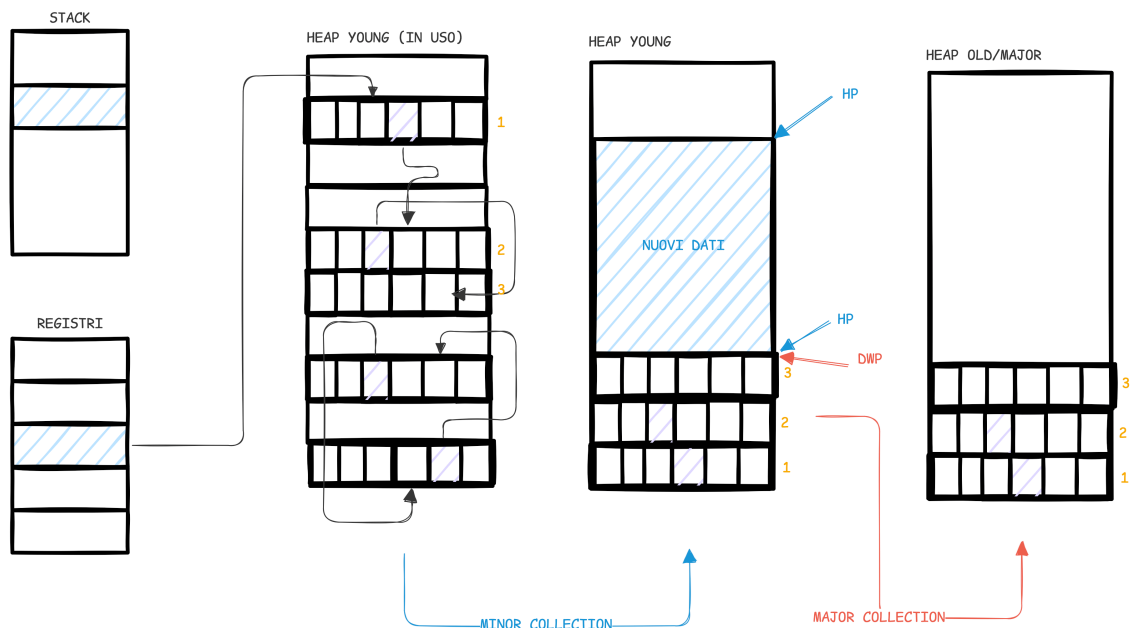


Figura 6: Algoritmo Mark & Sweep sotto l'ipotesi generazionale

Lo heap a disposizione viene diviso in tre aree:

- Area Young (in uso)
- Area Young (non in uso) che svolge il ruolo dell'heap inutilizzato
- Area Old (in uso sempre)

I dati che capiamo essere vivi a lungo vengono posti in Area Old (o Major). La versione base ha le tre aree grandi uguali in realtà l'area old viene inizializzata con capacità maggiore rispetto alle aree young. Per reclamare memoria, le aree young hanno delle metodologie che vedremo.

Per capire se un tipo di dato è di tipologia 1 o 2 (eq. (1)) possiamo vedere se muore in fretta spostandolo tra gli heap diverse volte. Se il dato sopravvive diverse volte al Mark & Sweep (e.g., sopravvive due volte) allora viene reputato di tipologia 1 e viene posto nell'area old.

L'implementazione è la seguente: alla prima passata spostiamo un dato dall'area minor (in uso) all'altra area minor, se serve muoverlo nuovamente si sposta nell'area major. Per capire se un dato è già stato mosso in precedenza possiamo utilizzare l'HP. Tutte le allocazioni recenti (nuove) avvengono sopra l'HP, tutti i dati che si trovano sotto l'HP terminata una minor collection sono già stati spostati almeno una volta. Possiamo usare un nuovo puntatore il Deep Water Mark (DWP) per tenere traccia del punto sotto il quale i dati sono sopravvissuti. Finita la minor collection l'HP mano a mano si sposterà verso l'alto e quando il GC riparte tutti i dati che sono sotto il DWP vengono spostati nell'heap major mentre quelli sopra vengono spostati nello heap minor che diventerà in uso per questo ciclo di GC.

Cosa succede se a un certo punto la minor collection non recupera nulla e l'heap major è pieno e non ci sta più nulla? Si effettua una passata minor e il major diventa l'heap young non in uso (che è considerato vuoto). L'alternativa è fare un defrag in-place (come faceva Windows di un tempo).

Analizziamo ora il nuovo algoritmo e il suo costo computazionale. Innanzitutto non vogliamo seguire i puntatori che dalle radici ci portano allo heap major. Ci interessa visitare il grafo nello heap young in uso. Per non visitare lo heap major basta memorizzare il range dei suoi indirizzi così da capire se un puntatore punta a un dato nello heap major. Così siamo lineari nel numero di dati vivi nello heap young. Se un dato nello heap young ha un puntatore allo heap major allora non dobbiamo seguire tale arco!

Il costo computazionale in tempo è $O(\#radici + \#dati \text{ vivi nello heap minor})$. L'ipotesi generazionale ci dice che il numero di dati vivi nello heap young è piccolo.

Però se abbiamo un arco dallo heap old/major allo heap young rischiamo di deallocare qualcosa che però è referenziato ([Figura 7](#)). Ma è possibile che un dato old punti a un dato young? non è possibile avere un puntatore a un dato young da un dato old se il linguaggio non ha mutabilità.

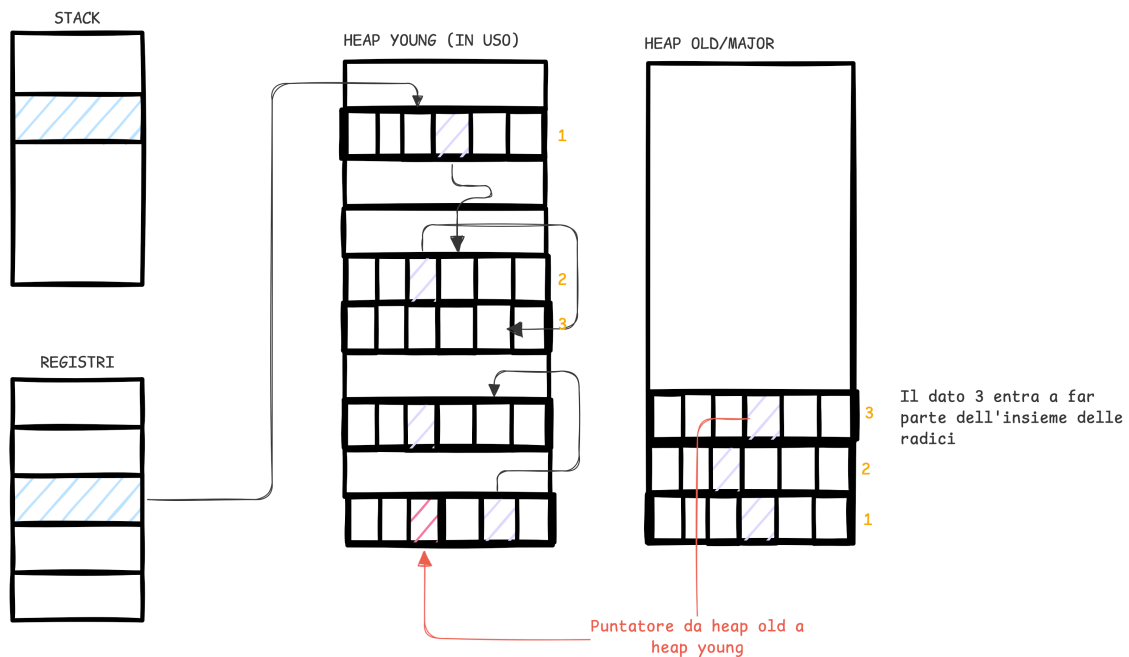


Figura 7: Estensione del concetto di radici per il corretto funzionamento dell'algoritmo Mark & Sweep

- Se un linguaggio non ha mutabilità non ci sono archi da old a young
- Se un linguaggio ha mutabilità dobbiamo estendere il concetto delle radici

Le radici comprendono anche una lista di celle nello heap major.

L'assegnamento di un puntatore diventa un'operazione complessa perché dobbiamo mantenere le celle nell'heap major che puntano a celle young e dobbiamo modificare la lista nel caso il puntatore venga modificato. Introduciamo un overhead chiamato **write barrier**.

Un'ulteriore lista di radici si configura quando ad esempio vogliamo far interagire due linguaggi di programmazione. A questo scopo serve un binder ovvero un programma che fa interagire i due linguaggi e i loro gestori della memoria. Se consideriamo C (che non ha gestore della memoria) e lo chiamiamo da Erlang, serve la garanzia che l'output prodotto dal codice C, se ancora referenziato dal codice C sia aggiunto alle radici, in modo tale da conoscere se è possibile o meno manipolare il dato in maniera safe con il gestore della memoria di Erlang. Se modificassimo il dato ancora referenziato dal codice C a quel punto C lo perderebbe in quanto il puntatore non punterebbe più al dato. Se C può accedere al dato allora lo si blocca in memoria. Un dato può essere reachable solo a partire da strutture dati del C [4].

Ad esempio in OCaml la GC richiede in genere il 25-35% del CPU-time di un programma.

5. Programmazione Parallela in Erlang

Vediamo come parallelizzare i programmi in Erlang, distribuendo la computazione su diversi core. Per farlo useremo un'implementazione dell'algoritmo QuickSort. Scriveremo un QuickSort funzionale che è leggermente diverso dal QuickSort sequenziale. Il QuickSort sequenziale è in-place mentre quello funzionale no e questo cambia tutto.

Per parallelizzare il lavoro un attore può spawnare dei figli che lavorino, in questo caso viene spawnato un solo figlio che lavora su una parte della lista, l'altra parte della lista la gestisce self. In C se questo avvenisse attraverso con una fork i due processi avrebbero tutto identico (stessa RAM stessi FileDescriptor, ...) l'unica cosa che cambia è che la fork restituisce al padre il PID

del figlio e al figlio -1. In Erlang il nuovo attore creato con spawn è completamente disgiunto dal padre, non condivide nulla ad eccezione dei dati che sono utilizzati nella funzione che l'attore esegue (psort(L2), quindi L2 sarà l'unico dato condiviso). I due attori avranno 2 copie differenti di L2 in memoria. I motivi di questa decisione è legata alla design decision let it fail [Sezione 2.1](#).

È meglio che sia il padre o il figlio a processare L2? dipende, parallelizzare la list-comprehension non sempre è buono, potrebbe essere addirittura più efficiente passare tutta la lista al figlio anziché dividere la lista. Parallelizzare la list-comprehension ha il costo nascosto che comunque tutta la lista viene copiata⁴. Vediamo l'implementazione di QuickSort in versione parallela (psort).

```
1  -module(qsort).
2  -export([main/0, qsort/1, psort/1]).
3
4  qsort([]) -> [] ;
5  qsort([H|L]) ->
6      L1 = [ X || X <- L, X <= H ],
7      L2 = [ X || X <- L, X > H ],
8      qsort(L1) ++ [ H | qsort(L2) ].
9
10 psort([]) -> [] ;
11 psort([H|L]) ->
12     L1 = [ X || X <- L, X <= H ],
13     L2 = [ X || X <- L, X > H ],
14     SELF = self(),
15     spawn(fun () -> SELF ! psort(L2) end),
16     psort(L1) ++ [ H | receive SL2 -> SL2 end ].
17
18 benchmark(F, L) ->
19     T = [ timer:tc(?MODULE, F, L) || _ <- lists:seq(1,10) ],
20     lists:sum([X || {X,_} <- T]) / (1000 * length(T)).
21
22 main() ->
23     L = [ rand:uniform(10000) || _ <- lists:seq(1,10000) ],
24     io:format("Sequenziale: ~p~n", [benchmark(qsort, [L])]),
25     erlang:garbage_collect(). % forza una minor collection
```

Ma il codice iniziale è chiaramente errato perché il padre non sa in che ordine arrivano i messaggi dai figli che spawna!

```
1  1> c(qsort).
2  {ok,qsort}
3  2> qsort:psort([1,3,27,2,8,4,105,12,29,7,2]).
4  [1,2,2,3,4,29,105,8,12,27,7]
```

⁴I figli in Erlang condividono con i padri anche la tabella di conversione degli atomi ([Sezione 2.4](#))

Per essere sicuri che un messaggio sia la risposta a una domanda possiamo usare dei nonces utilizzando la primitiva `make_ref()`. Non è semplice divinare un nonce così come è altamente improbabile generare due nonces simili in un arco temporale piccolo⁵.

Questo risolve il problema

```
1 1> c(qsort).
2 {ok,qsort}
3 2> qsort:psort([1,3,27,2,8,4,105,12,29,7,2]).
4 [1,2,2,3,4,7,8,12,27,29,105]
```

Shell

Ma stiamo introducendo un nuovo problema, ora il pattern è diventato `{REF, SL2}` ma cosa ci assicura che riceveremo un messaggio proprio con quella REF. Inoltre ricordiamoci che l'ordine nella mailbox non è rilevante, se siamo pronti a ricevere REF, ... e arrivano altri messaggi che non matchano, questi non saranno processati e tornano in coda. In questo modo garantiamo l'ordine.

```
1 -module(qsort).
2 -export([main/0, qsort/1, psort/1]).
3
4 qsort([]) -> [] ;
5 qsort([H|L]) ->
6   L1 = [ X || X <- L, X <= H ],
7   L2 = [ X || X <- L, X > H ],
8   qsort(L1) ++ [ H | qsort(L2) ].
9
10 psort([]) -> [] ;
11 psort([H|L]) ->
12   L1 = [ X || X <- L, X <= H ],
13   L2 = [ X || X <- L, X > H ],
14   SELF = self(),
15   REF = make_ref(),
16   spawn(fun () -> SELF ! {REF, psort(L2)} end),
17   psort(L1) ++ [ H | receive {REF, SL2} -> SL2 end ].
18
19 benchmark(F, L) ->
20   T = [ timer:tc(?MODULE, F, L) || _ <- lists:seq(1,10) ],
21   lists:sum([X || {X,_} <- T]) / (1000 * length(T)).
22
23 main() ->
24   L = [ rand:uniform(10000) || _ <- lists:seq(1,10000) ],
25   io:format("Sequenziale: ~p~n", [benchmark(qsort, [L])]),
26   erlang:garbage_collect(),
27   io:format("Parallelo: ~p~n", [benchmark(psort, [L])]),
```

Erlang

⁵Curiosamente Erlang non ha nessun controllo o primitive per la sicurezza, tutti i nodi sono trusted perché in origine Erlang girava su rete privata (telefonica)

```
28 erlang:garbage_collect(). % forza una minor collection
```

Notiamo però come non abbiamo un vantaggio dalla parallelizzazione immediato.

```
1 5> qsort:main().
2 Sequenziale: 3.6489
3 Parallelo: 5.3452
```

Shell

Evidentemente c'è un overhead nascosto che annulla i guadagni della parallelizzazione. Ad esempio se la lista è troppa piccola c'è ricorsione inefficiente, converrebbe ordinare direttamente la lista senza spawnare un nuovo attore. Un altro problema è dovuto al fatto che abbiamo un overhead relativo allo scheduling degli attori perché lo spawn di nuovi attori determina un costo nella loro gestione di cpu-time. Non abbiamo nessun vantaggio ad avere 10000 attori se abbiamo 12 core ad esempio.

L'idea è di limitare il numero di spawn per sfruttare al massimo la parallelizzazione senza l'overhead dello scheduling.

Nella soluzione precedente non abbiamo un controllo fine sugli attori, abbiamo solo il parametro N che ne controlla il numero massimo. Proviamo ad utilizzare un pool di job. Un job è la descrizione della computazione da svolgere, viene memorizzato in una struttura dati e man mano la computazione avviene per mezzo di attori che accedono al pool. Usiamo un attore che agisce come scheduler perché altrimenti il pool apparterebbe solo alla funzione e dovremmo passarlo in giro, serve però il PID dello scheduler. Non è una buona idea passare in giro il PID dello scheduler. Chi chiama la jsort se ha il PID dello scheduler, o è un discendente dello scheduler oppure esiste un discendente comune, il discendente comune in locale è la shell. In un sistema distribuito non funziona. Non ci sarebbe un antenato comune come è la shell in locale. Dobbiamo utilizzare un nome logico, ad esempio per contattare un server web ci serve ip+port. Ci serve un'operazione bind che associ un nome logico a un PID.

Introduciamo la funzione register per effettuare il binding tra un PID a un atomo. Questo è uno dei pochi costrutti imperativi in Erlang. Questo comando scrive da qualche parte in memoria sulla BEAM il binding. Esiste ovviamente anche il comando inverso unregister. La funzione può fallire tirando eccezione se è già presente il binding tra l'atomo e qualcos'altro (come per le porte).

```
1 SCHEDULER = spawn(?MODULE, scheduler, []),
2 register(scheduler, SCHEDULER).
```

Erlang

In presenza di fallimenti del server il PID cambia, quindi cosa facciamo? uccidiamo tutti i processi che usano il server? questo è un altro motivo per non passare in giro il PID di un attore. Non c'è questo problema se usiamo i nomi logici! implementiamo uno scheduler.

```
1 -module(qsort).
2 -export([main/0, jsort/1, worker/0, scheduler/1]).
3
4 scheduler(JOBS) ->
5   receive
6     {require, _, _, _} = JOB -> scheduler([JOB|JOBS]) ;
7     {get, REF, WORKER} when JOBS /= [] ->
8       [JOB|L] = JOBS,
```

Erlang

```

9      WORKER ! {REF, JOB},
10      scheduler(L)
11  end.
12
13  worker() ->
14      REF = make_ref(),
15      scheduler ! {get, REF, self()},
16      receive
17          {REF, {require, PID, REF2, F}} ->
18              PID ! {REF2, F()},
19              worker()
20      end.
21
22  jsort([]) -> [] ;
23  jsort([H|L]) ->
24      L1 = [ X || X <- L, X <= H],
25      L2 = [ X || X <- L, X > H],
26      JOB = fun () -> psort(1, L2) end,
27      SELF = self(),
28      REF = make_ref(),
29      scheduler ! {require, SELF, REF, JOB},
30      SL1 = jsort(L1),
31      SL2 = receive {REF, RES} -> RES end,
32      SL1 ++ [H | SL2].
33
34  benchmark(F, L) ->
35      T = [ timer:tc(?MODULE, F, L) || _ <- lists:seq(1,10) ],
36      lists:sum([X || {X,_} <- T]) / (1000 * length(T)).
37
38  main() ->
39      SCHED = spawn(?MODULE, scheduler, [[]]),
40      register(scheduler, SCHED),
41      [ spawn(?MODULE, worker, []) || _ <- lists:seq(1,24) ],
42      L = [ rand:uniform(10000) || _ <- lists:seq(1,10000) ],
43      io:format("Jsor: ~p~n", [benchmark(jsort, [L])]),
44      erlang:garbage_collect(),
45      unregister(scheduler).

```

Notiamo che la chiamata `JOB = fun () -> psort(1, L2) end`, dovrebbe essere `JOB = fun () -> jsort(L2) end`, ma abbiamo notato che causa deadlock quando tutti i worker sono impegnati. All'interno di `jsort(L2)`, viene chiamato `scheduler ! {require, ..., fun () -> jsort(...)} end` chiedendo di elaborare un'altra parte della lista. Poi si blocca in un `receive` in attesa del risultato. Ma tutti i worker sono già occupati, quindi nessuno è libero per prendere questi nuovi job. Lo scheduler ha lavoro da assegnare, ma nessuno chiede (`{get, REF, self()}`), perché tutti i worker sono in attesa dei job che loro stessi hanno creato.

Possiamo risolverlo con qualcosa del genere

```
1  SL2 =
2    LOOP = fun() ->
3      receive
4        {REF, RES} -> RES
5      % dopo 10ms
6      after 10 ->
7        scheduler ! fun() -> SL2 = LOOP(), SL1 ++ [H | SL2] end
8      end
9    end,
10   LOOP,
```

6. Hot Code Swap in Erlang

Abbiamo visto la politica let it fail di Erlang [Sezione 2.1](#). La BEAM permette di modificare il codice in produzione di un sistema che utilizza Erlang. L'hot code swap è la possibilità di modificare il codice del sistema senza downtime. L'hot code swap non utilizza la politica let it fail, ovvero gli attori non perdono l'esecuzione e vengono ripristinati con la nuova versione del codice, ma viene utilizzato un approccio differente che vedremo a breve. Ci saranno addirittura momenti di transizione con versioni vecchie del codice che comunicano con versioni nuove del codice (e.g., in un sistema distribuito). Notiamo alcuni dettagli:

- I tipi di dato possono cambiare da una versione all'altra
- Il formato e il numero dei messaggi possono cambiare da una versione all'altra.
- Lo stack delle chiamate di funzione puntano al vecchio codice (e.g., gli indirizzi di ritorno delle funzioni sono sbagliati).

Modificare il codice di produzione non è ovviamente una pratica comune e necessita numerosi test. Vediamo come scrivere codice in Erlang che permetta l'hot code swap.

```
1  -module(hot_swap).
2  -export([]).
3
4  loop(N,M) ->
5    receive
6      {add1, X} -> loop(N+X, M) ;
7      {add2, X} -> loop(N, M+X) ;
8    end.
```


È importante che la funzione del servizio sia tail ricorsiva, in questo modo lo stack è vuoto! L'hot code swap non funziona semplicemente sostituendo il codice in memoria, perché se siamo all'interno dell'esecuzione di una funzione la romperemmo. Deve essere il programmatore che decide quando effettuare hot code swap. La prima volta che un modulo viene usato la BEAM carica in memoria il codice del modulo. Quando il modulo viene ricaricato, la BEAM mantiene entrambe le versioni. La BEAM mantiene solo due copie in memoria. Anche dopo avere caricato il nuovo codice, il vecchio codice continua a girare, fino a quando non viene esplicitato dal programmatore. Per specificare quale versione del codice la BEAM deve utilizzare esistono due diverse sintassi per questo scopo:

- chiamare la funzione con il suo nome come abbiamo sempre fatto func_name
- per invocare l'ultima versione del codice usiamo ?MODULE:func_name

```

1  -module(hot_swap).
2  -export([loop/2, new_loop/2]).
3
4  loop(N,M) ->
5      io:format("Versione 1: ~p, ~p~n", [N, M]),
6      receive
7          {add1, X} -> loop(N+X, M) ;
8          {add2, X} -> loop(N, M+X) ;
9          upgrade -> ?MODULE:new_loop(N,M)
10     end.
11
12 new_loop(N,M) -> loop(N,M).


```

 Erlang

```

1  3> PID = spawn(hot_swap, loop, [2,3]).
2  Versione 1: 2, 3
3  <0.96.0>
4  4> PID ! {add1, 5}.
5  Versione 1: 7, 3
6  {add1,5}

```


 Shell

Ora vogliamo implementare una nuova versione della funzione loop. Ricordando che non tutto il sistema distribuito viene aggiornato simultaneamente, quindi serve essere in grado gestire sia i messaggi nuovi, sia i messaggi vecchi (utilizzando una sorta di versione 1.5).

```

1  9> c(hot_swap).
2  {ok,hot_swap}
3  10> PID ! {add1, 5}.
4  Versione 1: 12, 3
5  {add1,5}
6  11> PID ! {add1, 5}.
7  Versione 1: 17, 3
8  {add1,5}
9  12> PID ! {add1, 5}.
10 Versione 1: 22, 3
11 {add1,5}
12 13> PID ! upgrade.
13 Versione 1.5: 22, 3
14 upgrade
15 14> PID ! {add, 1, 5}.
16 Versione 1.5: 27, 3
17 {add,1,5}
18 15> PID ! {add, 2, 3}.
19 Versione 1.5: 27, 6


```

 Shell

```
20 {add,2,3}
```


Infine possiamo transire definitivamente alla versione 2 che gestisce un solo tipo di messaggi (una tripla con istruzione e due valori).

```
1  -module(hot_swap).
2  -export([loop/1, new_loop/1]).
3
4  loop({N,M}) ->
5      io:format("Versione 1.5: ~p, ~p~n", [N, M]),
6      receive
7          {add, 1, X} -> loop({N+X, M}) ;
8          {add, 2, X} -> loop({N, M+X}) ;
9          upgrade -> ?MODULE:new_loop({N,M})
10     end.
11
12 new_loop({N,M}) -> loop({N,M}).
```

 Erlang

Per effettuare l'hot code swap nuovamente eseguiamo:

```
1  16> c(hot_swap).
2  {ok,hot_swap}
3  17> PID ! {add, 2, 3}.
4  Versione 1.5: 27, 9
5  {add,2,3}
6  18> PID ! {add, 2, 3}.
7  Versione 1.5: 27, 12
8  {add,2,3}
9  19> PID ! {add2, 3}.
10 Versione 1.5: 27, 15
11 {add2,3}
12 20> PID ! upgrade.
13 Versione 1.5: 27, 15
14 upgrade
15 21> PID ! {add2, 3}.
16 {add2,3}
17 22> PID ! {add,2, 3}.
18 Versione 1.5: 27, 18
19 {add,2,3}
```


 Shell

7. Failure Handling in Erlang

Quando abbiamo descritto il paradigma let it fail ([Sezione 2.11](#)), abbiamo visto come un attore che incontra un guasto viene terminato e di conseguenza tutti gli attori che collaborano con l'attore vengono terminati. Ma in un sistema distribuito chi sono gli attori che collaborano? sono gli attori che attendono messaggi dall'attore (messaggi che non verranno mai processati) o attori che devono mandare messaggi all'attore (messaggi non arriveranno mai). Non è semplice sapere a priori chi

sono gli attori che collaborano, di fatto è un problema indecidibile. In Erlang è il programmatore che esplicitamente tramite primitive del linguaggio dichiara chi sono gli attori che collaborano. È inoltre possibile definire un sistema gerarchico di controllo sul paradigma let it fail. Dobbiamo vedere la collaborazione come relazione simmetrica e transitiva. Se *A* collabora con *B* allora *B* collabora con *A* e se *A* e *B* collaborano e *B* e *C* collaborano, allora *A* e *C* collaborano.

```
1  -module(link).
2  -export([test/0, sleep/1, make_link/2]).
3
4  sleep(N) -> receive after N -> ok end.
5
6  make_link(SHELL, 0) ->
7    io:format("0: send to SHELL ~p~n", [SHELL]),
8    _ = 1 / 0,
9    SHELL ! self(),
10   sleep(200000),
11   io:format("0: Termino~n", []),
12   ok;
13
14  make_link(SHELL, N) ->
15    process_flag(trap_exit, true),
16    spawn_link(?MODULE, make_link, [SHELL, N-1]),
17    sleep(6000),
18    receive
19      MSG -> io:format("~p: Ho ricevuto ~p~n", [N,MSG])
20    after 1000 -> ok
21    end,
22    io:format("~p: Termino~n", [N]).
23
24  % Crea una catena di 10 attori ognuno linkato al precedente
25  test() ->
26    spawn(?MODULE, make_link, [self(), 10]),
27    receive
28      PID ->
29        io:format("Shell ha ricevuto da ~p~n", [PID]),
30        % exit(PID, kill),
31        ok
32    end.
```


 Erlang

In molti linguaggi è sconsigliato uccidere un attore o un thread, in Erlang invece è pratica comune prevista dal linguaggio e questo scatena il meccanismo let it fail. Per uccidere un attore esiste la primitiva `exit(PID)` dove `PID` è il process id dell'attore che vogliamo uccidere. La primitiva `link` è una primitiva bidirezionale, idempotente e transitiva che lega in una catena due attori. Quando viene ucciso un attore sulla catena muoiono tutti gli attori sulla catena. A livello di BEAM questo meccanismo viene implementato con dei messaggi nascosti (non catturati dalla `receive`) che gli attori si scambiano per uccidersi a vicenda. Se pensiamo a `spawn` come alla `fork` del C, e alla

link come alla kill di Unix potrebbe succedere qualcosa di male, si aprirebbe una race-condition (infatti esiste la primitiva `spawn_link` che rende atomiche le due operazioni unendole in una sola). La race-condition si verificherebbe se un attore uccide un processo al quale ci vogliamo linkare, prima che il link avvenga. Il processo non sarebbe notificato e non partirebbe la “catena della morte” dato che il link non è avvenuto.


Possiamo in maniera arbitraria scegliere come i diversi processi possono gestire i segnali di exit (i messaggi nascosti), ad esempio `process_flag(trap_exit, true)` evita che l’onda di exit si propaghi lungo la catena catturando il segnale di exit. Evitare di essere uccisi quando un attore vuole ucciderci potrebbe essere problematico. Se un attore fa `trap_exit` e non permette a nessuno di essere ucciso allora consuma risorse inutilmente e se per qualche motivo si trova in error loop è un problema. Ci serve un messaggio che non è processabile come lo è invece `trap_exit` che uccida gli attori. Esiste infatti `exit(PID, kill)` che non permette di essere parato. Ma in questo modo se in un nodo abbiamo tutti gli attori linked buttiamo giù l’intero nodo se nessuno può gestire questo messaggio. Quando un attore viene ucciso con `kill` propaga il segnale mandando però un messaggio `killed` agli attori linkati e `killed` può essere catturato se attivo `process_flag(trap_exit, true)`.

```
1 exit(PID, kill) % equivale al -9
2 exit(PID, killed) % può essere catturato
3 exit(PID, normal) % terminazione normale
```

 Erlang

L’operazione `link` è tipicamente usata nel seguente scenario: dati due attori, il corrente e PID, il corrente vuole richiedere un servizio a PID dal quale vuole una risposta, come fare?

```
1 sync_call(REQUEST, PID, TIMEOUT) ->
2   REF = make_ref(),
3   link(PID),
4   PID ! {REF, REQUEST},
5   receive
6     {REF, ANSWER} -> ANSWER
7   after TIMEOUT -> exit(timeout)
8   end,
9   unlink(PID),
10  RES.
```

 Erlang

Serve la REF come abbiamo visto per assicurarci di rispondere al messaggio corretto. Il `link` serve per assicurarci il bind client/server, se il client muore non è necessario che il server contiui a computare e quindi può morire.

Non sempre i link sono bidirezionali, se muore un client non necessariamente deve morire il server. Sarebbe un disastro. Esiste infatti un’altra primitiva `monitor` (e il corrispettivo `demonitor`) che è

- a) unidirezionale
- b) non idempotente
- c) non transitiva (up to `trap_link`)

Ad esempio un browser monitora e non linka un server web, se il browser crasha non deve uccidere il server (ovviamente). Oppure se uso due librerie B e C, sia B che C usano una libreria A. Tutte le

librerie sono implementate come attori (attori A,B e C). Poiché sia B e C usano A, sono in qualche modo collegati ad A. Se A muore muoiono B e C, quindi anche in questo caso usiamo monitor.

Link vs monitor:

- link e monitor prendono il PID dell'altro attore
- link non ritorna nulla; monitor ritorna un handle (identificatore univoco della connessione utile per fare demonitor)
- unlink prende il PID: demonitor prende l'handle e il PID
- trap_exit funziona solo con link.
- I messaggi legati al monitoring sono sempre in user space e non engono propagati, quando si monitora un nodo che muore si riceve un messaggio {'DOWN', Ref, process, PID, Reason} che può essere gestito come un normale messaggio. Il meccanismo di propagazione automatica non esiste nel monitor. Non serve fare trap_exit.
- Chiamare monitor più volte crea più monitor indipendenti che invieranno vari {'DOWN', Ref, process, PID, Reason}.

Per i sistemi distribuiti è possibile effettuare monitoring di un singolo nodo. Un nodo è una BEAM in esecuzione.

```
1 erl -sname pippo
2
3 (pippo@Mac)1> nodes().
4 []
5 (pippo@Mac)2> net_adm:ping(pluto@Mac).
6 pong
7 (pippo@Mac)3> nodes().
8 [pluto@Mac]
```

```
1 erl -sname pluto
2
3 (pluto@Mac)2> register(shell, self()).
4 true
5 (pluto@Mac)3> shell ! ciao.
6 ciao
7 (pluto@Mac)8> shell ! ciao.
8 ciao
9 (pluto@Mac)9> shell ! ciao.
10 ciao
11 (pluto@Mac)10> flush().
12 Shell got ciao
13 Shell got ciao
14 ok
```

La prima volta che un nodo manda un messaggio a un altro nodo può farlo usando una coppia con atomo e nome remoto. In questo modo mandiamo anche il PID del nodo pippo al nodo pluto. Possiamo salvare il PID e usarlo per comunicare.

```
1 (pippo@Mac)4> {shell, pluto@Mac} ! {ciao, self()}.
```

```

2 {ciao,<0.90.0>}
3
4 (pluto@Mac)11> flush().
5 Shell got {ciao,<10964.90.0>}
6
7 (pluto@Mac)12> PID = receive {ciao, PID} -> PID end.
8 <10964.90.0>
9
10 (pluto@Mac)14> PID ! saluto.
11 (pippo@Mac)6> flush().
12 Shell got saluto
13 ok

```

Quando si collegano due nodi in background parte un meccanismo di monitoring automatico che controlla lo stato di connettività. Quando due nodi vengono collegati ricevono l'accesso ai conoscenti degli altri nodi.

Una funzione molto carina è poter spawnare un attore su un nodo remoto (con qualche workaround). Come avevamo già accennato Erlang non implementa meccanismi di sicurezza (se non minimi) perché nasce per girare su reti chiuse. Tutti i messaggi passano in chiaro. Il meccanismo di sicurezza minimo si basa sulla condivisione di una chiave, ovvero quando due cluster di nodi richiedono l'unione è necessario che condividano una chiave che si trova in `~/erlang.cookie` e contiene qualcosa del tipo `DQXPHXTSAUUNNXPWLIA`.

Erlang rende anche possibile trasferire attori da un nodo a un altro. Questo sicuramente è utile per bilanciare il carico tra i vari nodi e anche per prossimità geografica ai dati. Può essere conveniente trasferire un attore vicino ai dati che necessita anziché spostare i dati stessi. Per trasferire l'esecuzione su altri nodi possiamo trasferire codice su un altro attore usando `spawn` e il nodo agisce come *man-in-the-middle* inoltrando i messaggi che riceve al nuovo attore. Questo viene implementato in [Codice 10](#). Possiamo usare un messaggio speciale `comeback` che permette di ripristinare l'esecuzione sull'attore corrente.

```

1  -module(migrate).
2  -export([migrate/2, server/1]).
3
4  migrate(NODE, F) ->
5      PID = spawn(NODE, F),
6      Forward =
7          fun Forward() ->
8              receive
9                  {comeback, F} -> F() ;
10                 Msg -> PID ! Msg, Forward()
11             end
12         end,
13         Forward().
14
15  server(Dati) ->
16      receive
17          msg1 -> server(Dati+1) ;
18          {migrate, NODE} ->
19              migrate(NODE, fun () -> server(Dati) end)
20      end.

```

Codice 10: Codice Erlang per spawnare attori che eseguono codice arbitrario su nodi remoti

8. Rust

Sviluppato nel 2006 da un dipendente di Mozilla. Rust è uno dei pochi di linguaggi di sistema che possediamo oggi. Un linguaggio di sistema si definisce tale se non necessita nessun supporto esterno a run-time, si usa per programmare su bare-metal o sistemi operativi. Rust ha subito forti influenze da OCaml e C++. Il primo compilatore fu scritto in OCaml mentre ora viene compilato in Rust stesso. LLVM come backend. Ricapitolando:

- Linguaggio di sistema
- Non ha runtime, anche se in realtà ha un minimo supporto
 - System threads
 - Ma non garbage collection
- Zero-cost abstractions, ha costrutti di alto livello ma a costo zero, non viene introdotto overhead ma non si propaga
 - Polimorfismo parametrico via monomorfizzazione
 - Complesso ownership system dei dati in memoria imposto a compile-time
- Garantisce memory safety
- Concorrenza senza paura
 - Type systems (e smart pointers) minimizzano i problemi legati alla concorrenza
- Chiusure
- Algebraic Data Types e pattern matching
- No null value (option type)
- Polimorfismo parametrico bounded, possibile scrivere funzioni parametriche

- Traits (le classi inizialmente presenti sono state poi rimosse). Trait objects per dynamic dispatch.
- Moduli annidabili

8.1. Gestione della memoria

Il primo meccanismo è tipico di Rust è un sistema di ownership e borrowing in lettura e scrittura che si basa sulla logica lineare. Il secondo meccanismo si basa sul concetto di smart pointer, presente anche in C++, rendendo esplicite le invarianti che il programmatore ha in mente.

8.1.1. Ownership

Il concetto di ownership è il seguente: ogni cella di memoria sullo heap ha un owner ovvero colui che è responsabile della sua deallocazione. Questo garantisce che essa sia deallocata una e una sola volta. Quando una cella viene allocata sullo heap viene creato un puntatore, la ownership del puntatore viene assegnata a una variabile sullo stack. Quando invece il puntatore viene assegnato a un'altra cella sullo heap, la cella sullo heap ne diventa l'owner. Quando un owner viene deallocato le celle ricorsivamente possedute vengono deallocate. Il compilatore inserisce automaticamente le istruzioni di deallocazione (cool!).

Una cella sullo heap ha sempre uno e un solo owner. Quando avvengono assegnamenti oppure avviene un passaggio come parametro di funzione della variabile/cella, l'ownership viene trasferita (move). Se una variabile perde l'ownership non può più essere utilizzata.

La deallocazione delle variabili avviene quando si incontra il simbolo }. Le variabili di default sono immutabili.

```

1  fn main() {
2      let x = 4; // x è unboxed quindi sullo stack
3      let s = String::from("ciao"); // s sullo stack punta a una stringa nello
                                     heap, s è owner del dato
4      let y = x; // non succede nulla di male dato che x è unboxed
5
6      // sarebbe una move e s per il sistema di tipi s è morta perché t diventa
                                     owner del dato sullo heap
7      // let t = s;
8      let t = String::from("ciao");
9      println!("x = {}, y = {}", x, y);
10
11     println!("s = {}, t = {}", s, t);
12 } // per ogni owner il compilatore inserisce la deallocazione

```

A ritroso dopo la parentesi } il sistema di tipi determina chi è owner o meno e aggiunge il codice della deallocazione. Interessante il commento `let t = s` che viene reputata come un'operazione move quindi decommentando non è possibile usare `s` nella stampa, il compilatore non compila.

Anche chiamare una funzione trasferisce la ownership

```

1  fn main() {
2      let s1 = gives_ownership();
3      let s2 = String::from("hello");
4      let s3 = takes_and_gives_back(s2);

```



```

5 } // s1 e s3 sono deallocate
6
7 // scrivere -> String implica trasferire l'ownership
8 fn gives_ownership() -> String {
9     let some_string = String::from("hello"); // alloca sullo heap una stringa
        e assegna un puntatore alla variabile
10    some_string // ownership trasferita, restituirlo vuol dire copiarlo nello
        stack frame delle chiamate
11 }
12
13 fn takes_and_gives_back(a_string: String) -> String { // onwership taken
14     a_string // owneship transferred
15 }

```

Il meccanismo dell'ownership risponde alla domanda “chi deve deallocare il dato?”.

8.1.2. Borrowing

Il meccanismo del borrowing risponde alla domanda “chi può accedere al dato?” Finchè lavoriamo in sola lettura le references sono una semplice comodità, si potrebbe cedere l'ownership per poi riceverla nuovamente. Sono necessarie per la modifica/scrittura. L'owner può prestare la stessa variabile in lettura in parallelo.

- `&x` è una reference al contenuto di `x`
- `&mut x` è una reference al contenuto di `x` che permette di modificarne il contenuto, `x` deve essere dichiarato con `let mut` (mutabile). Volendo `&` e `&mut` sono costruttori di tipo.
- Se `x` ha tipo `T`, `&x` ha tipo `&T` e `&mut x` ha tipo `&mut T`
- Ricevere una reference di una variabile implica fare borrowing della variabile
- No data races (anche concorrentemente)
 - Se una variabile è borrowed mutably, nessun altro borrow è possibile e l'owner è frozen, ovvero non può accedere alla variabile finchè il borrowing non termina (tutto questo a compile-time)
 - Se l'ownership è mutable ma viene borrowed in lettura l'owner è frozen (altrimenti race-condition)

Vediamo quando termina il borrowing

```

1 fn main() {
2     let x = 4;
3     let y = &x; // unboxed
4     let t = String::from("ciao"); // immutable ownership
5     let s = &t; // borrowing immutabile
6     println!("x = {}, y = {}", x, y);
7     println!("s = {}, t = {}", s, t);
8 } // end of borrowing (s goes out of scope) and end of ownership (t goes out
    of scope and the string is deallocated)

```

```

1 fn main() {
2     let mut x = 4;

```

```

3   let y = &x;
4   x = 5; // error assignment to borrowed x
5   }
6
7   fn main() {
8       let mut x = 4;
9       { let y = &x; } // ok since the borrow ends with the inner block
10      x = 5;
11  }
12
13  fn main() {
14      let x = 4;
15      let z = &mut x; // error cannot borrow immutable local variable as
                        mutable
16  }

```

Ma il compilatore non può sforzarsi e cercare di capire che il programma 1 e 2 siano identici e compilare il programma 1 come il programma 2. Il problema è sempre un problema indecidibile, Rust comunque ci prova provando ad accomodare i pattern ripetuti dagli sviluppatori cercando di adeguare il compilatore. In generale i linguaggi o definiscono regole statiche sempre prevedibili oppure cercano di compilare adeguandosi.

```

1   fn increment(x: &mut i32) { // syntactic sugar at work! See later
2       *x = *x + 1;
3   }
4
5   fn main() {
6       let mut x = 4;          // x has mutable ownership
7       increment(&mut x);      // mutable borrows begins and ends
8       x = x + 1;              // so x can be used again here
9       println!("x = {}", x);
10  }

```

 Rust

8.1.3. Lifetime

Ma quando avviene effettivamente la fine del borrowing e come?

Chiamiamo lifetime il momento temporale simbolico in cui una cella verrà deallocata dallo heap, in prima battuta pare che lifetime sia molto simile al concetto di scope, non è così però perché abbiamo visto che possiamo trasferire ownership attraverso chiamate di funzione. In qualche modo lo scope sintattico determina un lifetime ma non sono i lifetime. Ogni reference ha di fatto due lifetime, quello della reference e quello al quale punta la reference. Non vogliamo dangling pointers! non vogliamo accedere a dati che non ci sono più. Quindi il lifetime della reference deve essere strettamente minore rispetto al lifetime del dato allocato. Quando il compilatore non è in grado di capirlo autonomamente serve esplicitare nel codice il lifetime di una reference. Il lifetime non è statico e dobbiamo inventarci qualcosa. Un orologio virtuale che fa tick a ogni istruzione assegnando un tempo logico e il programmatore specifica un lifetime in questa maniera (?), ovviamente impraticabile. La seconda strada è usare un tempo simbolico specificando le

relazioni temporali tra le variabili di lifetime α, β, γ definendo con 'a, 'b, 'c le variabili in Rust. Su queste variabili il programmatore definisce la relazione simbolica tra i lifetime ('a > 'b).

L'unico termine ground di tipo lifetime è 'static (vivo fino al termine del programma, praticamente una variabile globale). I lifetime vengono usati in due punti:

- Template astratti su variabili di lifetime (polimorfismo bounded)
- Reference tipate con il lifetime (&'a i32 è una reference a un i32 di lifetime 'a)

In molte situazioni il compilatore capisce da solo che i lifetime vanno bene e possiamo ometterli. Questo concetto prende il nome di **elisione**.

```
1 fn main() {
2     let reference_to_nothing = dangle();
3 }
4
5 fn dangle<'a>() -> &'a String {
6     let s = String::from("hello");
7     &s // ritorniamo la reference alla stringa
8 } // s viene deallocata in quanto la funzione è owner di s
9 // error the lifetime of s ends here and it should end at 'a
10 // praticamente bad closure example
```

 Rust

```
1 // i lifetime 'b e 'c devono terminare dopo il lifetime di 'a
2 fn max<'a, 'b: 'a, 'c: 'a>(x: &'b i32, y: &'c i32) -> &'a i32 {
3     std::cmp::max(x, y)
4 }
5
6 fn main() {
7     // let z; errore se dichiariamo z prima
8     let x = 4;
9     let y = 3;
10    let z;
11    z = max(&x, &y);
12    println!("max = {}", z); // qui il compilatore vuole essere sicuro che la
13    // reference a z sia un dato ancora vivo quindi che x sia viva
14 } // i lifetime finiscono in ordine inverso di dichiarazione
15 // a mano a mano che le variabili muoiono è possibile rilasciare pezzi di
16 // stack
```

 Rust

Se in qualche modo un dato sotto forma di puntatore viene salvato dentro una struttura dati (heap ad esempio) allora serve informare il compilatore annotando la struttura dati con un proprio lifetime e il rapporto coi lifetime dei puntatori che contengono, quando la struttura dati smette di essere accessibile allora lo è anche il dato

```

1 // record con un singolo campo che è una reference a una str
  (sottostringa in rust)
2 struct ImportantExcerpt<'a> {
3     part: &'a str,
4 }
5
6 fn main() {
7     // Allocando sull'heap qualcosa di tipo Stringa (in questo caso finisce
  nel data segment del programma essendo nota a compile-time)
8     let novel = String::from("Call me Ishmael. Some years ago...");
9     let first_sentence = novel.split('.')
10         .next()
11         .expect("Could not find a '.');
12     let i = ImportantExcerpt { part: first_sentence };
13 }

```

Codice 11: Dangling Pointers attraverso references

Stiamo inserendo dentro una struttura dati una sottostringa `str`. Dobbiamo esplicitare il lifetime di `str`, la struttura dati `ImportantExcerpt` è parametrica nel lifetime. Se ritornassimo `i` allora ci sarebbe un problema perché dopo `}` `novel` morirebbe così come `first_sentence` che però ha un puntatore dentro la struttura dati `ImportantExcerpt`.

```

1 struct Ref<'a, T: 'a>(&a T);
2 // tupla con un campo che contiene un valore di tipo
3 // T tale che tutte le reference in T vivono almeno
4 // quanto 'a
5
6 fn print_ref<'a, T>(t: &'a T) where
7     T: Debug + 'a {
8     println!("`print_ref`:t is {:?}", t);
9 }

```

Nell'esempio notiamo un predicato sul lifetime del tipo generico che `T` che viene accettato dalla struttura dati `Ref`. In questo caso senza le annotazioni dei lifetime il compilatore segnalerebbe che la reference non è sicuramente safe.

Per esempio una monade predica su un costruttore di tipo l'esistenza delle funzioni `bind` e `return` ma le funzioni non sono definite sul costruttore di tipo, bensì sull'applicazione del costruttore di tipo al tipo concreto (`m a`).

I teorici dicono che Rust è funzionalmente completo ma non alitmicamente, sostanzialmente non tutto il codice corretto è tipabile.

8.2. Slices

Una slice (la `str` incontrata in precedenza) è uno smart pointer per fare borrowing mutabile o meno di una parte di una struttura. In quanto smart pointers le slices hanno lifetimes. Il compilatore deve essere sicuro che può fare borrowing o meno di una parte della struttura dati e vedremo come.

Esempi di slices sono `str` (string slices) o `&[T]` (vector slices). Internamente le slices sono rappresentate come:

- Un puntatore (alla struttura dati)
- Numero Byte (quanti byte utilizzati)
- Capacità residua (spazio rimanente nell'array se usiamo un array di caratteri).

```
1 fn analyze_slice(slice: &[i32]) {  
2     println!("First element of the slice: {}", slice[0]);  
3     println!("The slice has: {} elements", slice.len());  
4 }  
5  
6 fn main() {  
7     // Fixed-size array (type signature is superluos)  
8     let xs: [i32; 5] = [1,2,3,4,5];  
9  
10    // Slice with the last 3 elemetns  
11    analyze_slice(&xs[2 .. 4])  
12 }
```

 Rust

Nei sistemi operativi moderni le pagine di codice sono flaggate read-only per incrementare la sicurezza. Ci sono vari segmenti in memoria, il code segment (read-only) e il data segment. Il data segment contiene dati globali inizializzate a compile-time. Nell'esempio `Codice 11` `String::from` estrae la stringa dal data segment e viene copiata nello heap poi si effettua una slice che viene effettivamente usata.

8.3. Chiusure

Una reference potrebbe essere memorizzata implicitamente usando una chiusura, quindi Rust anche in questo caso si vuole proteggere per evitare dangling pointers.

Rust ha le chiusure che catturano le variabili libere nella chiusura. Per cattura si intende

- Trasferimento di ownership se la chiusura è di tipo move (move |params| { body }) (le variabili non sono più accessibili nello scope esterno).
- Borrowing mutabile/immutabile altrimenti

In Rust esistono vari concetti di chiusura, ci sono Trait per le funzioni, uno per le chiusure move, uno per le chiusure con borrowing mutabile, uno per quelle immutabili.

8.4. Smart Pointers

Sono strutture dati user o system defined ispirati al mondo C. Una struttura dati per essere chiamata smart pointer deve implementare uno o più traits, i quali definiscono le operazioni possibili sullo smart pointer, ad esempio se uno smart pointer è owner di un dato, quando questo viene deallocato serve deallocare il dato puntato. I trait per gli smart pointer grossomodo ricalcano le funzioni disponibili sui puntatori. Uno smart pointer che implementa diversi Trait diventa quasi indistinguibile da un puntatore/reference. Ma perché definirli allora? per definire una serie di politiche esplicite o invarianti sulla gestione della memoria. Rendono possibili numerose operazioni a run-time (e.g., allocazione nello heap con `Box<T>`, reference counting con `RC<T>`). Inoltre l'utilizzo degli smart pointers introduce un costo computazionale a run-time a differenza di ownership/borrowing che è a compile-time). Vediamo un esempio

```

1  fn main() {
2      let x = 5           // 5 allocato sullo stack
3      let b = Box::new(5) // 5 allocato nello heap
4                          // b è uno smart pointer allocato sullo stack
5      // Il dereferencing di b è automatico attraverso un trait
6      println!("x = {}, b = {}", x, b)
7  }
8  // sia x che b escono di scope
9  // lo stack frame della chiamata viene deallocato
10 // quando b viene deallocato il dato sullo heap a cui b
11 // punta viene deallocato anche lui

```

Codice 12: Smart Pointer Box

L'invariante fondamentale associata a `Box::new` è che il dato dello heap abbia uno e un solo puntatore entrante. Non avremo mai nessun'altro puntatore a quel dato. Nel caso mostrato in [Codice 12](#) non possiamo fare borrowing di `b`, in quanto l'utilizzo di `Box` implica un unico puntatore al dato in memoria per garantire l'invariante di un unico puntatore entrante. I dati in Rust occupano un numero non uniforme di Byte (come in C). `Box<T>` permette di allocare dati di grandi dimensioni sullo heap, facendo riferimento a essi con uno smart pointer di dimensione fissata e piccola. Di fatto necessari per implementare Algebraic Data Type (ADT) ricorsivi!

```

1  // Dichiarazione errata: List può avere una dimensione
2  // arbitraria e quindi un Cons richiederebbe dimensione
3  // arbitraria
4  enum List {
5      Cons(i32, List),
6      Nil,
7  }
8  // Dichiarazione corretta, allocando le celle sullo heap
9  enum List {
10     Cons(i32, Box<List>),
11     Nil,
12 }
13
14 use List::{Cons, Nil};
15 fn main() {
16     let list = Cons(1, // cella allocata sullo stack
17                     Box::new(Cons(2, // cella allocata sullo heap
18                                   Box::new(Cons(3,
19                                                 Box::new(Nil))))));
20 } // quando list va out-of-scope, tutte le celle vengono deallocate tramite
    il metodo del trait implementato dallo smart pointer Box

```

Questa è una lista che ammette un solo puntatore entrante dato che stiamo usando `Box` quindi non possiamo sfruttare lo sharing. Se volessimo implementare sharing di sotto-liste? esiste uno smart pointer `RC<T>`.

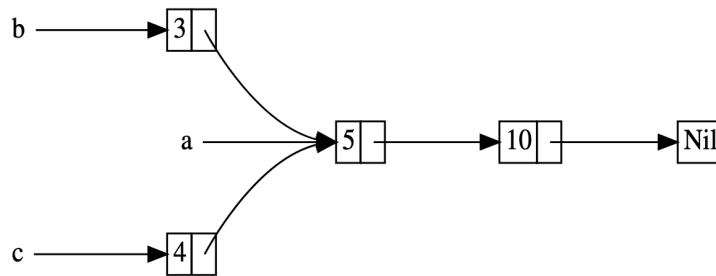


Figura 8: Implementazione di sharing di liste usando lo smart pointer `Rc<T>`. La cella 5 ha tre puntatori entranti quindi non è possibile usare `Box`.

In [Figura 8](#) notiamo che non è semplice capire a compile-time chi è l'owner della cella 5. In questo caso l'ownership è distribuita e quindi sarebbe utile usare un meccanismo di garbage collection.

```

1  enum List {
2      Cons(i32, Rc<List>),
3      Nil, // Come un atomo
4  }
5
6  use List::{Cons, Nil};
7  use std::rc::Rc;
8
9  // Codice corretto:
10 // Rc::new alloca spazio nello heap per un dato + il suo contatore di
    riferimenti
11 // Rc::clone incrementa il reference counter data una reference
12 fn main() {
13     let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
14     let b = Cons(3, Rc::clone(&a));
15     let c = Cons(4, Rc::clone(&a));
16 } // quando b e c vanno out-of-scope il counter di a viene decrementato
17 // di 2 e, raggiunto lo 0, anche a viene deallocato dallo heap

```

Codice 13:

Resta il problema del Reference Counting che non reclama le strutture dati cicliche.

Come abbiamo visto, Rust garantisce che un dato possa avere al più una reference mutabile oppure un numero arbitrario di reference in sola lettura (no data races!). `Box<T>` non ha restrizioni da questo punto di vista. `Rc<T>` permette solamente reference di sola lettura! `Rc<T>` non implementa il Trait per la concorrenza, se due thread creano una reference con lo smart pointer potrebbero creare una race-condition incrementando la reference.

Nei casi in cui vogliamo anche mutare delle celle in memoria allora dobbiamo proteggerci usando lo smart pointer `RefCell<T>`, uno smart pointer che a **run-time** verifica solo la condizione “al più un mutable reference o n immutable ones”. Solleva panic! in caso contrario, meglio affidarsi al controllo statico a compile-time di Rust. Lo fa utilizzando un meccanismo di lock.

```

1  enum List {
2      // le teste possono essere mutate e condivise, le code solo condivise
3      Cons(Rc<RefCell<i32>>, Rc<List>),
4      Nil,
5  }
6
7  fn main() {
8      let value = Rc::new(RefCell::new(5)); // value punta allo heap
9      // come coda nil condiviso e come testa il value condiviso
10     let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));
11     let b = Cons(Rc::new(RefCell::new(6)), Rc::clone(&a));
12     let c = Cons(Rc::new(RefCell::new(10)), Rc::clone(&a));
13
14     *value.borrow_mut() += 10; // per mutare la cella acquisisco un lock in
        scrittura
15
16     println!("a after = {:?}", a);
17     println!("b after = {:?}", b);
18     println!("c after = {:?}", c);
19 }

```

Lo smart pointer `Weak<T>` gestisce i cicli che possono causare memory leaks. Spesso in un ciclo vi sono dei puntatori che rappresentano che sono naturalmente forti e altri deboli, una cella è viva quando ha almeno un puntatore forte entrante; morta altrimenti. Ovvero: i deboli non contano ai fini del RC.

Esempio: alberi. Un nodo ha due puntatori forti (e.g. `Rc<Node>`) ai figli dx e sx, un puntatore debole (`Weak<Node>`) al padre. Il ragionamento è il seguente:

- Se un nodo viene sganciato da un'albero (ovvero il padre non punta più a lui) allora diventa unreachable e deve essere eliminato
- Tutti i suoi discendenti sono unreachable e vanno reclamati
- Questo anche se il nodo e tutti i discendenti si puntano fra di loro (i figli puntano al padre, ma debolmente).

Per il tipo `Weak` viene definita l'operazione `upgrade` data una weak reference e ritorna una `Rc<T>` reference se l'oggetto è ancora allocato `upgrade(r Weak<T>) -> Option<Rc<T>>`. L'operazione `downgrade` restituisce un weak pointer per un `Rc<T>` `downgrade(r Rc<T>) -> Weak<T>`.

Ci sono molti altri smart pointers, numerosi sono pensati per scenari concorrenti. `Mutex<T>` modella un semaforo e ha a disposizione `lock()` per ottenere una reference al contenuto di `T`. `Arc<T>` invece modella un reference counter atomico, utile se in scenari concorrenti siamo disposti a pagare il prezzo del meccanismo di locking.

Gli smart pointer sono implementabili in user-space.

```

1  struct MyBox<T>(T) // una tupla con un solo campo ovvero il tipo
        di dato più piccolo possibile
2
3  impl<T> MyBox<T> {
4      fn new(x: T) -> MyBox<T> { MyBox(x) }

```



```

5  }
6
7  impl<T> Deref for MyBox<T> {
8      type Target t // il dereferencing restituisce una reference al targer che
        è T
9
10     fn deref(&self) -> &T { &self.0 } // il T è il valore dell'unico campo
11 }
12
13 impl<T> Drop for MyBox<T> {
14     fn drop(&mut self) {
15         println!("Dropping MyBox<T> with data `{}`!", self.0)
16     }
17 }
18
19 fn main() {
20     let x = MyBox::new(4);
21     let y = *x; // borrow di x
22     println!("x = {}, y={}", x, y)
23 }

```

9. Generalized Algebraic Data Type (GADT)

I tipi di GADT sono un modo più furbo per tipare i tipi di dato algebrici. Un esempio è l'implementazione `printf` di C o di altri linguaggi. Non riusciamo ad implementare in user-space una funzione che accetta un numero arbitrario di parametri e che gestisca i tipi in base al tipo degli argomenti (`%s`, `%d`, ... vengono tradotti in stringhe, interi, ecc...). Riusciamo a farlo con GADT.

I GADT sono una zero-cost-abstraction, non introducono overhead a run-time. Li troviamo in forma embrionale su Kotlin e in maniera matura in OCaml.

Siamo sempre stati abituati a leggere invarianti e applicarle algoritmicamente, ma è molto difficile rispettarle a livello di tipi. Ad esempio gli alberi red black hanno invarianti sul colore dei nodi dei figli. Rispettare l'invariante tipando l'albero è difficile.


Il primo vantaggio dei GADT è che annullano il vantaggio di usare i linguaggi non tipati (Python). Vediamo quindi un esempio con Python. (Senza un type checker comunque Python non controlla se abbiamo scritto correttamente i tipi, le invarianti che intendiamo non possono avere un riscontro col sistema di tipi.)

Vogliamo rappresentare con una struttura datigli alberi di sintassi astratta delle espressioni di un linguaggio di programmazione. Un tipo di dato algebrico elenca un numero finito di forme del dato e ogni forma ha un certo numero di argomenti. Concatenando le definizioni delle singole classi otteniamo il tipo di dato `Expr`. `eval` non ha un tipo definibile perché in base al branch torna un tipo differente.

```

1  from __future__ import annotations
2  from dataclasses import dataclass
3
4  @dataclass

```

 Python

```

5  class Num:
6      n: int
7
8  @dataclass
9  class String:
10     s: str
11
12 @dataclass
13 class Mult:
14     e1: Expr
15     e2: Expr
16
17 @dataclass
18 class Concat:
19     e1: Expr
20     e2: Expr
21
22 @dataclass
23 class Eq:
24     e1: Expr
25     e2: Expr
26
27 Expr = Num | String | Mult | Concat | Eq
28
29 def eval(e: Expr) -> int | str | bool:
30     match e:
31         case Num(n):
32             return n
33         case String(s):
34             return s
35         case Mult(e1,e2):
36             # l'invariante e che le espressioni sono numeriche
37             return eval(e1) * eval(e2)
38         case Concat(e1,e2):
39             return eval(e1) + eval(e2)
40         case Eq(e1,e2):
41             # l'invariante e che gli alberi sx e dx di sintassi
42             # astratta abbiano le stesse rappresentazioni
43             return eval(e1) == eval(e2)
44
45 good    = Eq(Mult(Num(3),Num(4)),Num(5))
46 # TypeError: can only concatenate str (not "int") to str
47 bad     = Eq(Concat(String("ciao"), Num(2)), Num(3))
48

```

```

49 print(f"Good: {eval(good)}")
50 print(f"Bad: {eval(bad)}")

```

In OCaml riusciamo ad ottenere la stessa cosa annegando però la business logic.

```

1  type expr = OCaml
2    | Num : int -> expr
3    | Bool : bool -> expr
4    | Mult : expr * expr -> expr
5    | And : expr * expr -> expr
6    | Eq : expr * expr -> expr
7
8  type res = I : int -> res | B : bool -> res | E : res
9
10 let to_string = function
11   | I n -> string_of_int n
12   | B b -> string_of_bool b
13   | E -> "error"
14
15 let rec eval : expr -> res = function
16   | Num n -> I n
17   | Bool b -> B b
18   | Mult (e1, e2) -> (
19     match (eval e1, eval e2) with I n1, I n2 -> I (n1 * n2) | _, _ -> E)
20   | And (e1, e2) -> (
21     match (eval e1, eval e2) with B b1, B b2 -> B (b1 && b2) | _, _ -> E)
22   | Eq (e1, e2) -> (
23     match (eval e1, eval e2) with
24     | I n1, I n2 -> B (n1 == n2)
25     | B b1, B b2 -> B (b1 == b2)
26     | _, _ -> E)
27
28 let good = Eq (Mult (Num 3, Num 4), Num 5)
29 let bad = Eq (And (Bool true, Num 4), Num 5)

```

I GADT dato un tipo dato sul quale vorremmo esprimere un'invariante, possiamo spiegare l'invariante al compilatore cercando di rappresentare dei predicati sulla struttura dati e.g., per un albero red and black potremmo esprimere un predicato sul colore dei figli. Il tipo `expr` permette di non aver duplicazione di codice che invece avremmo avuto se avessimo definito tipi diversi per ogni combinazione.

Introduciamo le alpha espressioni. Con esse riusciamo ad esprimere dei vincoli di tipo sulle espressioni, il compilatore in questo modo ha informazioni sulle invarianti (anche se il compilato sarà lo stesso di prima. 'a indica che `expr` diventa costruttore di tipo, spesso si usa `_`, infatti in realtà 'a non viene legato da nessuna parte indica solo che viene accettato un parametro di tipo.

```

1  (* forall x, P(x) -> Q = exists x, P(x) -> Q *)

```

```

2  type 'a expr =
3      Num : int -> int expr
4  | Bool : bool -> bool expr
5  | Mult : int expr * int expr -> int expr
6  | And : bool expr * bool expr -> bool expr
7  | Eq : 'b expr * 'b expr -> bool expr
8
9  (* La cosa interessante e che il tipo dell'argomento determina il tipo
10     dell'output *)
11  let rec eval : type a. a expr -> a =
12      (* Il tipo dell'input:  a expr
13         Il tipo dell'output: a
14      *)
15      function
16      | Num n ->
17          (* Sono nel caso Num, Num dichiarato come segue
18             Num : int -> int expr
19             quindi n: int
20             input Num n ha tipo int expr
21             quindi
22             a expr = int expr
23             Per iniettivita dei costruttore di GADTs
24             a = int
25             Quindi il tipo di ritorno che era a ora diventa int!
26          *)
27          n
28      | Bool b -> b
29      | Mult(e1,e2) ->
30          (* Sono nel caso Mult, Mult dichiarato come segue
31             Mult : int -> int expr * int expr -> int expr
32             Quindi:
33             e1: int expr
34             e2: int expr
35             Mult(e1, e1) ha tipo int expr
36             Quindi
37             il tipo di ritorno che era a ora diventa int
38             Inoltre la eval ha tipo 'a expr -> 'a
39             Quindi
40             eval e1 ha tipo int
41             eval e2 ha tipo int
42          *)
43          eval e1 * eval e2
44      | And(e1,e2) -> eval e1 && eval e2
45      | Eq(e1,e2) ->

```

```

46      (* Sono nel caso Eq, Eq dichiarato come
47          Eq : forall 'b. 'b expr * 'b expr -> bool expr
48      Sia 'b un tipo ignoto ma fissato
49      Quindi
50          e1 : 'b expr
51          e2 : 'b expr
52      L'input Eq(e1,e2) ha tipo bool expr
53      Quindi
54          a expr = bool expr
55      Per inettivita dei costruttori GADTs
56          a = bool
57      Quindi il tipo di ritorno che era a ora e bool
58      Inoltre la eval ha tipo
59          'a expr -> 'a
60      Quindi:
61          eval e1 ha tipo 'b
62          eval e2 ha tipo 'b
63      *)
64      eval e1 == eval e2
65
66  let good = Eq (Mult( Num 3, Num 4), Num 5)
67  (* let bad = Eq (And (Bool true, Num 4), Num 5) *)
68
69  let _ =
70    Printf.printf "%b/n", (eval good)


```

Vogliamo scrivere una funzione che date due espressioni dica se sono uguali o diverse (utilizzabile magari in un parser di un compilatore)

```

1  let equal : type a b. a expr -> b expr -> bool =
2    fun e1 e2 ->
3      e1 == e2

```


 OCaml

Ma questo non compila, perché in OCaml non viene mantenuta informazione sui tipi a run-time (Sezione 2.4).

```

1  let same_type : type a b. a expr -> b expr -> bool =
2    fun e1 e2 ->
3      match e1 == e2 with
4      | (Num _ | Mult _), (Num _ | Mult _) -> true
5      | (Bool _ | And _, Eq _), (Bool _ | And _, Eq _) -> true
6      | _, _ -> false
7
8  let equal : type a b. a expr -> b expr -> bool =
9    fun e1 e2 ->
10     if same_type e1 e2 then

```

 OCaml

```

11     e1 = e2
12     else
13     false

```

Nonostante `same_type` la funzione `equal` continuerà a non compilare, perché a compile-time non permette di confrontare tipi senza cast, dato che OCaml è un linguaggio serio non ha i cast, eccetto `Obj.magic`. Proviamo a cambiare il tipo di output di `same_type` che ritornerà una dimostrazione del fatto che i due tipi sono uguali. Ci serve il tipo delle dimostrazioni che due tipi siano uguali.

```

1 type ('a, 'b) Eq =
2   | Refl : ('c, 'c) eq

```


 OCaml

Definiamo un tipo di dato algebrico `Eq` con due input `'a` e `'b`. L'uguaglianza è il più piccolo predicato riflessivo ($x=x$ è un predicato riflessivo). Per effettuare una dimostrazione della riflessività possiamo scrivere

```

1 let refl : type a. (a, a) eq = Refl

```


 OCaml

Per la simmetria

```

1 let symm : type a b. (a, b) eq = (b, a) eq =
2   function
3     Refl -> Refl

```


 OCaml

Possiamo riscrivere `same_type` usando le dimostrazioni prodotte

```

1 let same_type : type a b. a expr -> b expr -> (a,b) eq option =
2   fun e1 e2 ->
3     match e1 == e2 with
4     | (Num _ | Mult _), (Num _ | Mult _) -> true
5     | (Bool _ | And _, Eq _), (Bool _ | And _, Eq _) -> Some Refl
6     | _, _ -> None
7
8 let cast : type a b. (a, b) eq -> b -> a =
9   function
10    Refl ->
11      (*
12       Dato che Refl : ('c, 'c) eq
13       si ha (a,b) eq = ('c, 'c) eq
14       quindi a = b = 'c
15       e il tipo di ritorno a -> b diventa 'c -> 'c
16      *)
17    fun x -> x
18
19 let equal : type a b. a expr -> b expr -> bool =
20   fun e1 e2 ->
21     match same_type e1 e2 with
22     | Some p -> e1 = cast p e2

```

 OCaml

I GADT sono sufficientemente espressivi per generare delle dimostrazioni che bastano al compilatore per usarle per effettuare cast.

10. Storia dei Linguaggi

Esistono diversi paradigmi di programmazione ma quasi tutti i linguaggi in uso sono multi-paradigma.

Paradigma	Programma	Linguaggi
Imperativo	Sequenza di azioni	C, Pascal, C++, Java, Scala, Python, Scheme
Object oriented	Oggetti che comunicano	Smalltalk, C++, OCaml, Java, Scala, Python
Funzionale	Espressione	Haskell, OCaml, Scala, Scheme, F#

Tabella 1: Diversi paradigmi di programmazione

Negli anni '50

- Assembly: corrispondenza 1-1 con codice macchina, mancano visioni su linguaggi ad alto livello, si teme che non sarà mai possibile scrivere programmi complessi.
- FORTRAN (formula transactor): linguaggio per calcoli numerici, introduce procedure (no ricorsione), cicli, espressioni in notazione infissa, I/O formattato, usato ancora oggi (con raffinamenti), richiede 10 anni/uomo di sviluppo, bootstrap difficilissimo (gli studi contemporanei di Chomsky furono applicati solo in seguito). Viene introdotto da John Backus. Fortran necessita di un compilatore ma che comunque ha richiesto uno sforzo enorme dato che andava scritto in linguaggio macchina.
- COBOL: Linguaggio per gestione aziendale, introduce il record (raccolta di dati primitivi che insieme formano un'entità).

Negli anni '60

- ALGOL: grammatica in forma BNF (Backus-Naur Form), primo linguaggio indipendente dall'architettura, introduce blocchi con variabili locali, ricorsione.
- BASIC: "programmazione per tutti", sarà incorporato nei primi Personal Computer (IBM e Apple) 20 anni dopo, facile da imparare ma poco strutturato (goto).
- Simula67: astrazioni di più alto livello, introduce classi, oggetti e ereditarietà

Negli anni '70

- Pascal: programmazione strutturata, ruolo importante dei tipi per evitare errori di programmazione, ideale per imparare a programmare.
- Smalltalk: programmazione a oggetti estrema, tutto è un oggetto (numeri, classi, ecc.), l'unica operazione possibile è l'invio di un messaggio (metodo). Con SmallTalk nasce l'idea di oggetti che scambiano messaggi (oggi li chiamiamo metodi).
- C: linguaggi di medio livello per facilitare l'accesso all'hardware e implementazione di sistemi operativi (UNIX).

Negli anni '80 i processori iniziano ad essere potenti abbastanza da poter interpretare i linguaggi e non per forza compilarli.

- Ada: sviluppato dal ministero della difesa USA, evoluzione del Pascal con concorrenza e tipi astratti (rappresentazione privata + interfaccia pubblica).
- C++: estensione di C con classi e oggetti, introduce i template (classi generiche ottenute per espansione) e overloading di operatori e metodi.
- PostScript: linguaggio stack-based creato da Adobe per descrivere documenti, interprete in stampanti, verrà semplificato e reso efficiente in PDF.
- Perl: prestazioni PC rendono possibile uso di linguaggi interpretati, Perl è per “sistemisti”, manipolazione di file di configurazione del S.O., find/replace con espressioni regolari, linguaggio write-only (programmi difficili da leggere, è difficile anche solo giorni dopo capire cosa fanno i programmi).

Negli anni ‘90

- Python: linguaggio di scripting inventato per noia da uno studente durante le vacanze natalizie (Guido van Rossum).
- Java: il prodotto dei programmi java è bytecode, istruzioni per una macchina virtuale (la JVM). Ora linguaggio general-purpose utilizzato praticamente ovunque.
- Php: per web dinamico.
- Javascript: per web dinamico.
- Erlang: estremamente resiliente, sviluppato per la concorrenza assoluta.


Negli anni 2000

- C#: vuole essere C++ “fatto bene”, compilatore per la piattaforma .NET di Microsoft e successivamente altri S.O. (mono), contributi da un comitato di esperti.
- Scala: linguaggio a oggetti con sintassi “flessibile”, compilatore per JVM.
- Go: linguaggio di Google per programmazione concorrente e distribuita su scala globale, comunicazione su canali, elimina classi a favore delle interfacce.

Nel 1997 però accade un fatto molto importante. John Backus (1924–2007) [5] vince il premio Turing per il FORTRAN nel 1977, ma durante la lezione d’onore critica aspramente il paradigma imperativo e introduce il Functional Programming (FP). “Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming [...]” [6]. Sostiene che i linguaggi di programmazione sono sempre più ingombranti ma non efficaci per programmi complessi. I linguaggi hanno costrutti non interessati, poco basati sulla matematica e devono sottostare al concetto di “word at a time” dove con word si intende parola in senso informatico ovvero tra memoria e CPU vengono scambiate parole, sequenze di bit larghe quanto il bus che collega memoria centrale e CPU.

Facciamo un passo indietro: tutto nasce negli anni ‘30 con l’invenzione da parte di Alonzo Church del λ -calcolo dove tutto è una funzione (numeri, liste, costrutti di controllo, ecc.). I linguaggi funzionali in realtà fiorirono fin dagli albori dell’informatica, ad esempio LISP di John McCarthy (altro premio Turing). Un linguaggio per l’elaborazione dell’informazione non-numerica e simbolica.

```
1 (defun factorial (n)
2   (if (= n 0)
3       1
4       (* n (factorial (- n 1)))))
5
6 (defun append (l1 l2)
```

 Lisp


```

7  (if (null l1)
8    l2
9    (cons (first l1) (append (rest l1) l2))))

```

Nasce anche ML negli anni '80 per opera di Robin Milner, dove introduce il pattern-matching.

Infine viene Haskell anni '90 per opera di Simon Peyton-Jones, Phil Wadler e molti altri (comitato scientifico). Haskell è un linguaggio lazy. Quando applichiamo una funzione ad un argomento esso viene valutato solo se strettamente necessario (CBN in Scala). Introduce anche separazione tra parte pura e parte impura per mezzo di monadi e overloading ovvero lo stesso simbolo cambia comportamento in base al tipo degli operandi al quale viene applicato (simbolo + in Java con stringhe o numeri). Vediamo un confronto concreto Java/Haskell in [Codice 14](#) e [Codice 15](#).

```

1 factorial 0 = 1
2 factorial n = n * factorial (n - 1)
3
4 append [] ys = ys
5 append (x : xs) ys = x : append xs ys

```

» Haskell

Nel 1993 l'NSWC effettua uno studio sulla rapidità di sviluppo di prototipi in vari linguaggi di programmazione. Haskell si dimostra essere conciso e molto efficace, anche per i neofiti (paradossalmente non verrà scelto) [\[7\]](#).

Vediamo ora un esempio di word-at-a-time-programming, inizializzazione, confronto e modifica degli indici, accumulo dei prodotti l'unico ciclo disponibile è il `for`, quello primitivo il codice ricco di assegnamenti.

```

1 int[][] product(int[][] a, int[][] b) {
2   int[][] r = new int[a.length][b[0].length];
3   for (int i = 0; i < a.length; i++)
4     for (int j = 0; j < b[0].length; j++)
5       for (int k = 0; k < a[0].length; k++)
6         r[i][j] += a[i][k] * b[k][j];
7   return r;
8 }

```

» Java

Codice 14: Matrix multiplication in Java

```

1 (·) v w = sum (zipWith (*) v w)
2 (×) a b = map (\r -> map (· r) (columns b)) (rows a)

```

» Haskell

Codice 15: Matrix multiplication in Haskell

Esempio di wholemeal programming, e.g., “moltiplica ogni riga `r` di `a` con una colonna di `b`”. `sum`, `zipWith`, `map` sono funzioni già definite (ma non primitive) nella libreria standard del linguaggio le forme di iterazione, riduzione, composizione di liste (e altre strutture dati) sono programmabili, generiche, utili in molti contesti diversi, non ci sono assegnamenti! quindi non esistono conflitti nell'esecuzione concorrente di questo programma.

Questo torna utile dal 2004 perché cambia qualcosa:

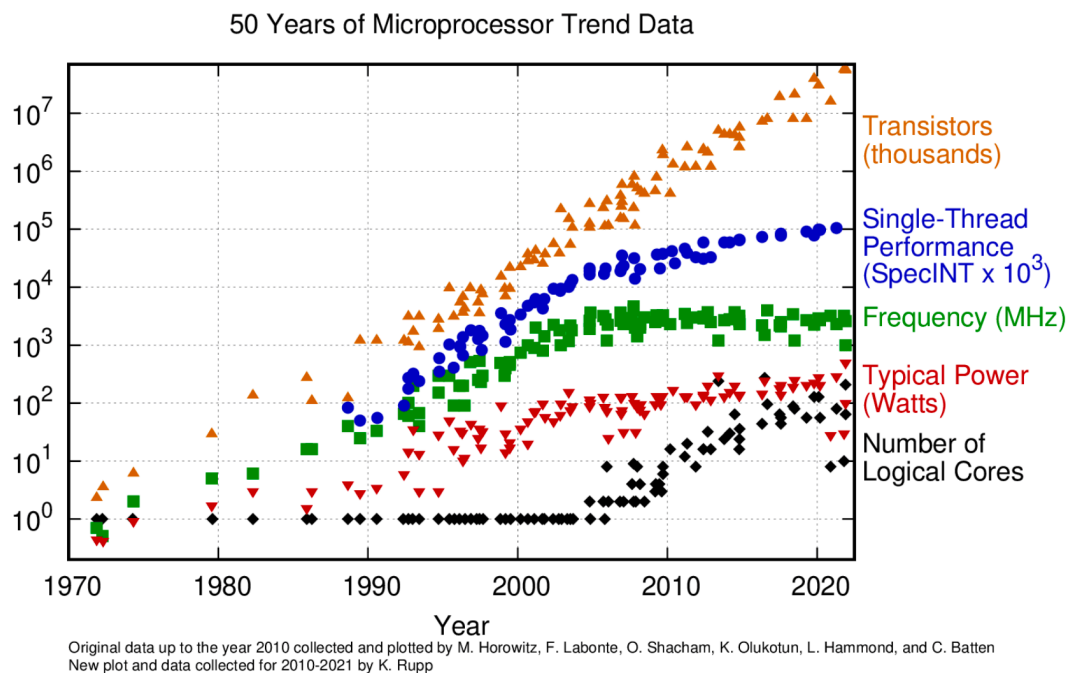


Figura 9: Come cambia la capacità di incorporare transistori in micro processori. La scala è logaritmica. Herb Sutter, A Fundamental Turn Toward Concurrency in Software, 2005 [8]

Fino al 2004 per raddoppiare la velocità di esecuzione dei programmi bastava aspettare ~18 mesi, più o meno il tempo in cui i produttori riuscivano a raddoppiare la velocità dei microprocessori grazie al progresso tecnologico. Nel 2004 la velocità dei microprocessori si attesta attorno ai ~3GHz, andando oltre si incontrano limiti fisici difficili da aggirare (troppa energia per funzionare e troppo calore da dissipare). Dopo il 2004 il numero di transistor continua ad aumentare seguendo grosso modo la legge di Moore (raddoppio di densità ogni ~18 mesi), ma invece di investire i transistor in più per realizzare un singolo core molto veloce, si realizzano tanti core alla velocità massima possibile (~3GHz). Per sfruttare efficacemente la disponibilità di tanti core occorre scrivere programmi in cui attività indipendenti possano essere eseguite in parallelo su core diversi. Questo è facile a dirsi ma difficile (o impossibile) a farsi, di certo non basta più aspettare ~18 mesi come succedeva fino al 2004. Ad aggravare lo scenario, molti linguaggi di programmazione sono mal equipaggiati per la scrittura di programmi paralleli e concorrenti i side effect (assegnamenti) sono spesso causa di conflitti tra attività lo stile “word-at-a-time programming” rende difficile l’identificazione di attività complesse da eseguire in parallelo. L’assenza di side effects e wholemeal programming sono proprio tra le caratteristiche fondamentali della programmazione funzionale.

11. Entità di Prima Classe e Chiusure

Nei linguaggi funzionali le funzioni sono entità di prima classe. Un’entità di prima classe ha diverse caratteristiche

- Può essere l’argomento di una funzione.
- Può essere il risultato di una funzione.
- Può essere definita dentro una funzione.
- Può essere assegnata a una variabile.
- Può essere memorizzata in una struttura (array, lista, albero, ...).

Un’entità di prima classe in un linguaggio tipato ha un tipo che la descrive.

Il C è un linguaggio funzionale? no! in C esistono puntatori a funzioni (**Codice 16-2**) ma non è possibile definire funzioni dentro funzioni ad esempio. Scrivere $f \circ g$ in C è impossibile.

```
1 void qsort(void* base, size_t num, size_t size,
2           int (*compare)(const void*, const void*));
3
4 int compare(const void* a, const void* b) {
5     return (*(intptr_t) a - *(intptr_t) b);
6 }
7
8 int main() {
9     qsort(values, 6, sizeof(int), compare);
10 }
```

Codice 16: Qsort in C

Sembra che in C le funzioni siano entità di prima classe ma in maniera molto limitata, infatti possiamo passare come argomento di funzioni solamente puntatori a funzioni. Notare che in C gli array vengono allocati in un'area contigua di memoria.

```
1 function E(x: real): real;
2     function F(y: real): real;
3     begin
4         F := x + y
5     end;
6 begin
7     E := F(3) + F(4)
8 end;
```

Codice 17: Esempio di funzioni annidate in Pascal

In Pascal è possibile dichiarare funzioni dentro ad altre funzioni ma anche in Pascal (fortemente tipato) non esiste il tipo funzione. Dichiarare funzioni dentro funzioni è interessante, notiamo come la funzione F accede alla variabile x che viene presa in input dalla funzione E. Se spostassimo la definizione di F fuori dalla definizione di E allora si perderebbe l'accesso a x, vedremo perché questo fenomeno è interessante.


Haskell invece è un linguaggio funzionale, le funzioni hanno un tipo, possono essere definite dentro altre funzioni.

```
1 fibo :: Int -> Int
2 fibo k = aux 0 1 k
3 where
4     -- aux :: Int -> Int -> Int -> Int
5     aux m _ 0 = m
6     aux m n k = aux n (m + n) (k - 1)
```

In Haskell la funzione fibo si aspetta in input numeri interi e produce in output numeri interi. Il simbolo `->` separa il tipo del dominio (ciò che la funzione accetta) dal tipo del codominio (ciò che la funzione produce). La funzione fibo applicata ad un tipo Int produce un tipo Int. Il tipo della funzione fibo è determinata dal suo dominio e codominio. Per definire la funzione

fibonacci che calcola il k-esimo numero nella sequenza di fibonacci usiamo una funzione ausiliare che chiamiamo `aux`. La funzione `aux` è definita per casi. Quando il terzo argomento di `aux` è 0, ignoriamo il secondo argomento `_` e il risultato sarà `m` (il primo argomento). In tutti gli altri casi i tre argomenti `m n k` vengono usati ricorsivamente. Il tipo esplicito della funzione di `aux` sarebbe `aux :: Int -> Int -> Int -> Int`, quindi `aux` accetta 3 argomenti di tipo `Int` e produce un tipo `Int`. In Haskell è possibile omettere le dichiarazioni esplicite dei tipi delle funzioni. Il simbolo `->` in Haskell è associativo a destra⁶.


```
1 aux :: Int -> (Int -> (Int -> Int))
```

 Haskell

Il numero di argomenti corrisponde al numero di frecce. `aux` riceve un `Int` e produce una funzione che riceve un `Int` che produce una funzione che riceve un `Int` che ritorna un tipo `Int`. Così è come si legge il tipo di `aux`.

Java, Python, C++ sono linguaggi più o meno funzionali, sicuramente non sono puri.

```
1 static <T> void sort(T[] a, Comparator<? super T> c)
2
3 public class sort {
4     public static void main(String[] args) {
5         System.out.println(Arrays.toString(args));
6         Arrays.sort(args, (a,b) -> a.length() - b.length());
7         System.out.println(Arrays.toString(args));
8     }
9 }
```

 Java


Lo sono più o meno, praticamente le funzioni vengono modellate come oggetti (incorporate a posteriori dai progettisti del linguaggio) ed è possibile passarle come argomento dei metodi. È possibile scrivere λ espressioni che Java interpreta come oggetti che rappresenta una funzione $(a, b) \rightarrow a.length() - b.length()$. `Comparator<? super T>` è una semplice interfaccia Java che viene detta interfaccia funzionale⁷.

La regola di massima è: se un linguaggio è possibile definire la composizione funzionale è (o incorpora elementi del paradigma) funzionale.

$$(f \circ g)(x) = f(g(x)) \quad (2)$$


In Haskell

```
1 (.) :: (b -> c) -> (a -> b) -> (a -> c)
2 (.) f g = \x -> f (g x)
```

 Haskell

In Java

```
1 public static <A,B,C>
2 Function<A,C> compose(Function<B,C> f, Function<A,B> g) {
3     return x -> f.apply(g.apply(x));
```

 Java

⁶Nell'espressione $x * y * z$ l'operando y è conteso tra le due occorrenze di $*$. Se $*$ è associativo a destra significa che l'occorrenza a destra cattura l'operando conteso i.e., $x * (y * z)$. La moltiplicazione è associativa quindi non cambia nulla in questo caso.

⁷The term `? super T` means “unknown type that is, or is a super class of, `T`”, which in generics parlance means its lower bound is `T`.

```
4 }
```

Ancora una volta Java dà l'illusione di lavorare con funzioni con oggetti di prima classe, in realtà dietro le quinte crea due oggetti che implementano l'interfaccia `Function`.

In C è molto più complesso

```
1  int (*)(int) compose(int (*f)(int), int (*g)(int)) {
2      int aux(int x) {
3          return f(g(x));
4      }
5      return aux;
6  }
7
8  int main() {
9      int (*plus2)(int) = compose(succ, succ);
10     printf("%d", plus2(1));
11 }
```

Il problema è che quando viene eseguito il corpo di `aux` gli slot che contengono i valori di `f` e `g` non esistono più, il record di attivazione di `compose` è già stato scartato dallo stack. Al momento dell'invocazione di `plus2`, `f` e `g` non esistono. Una funzione è una computazione ritardata che può accedere a nomi definiti all'esterno del suo corpo ma che potrebbero non esistere più quando il suo corpo viene eseguito.

In Pascal abbiamo visto che è possibile annidare le funzioni ma non ritornare come risultato. Avevamo visto però che in qualche modo in Pascal è possibile accedere a nomi definiti all'esterno del suo corpo (funzione `F` [Codice 17](#)). Tutte le chiamate al corpo di `F` devono essere esaurite all'interno del corpo di `E`.

In un linguaggio funzionale deve essere possibile invocare una funzione in luogo e tempo molto diversi da quelli in cui una funzione è stata definita. Per assicurare che l'esecuzione della funzione avvenga con successo (i dati di cui ha bisogno esistono) i dati sono copiati e impacchettati insieme al codice della funzione in una struttura detta **chiusura**, allocata nello heap. La chiusura è composta da

- Codice della funzione
- Valori delle variabili libere

Vediamo un esempio di chiusura in Haskell

```
1 add = \x -> \y -> x + y
```

`add` non presenta variabili non locali quindi la chiusura prodotta sarà semplicemente il codice di `add`.

```
1 plus1 = add 1
```

La chiusura prodotta sarà


```
1 +-----+
2 | codice  | ----> \y -> x + y
3 | x = 1   |
```

```
4 +-----+
```

Quindi una chiusura è una struttura che contiene il puntatore al codice della funzione e una copia di tutte le variabili non locali a cui la funzione fa riferimento (serviranno per quando applicheremo la funzione ai suoi argomenti). Le chiusure sono spesso utilizzate in vari casi:


- Currying, le funzioni a più argomenti sono cascate di funzioni a un singolo argomento

```
1 add :: Int -> Int -> Int
2 add = \x -> \y -> x + y
```

 Haskell


Solitamente la funzione `add` la scriviamo a due argomenti, ma in Haskell possiamo scrivere la funzione `add` come una funzione che produce un'altra funzione che somma l'input della prima con l'input della seconda. In questo modo il linguaggio rimane semplice e le funzioni possono essere specializzate

```
1 ghci> let succ = add 1
2 ghci> succ 5
3 6
4 ghci> succ 12
5 13
```

 Shell

La funzione `succ` è la funzione *successore* ma in realtà è la chiusura che otteniamo applicando `add` a `1` (ovvero `x`) che viene congelato a `1` nella funzione `succ`. Questo ci permette di definire la funzione *successore* specializzando funzioni pre-esistenti oppure applicarle parzialmente. In un esempio molto più fine analizziamo il codice Haskell che implementa l'algoritmo *QuickSort*.

```
1 sort :: [Int] -> [Int]
2 sort [] = []
3 sort (x : xs) =
4   sort (filter (<= x) xs) ++ [x] ++ sort (filter (> x) xs)
```


 Haskell

Codice 18: QuickSort in Haskell

La funzione `sort` è definita per *pattern-matching*. La prima definizione [Codice 18-2](#) matcha la lista vuota. Il secondo la lista generica con l'elemento `x` in testa e `xs` indica una lista in coda. `(<= x)` è l'applicazione parziale della funzione `<=` congelando il secondo operando a `x`. Dove `<=` corrisponde a `(<=) :: Ord a => a -> a -> Bool`. In questo modo otteniamo il predicato “essere minore o uguale a `x`”.

- Incapsulamento, la chiusura può incapsulare un dato sensibile creato localmente da un'altra funzione.

```
1 let make_counter () =
2   let c = ref 0 in          (* crea una reference mutabile nell'heap *)
3   (fun () -> !c),           (* lettura ! fa dereference *)
4   (fun () -> c := !c + 1),   (* incremento *)
5   (fun () -> c := 0),        (* reset *)
6
7 let get, inc, reset = make_counter ()
```

 OCaml

In questo esempio in OCaml tutte le funzioni non accettano argomenti e fanno riferimento a `c` che però è definita internamente a `make_counter()`. Il contatore è utilizzabile solo attraverso le funzioni `get`, `inc`, `reset`. Se scrivessimo solo `c = 0` creeremmo una variabile immutabile. In questo caso la chiusura permette di mantenere privato l'accesso alla variabile `c`.

- Laziness (Call By Need⁸), le chiusure sono usate per impacchettare espressioni che non devono essere valutate subito, ma solo se necessario.

```
1 ghci> zip [1,2,3] ['a','c']
2 [(1,'a'),(2,'c')]
```

Shell

```
1 zif :: (t1 -> t2 -> a) -> [t1] -> [t2] -> [a]
2 zif f [] _ = []
3 zif f _ [] = []
4 zif f (x : xs) (y : ys) = f x y : zif f xs ys
```

Haskell

```
1 ghci> zip [1,2,3] ['a','c']
2 [(1,'a'),(2,'c')]
3 zif (\x -> \y -> x + y) [1,2,3] [4,5,6]
4 [5,7,9]
5 ghci> sorted xs = and (zif (<=) xs (tail xs))
6 ghci> sorted [-1,2,-3]
7 False
```

Shell

`tail xs` data una lista calcola la sua coda ma la funzione `tail` è una funzione parziale, sulla lista vuota lancia eccezione. Ma quindi la funzione `sorted` (che combina usando `and` il risultato di `zif` con la funzione `<=` su una lista, quindi determina se gli elementi sono in ordine crescente) in realtà crea una chiusura per garantire di sapere chi è `xs` quando è richiesta l'applicazione di `tail` su `xs`, come detto sopra `tail xs` fallirebbe se `xs` è la lista vuota. `sorted` è corretta perché `tail xs` non viene valutata mai se il secondo argomento è la lista vuota.

11.1. Sulla Laziness di Haskell

Proviamo un esperimento. La funzione `const` è una funzione di Haskell che rifiuta il primo argomento e ritorna il secondo, accettando due argomenti con tipi `a` e `b`.

```
1 ghci> :t const
2 const :: a -> b -> a
3 ghci> const 1 True
4 1
5 ghci> const 1 2
6 1
7 ghci> const 1 (1 `div` 0)
8 1
9 ghci> const (1 `div` 0) 1
10 *** Exception: divide by zero
```

Shell


⁸Call by need is similar to call by name; the only difference is that in call by need the actual parameter is evaluated only the first time it is needed. Then the value is stored and used whenever it is needed again (memoization).

Il secondo parametro non viene neanche valutato.

11.2. Chiusure in Java

Nel primo esempio `x` è *effectively final*. L'oggetto che Java crea è in realtà una chiusura che contiene un oggetto con un metodo che dato un argomento `y` calcola la somma `x` e `y`. `x` sarà un campo dell'oggetto. I campi dell'oggetto contengono i valori non locali ai quali la λ espressione fa riferimento. Nel secondo esempio invece la chiusura è alterata. Java consente di far riferimento a nomi non locali nelle λ espressioni sse le variabili non locali sono *effectively final* ovvero una nozione debole della kw `final` che impone che una variabile non sia mai alterata. Questo garantisce che non è problematico copiare `x` nella chiusura (non verrà modificato in seguito dopo la copia).

```
1 public static Function<Integer, Integer> add(int x) {
2     return y -> x + y;
3 }
4
5 public static Function<Integer, Integer> add(int x) {
6     x++;
7     return y -> x + y;
8 }
```

 Java

Se `x` fosse una reference a un oggetto il compilatore Java sarebbe comunque felice probabilmente ma non ci sarebbero garanzie sullo stato interno della reference quando il codice della chiusura viene eseguito.

Gli oggetti presentano al loro interno i campi e i riferimenti ai metodi. I campi sono mutabili e i metodi hanno un parametro implicito (`self` o `this`) con riferimento all'oggetto ricevente. Le chiusure mantengono al loro interno i valori delle variabili libere e il riferimento a una funzione. I valori sono immutabili e la funzione ha un parametro implicito con riferimento alla chiusura in cui la funzione trova i valori delle variabili libere.

Un thunk è una sospensione di un'espressione, una generalizzazione di una chiusura: una struttura che ha un puntatore ad un'espressione in cui sono presenti tutti i valori riferiti dall'interno di quell'espressione. In Haskell ogni espressione non viene valutata subito, quindi ogni espressione produce un thunk.

12. Classi, Interfacce e Traits

La programmazione imperativa è caratterizzata dall'architettura di Von Neumann dove lo stato corrisponde alla memoria e un programma corrisponde a funzioni che alterano lo stato. La programmazione ad oggetti permette di dividere lo stato, partizionandolo all'interno di oggetti i quali sono responsabili della propria partizione. Gli oggetti possono comunicare scambiandosi messaggi (metodi).

In Erlang abbiamo visto come concretamente gli attori si scambiano messaggi attraverso le mailbox quindi è una metafora che in certi contesti (programmazione ad attori) trova un'implementazione reale.

12.1. Linguaggi Object-Based

I linguaggi object-based sono linguaggi con costrutti del tipo


```

1 obj = object {
2   x      = 2
3   set(n)  = x <- n
4   double() = set(2*x)
5 }
6 obj.double()

```

Dove gli oggetti vengono creati all'occorrenza. L'implementazione sarà qualcosa del tipo

```

1 obj = struct {
2   x      = 2
3   set     = set'
4   double  = double'
5 }
6 set'(self, n) = self.x <- n
7 double'(self, n) = self.set(self, 2 * self.x)
8 obj.double(obj)

```

Dove i metodi diventano campi degli oggetti (con 'set' indichiamo un puntatore e con self il parametro implicito ovvero l'oggetto stesso). Javascript è un linguaggio object-based, Python è un ibrido perché esiste la nozione di classe però è comunque possibile modificare i campi di un oggetto a run-time.

Nei linguaggi object-based chi riceve un oggetto può fare molto di più di quello che potrebbe essere inteso fare con l'oggetto (può chiamare metodi su esso). È difficile ragionare sulla correttezza perché non c'è la garanzia che un oggetto non venga modificato a run-time (il metodo set potrebbe cambiare e double non avrebbe più lo stesso comportamento). Attribuire tipi agli oggetti è difficile perché se i campi di un oggetto possono variare a run-time allora un compilatore fa molta fatica a capire le invarianti sugli oggetti. In C e in Java le variabili di tipo int determinano delle invarianti, ovvero finché esiste sarà di tipo int. Nei linguaggi object-based questo non è garantito. Il risultato è che le informazioni sui tipi nei linguaggi object-based sono rilevanti ma il tipaggio avviene dinamicamente quindi il fatto che il linguaggio sia tipato a run-time potrebbe essere renderlo poco robusto in merito alla correttezza dei programmi.

12.2. Linguaggi Class-Based

I linguaggi class-based sono linguaggi con costrutti del tipo

```

1 class C {
2   x      = 2
3   set(n)  = x <- n
4   double() = set(2 * x)
5 }
6
7 obj = new C()
8 obj.double()

```

Le classi sono famiglie di oggetti che condividono le stesse operazioni e fanno parte di una gerarchia. Notiamo che l'insieme dei metodi è fissato una volta per tutte. È inoltre possibile

assegnare un tipo statico agli oggetti ed è possibile ragionare sulla correttezza di una classe. L'implementazione di una classe è qualcosa del tipo

```
1  struct C {
2    vtab : C_virtual_table
3    x    : int
4  }
5
6  struct C_virtual_table {
7    set    : int -> void = set'
8    double : void -> void = double'
9  }
10
11 set'(self, n) = self.x <- n
12 double'(self) = self.vtab.set(self, 2 * self.x)
```

L'insieme dei metodi di una classe è nota a priori (a compile-time) ed è possibile raccogliarli in una virtual table, ovvero una struttura statica che aggiunge un livello di indirizzione che però viene calcolata una volta per tutte. La virtual table è unica per tutti gli oggetti istanza di una certa classe.

Anche i linguaggi class-based hanno dei problemi, uno dei più discussi è l'ereditarietà. L'idea dell'ereditarietà è riusare il codice di una classe per definire una sottoclasse più specifica permettendo di aggiungere campi e ridefinire metodi.

```
1  class D extends C {
2    y = 3
3    get() = y
4    double() = super.double(); y <- 2 * y
5  }
```

Dove la classe D viene implementata con qualcosa del tipo

```
1  struct D {
2    vtab : D_virtual_table
3    x    : int
4    y    : int
5  }
6
7  struct D_virtual_table {
8    set    : int -> void = set'
9    double : void -> void = double'
10   get    : void -> int = get'
11 }
12
13 set'(self, n) = self.x <- n
14 double''(self) = double'(self); self.y <- 2 * self.y
```

Possiamo anche pensare che anzichè ricopiare i metodi della virtual table di C in realtà venga mantenuto un puntatore alla virtual table della classe C in modo tale da riusare i metodi e ridefinire solo quelli effettivamente ridefiniti.

12.3. Ereditarietà e Polimorfismo

Il meccanismo dell'ereditarietà purtroppo è andato a braccetto con il concetto del polimorfismo quando invece dovrebbero essere concetti tenuti ben lontani.

```
1  abstract class Figure {
2      private float x, y;
3      public abstract float area();
4      public abstract float perimeter();
5  }
6
7  class Square extends Figure {
8      private float side;
9      public float area() { return side * side; }
10     public float perimeter() { return 4 * side; }
11 }
12
13 class Circle extends Figure {
14     private float radius;
15     public float area() { return radius * radius * PI; }
16     public float perimeter() { return 2 * PI * radius; }
17 }
18
19 float sum(Figure f, Figure g) { return f.area() + g.area(); }
```

Il metodo `sum(Figure f, Figure g)` è un metodo polimorfo, possiamo passare due Figure qualsiasi e `sum` riesce a calcolare la somma delle aree. Ma `sum` non necessariamente deve ricevere delle Figure, per `sum` è necessario che gli oggetti passati a `sum` sappiano calcolare l'area, ma `sum` introduce rigidità in quanto accetta solo Figure. Il problema è che la gerarchia delle classi introduce rigidità e ci spiega le relazioni tra di esse, ma non sempre fornisce tutte le informazioni su cosa gli oggetti sanno fare. La struttura gerarchica funziona bene per stabilire relazioni del tipo “essere un/una” ma non fornisce informazioni su cosa sa fare un certo oggetto di una classe.


L'ereditarietà è per certi versi troppo fragile e vediamo i motivi: nell'esempio seguente in Java viene presentato il codice che descrive una pila. Notiamo come in generale non c'è indipendenza totale delle classi derivate dalla classe base e dalla sua implementazione. In questo caso il metodo `pushAll()` si basa sul metodo `push()`. Questo è un problema perché se cambia la classe base questo si ripercuoterà a cascata su tutte le classe derivate.

```
1  class Stack<T> {
2      ...
3      void push(T x) { ... }
4      T pop() { ... }
5      void pushAll(T[] a) { for (T x : a) push(x); }
```

```
6 }
```


La dimensione della pila potrebbe non essere immediatamente accessibile. Ad esempio se viene implementata con una `LinkedList` servirebbe seguire la catena di puntatori per contare quanti elementi ci sono all'interno, dunque il costo potrebbe non essere costante. Di seguito viene definita una pila che conserva la dimensione corrente utilizzando però i metodi che già esistono nella classe base.

```
1 class SizedStack<T> extends Stack<T> {
2     int size = 0;
3     void push(T x) { super.push(x); size++; }
4     T pop() { size--; return super.pop(); }
5 }
```

 Java


Il metodo `push()` dipende dal tipo reale sul quale viene chiamato. Anche se chiamiamo su un oggetto `SizedStack` il metodo `pushAll()` viene chiamato il metodo ereditato `pushAll()` della classe base ma grazie al **binding dinamico** viene poi ripassato il controllo al metodo `push()` ridefinito nella classe derivata. Assumiamo però che qualcuno alteri il metodo `pushAll()` della classe base.

```
1 class Stack<T> {
2     ...
3     void push(T x) { ... }
4     T pop() { ... }
5     void pushAll(T[] a) { ... } // non si usa più push
6 }
```

 Java

Questo però crea problemi per le classe derivate, perché chi definisce la classe `SizedStack` deve ripartire da zero. Occorrerebbe ridefinire anche il metodo `pushAll()` altrimenti la dimensione della pila non verrebbe aggiornata correttamente dato che il nuovo metodo `pushAll()` non chiama `push()` ridefinito nella classe derivata. Quando le classi vengono definite utilizzando l'ereditarietà rimangono dei legami molto forti che dipende da come le classi sono realizzate internamente. Si preferisce ormai utilizzare la composizione di oggetti, ovvero sviluppare oggetti che al loro interno hanno campi che sono riferimenti ad altri oggetti.

```
1 class SizedStack<T> {
2     Stack<T> s; // composizione
3     int size = 0;
4     void push(T x) { s.push(x); size++; }
5     T pop() { size--; return s.pop(); }
6     void pushAll(T[] a) { s.pushAll(a); size += a.length; }
7 }
```

 Java

Così facendo i tipi `SizedStack<T>` e `Stack<T>` non sono più in relazione nonostante descrivano oggetti che sanno fare le stesse cose. Usando l'ereditarietà grazie al principio di sostituzione potevamo passare un oggetto di tipo `SizedStack<T>` laddove era attesa un oggetto di tipo `Stack<T>`, in questo modo non possiamo sfruttare lo stesso concetto. Non possiamo neanche esporre il campo

s perché la pila potrebbe essere modificata usando i metodi sull'oggetto `Stack<T>` `push()` che disallineerebbero la dimensione della pila dimensionata.

Gosling: “If you could do Java over again, what would you change?” Gosling : “I’d leave out classes.” [9].

12.4. Interfacce

Le interfacce, attraverso le quali vogliamo realizzare il polimorfismo, possono risolvere parzialmente i problemi che introduce l’ereditarietà. Ci interessa sapere cosa fanno le pile, quindi possiamo definire un’interfaccia comune a tutte le pile.

```
1 interface StackInterface<T> {
2     void push(T x);
3     T pop();
4     void pushAll(T[] a);
5 }
6
7 class Stack<T> implements StackInterface<T> { ... }
8 class SizedStack<T> implements StackInterface<T> { ... }
```

In questo modo possiamo definire dei metodi che accettano oggetti di tipo `StackInterface<T>` e possiamo garantire che gli argomenti siano di tipo `StackInterface<T>` se implementano l’interfaccia `StackInterface<T>`. Ma quindi come facciamo a definire un linguaggio moderno e robusto rispetto a queste problematiche se:

- Rinunciando all’ereditarietà dobbiamo rinunciare alle classi
- Rinunciando alle classi rinunciamo agli oggetti
- Rinunciando agli oggetti otteniamo object-less methods (traits)

12.5. Go

Go è un linguaggio fortemente tipato e compilato con GC automatica sviluppato da Google. Abbandona il concetto di classe e abbraccia totalmente i Trait. Non introduce il polimorfismo parametrico nonostante i decenni di ricerca sull’argomento (stesso errore di Java che li introdusse a posteriori, non senza fatica). Ora sono stati introdotti i tipi generici. È interessante la scelta di mantenere la presenza del costrutto `goto`.

```
1 package main
2 import (
3     "fmt"
4     "math"
5 )
6
7 type Vector struct {
8     X, Y float64
9 }
10
11 func Mod(v Vector) float64 {
12     return math.Sqrt(v.X * v.X + v.Y * v.Y)
13 }
```

```

14
15 func main() {
16     v := Vector{3, 4}
17     fmt.Println(Mod(v))
18     fmt.Println(v.Mod())
19 }

```

`Mod(v Vector)` in questo caso ricorda il paradigma funzionale, non è un metodo sul tipo `Vector` ma viene accettato in input un tipo `Vector`.

In Go è possibile specificare per un metodo il receiver ovvero un tipo che riceve il metodo in modo da poter usare la sintassi `v.Mod()`. Il tipo del receiver deve essere definito nel file corrente.

```

1 func (v Vector) Mod() float64 {
2     return math.Sqrt(v.X * v.X + v.Y * v.Y)
3 }

```

 Go

La sintassi `o.m(arg1, ..., argn)` è equivalente a `m(o, arg1, ..., argn)` dove `o` è il receiver. Ma come mai questa scelta? non c'è late binding (dynamic dispatch) quindi non è per quello, bensì per aiutare l'autocompletamento dell'ide.

È possibile anche definire metodi su tipi alias. Notare che non è possibile estendere i metodi su tipi al di fuori dallo stesso package.

```

1 package main
2 import ("fmt")
3
4 type MyFloat float64
5
6 func (f MyFloat) Abs() float64 {
7     if f < 0 {
8         return float64(-f)
9     }
10    return float64(f)
11 }
12
13 func main() {
14     f := MyFloat(-math.Sqrt2)
15     fmt.Println(f.Abs())
16 }

```

 Go

Esistono anche i metodi sui puntatori (meglio dire reference). Per aggiornare le componenti dell'oggetto la sintassi resta comunque uguale perché il compilatore Go automaticamente fa dereferencing dell'area di memoria puntata da `v`, tutto in automatico. Omettendo l'asterisco il codice resta sintatticamente corretto ma l'aggiornamento dei campi diventa locale al metodo e non vengono aggiornati fuori. In Java tutti gli oggetti sono allocati nell'heap e gli oggetti vengono sempre passati come reference. In Go che è più di basso livello l'oggetto `Vector` nel nostro esempio è nello stack e omettendo l'asterisco staremmo passando una copia dell'oggetto.

```

1  func (v *Vector) Scale(f float64) {
2      v.X = v.X * f
3      v.Y = v.Y * f
4  }
5
6  func main() {
7      v := Vector{3,4}
8      v.Scale(10)
9      fmt.Println(v.Abs())
10 }

```

 Go

12.5.1. Interfacce in Go

Le interfacce in Go, sono un insieme di signature di metodi, usate per tipare gli argomenti delle funzioni polimorfe. Non c'è nessuna cerimonia sintattica, infatti un tipo T implementa automaticamente un'interfaccia I se definisce tutti i metodi elencati da I (con lo stesso tipo). This is duck typing “If it walks like a duck and it quacks like a duck, then it's a duck”.

```

1  type Square struct {
2      side float64
3  }
4
5  func (o Square) Area() float64 {
6      return o.side * o.side
7  }
8
9  func (o Square) Perimeter() float64 {
10     return o.side * 4
11 }
12
13 type Circle struct {
14     radius float64
15 }
16
17 func (o Circle) Area() float64 {
18     return math.Pi * o.radius * o.radius
19 }
20
21 func (o Circle) Perimeter() float64 {
22     return 2 * math.Pi * o.radius
23 }
24
25 type Measurable interface {
26     Area() float64
27     Perimeter() float64

```

 Go

```

28 }
29
30 func PrintMeasure(o Measurable) {
31     fmt.Println(o.Area())
32     fmt.Println(o.Perimeter())
33 }
34
35 func main() {
36     s := Square{1}
37     c := Circle{2}
38     PrintMeasure(s) // Square -> Measurable
39     PrintMeasure(c) // Circle -> Measurable
40 }

```

Dato che `Circle` e `Square` implementano entrambi i metodi `Area()` e `Perimeter()` quindi sono tipi che implementano l'interfaccia `Measurable`. `PrintMeasure()` è un metodo polimorfo. Dichiarare un'interfaccia equivale a dichiarare un nuovo tipo di dato, il che implicitamente significa che vengono definiti nuovi tipi di dato ogni volta che un tipo implementa tale interfaccia.

Il compilatore però deve svolgere lavoro non banale per permettere l'esecuzione del codice sopra. Al compilatore serve sapere quali metodi `Area()` e `Perimeter()` invocare e soprattutto su che oggetto agiscono. Staticamente non è possibile capire quale sia il metodo giusto da invocare, e su quale oggetto agire. L'unica garanzia per il compilatore è che un tipo `Measurable` avrà i metodi `Area()` e `Perimeter()`. Anche lo stesso oggetto non è noto al compilatore (il receiver è scritto a sinistra).

```

1 func PrintMeasure(o Measurable) {
2     fmt.Println(o.Area())
3     fmt.Println(o.Perimeter())
4 }

```

 Go

Un valore di tipo interfaccia in Go è quindi una coppia (dati, codice). Chiamiamo un valore di tipo interfaccia `I` una coppia (v, p) dove v è l'oggetto e p è la tabella dei metodi in `I` per l'oggetto v . Questa è una forma di late binding dinamico (dynamic dispatch) dato che la funzione da invocare è decisa a run-time dinamicamente. Passare `s` di tipo `Square` a una funzione che attende un argomento `Measurable` significa creare una coppia (s, p) dove p è una tabella simile a una virtual table (in maniera simile a come abbiamo visto nella [Sezione 12.2](#)) che contiene puntatori ai metodi `Area` e `Perimeter` per i receiver di tipo `Square`. Stesso vale per `c`.

Abbiamo visto che in Go possiamo definire i metodi per un tipo solamente nel package dove è stato definito il tipo per motivi di autocompletamento del codice. Questo principio di località viene sfruttato dal compilatore per cercare i metodi di un oggetto `Measurable` limitandosi al package corrente.

12.5.2. Ereditarietà di Interfacce

In Go (come anche in Java) è possibile estendere le interfacce.

```

1 type MeasurableColorable interface {
2     Measurable

```

 Go


```

3  GetColor() color
4  SetColor(c color)
5  }

```

In questo caso `MeasurableColorable` eredita tutti i metodi definiti in `Measurable` ed estende l'insieme dei metodi. Qui abbiamo una forma innocua di ereditarietà che non presenta i problemi visti in precedenza (non c'è codice ma solo dichiarazioni di intenti).

12.5.3. Composizione di Strutture

```

1  type ColoredSquare struct {
2      Square
3      c Color
4  }

```

 Go

Una struttura può avere come campo anonimo un'altra struttura ciò significa che la struttura sfrutta il concetto di ereditarietà per composizione. Tutti i campi della struttura inclusa sono accessibili come campi della struttura che la include. I metodi definiti per la struttura inclusa sono generati anche per la struttura che la include (`Area()` e `Perimeter()`).

- Per una chiusura viene definita una funzione che può accedere a dati diversi. Quindi una singola funzione e molteplici dati.
- Nel caso dei trait identifichiamo un singolo dato per il quale disponiamo diverse operazioni. Quindi un singolo dato e molteplici metodi

12.6. Rust

Rust è un linguaggio di basso livello (codice di sistema come C, C++) con un supporto run-time minimo (no GC). La gestione delle risorse deve essere oggetto di grande attenzione da parte del programmatore. Rust, a differenza di C presenta sistema di meccanismi di tipi che permette al compilatore di capire se avviene una gestione della memoria inopportuna. I trait in Rust devono essere implementati e definiti con una sintassi esplicita per passare oggetti dove è atteso un trait (metodo = funzione con invocazione su receiver).

```

1  struct Vector {
2      x : f64,
3      y : f64
4  }
5
6  impl Vector {
7      fn origin() -> Vector { // equivalent of static method
8          Vector { x : 0.0, y : 0.0 }
9      }
10
11     fn modulus(&self) -> f64 { // instance method
12         (self.x * self.x + self.y + self.y).sqrt()
13     }
14 }
15

```

 Rust

```

16 fn mod_origin() -> f64 {
17     let o = Vector::origin()
18     o.modulus()
19 }

```

Analogo di quanto visto in Go però in Rust. Notiamo il metodo `modulus(&self)` dove la `&` ha un significato particolare che approfondiremo. Per definire un trait invece la sintassi è la seguente:

```

1 trait Measurable {
2     fn area(&self) -> f64;
3     fn perimeter(&self) -> f64;
4 }

```


 Rust

In questo caso l'uso di `&self` come tipo segnaposto per l'argomento dell'oggetto ricevente è fondamentale perché non sappiamo che tipo avrà.

```

1 struct Circle {
2     radius : f64,
3 }
4
5 impl Measurable for Circle {
6     fn area(&self) -> f64 {
7         std::f64::consts::PI * self.radius * self.radius
8     }
9
10    fn perimeter(&self) -> f64 {
11        2 * std::f64::consts::PI * self.radius
12    }
13 }

```


 Rust

Le funzioni polimorfe possono presentare vincoli fini:

```

1 fn print_measure<T : Measurable>(shape : T) {
2     println!("Area = {}", shape.area());
3     println!("Perimeter = {}", shape.perimeter());
4 }

```

 Rust

La funzione `print_measure` è polimorfa del tipo `T` dell'argomento. Il trait `Measurable` è usato come vincolo di `T` sarebbe l'analogo del vincolo `extends` di Java. Vediamo alcuni tipi polimorfi con vincoli:

```

1 struct Rectangle<T> {
2     width : T,
3     height : T,
4 }
5
6 impl Rectangle<T : PartialEq + Display> Rectangle<T> {
7     fn is_square(&self) -> bool {

```

 Rust

```

8     println!("Width = {}", self.width);
9     self.width == self.height
10 }
11 }

```

Il tipo `Rectangle` diventa polimorfo nel tipo della lunghezza. Dietro le quinte il confronto (`self.width == self.height`) viene tradotto in `self.width.eq(self.height)`. Ma chi ci assicura che il tipo `T` di `width` e `height` sia definito? ce ne assicuriamo imponendo che il tipo `T` implementi sia il trait `PartialEq` sia il trait `Display`.

In Rust viene seguito un principio generale: per implementare il trait `T` per il tipo `U` almeno uno tra `U` e `T` deve essere definito nel file corrente.

I trait possono offrire implementazioni default che possono a loro volta essere ridefinite quando si implementa il trait. Potremmo accontentarci di implementare uno solo dei due metodi `is_valid` e `is_invalid`. Questa scelta non è particolarmente felice perché richiama alcuni problemi che avevamo visto con l'ereditarietà.

```

1 trait Foo {
2     fn is_valid(&self) -> bool { !self.is_invalid() }
3     fn is_invalid(&self) -> bool { !self.is_valid() }
4 }

```

 Rust

I trait possono estendere altri trait

```

1 trait Foo {
2     fn foo(&self);
3 }
4
5 trait FooBar : Foo {
6     fn foo_bar(&self);
7 }

```

 Rust

12.6.1. Binding Statico e Dinamico

Un metodo invocato su un tipo concreto viene risolto staticamente, mentre un metodo invocato su un oggetto il cui tipo è “generico” (ma concreto) viene risolto staticamente grazie alla monomorfizzazione. Inoltre vengono create (a compile-time) tante versioni di `smaller`, una per ogni istanziazione del parametro di tipo `T`. Nella seconda versione di `smaller` non si conosce a priori il tipo di ritorno e di confronto quindi il compilatore crea a compile-time tante versioni della stessa funzione.

```

1 // tipo concreto: binding statico
2 fn smaller(a : Circle, b : Circle) -> bool {
3     a.area() < b.area()
4 }
5
6 // tipo concreto ma generico: binding statico monomorfo
7 fn smaller<T>(a : Rectangle<T>, b : Rectangle<T>) -> bool {
8     a.width * a.height < b.width * b.height

```

 Rust

```
9 }
```

Un metodo invocato su un oggetto il cui tipo è un riferimento a un trait viene risolto dinamicamente:

```
1 fn print_measure(shape : &Measurable) {  
2     println!("Area = {}", shape.area());  
3     println!("Perimeter = {}", shape.perimeter());  
4 }  
5  
6 fn main() {  
7     let c = Circle{radius : 2};  
8     print_measure(&c as &Measurable);  
9 }
```

 Rust

Il passaggio di un oggetto di tipo trait deve essere indicato esplicitamente dal programmatore (&c as), l'implementazione è analoga a quella di Go (coppie valore + puntatore a tabelle di metodi).

13. Type Classes

Haskell appartiene a una famiglia di linguaggi funzionali (OCaml, StandardML) fortemente tipati dove il programmatore non è forzato a scrivere in maniera esplicita il tipo delle entità che definisce. Il compilatore Haskell è in grado di inferire il tipo delle entità che compaiono nel programma semplicemente analizzando come esse vengono usate nel programma.

Questo non è gratuito e possono sorgere diversi conflitti con l'overloading. Vediamo alcuni problemi che sorgono quando i linguaggi cercano di implementare la type-inference:

```
1 val a = 3 * 3 (* accettato da ML, a è int *)  
2 val b = 3.14 * 3.14 (* accettato da ML, b è float *)  
3  
4 fun square x = x * x (* rifiutato da ML *)  
5 val a = square 3  
6 val b = square 3.14
```

 SML

Il tipo di square non è inferibile per lo Standard ML. Se attribuisse `Int -> Int` come tipo di square la prima applicazione sarebbe ben tipata mentre la seconda no. In OCaml per evitare questi problemi i progettisti hanno deciso di utilizzare direttamente operatori diversi ma non è una soluzione scalabile se vengono introdotti nuovi tipi numerici.

```
1 val a = 3 * 3  
2 val b = 3.14 *. 3.14  
3  
4 let squareI x = x * x (* * moltiplica numeri interi *)  
5 let squareF x = x *. x (* *. moltiplica numeri floating point *)  
6 let a = square 3  
7 let b = square 3.14
```

 OCaml

Un altro caso molto spinoso è l'uguaglianza:

```

1 let rec member x = function
2   []          -> false
3   | (y :: ys) -> x = y || member x ys
4 ;;
5 val member : 'a -> 'a list -> bool = <fun>

```

 OCaml

Usare la funzione member su liste di funzioni causa un errore runtime! mentre OCaml dice che member può essere usata su qualsiasi lista di qualsiasi tipo. Il motore di inferenza di OCaml è troppo generoso dato che member è eccessivamente polimorfa. L'apice significa che la variable può essere di qualsiasi tipo.

SML fa un pochino meglio, introduce il doppio apice significa che il tipo "a deve definire la nozione di uguaglianza. Non scala su altri tipi di proprietà oltre all'uguaglianza.

```

1 fun member (x, []) = false
2   member (x, h :: y) = x == h orelse member x ys;
3 val member : 'a * 'a list -> bool = <fun>

```


 SML

La soluzione che adotta Haskell è la seguente: divide i tipi in classi di tipi, non necessariamente disgiunte, ogni classe è definita dalle operazioni supportate da quei tipi [10]. Di seguito alcuni esempi degli operatori overloaded in Haskell

```

1 (+)      :: Num a => a -> a -> a
2 negate   :: Num a => a -> a
3 (==)     :: Eq a  => a -> a -> Bool
4 (<=)     :: Ord a => a -> a -> Bool

```

 Haskell


Num è definito il contesto, il tipo di a non è arbitrario ma deve appartenere alla classe di tipi Num ovvero i tipi numerici. Num a vedremo che sarà anch'esso un dizionario di metodi.

Formalmente, il compito del compilatore è dato un programma

```

1 Gamma |- e :: t
2 -- x1 :: t1, x2 :: t2, ... |- e :: t

```

 Haskell

Dove e è un'espressione e Gamma è un contesto, ovvero una mappa del tipo delle variabili che possono apparire nell'espressione

Definizione: Problema del Type Checking

Noto il contesto Gamma (e noti i tipi delle variabili di Gamma) e noto il tipo t dell'espressione e vogliamo determinare se il tipo dell'espressione è quello corretto.

Definizione: Problema della Type Inference

Noto il contesto Gamma (e noti i tipi delle variabili di Gamma), nota l'espressione e vogliamo determinare se il tipo dell'espressione è quello corretto.

Questo è un problema più complicato, mentre nel problema della type checking serve solo controllare se il tipo è giusto, nel problema della type inference serve costruire un tipo da associare all'espressione e far sì che sia il tipo giusto.

Definizione: Problema della Type Reconstruction

Nota l'espressione e capire il tipo dell'espressione e.

Il compilatore ha poche informazioni, ha solo il programma in pratica e deve capire anche i tipi del contesto Gamma.

Haskell, OCaml e ML risolvono il problema della Type Reconstruction però c'è spesso ambiguità, in molti casi si usa il termine Type Inference al posto di Type Reconstruction (anche nelle slide).

Chiedendo ad Haskell che tipo ha l'operatore `*` ritorna qualcosa del tipo

```
1 ghci> :t (*)
2 (*) :: Num a => a -> a -> a
```

Shell

Risolve il problema posticipando la scelta su come debba essere istanziata la variabile di tipo a tenendo traccia però che il tipo con il quale verrà istanziata a debba appartenere alla classe di tipi numerici.

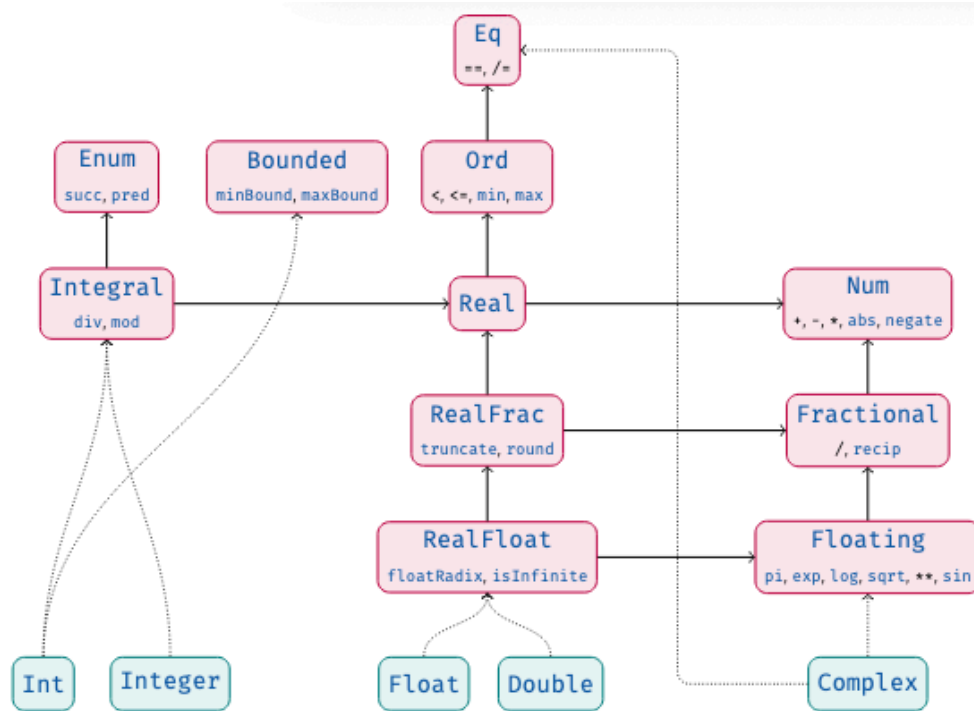


Figura 10: Classi di tipo più comuni in Haskell


In rosa sono espresse le classi mentre in azzurro i tipi. Le frecce solide determinano le relazioni di sotto-classe. Per esempio `Ord` è sotto-classe di `Eq`, questo significa Per tutti i tipi per i quali esiste una nozione di ordinamento esiste una nozione di uguaglianza ma non viceversa. Le frecce tratteggiate richiamano la nozione di appartenenza a una classe di tipo. I numeri complessi (`Complex`) appartengono alla classe di tipi `Eq` ma non alla classe di tipo `Ord`, non esiste infatti una relazione d'ordine totale per i numeri complessi. Per la classe di tipo `Num` non viene inclusa l'operazione di divisione, per i numeri `Real` è previsto un tipo di divisione mentre i numeri `Integral` un altro tipo di divisione (`div` e ha senso anche prevedere la funzione `mod`). `Real` è sottoclasse di `Ord` perché c'è una relazione di ordine totale. Si nota inoltre la differenza tra `Int` e `Integer`, per il tipo `Int`

esiste un bound dato che appartiene alla classe di tipo Bounded mentre per Integer questo non vale.

```
1 ghci> (2 ^ 1000) :: Int
2 0
3 ghci> (2 ^ 1000) :: Integer
4 10715086071862673209484250490600018105614048117055336074437503883703510511249
```

È possibile definire una classe di tipi


```
1 class Num a where
2     (+), (-), (*) :: a -> a -> a
3     negate, abs :: a -> a
4
5     square :: Num a => a -> a -> a
6     square = x * x
7
8     member :: Eq a => a -> [a] -> Bool
9     member _ [] = False
10    member x (y : ys) = x == y || member x ys
```

 Haskell

Per ogni tipo `a` della classe `Num` sono definite le operazioni `(+)`, `(-)`, `...` il cui significato dipende dal tipo effettivo con cui viene istanziata `a`. La freccia `=>` indica il contesto che esprime i vincoli sul tipo `a`.


È possibile definire istanze (`Int`) di una classe di tipi (`Num`)

```
1 instance Num Int where
2     (+) = addInt
3     (-) = subInt
4     abs = absInt
```

 Haskell


`addInt`, `...` sono le funzioni primitive per sommare numeri interi, moltiplicare numeri floating point ecc... Le classi `Show` e `Read` sono interessanti, ad esempio `Show` è l'equivalente di `toString` in Java.

```
1 show :: Show a => a -> String
2 read :: Read a => String -> a
```

 Haskell

Notiamo che solo i tipi di dato che appartengono alla classe `Show`, un tipo di dato che non appartiene a `Show` ad esempio sono le funzioni. `Read` è l'inverso di `Show` ma è un po' strana, sembra che data una `String` ritorni un tipo a qualunque. `Read` è una funzione parziale dato che ci saranno per forza `String` che non hanno una rappresentazione, ad esempio le funzioni.

```
1 ghci> id
2 <interactive>:7:1: error: [GHC-39999]
3     • No instance for 'Show (a0 -> a0)' arising from a use of 'print'
4       (maybe you haven't applied a function to enough arguments?)
5     • In a stmt of an interactive GHCi command: print it
```

 Shell

```

1  ghci> show 123
2  "123"
3  ghci> read "123" :: Int
4  123
5
6  ghci> read "Ciao" :: Int
7  *** Exception: Prelude.read: no parse
8
9  ghci> :t 384756
10 384756 :: Num a => a
11 # Haskell non committa su un tipo preciso e tiene ambiguità
12
13 ghci> :t 384756.235
14 384756.235 :: Fractional a => a
15 # Haskell qui capisce che si tratta di Fractional ma non disambigua fino in
16 fondo
17 ghci> minBound :: Int
18 -9223372036854775808
19 # vogliamo il numero intero più piccolo
20
21 ghci> minBound :: Bool
22 False
23 ghci> maxBound :: Bool
24 True
25 # I Bool sono Bounded e hanno ordinamento totale :)

```

Per definire le sottoclassi è possibile specificare un contesto

```

1  class Eq a => Ord a where
2  -- ogni istanza di Ord deve essere per forza istanza di Eq
3
4  class Eq a where
5      (==) :: a -> a -> Bool
6      (/=) :: a -> a -> Bool
7
8      x == y = not (x /= y) -- due valori sono uguali se non sono diversi
9      x /= y = not (x == y) -- due valori sono diversi se non sono uguali
10
11  -- basta implementare uno dei due

```

Haskell permette di fornire (oltre alla signature delle funzioni) anche delle definizioni di default (`==` e `/=`) e implementare solo una delle due.

È possibile anche definire istanze generiche, ovvero nell'esempio seguente il tipo delle liste di elementi di tipo `a` è istanza di `Eq`. Quindi ogni elemento `a` deve appartenere alla classe di tipo `Eq` e questo si ripercuote sull'intera lista che a sua volta apparterrà alla classe di tipo `Eq`.


```

1 instance Eq a => Eq [a] where
2   [] == []
3   (x :: xs) == (y :: ys) = x == y && xs == ys
4   _ == _
5   -- dato che in Eq c'è la negazione di uguaglianza siamo a posto

```

Haskell

Per definire una nozione di uguaglianza tra liste è necessario e sufficiente che esista una nozione di uguaglianza tra gli elementi della lista. La nozione di disuguaglianza è definita implicitamente.

13.1. Classi su Costruttori di Tipo

Un costruttore di tipo non è un tipo a sè stante ma possiamo pensarlo come una funzione che applicata a un tipo produce un tipo. Pensiamo alle liste, una lista non è un tipo, una lista di interi è un tipo. Lista è un costruttore di tipo infatti possiamo applicare il costruttore di tipo lista ad un intero per ottenere una lista di interi. Un altro esempio è il tipo Maybe che può essere istanziato con Just o Nothing.

```

1 ghci> Just 67
2 Just 67
3
4 ghci> Nothing
5 Nothing
6
7 ghci> :t Just 67
8 Just 67 :: Num a => Maybe a
9
10 ghci> :t Nothing
11 Nothing :: Maybe a

```

Shell

Due esempi di classi di costruttori di tipo molto comuni sono Foldable e Functor.

```

1 class Foldable t where
2   -- t a significa il costruttore di tipo t applicato al tipo a
3   length :: t a -> Int
4
5   -- Per ogni a che appartiene alla classe dei numeri applichiamo il
   -- costruttore di tipo t ad a
6   sum :: Num a => t a -> a
7   null :: t a -> Bool
8   elem :: Eq a => a -> t a -> Bool
9
10 class Functor t where
11   fmap :: (a -> b) -> t a -> t b
12   -- fmap è una funzione di ordine superiore (il primo parametro è una
   -- funzione) che applica una funzione agli elementi di tipo a trasformandoli
   -- in elementi di tipo b e ritornando un costruttore di tipo t applicato
   -- agli elementi di tipo b

```

Haskell

Guardiamo il tipo di `length`, è una funzione che accetta come argomento un tipo `t` di `a`. Pensiamo a `t` come un contenitore di elementi di tipo `a`. La signature di `length` lascia intendere che restituisca il numro di elementi dentro il contenitore `t`. `sum` ha senso farla solo su elementi di tipo numerico ed ecco allora che nel tipo di `sum` c'è il contesto (`Num a => ...`). `null` invece ci dice se il contenitore è vuoto. `elem` ci dice se un certo elemento è contenuto nel contenitore. Di seguito alcuni esempi

```
1 ghci> length Nothing
2 0
3 ghci> length (Just 67)
4 1
5 ghci> length [1,2,3,4,6]
6 5
7 ghci> length [1..100]
8 100
9 ghci> sum [1,2,3,4,5]
10 15
11
12 :t product
13 product :: (Foldable t, Num a) => t a -> a
```

Per `product` i vincoli sono 2, il contenitore deve essere `Foldable` e il tipo degli elementi contenuti deve essere di tipo `Num`.

Nell'esempio sopra vediamo anche i cosiddetti funtori `Functor`, innanzitutto notiamo che è una funzione di ordine superiore dato che accetta un'altra funzione, poi a partire da un contenitore di tipi `a` applica una funzione su tutti gli elementi sul contenitore ritornando un nuovo contenitore di tipi `b`.

```
1 ghci> fmap show (Just 67)
2 Just "67"
3 ghci> :t (Just 67)
4 (Just 67) :: Num a => Maybe a
```

Ad esempio vediamo in Haskell come implementiamo la deviazione standard

$$\sigma(x) = \sqrt{E[(X - E[X])^2]} \quad (3)$$

dentro ad un contenitore

```
1 -- fractional perché stiamo usando /
2 mu :: (Fractional a, Foldable t) => t a -> a
3 mu xs = sum xs / fromIntegral (length xs)
4
5 sigma :: (Floating a, Functor t, Foldable t) => t a -> a
6 sigma xs = sqrt (mu (fmap ((** 2) . (mu xs -)) xs))
```


`(mu xs -)` calcola la media degli elementi e sottrae un elemento dalla sua media e viene composta con la funzione `(**2)` che eleva al quadrato.

Esercizio 1

Dedurre il comportamento di funzioni che hanno i tipi seguenti (per ogni tipo ci possono essere più risposte ragionevoli)


– La signature

```
1 Ord a => [a] -> a
```

 Haskell


Indica che il contesto prevede che il tipo `a` appartenga alla classe di tipo `Ord` ovvero deve esistere una relazione d'ordine totale sul tipo `a`. La funzione riceve una lista contenente elementi di tipo `a` e ritorna un valore di tipo `a`. Potrebbe essere usata per ottenere il massimo (o il minimo) elemento della lista. Una possibile implementazione sarebbe

```
1 maximum (x:xs) = aux x xs
2 where
3   aux currentMax [] = currentMax
4   aux currentMax (y:ys) = aux (max currentMax y) ys
```

 Haskell


– La signature

```
1 Eq a => [a] -> Bool
```

 Haskell

Indica che il contesto prevede che il tipo `a` appartenga alla classe di tipo `Eq` ovvero supporti comparazioni per determinare l'uguaglianza. La funzione riceve una lista contenente elementi di tipo `a` e ritorna un valore di tipo `Bool`. Potrebbe essere usata per verificare se tutti gli elementi di una lista sono uguali. Una possibile implementazione potrebbe essere


```
1 allEq :: Eq a => [a] -> Bool
2 -- lista vuota si ritorna sempre True
3 allEq [] = True
4 -- verifica se tutti gli elementi della lista sono uguali al primo
5 allEq (x : xs) = all (== x) xs
```

 Haskell

`all` è una funzione d'ordine superiore con tipo `all :: Foldable t => (a -> Bool) -> t a -> Bool`.

– La signature


```
1 Ord a => [a] -> [a]
```

 Haskell

Indica che il contesto prevede che il tipo `a` appartenga alla classe di tipo `Ord` ovvero deve esistere una relazione d'ordine totale sul tipo `a`. La funzione riceve una lista contenente elementi di tipo `a` e ritorna una lista di elementi di tipo `a`. Potrebbe essere usata per ordinare una lista (vedi [Codice 18](#)).

– La signature

```
1 Ord a => [a] -> [a] -> [a]
```

 Haskell

Indica che il contesto prevede che il tipo `a` appartenga alla classe di tipo `Ord` ovvero deve esistere una relazione d'ordine totale sul tipo `a`. La funzione riceve in input due liste e ritorna una lista. Potrebbe essere usata per unire due liste mantenendo l'ordine.

```

1 merge :: Ord a => [a] -> [a] -> [a]
2 merge [] ys = ys
3 merge xs [] = xs
4 merge (x:xs) (y:ys)
5   | x <= y    = x : merge xs (y:ys)
6   | otherwise = y : merge (x:xs) ys

```

» Haskell

– La signature

```

1 (Num a, Foldable t) => t a -> a

```

» Haskell

Indica che il tipo `a` deve appartenere alla classe di tipi `Num` (i tipi numerici) e che il costruttore di tipo `t` deve appartenere alla classe di tipo `Foldable`. La funzione riceve un costruttore di tipo `t` applicato al tipo `a` e restituisce un tipo `a`. Potrebbe essere usata per calcolare somma o prodotto ad esempio.

```

1 sum :: (Num a, Foldable t) => t a -> a
2 sum xs = foldr (+) 0 xs

```

» Haskell

Esercizio 2

Realizzare le seguenti funzioni dando loro il tipo più generico possibile

– `pis xs` moltiplica tutti i numeri in `xs` per π .

```

1 pis :: (Floating a, Functor t) => t a -> t a
2 pis xs = fmap (* 3.14) xs

```

» Haskell

– `sort xs` ordina gli elementi della lista `xs` (vedi [Codice 18](#))

```

1 sort :: (Ord a) => [a] -> [a]

```

» Haskell

– `clip a b xs` trasforma ogni elemento di `xs` in modo che sia non inferiore ad `a` e non superiore ad `b`. Gli elementi che stanno nell'intervallo `[a, b]` non vengono modificati.

```

1 clip :: (Ord a, Functor t) => a -> a -> t a -> t a
2 clip low hi xs = fmap clipOne xs
3   where
4     clipOne x
5       | x < low    = low
6       | x > hi     = hi
7       | otherwise = x

```

» Haskell

A quanto pare non è possibile che il tipo degli elementi della lista (`a`) sia diverso dal tipo degli elementi di confronto, questo perché `(<) :: Ord a => a -> a -> Bool`

– `normalize xs`: normalizza il contenitore `xs` di numeri dividendo ogni elemento di `xs` per l'elemento più grande di `xs`.

```

1 normalize :: (Ord a, Fractional a, Foldable t, Functor t) => t
  a -> t a

```

» Haskell

```
2 normalize xs = fmap (/ maximum xs) xs
```

13.2. Interpretazione delle Classi di Tipo

Le classi di tipo permettono di avere overloading degli operatori in maniera type safe.

13.2.1. Overloading

Abbiamo visto che il compilatore in Haskell non riesce immediatamente a scegliere che tipo inferire sugli operandi in base all'operatore. Ad esempio data una somma (+) ritarda la scelta del tipo delle variabili sulle quali applicare l'operatore. Gli operatori però hanno senso solo su determinati tipi, quindi il compilatore associa dei vincoli ai tipi degli operandi, ad esempio data l'operazione $x + y$, i tipi di x e y devono essere tipi numerici `Num`, quindi il compilatore inferisce la classe di tipo di x e y . Vengono espressi dei vincoli che devono essere soddisfatti dai tipi con i quali vengono istanziati x e y . È possibile definire il polimorfismo in maniera limitata:

- $f :: a \rightarrow a$ equivale a $\forall \alpha. \alpha \rightarrow \alpha$
- $f :: C\ a \Rightarrow a \rightarrow a$ equivale a $\forall C. \alpha \rightarrow \alpha$

La quantificazione universale ($f :: a \rightarrow a$) può essere limitata usando delle classi di tipo.

In Java ad esempio è possibile definire dei metodi polimorfi ma con polimorfismo limitato come segue

```
1 public static <T extends Comparable<T>> int method(T arg) {  
2     return 0;  
3 }
```

Java

```
1 class C a where  
2     m1 :: T1  
3     ...  
4     mn :: Tn
```

Haskell

L'interpretazione è quindi per ogni tipo a istanza di C esistono i “metodi” $m1, \dots, mn$ con tipi $T1, \dots, Tn$.

13.3. Implementazione delle Classi di Tipo

Per implementare le classi di tipo prendiamo come esempio l'implementazione della classe di tipo `Num` in Haskell.

```
1 class Num a where  
2     add :: a -> a -> a  
3     mul :: a -> a -> a  
4     neg :: a -> a  
5  
6 instance Num Int where  
7     add = addInt  
8     mul = mulInt  
9     neg = negInt  
10  
11 instance Num Float where
```

Haskell

```

12  add = addFloat
13  mul = mulFloat
14  neg = negFloat
15
16  square :: Num a => a -> a
17  square x = mul x x
18
19  result :: Int
20  result = square 3

```

È possibile realizzare le classi di tipo senza le classi di tipo (ignoriamo l'algoritmo di type reconstruction/type inference che deve conoscere l'esistenza delle classi di tipo). Il run-time di Haskell contiene già tutte le informazioni per definire le type classes!

```

1  data Num a = Num
2    { add :: a -> a -> a,
3      mul :: a -> a -> a,
4      neg :: a -> a }
5
6  numIntD :: Num Int = Num
7    { add = addInt,
8      mul = mulInt,
9      neg = negInt }
10
11 numFloatD :: Num Float = Num
12   { add = addFloat,
13     mul = mulFloat,
14     neg = negFloat }
15
16 square :: Num a -> a -> a
17 square numD x = mul numD x x
18
19 result :: Int
20 result = square numIntD 3

```

» Haskell

Si definisce un record come un classe nuda senza campi, definire una classe di tipo significa definire un tipo record. Dichiarare che un certo tipo è istanza di una classe significa istanziare un record che ha come tipo il nome della classe e i valori sono quelli per il tipo che stiamo definendo. Il “metodo” square viene tradotto, passando da 1 a 2 argomenti. L'argomento Num a -> è il dizionario che specifica le operazioni possibili sul tipo di a, in questo caso Num. In Haskell otteniamo la funzione del record con la dicitura (mul numD) e la applichiamo ad x. (mul numD) estrae la funzione di moltiplicazione dal dizionario Num. Inoltre chiamiamo square, passando esplicitamente il dizionario numIntD.

Questo impone che i nomi dei campi dei record siano visibili a livello globale, serve assicurarsi che se due record hanno un campo con lo stesso nome serve disambiguare, se vengono definiti in

moduli distinti invece possiamo anteporre al nome dei campi il nome del modulo per evitare che due nomi si sovrappongano.

Consideriamo il codice seguente che mostra un esempio di fattorizzazione dei dizionari.

```
1 f :: C a => a -> a -> a
2 f :: C a -> a -> a -> a
```

» Haskell

Il dizionario con i metodi di C viene passato una volta sola mentre ogni oggetto o trait object porta con sé il dizionario. In altri linguaggi (non Haskell) un trait object viene passato come argomento ad una funzione come una coppia (oggetto, dizionario). In Haskell il dizionario e il valore sono mantenuti separati così da poterli passare una volta sola, invece in Go se avessimo due argomenti dello stesso trait comunque verrebbero passate due coppie, (oggetto-dizionario) (una per ogni argomento). Vedi [Hoogle](#).

14. Monadi

Le monadi nascono dal dogma “pigro è bello”. Solo quando una funzione ha necessità di ispezionare un argomento allora esso viene valutato, questo comportamento prende il nome di laziness (call by need). Dato che i progettisti di Haskell avevano scelto questo paradigma notiamo alcune conseguenze:

- Il linguaggio deve essere **puro**, ovvero privo di effetti collaterali (stampe su terminale, connessioni di rete, interruzioni anomale, assegnamenti). Un linguaggio lazy deve essere puro perché in un linguaggio lazy diventa difficile ricostruire l’ordine di valutazione delle istruzioni. Se la valutazione degli argomenti delle funzioni causa effetti collaterali e il corpo delle funzioni viene eseguito dopo la valutazione degli argomenti allora potrebbero esserci problemi. In un linguaggio pigro questo non avviene, dato che gli argomenti vengono valutati a discrezione del linguaggio quando è strettamente necessario. Questo concetto viene riassunto dal termine trasparenza referenziale.
- Modularità dei programmi (produce/filter/consume) [III]. Pensiamo ad un programma che deve calcolare la lista dei numeri primi. Sappiamo che essendo infinita il programma non termina ma considerando un linguaggio lazy il programma calcolerebbe il risultato on-demand (produce) a seconda di una richiesta (consume).
- Normalizzazione, sappiamo anche che un linguaggio lazy termina mentre non possiamo dire lo stesso di un programma di un linguaggio eager. `const 1 undefined` è l’espressione che corrisponde a un errore dato che la funzione `const` è definita come `const :: a -> b -> a` e valutare `undefined` causerebbe un loop (valutare `undefined` non termina). Ma un linguaggio lazy come Haskell ritorna 1 dato che non valuta un argomento se non necessario.
- Il parallelismo è molto semplice (map/reduce). Abbiamo visto come i microprocessori multi-core. Un linguaggio pieno di side effects non rende semplice il lavoro di parallelizzare nonostante la possibilità di spawnare task paralleli (serve lock management per aree di memoria condivise). Se il linguaggio è puro però non abbiamo problemi dato che non ci sono side effects.

Ci sono però anche conseguenze negative:

! La purezza potrebbe far sembrare i linguaggi inutili. Un linguaggio puro non comunica verso l’esterno dato che non ha side effects.

! È molto difficile prevedere il comportamento dei programmi (imprevedibilità) `const (print 1) (print 2)` cosa stampa?

Come fare I/O in un linguaggio di programmazione puro, come aprire un file da disco, come segnalare eccezioni?

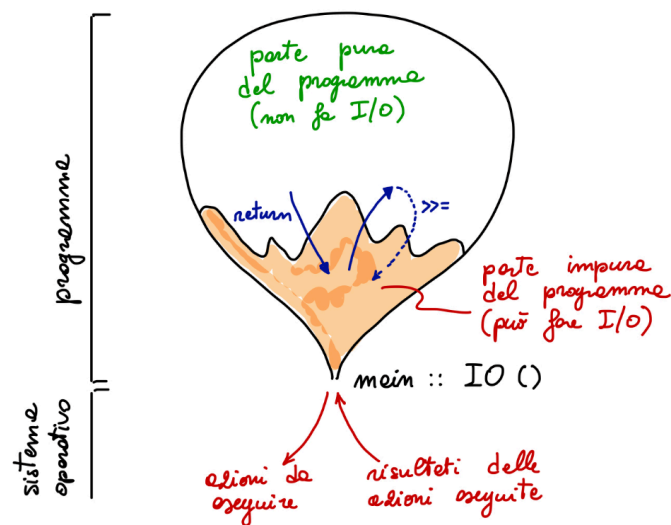


Figura 11: Sketch di una monade

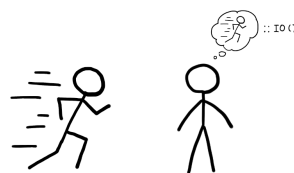


Figura 12: Pensare di fare qualcosa è diverso da farlo

Definizione: Trasparenza Referenziale

Se $a = b$ posso usare a al posto di b e viceversa.

```
1 f x = a + a
2   where
3     a = g x
4     b = h 3
```

» Haskell

Il compilatore può in tranquillità riscrivere la funzione in questo modo:

```
1 f x = g x + g x
```

» Haskell

Ma solo se il linguaggio non comporta side effects. Questo porta anche a ottimizzazione del codice.

Un programma basato su monadi è un programma che crea delle azioni che se eseguite generano side effects (stampa, leggi da file ecc..). Quando programiamo in maniera monadica il programma è puro, non esegue azioni ma produce idee di azioni che se eseguite hanno un effetto (dichiarazioni di intenti). Il programma ha una parte pura (predominante), una seconda parte del programma crea e combina delle azioni che se eseguite generano un effetto.

Inizialmente le monadi vennero introdotte per risolvere il problema dell'I/O, una volta sdoganata l'idea potremo descrivere matematicamente la computazione che può avere effetti collaterali.

Ci si è resi conto che le monadi possono risultare utili in una miriade di contesti. Troviamo contesti per lavorare con monadi anche in Scala, Python, Perl...

Le monadi sono un concetto nell'area della semantica dei linguaggi per descrivere matematicamente computazioni che possono avere effetti collaterali [12].

Studiamo ora tre diverse monadi con effetti diversi esprimibili con esse, partiamo descrivendo un valutatore di espressioni

```
1  data Expr = Const Int
2          | Add Expr Expr
3          | Sub Expr Expr
4          | Div Expr Expr
5
6  eval :: Expr -> Int
7  -- se valutiamo una costante ritorniamo sè stessa
8  eval (Const n) = n
9  -- valutazione ricorsiva sui due sotto alberi
10 eval (Add t s) = eval t + eval s
11 eval (Sub t s) = eval t - eval s
12 -- applica div (divisione intera con troncamento) ai due sottoalberi
13 eval (Div t s) = eval t `div` eval s
```

Codice 19: Valutatore di espressioni in Haskell

Proviamo ad aggiungere al valutatore di espressioni definito in Codice 19 la gestione della divisione per zero. La divisione è un'operazione parziale dato che non possiamo dividere per zero. Anche Haskell ha una sorta di impurità dato che esistono le eccezioni, per rappresentare però in maniera più esplicita che la valutazione che può fallire vogliamo definire eval come una funzione parziale (può fallire). Il risultato di eval non è un Int ma un MaybeInt.

```

1  eval :: Expr -> Maybe Int
2  eval (Const n) = Just n
3
4  eval (Div t s) = case eval t of
5      Nothing -> Nothing
6      Just n   -> case eval s of
7          Nothing -> Nothing
8          (Just 0) -> Just 0
9          (Just m) -> Just (n `div` m)
10
11 eval (Add t s) = case eval t of
12     Nothing -> Nothing
13     Just n   -> case eval s of
14         Nothing -> Nothing
15         Just m   -> Just (n + m)
16
17 eval (Sub t s) = case eval t of
18     Nothing -> Nothing
19     Just n   -> case eval s of
20         Nothing -> Nothing
21         Just m   -> Just (n - m)

```

» Haskell

Codice 20: Valutatore di espressioni in Haskell con gestione della divisione per zero

Notiamo come la business logic sia annegata da molto codice che deve gestire diversi casi. Non è una gestione soddisfacente, per quanto pura. Si inizia a insinuare in noi l'idea che il tipo `Maybe Int` non sia così diverso da un tipo `IO`. La funzione `eval` non ritorna più il risultato ma ritorna un'azione che se eseguita produce un risultato. `eval` resta comunque lontano dalla versione che otterremmo con un linguaggio impuro. Introduciamo ora un contatore delle operazioni di divisione, con un linguaggio impuro sarebbe semplicissimo, basta aggiungere una variabile intera che viene incrementata ogni volta che incontriamo una divisione. Siamo in grado di realizzarlo in un linguaggio di programmazione puro? non abbiamo variabili che sono incrementabili. Incrementare un contatore significa leggere il contatore, incrementarlo e produrre il nuovo valore del contatore dopo l'incremento. Possiamo realizzare una funzione che trasforma un valore in input e la funzione dice il nuovo valore del contatore.

```

1  -- simile a Maybe a
2  -- Maybe a ritorna Just a o Nothing
3  -- Counter a ritorna la coppia contatore e risultato dell'espressione
4  type Counter a = Int -> (a, Int)
5
6  data Expr = Const Int
7             | Add Expr Expr
8             | Sub Expr Expr
9             | Div Expr Expr
10
11 --eval :: Expr -> Counter Int
12 eval :: Expr -> Int -> (Int, Int)
13 eval (Const n) x = (n, x)
14 eval (Div t s) x =
15     let (m, y) = eval t x in
16     let (n, z) = eval s y in
17     (m `div` n, z+1)

```

Codice 21: Valutatore di espressioni in Haskell con contatore di divisioni

La computazione pura è il primo mattoncino fondamentale per definire le monadi. In [Codice 21](#) non consideriamo tutte le casistiche che abbiamo già gestito in precedenza (divisioni per zero). Notiamo però che anche in [Codice 21](#) la logica viene annegata da boilerplate e commettere errori è molto facile.

Ci sono diverse similitudini in [Codice 20](#) e [Codice 21](#), notiamo come nel caso di `const` la computazione è pura in entrambi i casi in quanto nel primo produciamo un valore `Just n` mentre nel secondo caso si produce `(n, x)`. La computazione pura è un concetto che **dipende dalla monade** ma in entrambi i casi di `Const` la computazione è pura. Vediamo ora se ci sono altre analogie per l'espressione `Div`. Notiamo come il concetto è il medesimo in entrambi i frammenti di codice ciò che cambia è la gestione degli effetti collaterali e come vengono composte le azioni. Comporre azioni significa controllare potenziali fallimenti in entrambi i casi. Nel primo caso l'effetto collaterale è la valutazione che fallisce, nel secondo è l'incremento del contatore.

Riusciamo ad esprimere in maniera astratta cosa significa computazione pura e cosa significa comporre azioni nei due casi? sì e lo facciamo definendo i due operatori fondamentali di ogni monade. Data una generica monade `M` (dove `M` può essere `Maybe`, `Counter` o `Output`) sono definite le seguenti:

- `return` produce una computazione all'interno della monade il cui risultato è direttamente il valore dell'argomento

```

1  return :: a -> M a

```

- `bind` che serve per combinare in sequenza due computazioni della monade `M` dove però la seconda può richiedere il risultato della prima. Il tipo può essere spaventoso ma in realtà il primo argomento è la prima computazione della monade. Il secondo argomento è una funzione porta a all'interno di una computazione della stessa monade `M`. Non a caso il dominio della funzione nel secondo argomento coincide con il tipo del risultato della prima computazione. `M a` non restituisce un valore di tipo `a` direttamente, ma rappresenta una computazione che potenzialmente produrrà un valore di tipo `a`.

```
1 (>=) :: M a -> (a -> M b) -> M b
```

» Haskell

Inoltre è possibile definire delle operazioni specifiche per la monade.

```
1  -- Disabilita le funzioni default return e >=>
2  import Prelude hiding (return, (>=>))
3
4  data Expr = Const Int | Div Expr Expr
5
6  return :: a -> Maybe a
7  return = Just
8
9  (>=>) :: Maybe a -> (a -> Maybe b) -> Maybe b
10 (>=>) Nothing _    = Nothing
11 (>=>) (Just a) f    = f a
12
13 abort :: Maybe a
14 abort = Nothing
15
16 evalM :: Expr -> Maybe Int
17 evalM (Const n) = return n
18 evalM (Div t s) =
19   evalM t >=> \m ->
20   evalM s >=> \n ->
21   if n == 0 then Nothing
22   else return (m `div` n)
```

» Haskell

Codice 22: Gestione di divisioni per zero in Haskell con versione monadica

`evalM t`: valuta il numeratore `t`, ottiene `Maybe Int`. Quindi `(>=>)` non significa semplicemente “fai A poi fai B” ma significa “fai A, prendi il risultato, poi calcola B in base a quel risultato.”
Scrivere

```
1 evalM s >=> \n -> return (n + 1)
```

» Haskell

Equivale a scrivere

```
1 (>=) (evalM s) (\n -> return (n+1))
```

» Haskell

```

1  return :: a -> Counter a
2  -- il significato di return è "computazione pura"
3  -- una computazione è pura se il contatore non viene alterato
4  return a = \x -> (a, x)
5  -- questo significa essere computazione pura nella monade Counter
6  -- avremmo anche potuto definire return come
7  return a x = (a, x)
8  -- ma vogliamo enfatizzare il fatto che Counter a è il tipo di una funzione
9
10 type Counter a = Int -> (a, Int)
11 (>=>) :: Counter a -> (a -> Counter b) -> Counter b
12 (>=>) :: m f = \x -> let (a, y) = m x in f a y
13
14 -- dobbiamo produrre qualcosa di tipo Counter () "()" letto come unit"
   ovvero tupla vuota
15 tick :: Counter ()
16 -- dato il valore corrente produce unit e il nuovo contatore
17 tick = \x -> ((), x + 1)
18
19 evalM :: Expr -> Counter Int
20 evalM (Const n) = return n
21 evalM (Div t s) =
22     evalM t >=> \m ->
23     evalM s >=> \n ->
24     tick >=> \() ->
25     return (m `div` n)

```

Codice 23: Conteggio di divisioni in Haskell con versione monadica

Riepiloghiamo dando una definizione di Monade, il termine Monade deriva dal termine monoide, una struttura algebrica dotata di un'operazione associativa $(a * b) * c = a * (b * c)$ che equivale all'operatore bind $>=>$. Un monoide ha un elemento neutro $a * e = e * a = a$, ovvero a composto con e ed e composta con a si equivalgono e sono proprio a . La legge di neutralità di e ci informa sul fatto che comporre una computazione che non ha effetti collaterali con un'altra operazione equivale a considerare solo quest'ultima. Una monade non è propriamente un monoide perché quando componiamo due computazioni in una monade è necessario trasferire informazioni dalla prima alla seconda computazione. Bind è sbilanciato in quanto serve definirlo proprio a questo scopo. Possiamo comunque formulare in maniera quasi equivalente le proprietà di un monoide con 3 leggi.


14.1. Le 3 Leggi di una Monade

1. $\text{return } a \gg= f \equiv f \ a$ notiamo che return deve essere definito in modo tale che non aggiunga nessun effetto collaterale alla nostra computazione. Usarlo a sinistra del bind equivale ad applicare direttamente a alla funzione f .
2. $m \gg= \text{return} \equiv m$ notiamo return non aggiunge effetti collaterali e possiamo scartarlo.

3. $(m \gg= \lambda a \rightarrow n) \gg= f \equiv m \gg= (\lambda a \rightarrow n \gg= f)$ ⁹ Componiamo un'azione che a sua volta è un'azione composta. Dato che il bind in qualche modo è associativo possiamo riscrivere l'operazione componendo in maniera associativa le operazioni

Riassumendo: le leggi 1. e 2. dicono che `return` si comporta come elemento neutro (computazione pura priva di effetti) mentre la legge 3. dice che `>>=` è “associativo”. Queste leggi però vanno dimostrate per ogni monade. Vediamo se valgono per la monade `Maybe`.

```
1  return :: a -> Maybe a
2  return a = Just a
3
4  (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
5  (>>=) Nothing _ = Nothing
6  (>>=) (Just a) f = f a
7
8  -- la prima legge afferma che return a >>= f a
9    return a >>= f =
10 = Just a >>= f   =
11   f a
12
13 -- la seconda legge afferma che m >> return = m
14 -- se non facciamo assunzioni su m non andiamo avanti
15 m >>= return
16
17 -- caso m = Nothing
18 = Nothing >>= return
19 = Nothing = m
20
21 -- caso m = Just a
22 = return a
23 = Just a
24 = m
25
26 -- la terza legge afferma che (m >> \a -> n) >>= f
27 (m >> \a -> n) >>= f
28
29 -- caso m = Nothing
30 = (Nothing >>= \a -> n) >>= f
31 = Nothing >>= f
32 = Nothing
33
34 m >>= (\a -> n >>= f)
35
36 -- caso m = Nothing
```

 Haskell

⁹Se `a` non compare libera in `f`

```

37   = Nothing >=> (\a -> n >=> f)
38   = Nothing

```

14.2. La classe Monad

Se m è un costruttore di tipo che appartiene alla classe `Monad` sono definite le funzioni `return` e `bind`.

```

1  class Monad m where
2    return :: a -> m a
3    (>=>) :: m a -> (a -> m b) -> m b

```

» Haskell

`return` e `bind` sono funzioni polimorfe in a e b .

m non è un tipo bensì un costruttore di tipo! stiamo definendo una famiglia di costruttori di tipo, quindi m è una variabile su costruttori di tipo. `Monad` è semplicemente una famiglia di costruttori di tipo. Un'utilità di libreria molto utile è la funzione “and then” `>>`. Serve perché in alcuni casi il risultato della prima computazione non ci interessa (ad esempio quando abbiamo combinato l'azione `tick` non ci siamo interessati del suo risultato), capita a volte che è necessario scartare il risultato ma comunque combinare gli effetti collaterali delle due azioni.

```

1  (>>) :: Counter a -> Counter b -> Counter b
2  (>>) as bs = as >=> const bs -- caso speciale di bind praticamente
   scartiamo il primo argomento
3
4  evalN :: Expr -> Counter Int
5  evalN (Const n) = return n
6  evalN (Div t s) =
7    evalN t >=> \m ->
8    evalN s >=> \n ->
9    tick >>
10   return (m `div` n)

```

» Haskell

Quando usiamo “and then” anziché “bind” la terza legge della monade è formulabile in maniera molto simile ai monoidi.

```

1  (as >> bs) >> cs = as >> (bs >> cs)

```

» Haskell

Un'altro aspetto sintattico molto importante è la notazione `do`, sostanzialmente zucchero sintattico per definire monadi senza utilizzare lambda expressions.

```

1  evalN (Div t s) = do
2    m <- evalM t
3    n <- evalM s
4    if n == 0 then abort
5    else return (m `div` n)

```

» Haskell

14.3. Monadi Composte

E se volessimo un valutatore che gestisce le divisioni per zero e conta le divisioni? è concepibile una monade che incorpora entrambi gli effetti, non è impossibile partire da zero e definire `return`

e `>=>` per esprimere operazioni che possono fallire e gestiscono un contatore. È possibile ma non auspicabile.

Consideriamo un problema di natura concettuale ovvero la soluzione causerebbe ambiguità semantica. Pensandoci bene `eval` può avere due tipi distinti. In base al tipo che scegliamo di dare ad `eval` cambia totalmente il modo con cui gestiremo le operazioni.

```
1 -- semantica "tradizionale"
2 eval :: Expr -> Int -> (Maybe Int, Int)
3
4 -- semantica "transazionale"
5 eval :: Expr -> Int -> Maybe (Int, Int)
```

» Haskell

Nella semantica tradizionale (in un linguaggio impuro) quando un'espressione viene valutata la valutazione avanza per un po' con successo (magari il contatore viene incrementato) poi si incontra una divisione per zero e si lancia un'eccezione. Ma il contatore è stato incrementato! l'effetto continua ad essere presente. La semantica transazionale, invece, ci dice che la valutazione arriva alla fine, ottenendo il risultato e il contatore oppure `Nothing`. È come dire che una divisione per zero comporta non modificare il contatore. Questa è la semantica utilizzata in molti database distribuiti.

Per definire monadi composte serve riflettere sul significato delle singole computazioni. Se le computazioni hanno dei tipi cambiarli cambia radicalmente tutto.

Invece di definire una monade per rappresentare computazioni che possono avere certi effetti definiamo un "monad transformer" ovvero una trasformazione di monade che aggiunge certi effetti ad un'altra monade già esistente. Si ragiona in maniera incrementale.

```
1 type MaybeT m a = m (Maybe a)
2
3 instance Monad m => Monad (MaybeT m) where
4   return = return . Just -- equivalente a return a = return (Just a)
5   (>=>) m f = m >=> \x ->
6     case x of
7       Nothing -> return Nothing
8       Just a -> f a
9
10 abort :: Monad m => MaybeT m a
11 abort = return Nothing
```

» Haskell

`MaybeT` aggiunge il fallimento all'interno di un'altra monade `m`. Haskell rifiuterebbe questa definizione per i limiti imposti sugli alias di tipo (per il tipo `m` applicato a `Maybe a`). Il codice è comunque corretto dal punto di vista operativo. `m` rappresenta un'altra monade e `MaybeT` aggiunge la possibilità di fallimento della computazione all'interno di un'altra monade `m`. Stiamo sostanzialmente inserendo `Maybe` dentro la monade `m`.

Cosa significa avere una computazione pura nella monade `MaybeT m`? `return` è definito come

```
1 type MaybeT m a = m (Maybe a)
2 return :: a -> MaybeT m a
3 return :: a -> m (Maybe a)
```

» Haskell

Attenzione che `return` non è definita ricorsivamente. `return` (come `>=>`) sono funzioni overloaded quindi il `return` interno così come `>=>` sono definiti internamente alla monade `m`. Proviamo a definire la combinazione della monade `Counter` e della monade `MaybeT`. Qui sorge un altro problema, l'azione `tick` è specifica della monade `Counter` (interna) che però stiamo trasformando usando `MaybeT`. Vorremmo continuare ad eseguire sulla monade interna le operazioni che avevamo definito per essa.

```
1  type MaybeCounter = MaybeT Counter
2
3  evalM :: Expr -> MaybeCounter Int
4
5  evalM (Const n) = return n
6  evalM (Div t s) =
7      evalM t >=> \m ->
8      evalM s >=> \n ->
9      tick >>          -- ERRORE!
10     -- lift tick >> ->  Ok
11     if n == 0 then abort
12     else return (m 'div' n)
13
14     -- tick è un'azione della monade interna Counter e non della monade
15     MaybeCounter, vorremo poter continuare ad utilizzare azioni della monade
16     interna
```

Non possiamo immediatamente utilizzare o mescolare azioni di monadi diverse. Dobbiamo effettuare il cosiddetto `lifting`.

```
1  class MonadT t where
2      lift :: Monad m => m a -> t m a
3
4  instance MonadT MaybeT where
5      lift m = m >=> (return . Just)
```

Affinché `MaybeT` sia un monad transformer dobbiamo fornire ad Haskell la funzione `lift` sul monad transformer `MaybeT`. In questo modo riusciamo a promuovere azioni di una monade `m` in azioni della monade `MaybeT m`. Questo può richiedere un certo sforzo mentale, tutto per soddisfare il type checker di Haskell. In questo modo ricicliamo tutte le azioni disponibili per la monade originale sulla monade trasformata.

La versione finale però è ancora diversa perché Haskell non accetta applicazioni parziali di alias di tipo. `type MaybeT m a = m (Maybe a)` viene applicato solo parzialmente solo a `m` `instance Monad m => Monad (MaybeT m) where` (non viene applicato ad `a`). La soluzione è definire `MaybeT` non come alias di tipo ma come un nuovo tipo (definendo un tipo record).

```
1  newtype MaybeT m a = MaybeT { unMaybeT :: m (Maybe a) }
2
3  instance Monad m => Monad (MaybeT m) where
4      return = MaybeT . return . Just
```

```

5  (>>=) m f = MaybeT $
6      unMaybeT m >>= \x ->
7          case x of
8              Nothing -> return Nothing
9              Just a -> unMaybeT (f a)

```

Con MaybeT possiamo aggiungere fallimenti a qualsiasi monade, inclusa Counter, infatti se definiamo la trasformazione di una monade CounterT possiamo aggiungere un contatore a qualsiasi monade inclusa Maybe... avremmo 2 trasformazioni di monadi + 2 monadi. Non risolviamo il problema della duplicazione di codice, prolifera codice ripetuto. Però se definissimo una monade banale che rappresenta computazioni pure prive di effetti collaterali chiamandola **monade identità** possiamo ottenere tutte le altre monadi per trasformazioni successive dalla monade identità.

```

1  -- nella lib std è chiamata identity
2  type Id a = a
3  instance Monad Id where
4      return = id -- la funzione identità
5      (>>=) a f = f a -- banale applicazione funzionale
6
7  -- Maybe = MaybeT Id
8  -- Counter = CounterT Id
9  -- praticamente decoriamo la monade identità

```

» Haskell

Codice 24: Monade Identità in Haskell

Come diventa la monade Counter con l'introduzione della monade identità.

```

1  type CounterT m a = Int -> m (a, Int)
2  instance Monad m => Monad (CounterT m) where
3      return a = \x -> return (a, x) -- si riceve il valore corrente del
4      contatore e creiamo la computazione pura nella monade m che produce la
5      coppia
6      (>>=) m f = \x -> m x >>= \(a, y) -> f a y
7
8  tick :: Monad m => CounterT m ()
9  tick = \x -> return ((), x + 1)

```

» Haskell

In un linguaggio lazy come Haskell si perde il controllo dell'ordine di esecuzione, uno statement print (presente in altri linguaggi) causerebbe confusione perché non sempre riusciremmo a capire quando e se lo statement viene eseguito.

```

1  length [print 2, print 4]
2  -- riceviamo 2

```

» Haskell

Viene necessario introdurre una monade specifica per I/O, che è molto simile alla monade Counter.

```

1  type IO a = World -> (a, World)
2

```

» Haskell

```
3  -- return
4  -- (>=)
5
6  main :: IO ()
7  main =
8      putStrLn "ciao" >= \() ->
9      print 3 >= \() ->
10     putStrLn (show True)
```

Dove World è un tipo astratto (non forgiabile dal programmatore) che descrive lo stato del “mondo” prima e dopo l’esecuzione di una data azione. Abbiamo la garanzia che scrivere un programma con la monade I/O genera stampe sequenziali nell’ordine atteso

Bibliografia

- [1] C. Hewitt, P. Bishop, e R. Steiger, «A universal modular ACTOR formalism for artificial intelligence», in Proceedings of the 3rd International Joint Conference on Artificial Intelligence, in IJCAI'73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.
- [2] J. Armstrong, «A history of Erlang», in Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, in HOPL III. San Diego, California: Association for Computing Machinery, 2007, pp. 6–1. doi: [10.1145/1238844.1238850](https://doi.org/10.1145/1238844.1238850).
- [3] Github, «Github Receive Implementation in Erlang». [Online]. Disponibile su: <https://github.com/erlang/otp/issues/6176>
- [4] M. Dinelli, «Excalidraw Notes on Mark & Sweep Algorithm». [Online]. Disponibile su: <https://excalidraw.com/#json=u6dCV9RQHjD4pDfbHL2Ds,N5Uk-IZ59dcklYFJnWd2wA>
- [5] Wikipedia, «John Backus — Wikipedia, L'enciclopedia libera». [Online]. Disponibile su: http://it.wikipedia.org/w/index.php?title=John_Backus&oldid=142937489
- [6] J. Backus, «Can programming be liberated from the von Neumann style? a functional style and its algebra of programs», Commun. ACM, vol. 21, fasc. 8, pp. 613–641, ago. 1978, doi: [10.1145/359576.359579](https://doi.org/10.1145/359576.359579).
- [7] P. Hudak e M. P. Jones, «Haskell vs. Ada vs. C++ vs. awk vs.... an experiment in software prototyping productivity», Contract, vol. 14, fasc. 92-C, p. 153, 1994.
- [8] H. Sutter, «A Fundamental Turn Toward Concurrency in Software», 2008. [Online]. Disponibile su: <https://api.semanticscholar.org/CorpusID:59749079>
- [9] Wikipedia contributors, «James Gosling — Wikipedia, The Free Encyclopedia». [Online]. Disponibile su: https://en.wikipedia.org/w/index.php?title=James_Gosling&oldid=1294131107
- [10] P. Hudak, J. Hughes, S. Peyton Jones, e P. Wadler, «A history of Haskell: being lazy with class», in Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, in HOPL III. San Diego, California: Association for Computing Machinery, 2007, pp. 12–11. doi: [10.1145/1238844.1238856](https://doi.org/10.1145/1238844.1238856).
- [11] J. Hughes, «Why Functional Programming Matters», The Computer Journal, vol. 32, fasc. 2, pp. 98–107, 1989, doi: [10.1093/comjnl/32.2.98](https://doi.org/10.1093/comjnl/32.2.98).
- [12] E. Moggi, «Notions of computation and monads», Information and Computation, vol. 93, fasc. 1, pp. 55–92, 1991, doi: [https://doi.org/10.1016/0890-5401\(91\)90052-4](https://doi.org/10.1016/0890-5401(91)90052-4).