

Emerging Programming Paradigms

A.A. 2024/2025

Indice

1. Erlang	4
1.1. Polimorfismo	7
1.1.1. Monomorfizzazione (C++, Rust)	8
1.1.2. «Alla C» (Rust in casi residuali)	8
1.1.3. Rappresentazione uniforme dei dati (Erlang, OCaml, Haskell, ...)	9
1.2. Rappresentazione degli atomi	12
1.3. Pattern matching	13
1.4. Ricorsione	15
1.5. Eccezioni	17
1.6. Programmazione parallela e concorrente	20
1.7. Hot swap	24
1.8. Gestione fallimenti	25
1.8.1. Link	25
1.8.2. Comando che solleva eccezione	26
1.8.3. Monitor	26
1.9. Distribuzione	27
2. Effetti algebrici	29
3. Gestione della memoria	31
3.1. Tecniche di gestione della memoria	31
3.1.1. Reference Counting	32
3.1.2. Mark & sweep	35
3.1.3. Mark & sweep generazionale	37
4. Funzioni di prima classe e chiusure	40
4.1. Chiusura	41
4.1.1. Currying	41
4.1.2. Incapsulamento	42
4.1.3. Laziness	42
4.1.4. Comportamento in Java	43
4.2. Oggetti vs chiusure	43
4.2.1. Comportamento in C	43
5. Classi, interfacce e trait	46
5.1. Linguaggi object-based	46
5.2. Linguaggi class-based	46
5.3. Ereditarietà	47
5.4. Composizione	48
5.5. Interfacce	49
5.6. Trait	49
5.6.1. Trait in Rust	51
6. Classi di tipi	54
6.1. Type checking/inference/reconstruction	55
6.2. Costruttore di tipo	58
6.3. Interpretazioni delle classi di tipo	59
7. Monadi	62
7.1. Esempio: valutatore di espressioni	63

7.2. Monoide	67
7.3. Leggi di una monade	67
7.3.1. Dimostrazione per <code>Maybe</code>	67
7.3.2. Dimostrazione per <code>Counter</code>	69
7.4. Classe <code>Monad</code>	69
7.5. Operatore and-then <code>>></code>	70
7.6. Notazione <code>do</code>	71
7.7. Monadi composte	71
7.7.1. Lifting di operazioni	73
7.8. Monade identità	74
8. Rust	75
8.1. Ownership	75
8.2. Reference	77
8.3. Borrowing	77
8.4. Lifetime	78
8.5. Slice	80
8.6. Chiusure	81
8.7. Smart pointer	81
8.7.1. Box	81
8.7.2. Reference counter (<code>Rc</code>)	82
8.7.3. RefCell	84
8.7.4. Weak	84
8.7.5. Mutex	85
8.7.6. Arc	85
9. GADT	86
9.1. Uguaglianza	91

1. Erlang

Cambiare paradigma vuol dire cambiare punto di vista e dunque cambiare approcci, metodi e costrutti.

Erlang è il linguaggio ad attori più mainstream. Elixir usa sempre la VM di Erlang ma tiene una sintassi molto più simile a linguaggi più in voga, come Ruby.

Nasce come linguaggio logico, infatti riprende molti aspetti sintattici di Prolog. È funzionale ma non è tipato.

I suoi programmi sono distribuiti (thread concorrenti su macchine diverse nella rete). In un linguaggio multi paradigma dobbiamo garantire delle proprietà in output da quelle in input, e questo multi paradigma semplifica l'output ma complica le proprietà in input.

Un attore è composta da:

- PID (id univoco dell'attore, un nome logico);

Un **nome logico** identifica qualcuno senza esplicitare la tipologia della rete; dunque può essere un attore anche presente nella propria macchina ma in un diverso core. È il motivo per cui Erlang è nato già per funzionare in maniera multi-core. Solo chi conosce il nome logico dell'attore è capace di comunicare con tale attore.

Un nome fisico descrive come raggiungere e comunicare con un processo (in genere IP + porta)

- Mailbox (coda di ricezione dei messaggi);

Un attore riceve i messaggi e li mette in una coda di esecuzione.

Un'azione fa computazione interna e può inviare messaggi in maniera async ad altri attori.

Un attore può creare e terminare altri attori.

- Behaviour (map di messaggi → azioni).

In ogni attore vi è 1 solo thread, e non condivide risorse con altri attori.


In un ambiente ad attori, in genere, si hanno moltissimi attori ma solo pochi di loro lavorano in contemporanea.

Cosa succede quando vi è un guasto?

- In un ambiente concorrente si possono controllare tutti i thread in concorrenza e poi sincronizzarli per riparare tale guasto.
- In un ambiente distribuito bisogna anche considerare i guasti che avvengono nella rete.

Esempio di codice in Erlang:

```
1  % "-" è perché è una direttiva. Il modulo è un file.
2  -module(hello).
3  % quali funzioni esportare all'esterno. cc/1 vuol dire che cc è la funzione e 1
   l'arietà (numero di parametri).
4  -export([cc/1]).
5
6  % le funzioni si scrivono con le lettere minuscole
7  % i parametri formali iniziano con la maiuscola
8  % "->" separa il behaviour
```

 Erlang

```

9  cc(Bal) ->
10  receive
11      % un atomo è qualcosa che si scrive con la minuscola ma non è una funzione
12      print -> io:format("Balance is ~p ~n", [Bal]),
13              cc(Bal) ;
14      % è una coppia, passo l'atomo e una variabile
15      {put, N} -> cc(Bal+N) ;
16      % l'operazione di send è data da PID ! <messaggio>
17      {get, PID} -> PID ! Bal, cc(Bal) ;
18      % qui ritorna un atomo e basta
19      exit -> ok
20  end.

```

Un pattern è una forma dell'input, simil modo come qualsiasi altro linguaggio funzionale.

Il codice di Erlang gira in maniera simile a Java: la VM è chiamata **BEAM** e dunque il codice viene compilato in bytecode e poi interpretato dalla VM.

In una macchina fisica c'è una VM chiamata **nodo**. Quando si lancia un nodo si avvia un altro attore, il quale è la shell. Erlang rappresenta come attore qualsiasi cosa estranea al linguaggio, ecco perché la shell è considerata un attore.

```

1  Erlang/OTP 27 [erts-15.2.2] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-
   threads:1] [jit:ns]
2
3  Eshell V15.2.2 (press Ctrl+G to abort, type help(). for help)
4  1> c(hello).
5  {ok,hello}
6
7  2> PID = spawn(hello, cc, [10]).
8  <0.97.0>
9
10 3> PID ! print.
11 Balance is 10
12 print
13
14 4> PID ! {put, 2}.
15 {put,2}
16
17 5> PID ! print.
18 Balance is 12
19 print
20
21 6> PID ! {get, self()}.
22 {get,<0.103.0>}
23
24 7> receive MSG -> MSG end.
25 12

```

26

27 % questo rimane in deadlock perché non ha nessun messaggio da processare ma non
c'è modo di inviarne nuovi

28 8> receive MSG -> MSG end.

La funzione `spawn` restituisce il PID. Il PID può essere diverso sui vari nodi, l'importante è come viene interpretato.

Nella BEAM si può fare aggiornamento del codice senza metterlo down, dunque a runtime.

La libreria standard di Erlang è chiamata OTP, storicamente perché si interfacciava con la rete telefonica.

La descrizione originale di OOP è basata su scambio di messaggi (non invocazione di metodi) sincrono e monothread: il controllo passa da un oggetto all'altro. Nella programmazione ad attori si ha un concetto simile ma asincrono.

Un ambiente con memoria condivisa non è naturale: immaginiamo 50 core che tentano di accedere alla stessa memoria nello stesso istante. Ambienti del genere sono pessimi anche per fault tolerance.

Il crash di un attore, il deadlock e la recovery, sono gestiti in maniera più semplice.

L'ordine è causale, un messaggio 2 è causato da un messaggio 1. Questo permette di avere una sorta di ordine in un ambiente asincrono. Message-passing async e ricezione out-of-order.

La fault tolerance è gestita in maniera «Let it fail»: il processo in errore viene distrutto e così anche tutti gli attori che comunicavano con esso. Brutalmente risolve la questione.

Un attore genera un tot di attori figli che aiutano il processo; il padre attende la fine del processo. Si ha un sistema gerarchico di computazioni in modo tale che:

1. se uno dei figli fallisce, uccide tutti i fratelli;
2. il padre ricrea tutti i figli;
3. se i figli vengono uccisi troppo spesso, il padre si suicida;
4. il padre uccide tutti i fratelli;
5. il padre del padre ricrea il padre + fratelli del padre;
6. il padre ricrea i figli.

Dunque la creazione/uccisione degli attori è qualcosa che avviene in maniera veloce. Nella programmazione concorrente invece non bisogna uccidere altri thread; ma questo è perché si ha la memoria condivisa. Per fortuna che Erlang non condivide la memoria.

1 attore = 1 language thread: ogni attore è rappresentato a runtime del sistema, ed è esso che schedula gli attori alternandoli, senza che l'OS lo sappia. Il context switch (che avviene mediante interrupt stoppando il codice per avviarne un altro e così via; in fase di bootstrap si salva una tabella con un codice del processo) per language thread (aka collaborativo) è una versione di passaggio al controllo con user space (tipo lo `yield`) più veloce. Erlang si basa su questo.

La creazione di un attore in C avviene allocando in stack una dimensione sufficiente per gestire tale attore. Dato che Erlang non fa così, esso evita stackoverflow: ha una memory footprint molto piccola. La crescita di uno stack è costante ammortizzata perché la parte di spostamento degli n elementi precedenti viene fatta dagli altri.

Elixir permette una specie di polimorfismo e si hanno macro, a differenza di Erlang.

In Erlang non c'è un costrutto per dichiarare i tipi. I booleani in genere vengono definiti in userspace; in Erlang sono degli atomi `true` e `false`, non un valore di tipo booleano. I tipi di dati atomici non contengono altri dati, dunque non sono divisibili. `==` e `/=` sono il test di uguaglianza e disuguaglianza per tipi diversi (float e int ad esempio).

Si distinguono processi interi e parziali dai PID per i primi e le porte per gli altri.

Erlang ha due tipi di dati non atomici (system defined): array/tuple e liste. Una lista impropria (tipo `[2|4]`) solleva un'eccezione. La sottrazione di lista `--` è associativa a destra.

Le funzioni sono oggetti di prima classe: manipolate come se fossero un numero.

Quando si fa pattern-matching Erlang controlla se la guardia ha side effect (= operazioni diverse dalla semplice lettura di variabili od operazioni di return; esempi sono variabili non locali, variabili statiche o modificate per reference, raise di eccezioni, I/O e chiamate a funzioni con side effect. Non avere side effect mi permette una migliore verifica formale). All'interno delle guardie si possono fare richieste di funzioni come `is_integer` e `length`.

Un esempio di comprensione di lista è dato da:

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

Si può fare pattern matching sulla rappresentazione dei bit.

```
1 1> N = 16#7AC.
2 1964
3
4 2> <<R:4, G:4, B:4>> = <<N:12>>.
5 <<122,12:4>>
6
7 3> R.
8 7
9
10 4> G.
11 10
12
13 5> B.
14 12
15
16 6> <<N>>.
17 <<"-">>
```

1.1. Polimorfismo

Un sistema di tipi è un meccanismo a compile-time che cerca di capire se il programma rispetta delle proprietà che altrimenti sarebbero indecidibili.

Un «tipo» è una proprietà che viene associata ad un termine. Un sistema di tipi descrive attraverso delle regole le operazioni che possono essere calcolate da un termine.

Funzioni monoforme: lavorano correttamente se il dato in input ha una certa implementazione.

Allocare, deallocare, spostare e copiare sono operazioni che non richiedono il tipo del dato, basta guardare la rappresentazione dei bit in memoria.

Funzioni polimorfe: si può ricevere un input di diverso tipo.

```
1  # let swap (x, y) = (y, x);;
2  val swap : 'a * 'b -> 'b * 'a = <fun>
3
4  # let f g x = g x x;;
5  val f : ('a -> 'a -> 'b) -> 'a -> 'b = <fun>
6
7  # f (+) 3;;
8  - : int = 6
9
10 # f (+) "hi";;
11 Error: This expression has type string but an expression was expected of type
12         int
```

La `g` deve essere coerente con tutta la `f`. È chiamato poliformo uniforme oppure generico/template.

Ci sono i tipi per misurare la dimensione del dato e quelli che garantiscono la sequenza di bit se rispetta quello che noi intendiamo per «tipo».

1.1.1. Monomorfizzazione (C++, Rust)

Bisogna avere un linguaggio di programmazione tipato. Dato un programma, è possibile calcolare un insieme finito di tipi su cui ogni funzione lavorerà.

Nel codice oggetto prodotto si hanno diverse implementazioni della stessa funzione dipendenti dal tipo. Tipo i template di C++.

Pro:

- non ci sono vincoli sulla rappresentazione dei dati.
- scattano ottimizzazioni ad-hoc sui dati.

Cons:

- limitata a quando i vincoli sono soddisfatti.
- tempi di compilazione e dimensioni dell'eseguibile più grandi.

1.1.2. «Alla C» (Rust in casi residuali)

In C per dichiarare una funzione polimorfa:

- si usa un puntatore per accedere al dato I/O;
- si usa il tipo `void *` per rappresentare il puntato (ovvero si ignora la dimensione del dato).
- la funzione prende in input coppie puntatore e dimensione del dato (`void*`, `size_t`).

Un esempio è il `qsort` di C.

Cons:

- a runtime bisogna preservare e passare la dimensione dei dati.

- necessitano di aiuto da parte del programmatore: in C infatti dobbiamo passare sempre la dimensione.
- inefficiente as a fuck.

1.1.3. Rappresentazione uniforme dei dati (Erlang, OCaml, Haskell, ...)

Se tutti i dati avessero la stessa dimensione si potrebbe fare la swap serenamente. L'idea è quella di rappresentare i dati con una word (dentro ci sta un puntatore, in un byte non potrebbe starci).

Come?

- i tipi di dati che occupano MENO bit di una word sprecano bit (in certi linguaggi: values types o unboxed).
- i tipi di dati di dimensione MAGGIORE di una word:
 - (a) vengono allocati sullo heap.
 - (b) lo rappresento come il puntatore nello heap (in certi linguaggi: reference types o boxed).

Pro:

- tempi di compilazione e dimensione dell'eseguibile ridotti.


Cons:

- introduce indirezioni (o anche dette reference: invece di salvare il valore in memoria, si riferenzia qualcosa usando un nome) e questo ha minore efficienza.

OCaml usa gli utenti boxed, quindi operazioni numeriche pesanti vanno a fare numerose operazioni sull'heap.

Ad esempio, prendiamo un albero in Erlang:

```
1 % Tree K V ::= { leaf, K, V } | { node, Tree K V, K, Tree K V }
2 T = { node, { leaf, 4, true }, 5, { leaf, 6, false } }
```

 Erlang


T finisce nell'heap.

```
1 T -----> node
2
3           5      |
4     -----      |
5     |              |
6     |----> leaf    |
7           6      |
8     false         |
9                   |
10          leaf <-----|
11                4
12               true
```

Anche in C viene rappresentato così, attraverso un puntatore a struct.

Diverso comportamento invece per le liste del C. In Erlang abbiamo:

```
1 % List T ::= nil | { cons, T, List T }
2 L = { cons, 0, { cons, 1, nil } } % [0 | [ 1 | []]]
```

 Erlang

Qui il `cons` non viene rappresentato nelle liste del C.

Nei linguaggi con GC automatico devo essere capace di distinguere le word pensate come puntatori da word pensate come non puntatori.

La soluzione da parte di chi non monomorfizza è quella di usare un bit nella word per fare distinzione.

`xxxxxxxxxxxxxxxx0` (OK) Preso il bit meno significativo per rappresentare i puntatori spreco la metà degli indirizzi ma è meglio di usare il bit più significativo. Qui si alternano i puntatori dai non puntatori. Sono dati boxed.

`xxxxxxxxxxxxxxxx1` (NO) Non va bene, perché si accedrebbe a dati non allineati in memoria. I dati vengono passati nel bus e in genere gli allineamenti vengono fatti per multipli di valori pari.

`0xxxxxxxxxxxxxxxx` (NO) Qui spreco sempre la metà degli indirizzi ma proprio un 50% consecutivo.

Per accedere alla terza word di un dato boxed puntato dal puntatore `p` si accede come `*(p+3)`.

I numeri vengono rappresentati nel payload.

```
1 0000001 = 0
2 0000011 = 1
3 0000101 = 2
4 0000111 = 3
```

Per rimuovere il riporto nell'operazione di somma bisogna fare due operazioni.

```
x + y = ADD x y; SUB 1
```

Però per le moltiplicazioni si ha:

```
x * y = SUB x 1; SUB y 1; MUL x y; ADD 1
```

Cons:

- 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 aritmetiche-logiche per number crunching (calcoli numerici veloci su tanti dati).
- interazione con altri linguaggi complessa.

Tutti i valori diversi dai tipi di dato di Erlang (come atomi, PID, int, floating int) vengono rappresentati in maniera diversa. Quindi non vengono riutilizzate le stesse rappresentazioni di bit.

```
1 1 > 0 == [].
2 false
```

Erlang però dà la rappresentazione delle liste in modo confusionale rispetto alle stringhe.

```
1 1> [97, 98, 99].
2 "abc"
3 2> [97, 98, 99, -1].
4 [97,98,99,-1]
```

```
5 3> [97] := "a".
6 true
```

Diverso dal bellissimo OCaml che è tipato e invece:

```
1 # "a" = "a";;
2 - : bool = true
3 # 4 = "a";;
4 Error: This constant has type string but an expression was expected of type
5         int
```

A runtime, in realtà, `Autunno` e `Cuori` hanno stesso tipo perché vengono generati con la stessa rappresentazione, tipo `0` in questo caso.

```
1 # type semi = Cuori | Fiori;;
2 type semi = Cuori | Fiori
3 # type stagioni = Autunno | Inverno;;
4 type stagioni = Autunno | Inverno
5 # Cuori = Autunno;;
6 Error: This variant expression is expected to have type semi
7         There is no constructor Autunno within type semi
```

Erlang permette di fare confronti cross-type a runtime. Inoltre Erlang usa un tagging per i dati in memoria. Non si può riusare la stessa rappresentazione per tipi diversi.

Haskell e OCaml non permette di fare confronti cross-type perché fortemente tipizzati. Usano pseudo-tagging. Visto che il cross-type non è ammesso, si può riutilizzare la stessa rappresentazione dei bit per più tipi (il tipo è noto staticamente).

Dunque, se il linguaggio:

1. Ha tipi dinamicamente distinguibili a runtime (i valori sono taggati);
2. Ammette confronti fra valori di tipi di dati diversi e si aspetta `false` come risposta (invece di un errore).

Allora deve usare rappresentazioni disgiunte a livello di bit per i diversi tipi, per poterli distinguere.

Come si distinguono i tipi di dato:

- **Boxed**

Il dato è formato da parole consecutive sullo heap. Si aggiunge una prima parola che contiene un tag per distinguere i dati.

La parola con il tag in genere contiene anche la dimensione del dato sullo heap per la GC.

C'è uno spreco, ma in genere nello heap vanno i dati grandi, quindi diviene stupido lo spreco.

- **Unboxed**

Si usano i bit meno significativi per mantenere il tag. Avere i tag di lunghezza fissa è molto costoso per i puntatori. Quindi, si usano i tag a lunghezza variabile. L'idea è di ordinare i tipi di dati e assegnare i tag in base a quelli che servono (booleani tag lungo, puntatori tag corti), questo perché in base al tipo serve un payload più o meno lungo.

xxxxxxxxxxxxxTTT

Esempio di insieme di tag:

- 0 puntatori;
- 01 qualcosa;
- 011 qualcosa 2;
- 111 qualcosa 3.

1.2. Rappresentazione degli atomi

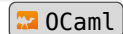
Si differenzia la rappresentazione in base a come appare un atomo nel linguaggio.

- Insieme di atomi noto interamente a compile-time e unicità degli atomi rispetto all'insieme dei tipi (= ogni atomo appartiene a uno e un solo tipo). Un esempio sono gli ADT in OCaml e Haskell. In tal caso, spesso, si riciclano per tipi diversi.
- Linguaggi in cui gli atomi possono comparire senza essere dichiarati.

Si suddividono in:

1. **Al momento del linking** (Erlang, Prolog, OCaml). Ad esempio due librerie A e B che usano il loro insieme degli atomi: in fase di linking si deve fare l'unione dei due insiemi di atomi.

```
1 # `Foo 3;;  
2 - : [> `Foo of int ] = `Foo 3
```



Qui l'interprete fa inferenza per capirne il tipo.

2. **A runtime** (Erlang). Non è un problema perché i messaggi possono semplicemente transitare nella rete. L'insieme degli atomi in Erlang può crescere a ogni messaggio.

In Erlang/Prolog/OCaml atomi con lo stesso nome usati da attori/moduli/librerie vanno identificati.

Problema: durante la compilazione/interpretazione associare a ogni atomo una sequenza di bit in maniera tale che sia possibile il linking/message-passing rispettando l'identificazione degli atomi. Questo avviene perché io compilo indipendentemente i due moduli scegliendo una sequenza di bit per l'atomo in ogni modulo, ma poi quando li metto insieme come faccio a scegliere qual è la sequenza di bit da usare?

Soluzione con **hash per ogni atomo** (usata da OCaml). Vi sono possibili collisioni dato che una funzione hash non è iniettiva. Nel file oggetto si può mettere l'associazione dell'atomo con l'hash; in fase di linking si guardano tutte le associazioni e in caso si genera un hash conflict. Il fatto che in fase di linking ti dia un errore è qualcosa che non scala: bisognerebbe modificare l'atomo della libreria e ricompilarla. Qui però vi sono sequenze di bit sparse (tocca fare if-else infiniti) dunque l'implementazione è inefficiente.

Soluzione con **sequenze consecutive di bit a mano a mano che vengono incontrati gli atomi** (usata da Erlang). Ogni nodo ha la sua tabella. Quando un messaggio contenente un atomo viene instradato nella rete, il programma crea un record dell'atomo con la sua rappresentazione: in questo modo ogni volta che si incontra un atomo si controlla la rappresentazione di esso se presente in tabella.

Cons di quest'ultimo:

- Richiede di mantenere diverse tabelle di traduzione;
- Richiede di tradurre ogni messaggio scambiato fra nodi diversi per rimappare le sequenze di bit degli atomi.

Pro:

- Funziona sempre!

1.3. Pattern matching

Preso un esempio di pattern matching con l'ADT dell'albero visto sopra.

```
1 [ | pattern, p |]
```

Dove `pattern` è il pattern e `p` è il dato (= 1 word). `[|...|]` è una funzione a tempo di compilazione (crea codice).

```
1 [| _, p |] =
2 [| X, p |] = word X = p;           % X è una variabile nuova (**)
3 [| k, p |] = if (p != k) return -1; % k è una costante di un tipo
    atomico, unboxed
4 [| P1 = P2, p |] =
5     [| P1, p |]
6     [| P2, p |]
7 [| {P1, ..., Pn}, p |] =
8     if (is_unboxed(p)) return -1;
9     if (p[0].length != n) return -1;
10    [| P1, p[1] |]
11    ...
12    [| Pn, p[n] |]
```

(**) in OCaml si definisce una nuova variabile ogni volta.

```
1 # let (x, y) = (4, 5);;
2 val x : int = 4
3 val y : int = 5
4 # let (x, y) = (2, 3);;
5 val x : int = 2
6 val y : int = 3
```

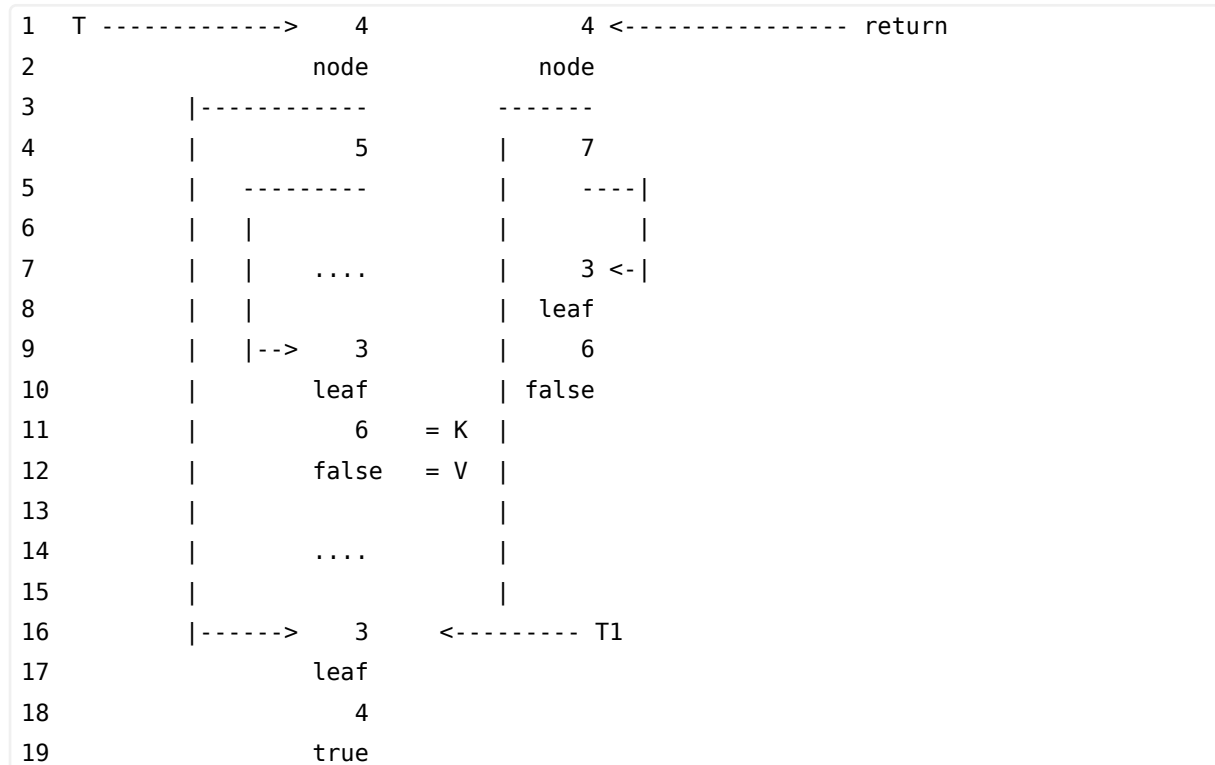
Mentre in Erlang non va bene questo: il secondo let non funzionerebbe perché dipende da un contesto molto ampio. Questo perché nel secondo caso intenderebbe il valore della variabile, ma questo scatta a runtime.

```
1 1> { X, Y } = { 4, 5 }.
2 {4,5}
3 2> { X, Y } = { 2, 3 }.
4 ** exception error: no match of right hand side value {2,3}
```

(Basta guardare il primo bit per vedere se è unboxed).

Se X è l'albero io assegno (non visito l'albero) dunque ho una complessità $O(1)$.

Va fatta una visione diversa:



Ad esempio,

```

1  f({_, T1, 5, {leaf, K, V}}) -> {node, T1, 7, {leaf, K, V}}
2
3  if (is_unboxed(T)) return -1;
4  if (T[0].length != 4) return -1;
5  word T1 = T[2];
6  if (5 != T[3]) return -1;
7  if (is_unboxed(T[4]) return -1;
8  if (T[4][0].length != 3) return -1;
9  if (leaf != T[4][1]) return -1;
10 word K = T[4][2];
11 word V = T[4][3];

```

Erlang

Quando ricopio una sequenza di bit, vuol dire inserire il puntatore `T1`. Esso fa sharing: `T1` non viene modificata ma viene condivisa da due parti; quello scritto esplicitamente viene allocato.

Si perde lo sharing della struttura dati `{leaf, K, V}` perché viene riallocato.

Costo computazione del pattern-matching: $O(n)$ in spazio e in tempo, dove n è la lunghezza del pattern.

Costo computazione dell'allocazione del risultato: $O(n)$ in spazio e in tempo, dove n è la lunghezza dell'espressione usata in output.

Nota: in realtà sono costi $O(1)$ a runtime perché `n` non è un parametro dell'input a runtime.

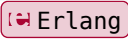
Ogni variabile che ha catturato un tipo di dato boxed ha generato sharing. Questo non è un problema se il linguaggio di programmazione non ammette mutabilità (dello heap). In C, ad

esempio, è difficile condividere una struttura dati a causa della sua mutabilità. Questo permette di sfruttare lo sharing nei linguaggi funzionali, anche se fanno tante assegnazioni.

Nei linguaggi che hanno nativamente pattern-matching profondi (come Erlang/OCaml/Haskell; ma *non* Scala/JS/Python) quando si compilano dichiarazioni di funzioni non si compilano i pattern uno alla volta.

Ad esempio,

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



La semantica del linguaggio dice che viene usato il primo pattern in ordine per fare match. Dato che vi sono due pattern che vi somigliano, un linguaggio come Scala farebbe troppi check ripetuti. Una soluzione è quella d'usare un albero di decisione ma non sarebbe la semantica corretta, perché un match più generico potrebbe essere prima in ordine ma non scelto dall'albero di decisione.

Complessità: sub-lineare sulla somma della lunghezza dei pattern.

1.4. Ricorsione

Idea: a runtime viene mantenuto uno stack di record di attivazione per ogni chiamata di funzione.

Convenzione: lo stack cresce verso il basso.

Un AR (Activation Record) mantiene:

- Variabili locali;
- Parametri della funzione;
- Valore dei registri salvati;
- Indirizzo del valore di ritorno (dove salvare l'output);
- Indirizzo di ritorno.


Costo di una chiamata di funzione: tempo e spazio $O(1)$, basta allora l'AR indipendentemente da quanto è grande.

Se eseguo una funzione `f()` e invoco `g()`, l'AR di `f()` deve contenere tutta e sola l'informazione del server per terminare l'esecuzione di `f()` dopo che `g()` sarà terminata.

Non tutta l'informazione contenuta nell'AR di `f()` è necessaria dopo la chiamata di `g()`.

Ad esempio,

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



L'AR di `f` contiene:

```

1 > w
2   z
3   p
4   y
5 > x
6   REGISTRI
7   RV
8   RA

```

Le variabili `w` e `x` non servono dopo l'invocazione di `g`.

Idea 1. Prima di invocare `g()` decremento lo SP così da rilasciare `w`.

Idea 2. Con un'analisi statica posso determinare che sia più conveniente piazzare `x` subito dopo `w` nell'AR in modo da rilasciare anche `x`. Così posso fare POP dell'AR un po' alla volta. (Prolog implementa sempre questa ottimizzazione).

Una chiamata a `g()` dentro a `f()` è detta **chiamata di coda** (tail call) se e solo se:

1. L'unica istruzione eseguita dopo la `g()` è la return.
2. Il valore ritornato da `f()` è esattamente il valore ritornato da `g()`.
3. Non è contenuta all'interno di un blocco try/catch nella parte dell'espressione da valutare (non è protetta).

Quando ho una chiamata di coda posso fare POP anche di `REGISTRI`, `RV` e `RA`:

```

1 pop    registri, variabili locali, parametri
2 push   parametri attuali della g
3 jump   codice della g

```

La prima parte dell'AR di `g()` coincide con la prima parte dell'AR di `f()`.

Costo dell'ottimizzazione tail call: $O(1)$ in tempo, e praticamente $O(0)$ in spazio perché riciclo l'AR.

Una funzione `f()` è detta **tail ricorsiva** se e solo se tutte le sue chiamate ricorsive sono di coda.

L'ottimizzazione scatta solo se la funzione è tail ricorsiva.


Un compilatore per linguaggi funzionali deve implementare questa ottimizzazione perché sennò diviene difficile gestire lo spazio (esempio di WASM e JS). L'errore dei novizi nel paradigma funzionale è quello di scrivere codice non efficiente in spazio!

Anche «`=`» (che sembra assegnamento) è un pattern matching. In genere, dato che non c'è nessuna garanzia, bisogna fare il match di valori che sappiamo siano ok.

```

1 1> {A, B} = {7, 5}.
2 {7,5}
3 2> X = 2+3.
4 5

```


 Erlang

1.5. Eccezioni

Guardando il pattern-matching per la ricerca dentro l'albero possiamo vedere come sia inutile fare `case of` se già facciamo `pattern-matching`; Erlang usa uno dei due come zucchero sintattico dell'altro.

Ricordando che il BST ottimizza la ricerca per chiave, non per valore, possiamo definire:

```
1 search_val(V1, {leaf, K, V2}) when V1 == V2 -> {found, K};
2 search_val(_, {leaf, _, _}) -> not_found;
3 search_val(V, {node, T1, _, T2}) ->
4   case search_val(V, T1) of
5     { found, _ } = Res -> Res ;
6     not_found -> search_val(V, T2)
7   end.
```

 Erlang


È efficiente a livello computazionale (non si può fare meglio di così) ma non è efficiente a livello di simboli.

La `search_val` fa chiamate ricorsive sull'albero su tutti i nodi fino alle foglie. Quando arriva alle foglie, sia se trova il valore sia se non lo trova, fa un'analisi per casi e decide se andare a destra o sinistra. Il problema è che la risalita «del valore fino alla radice» costa una costante moltiplicativa dispendiosa: si dovrebbe troncare l'esecuzione subito una volta trovato il valore.

Ci serve un **operatore di controllo** (costrutto che può alterare l'ordine futuro del codice andando ad alterare lo stack). Dobbiamo restituire il controllo ad un altro punto spaziale (inizio della ricerca) e temporale (lo stack rappresenta un'istruzione futura quindi c'è uno stack entry che è un puntatore al return code; dobbiamo buttare tutto il lavoro che avevamo predisposto di fare).

La parte di stack che si è fatta prima del catch rimane lì ma fa abort quando legge il throw che sale in cima allo stack e dunque tiene traccia di quanto fatto fino a quell'istante.


```
1 1> 2 * try 3 + (5 + throw ({exc1, 5})) catch {exc1, V} -> V end.
2 10
```

 Erlang

Qui infatti vediamo come l'output sia 10 perché il risultato della catch viene moltiplicato per 2.


La riscrittura della search viene fatta ricorsivamente come:

```
1 search_val_aux(V, {node, T1, _, T2}) ->
2   search_val_aux(V, T1),
3   search_val_aux(V, T2).
```

 Erlang

Questo funziona perché nel caso di found viene definita come:

```
1 search_val_aux(V1, {leaf, K, V2}) when V1 == V2 -> throw({found, K}).
2
3 search_val(K, T) ->
4   try
5     search_val_aux(K, T)
6   catch
7     {found, _} = R -> R;
```

 Erlang

```

8     not_found -> ....
9     end.

```

e dunque se trova la chiave l'eccezione blocca l'esecuzione.

La sintassi è

```

1 try
2     E          % codice protetto
3 catch
4     ...        % codice non protetto

```

L'eccezione, pertanto, non viene usata solo per gli errori. Molte eccezioni è meglio gestirle localmente, come l'errore di scrittura sul disco di un file: non conviene fare un'eccezione che risale nello stack e quindi bisognerebbe reiniziare l'esecuzione di nuovo; meglio tenerla lì e gestirla.

Nel caso di `search_val` l'eccezione viene usata solo localmente.

Il costrutto `try` aggiunge un nuovo stack frame che però non è un AR bensì un record try-catch.

```

1  -----
2  Stack frame n+3      AR di chiamate di funzione durante l'esecuzione di E
3  -----
4  Stack frame n+2      AR di chiamate di funzione durante l'esecuzione di E
5  -----
6  Stack frame n+1
7  -----
8  Stack frame n        f(n) = ... try E catch ...
9  -----
10 Stack frame n-1
11 -----
12 Stack frame n-2      f2() = .... f(3) ...
13 -----
14 Stack frame          f1() = .... f(2) ...

```

Il record che sta in « Stack frame n+1 » è un record try-catch con:

- Puntatore al codice catch.
- Tag «record try-catch».

Al momento della `throw(E)` :

```

1 throw(E) {
2     finished = 0;
3     while (!finished) {
4         while (Stack[0].Tag != "record try-catch") Stack.pop();
5
6         addr = Stack[0].indirizzo_codice_catch
7         Stack.pop()
8
9         match CALL addr(E) with

```

```

10      | { caught, V } -> ; finished = true; V
11      | not_catched   -> nothing
12  }
13 }

```


A basso livello il codice dopo il catch analizza le varie eccezioni.

Vogliamo adesso modificare il codice `cc(Bal)` in modo da gestire le eccezioni per il caso di divisione per 0. Le chiamate ricorsive non si vogliono proteggere, quindi vanno messe fuori dal try/catch. Lui deve terminare e poi dire che vuole fare la chiamata ricorsiva.

```

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

```

 Erlang

È stata introdotto un po' di zucchero sintattico per modificare questo approccio. Il codice protetto è quello in cui non scatta la tail recursion.

Il nuovo costrutto try/of/catch è un mix di quel try/catch e case/of:

```

1  try
2      E                      % codice protetto
3  of
4      p1 -> c1 ;             % codice non protetto
5      ...
6      pn -> cn
7  catch
8      ...                    % codice non protetto
9  end.


```

Il codice modificato è

```

1  cc(Bal) ->

```

 Erlang

```

2    try
3        receive
4            print ->
5                io:format("Balance is ~p ~n", [Bal]),
6                { recur, Bal } ;
7            {put, N} -> { recur, Bal+N } ;
8            {tax, N} -> { recur, Bal/N } ;
9            {get, PID} -> PID ! Bal, { recur, Bal } ;
10           exit -> { result, ok }
11        end
12    of
13        { result, R } -> R;
14        { recur, Bal } -> cc(Bal) ; % tail recursion
15    catch
16        error:_ -> { recur, Bal }
17    end.

```

Erlang è puro: non c'è mutabilità. Quindi non abbiamo side-effect. In realtà, Erlang ha accesso alla rete e file system e quindi non lo rende totalmente puro.

Il costrutto si estende ulteriormente come try/of/catch/after. Una volta usato il codice di `after` non ho più l'uso di coda: ha senso perché non devo farci nulla dopo. Esso viene usato non per ritornare un valore, bensì per eseguire codice con side-effect. È comunque un qualcosa che viene eseguito sempre, anche se non becco un'eccezione.


```
1 try E of F catch G after C end
```

è zucchero sintattico di

```

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

```


 Erlang

L'uso di `throw()` - che è un meccanismo di controllo - è diverso dall'uso di `exit()` che viene usato per la terminazione dell'attore. Idealmente si catturano solo le throw. Il catch diretto di `exit` o `error` avviene come:

```

1 1> try exit({hello, 2}) catch exit:E -> ok end.
2 ok

```

 Erlang

1.6. Programmazione parallela e concorrente

Facciamo un esempio guardando una versione parallela dell'algoritmo Quicksort.

Utilizzare più di 1 core per mandare in parallelo porta uno speedup ma lo speedup non è lineare. Possiamo avere centinaia di attori ma devono reagire a stimoli, cosa che nel codice qui sotto è il contrario.

```
1  -module(qsort).
2  -export([qsort/1, psort/1]).
3
4  % Versione sequenziale
5  qsort([]) -> [];
6  qsort([H|L]) ->
7      L1 = [ X || X <- L, X <= H ],
8      L2 = [ X || X <- L, X > H ],
9      qsort(L1) ++ [ H | qsort(L2) ].
10
11
12 % Versione parallela
13 psort([]) -> [];
14 psort([H|L]) ->
15     L1 = [ X || X <- L, X <= H ],
16     L2 = [ X || X <- L, X > H ],
17     SELF = self(),
18     % una nonce per dire esplicitamente cosa si vuole ricevere
19     REF = make_ref(),
20     % SELF, REF, L2 sono copiate al figlio
21     spawn(fun () -> SELF ! {REF, psort(L2)} end),
22     psort(L1) ++ [ H | receive {REF, SL2} -> SL2 end ].
```

Se avessimo

```
1  % ...
2  SELF = self(),
3  spawn(fun () -> SELF ! psort(L2) end),
4  psort(L1) ++ [ H | receive SL2 -> SL2 end ].
```

Non ci sarebbe sync fra i figli e dunque potremmo avere dei risultati che non ci aspettiamo.

Una variante (comunque sbagliata) dello spawn è quella di chiamarla come qui sotto, ma non ci sarebbe modo di prendere il valore del risultato.

```
1  PID = spawn(?MODULE, psort, [L2])
```


In C, una fork permette di condividere tutto ai processi, ad eccezione del valore di ritorno della fork: i due processi sono identici. In Erlang, il nuovo attore è completamente disgiunto dal padre e non condivide nulla a meno dei dati che vengono utilizzati nella funzione che vado ad eseguire (solo `L2` in questo caso); in realtà fa una copia di `L2`, e questo è utile quando si vuole far fallire i processi velocemente nel gestire dei guasti; se condividesse la memoria sarebbe più complesso perché bisognerebbe vedere quali parti della memoria reclamare.

Parallelizzare l'intera list comprehension vuol dire copiare l'intera lista, quindi in quantitativo di spazio non è buono.

In un sistema distribuito l'ordine dell'arrivo dei messaggi non è significativo. Bisogna scrivere un pattern tale che l'ordine dei messaggi nella mailbox non influisce la semantica del programma.

Il modo in cui si esplicita la chiamata al GC in Erlang è

```
1 erlang::garbage_collect().
```

 Erlang


Per liste di valori molto grandi il risultato di `psort` è molto più lento rispetto a `qsort`. L'overhead in questo caso per la parallelizzazione dipende da due motivi:

1. Per le liste piccole non ha senso parallelizzare;
2. La creazione di un numero di attori che è molto superiore al numero dei core del PC: ogni attore ha un overhead dato dal costo dell'alternanza dei core degli attori.

Una possibile soluzione è quella di avere un elemento extra `N` per monitorare quando fare il calcolo direttamente OPPURE fare spawn di un attore. Questo non funziona perché la parallelizzazione non avviene se blocco la `receive` dopo uno `spawn` (la `receive` deve rimanere asincrona!).

Un'altra possibile soluzione che usa sempre un sistema a `case of` per il valore `N` si ha con l'invio del messaggio a se stessi.

```
1 psort2(0, L) -> qsort(L);
2 psort2(N, [H|L]) ->
3   L1 = [X || X <- L, X <= H],
4   L2 = [X || X <- L, X > H],
5   SELF = self(),
6   REF = make_ref(),
7   spawn(fun () -> SELF ! {REF, psort2(N-1, L2)} end),
8   psort2(N-1, L1) ++ [H | receive {REF, RES} -> RES end ].
```

 Erlang


Invece di creare un attore si può creare un job, che è una struttura dati. Sottomettere un job al pool dei job vuol dire invocare una funzione che aggiunge tale job ad una struttura dati ausiliaria pool dei job. Ci saranno attori, capaci di manipolare il tool dei job, che saranno i responsabili dell'esecuzione dei job.

Affinché chi chiama la job-sort abbia il PID dello scheduler vuol dire:

1. Chi invoca la job-sort è un discendente dello scheduler;
2. Avere un discendente comune (in genere la shell che lancia lo scheduler e ottiene il PID: funziona in un sistema concorrente ma non in un sistema distribuito).

Una soluzione è quella di registrare un attore attraverso un nome logico e non come attore. L'identificazione che mappa il nome logico ad un PID avviene mediante chiamata a `register`.

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


 Erlang

C'è una tabella all'interno della BEAM che monitora coppie di atomi/PID. Infatti, con la `unregister` si modifica proprio la tabella. In caso di fallimenti è conveniente usare un atomo per rappresentare il processo: non è importante che il PID sia diverso una volta che si riavvia, perché tanto ci si appropcherà a quel processo mediante l'atomo.

Se nel costrutto «bang» uso un atomo (come `scheduler ! {...}`) esso automaticamente andrà a vedere se nella macchina da cui eseguo il comando esiste la corrispondenza atomo/PID all'interno della tabella.

Lo scheduler inserisce mediante ricorsione di coda tutti i job da fare, ma bisogna controllare se ha senso estrarre un messaggio quando la lista dei jobs non è vuota:


```
1 scheduler(JOBS) ->
2   receive
3     { required, _, _, _ } = JOB -> scheduler([JOB | JOBS]);
4     { get, REF, WORKER } when JOBS /= [] ->
5       [JOB|L] = JOBS,
6       WORKER ! {REF, JOB},
7       scheduler(L)
8   end.
```

 Erlang

Le richieste `get` rimangono in coda finché la lista di jobs non si popola.

Invece i worker sono gli attori che si occupano di processare il job:


```
1 worker() ->
2   REF = make_ref(),
3   SELF = self(),
4   scheduler ! {get, REF, SELF}
5   receive
6     % PID = pid di chi ha creato il job e vuole la risposta
7     % REF2 = reference di chi ha chiamato la risposta
8     % F = funzione job
9     {REF, {require, PID, REF2}} ->
10      PID ! {REF2, F()},
11      worker()
12   end.
```

 Erlang

Lo scheduler deve rispondere col job, ecco il motivo del PID `self()`. In ricezione riceve il job da processare dallo scheduler. Finché si è in ambito sequenziale nella propria macchina si può anche non avere la reference. Si fa una chiamata ricorsiva di coda sul worker perché così resta in attesa di ricezione di un altro job da processare.

Il programma che usa lo scheduler può richiedere «qualcosa» usando qualcosa del tipo:

```
1 jsort(...) ->
2   ...
3   Job = fun () -> ... end,
4   SELF = self(),
5   REF = make_ref(),
6   scheduler ! {require, SELF, REF, Job},
7   ...
```

 Erlang

Se il `Job` di `jsort` chiamasse ricorsivamente `jsort` ci sarebbe un deadlock: tutti i worker sono impegnati ma aspettano risposte dei job che hanno inviato loro stessi.

Il client manda la richiesta allo scheduler. Lo scheduler manda i job ai worker. I worker mandano la risposta ai client.

Non c'è un sistema di chiamate in cui ho subito una risposta: questa è la particolarità della programmazione asincrona.

Questo codice però va ad eseguire parallelizzazione anche per liste molto piccole (con annesso aumento di overhead).

Si lancia da shell, e sarà quest'ultimo a monitorare i vari processi che metterà up. Si passa da questo processo intermedio perché se li mettesse up tutti la shell, non ci sarebbe nessuno a monitorarli.

1.7. Hot swap

Quando si aggiorna il codice runtime bisogna:

1. Cambiare strutture dati dell'attore;
2. Cambiare lo stack di chiamate.

La funzione deve essere tail ricorsiva perché lo stack deve essere vuoto per aggiornare il codice runtime.

Non posso prendere un codice e modificarlo in memoria direttamente: è compito del programmatore attivare il nuovo codice.

La prima volta la BEAM carica il modulo e lo esegue; quando lo carica di nuovo la BEAM tiene entrambe in memoria e invoca quello vecchio finché non viene esplicitato lo swap con quello nuovo; quando il codice vecchio non viene più usato allora la BEAM lo mette offline.

La BEAM tiene al massimo 2 copie del codice, se si caricasse un terzo ci sarebbe un errore runtime.

Ci sono due modi per invocare una funzione:


1. Usando il nome della funzione;
2. Usando il nome della funzione con prefisso il modulo.

Quando chiamo `nome_modulo:funzione` invoco l'ultima versione del codice di quella `funzione`.

Il passaggio al nuovo codice (che è caricato in parallelo nella BEAM) viene spesso triggerato mediante un evento (un messaggio nel nostro caso).

Prendiamo ad esempio il seguente modulo:


```
1  -export([loop/2, new_loop/2]).
2
3  loop(N, M) ->
4      receive
5          {add1, X} -> loop(N+X, M) ;
6          {add2, X} -> loop(N, M+X) ;
7          upgrade -> ?MODULE:new_loop(N, M)
8      end.
9
10 new_loop(N, M) -> loop(N, M).
```

 Erlang

Serve dunque una funzione ausiliaria `new_loop`. Si sarebbe potuto chiamare direttamente un `?MODULE:loop(f(N, M))` ma `f` avrebbe ritornato un solo valore.


In realtà per la transizione si usa una versione intermedia tra quella vecchia e quella nuova: la versione transitoria usata dovrà gestire sia i messaggi vecchi che quelli nuovi.

```
1  -export([loop/1, new_loop/1, new_loop/2]).
2
3  loop({N, M}) ->
4      receive
5          {add1, X} -> loop({N+X, M}) ;
6          {add2, X} -> loop({N, M+X}) ;
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}).
13 new_loop({N, M}) -> loop({N, M}).
```

 Erlang

Dopo di questa si potrà caricare l'ultima versione nuova finale.

```
1  -export([loop/1, new_loop/1]).
2
3  loop({N, M}) ->
4      receive
5          {add, 1, X} -> loop({N+X, M}) ;
6          {add, 2, X} -> loop({N, M+X}) ;
7          upgrade -> ?MODULE:new_loop({N, M})
8      end.
9
10 new_loop({N, M}) -> loop({N, M}).
```

 Erlang

1.8. Gestione fallimenti

Si usano due approcci per gestire i fallimenti in un ambito di attori, anche se è gestito totalmente dal programmatore. Gli attori possono accorgersi con quali attori si sta collaborando per una più completa gestione dei fallimenti. In genere si fa un sistema gerarchico in cui è il padre a monitorare i figli e se uno fallisce esso uccide tutti i fratelli e poi è il padre a re-istanziare tutti gli attori figli.

In genere, non è una buona idea terminare un thread in esecuzione perché ha risorse in uso, lock aperti, condivisione di memoria. In Erlang, invece, si uccide l'attore e può causare ricorsione di uccisioni.

1.8.1. Link

La collaborazione fra due attori è una relazione simmetrica (la primitiva di link tratta nel medesimo modo il link $A \rightarrow B$ e $B \rightarrow A$) e transitiva. La funzione link è:

- Bidirezionale: se A osserva B e muore, morirà anche l'altra.
- Idempotente: un link fra due attori è fatto 1 sola volta. A questo si accompagna una primitiva di unlink.
- Transitiva (up to trap link).


La creazione di un link subito dopo uno spawn potrebbe causare una race condition. Qui si risolve con una primitiva composta che fa spawn e link contemporaneamente.

Un attore può usare una receive per fare catch di cambio di modalità e non essere ucciso. Questa funzione è `process_flag(trap_exit, true)`: quando arriva un messaggio di exit nascosto esso diviene un messaggio normale per essere gestito da una receive.

L'uccisione mediante `killed` è qualcosa che non può essere bloccato (come il `-9` di UNIX). In UNIX, però, non c'è la propagazione automatica del segnale come in Erlang. Chiamato `exit(PID, kill)` esso propaga il segnale come `exit(PID, killed)` (reason che può essere bloccata) e poi tutti gli altri linkati saranno `exit(PID, normal)`.

Esempio di uso: dati due attori - il corrente e PID - il corrente vuole richiedere un servizio a PID dal quale vuole una risposta.

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

 Erlang

Se il PID crasha per qualche minuto il processo rimane in attesa per sempre, dunque deve essere ucciso a ricorsione (grazie al link si risolve). Con un timeout mi assicuro di ignorare il problema, in caso di latenza di rete o altro.

Questo sistema di link chiaramente non ha sempre senso, come ad esempio il caso di un server web; non ha nemmeno senso che ci sia transitività perché se un client crasha non devono crashare tutti i client collegati al server.

Non ritorna nulla.

`unlink` prende il PID.

`trap_link` funziona solo con link.

1.8.2. Comando che solleva eccezione

Se non gestito mediante try-catch esso sale a top level dell'attore e morirà in maniera anormale che causerà un fail a tutti gli altri (tutti quanti di tipo normal).

1.8.3. Monitor

Un **monitor** è:

- Unidirezionale: un attore monitora un altro attore e viene informato se quell'attore termina.
- Non idempotente (se legate `n` volte poi servirà `n` volte l'operazione inversa per slegarle).
- Non transitiva: non c'è la «propagazione» del messaggio fra tutti gli attori linkati.

Ritorna un handle: un identificatore univoco di una connessione.

`demonitor` prende l'handle e PID.

I messaggi legati al monitoring sono sempre in user space e non vengono propagati.

Si può fare monitoring di un intero nodo mediante un sistema di pooling.

Esempio 1: un browser (non linka un server web).

Esempio 2: uso due librerie B e C, ed entrambi usano una libreria A. Se tutte le librerie sono implementate come attori si ha B link A e C link A ma col classico link avremmo che sia B e C sono linkati allo stesso A: se uno dei due fallisse allora fallirebbe anche l'altro.

1.9. Distribuzione

Un nodo è una VM in esecuzione, e dunque può girare indipendente nella stessa macchina. Si usano comunque i PID che sono nomi logici, si usa l'indirizzo fisico solo quando si vuole collegare un nodo ad un altro alla stessa rete.

Posso creare una connessione da un nodo A ad un nodo B usando una qualunque funzione di libreria che usa comunicazione fra nodi. Questo collegamento è transitivo.

La comunicazione fra due cluster viene fatta solo se si condivide una chiave. Può essere data implicitamente se si creano due BEAM sulla stessa macchina perché entrambe si sincronizzano sulla chiave presente in `~/.erlang.cookie`

```
1 (alice@pc)1> nodes().
2 []
3 (alice@pc)2> net_adm:ping(bob@pc).
4 pong
5 (alice@pc)3> nodes().
6 [bob@pc]
```

Il ping è transitivo.

Il comando `flush()` è un loop su una receive del tipo `receive MSG -> MSG end`.

Il nodo B può usare una `register` su se stesso così da chiamare i messaggi con la semplice bang:

```
1 (bob@pc)1> register(shell, self()).

1 (alice@pc)4> {shell, bob@pc} ! {hello, self()}.
```


I messaggi fra i due nodi passano in chiaro.

Non c'è nessun modo per migrare un attore da un nodo all'altro ma può aver senso in alcuni casi:

- Distribuzione di carico;
- Dati usati da un attore più vicino geolocalmente.

C'è un modo per creare una specie di migrazione ma l'attore deve fare forwarding del nuovo PID a tutti gli altri nodi.

```
1 migrate(NODE, F) ->
2   PID = spawn(NODE, F),
3   Forward =
4     fun Forward() ->
5       receive
6         {comeback, F} -> F() ;
```

 Erlang

```
7      Msg -> PID ! Msg, Forward()  
8      end  
9      end,  
10     Forward().  
11
```

Il sistema dopo un po' si stabilizza, quindi non è così chissà quanto inefficiente. La nuova incarnazione, dopo la scoperta del nuovo PID, non passerà più dal nodo intermedio e quindi la comunicazione avverrà direttamente col nuovo attore; solo chi aveva il vecchio PID passerà dal forward.

Usando la chiamata `comeback` l'attore originale riprende il codice.

2. Effetti algebrici

Sono un insieme di operazioni atte a gestire effetti computazionali in maniera più efficiente rispetto alle monadi. Gli effetti collaterali (o side effect) si presentano in maniera più modulare. Dunque si dichiara un effetto e un handle per interpretarlo. Alla fine abbiamo il vantaggio di poter combinare e comporre handler.

Si trovano in linguaggi come OCaml ed Eff.

Usato, ad esempio, per centralizzare gli errori o implementare uno scheduler.

Anche i `setjmp/longjmp` di C possono essere usati per fare ciò.

Ad esempio,

`throw(E)`

- Passa il controllo remotamente alla 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,
4     N -> resume N+1
5   end
```

Il risultato dovrebbe essere `2 * (3 + 5 + 8)` . Chiaro che se non uso la nuova keyword `resume` , il catch lavora in modo classico.

- Nella exception la throw fa un ciclo while sullo stack e per ogni AR/try-catch record che non cattura l'eccezione fa `Stack.pop()` ;
- Negli algebraic effect (naive) la throw non fa `Stack.pop()` ma

```
1 AR = Stack.pop();
2 Detached.push(AR);
```

dove `Detached` è un secondo stack (ordinato in senso inverso) dove memorizzo temporaneamente gli stack frame staccati.

- La `resume` fa:

```
1 while (!Detached.is_empty()) {
2   AR = Detached.pop();
3   Stack.push(AR);
4 }
```

Scrivo il valore di `resume` come risultato dell'istruzione di `throw` .

- Se nel ramo catch non faccio `resume` mi ritrovo a fare:

```
1 while (!Detached.is_empty()) {
2   Detached.pop();
3 }
```

Questa cosa simile la fa l'OS con lo scheduler dei processi, dove usa uno stack di informazioni per ogni processo da dover eseguire.

In un linguaggio con effetti algebrici si aggiunge un tipo di dato astratto **fibra** (fiber) che rappresenta un frammento di stack `Detached`, il quale è un tipo di dato di prima classe (si può muovere, copiare, spostare, ...) e sul quale posso fare l'operazione `resume` per reinstallarlo in cima allo stack. Il ramo catch cattura la fibra. La fibra è veloce nell'esecuzione e ha uno scheduling cooperativo. Non gira in parallelo by default però può essere pensata come una coroutine o un thread user-space.

In Go ogni goroutine è una fibra.

```
1  yield() -> throw(yield).      (* yield e fork sono effetti *)
2  fork(G) -> throw(fork).
3
4  code_to_fiber
5
6  (* Main è il primo thread da eseguire *)
7  (* Queue è la coda dei thread sospesi *)
8  scheduler(Main, Queue) ->
9    try
10      resume (Main, ok)
11    catch
12      (**
13        K è una fibra, i pattern hanno sempre la forma
14        `pattern_eccezione + pattern K`
15      *)
16      yield, K ->
17        case Queue of
18          [] -> scheduler(K, [])
19          [F|L] -> scheduler(F, append(L, [K]))
20        end;
21      fork(G), K ->
22        scheduler(K, append(Queue, [code_to_fiber(G)]))
23    end.
24
25 scheduler(Main) ->
26   scheduler(code_to_fiber(Main), [])
```

Una implementazione efficiente:

- Stack diventa stack di fibre;
- Le fibre sono stack.

Il pro è che staccare/riattaccare fibre diventa $O(1)$, il contro è che lo stack non è consecutivo in memoria.

3. Gestione della memoria

In ambito sequenziale abbiamo dei possibili problemi:

- Il garbage è memoria non deallocata non più utile: nel C non vi è un garbage collector. Dire se è utile è un problema indecidibile: non è possibile decidere se il programma userà o meno quella locazione della memoria. Assolutamente incomprensibile anche da parte del programmatore se dovesse usare librerie esterne a causa delle invarianti esterne;
- Puntatori a memoria non allocata (accedere ad un puntatore ad una cella di memoria che non appartiene a tale programma);
- Riutilizzo di memoria deallocata ma in tal caso o vi accedo o lo rialloco;
- Accesso a memoria non allocata;
- Frammentazione della memoria, cosa che però in Erlang non c'è perché usa algoritmi che non frammentano la memoria.

In ambito concorrente abbiamo altri possibili problemi:

- Race condition: più attori tentano l'accesso alla stessa locazione di memoria.

Prendiamo ad esempio la seguente signature di funzione e vediamo dei diversi scenari:

```
1 T f(..., T x);
```

Scenario 1. Il dato `T` in output è interamente allocato a ogni chiamata.

Non ho puntatori da input a output e viceversa e non ho puntatori al dato in variabili globali (top level) / statiche (variabile globale ma con visibilità solo dentro la funzione).

L'unico modo per capire se il dato è ancora utile è compito del chiamante.

Scenario 2. L'input è condiviso con l'output.

Ci sono puntatori dall'output all'input.

L'output diviene garbage quando non è più usato ma l'input non diviene garbage finché l'output non diviene garbage.

Scenario 3. L'output è condiviso con l'input.

Ci sono puntatori dall'input all'output. [Esempi sono una funzione che dato un array ritorna una slice e una funzione che, dato un albero, restituisce un puntatore ad un sottoalbero].

Non posso rilasciare l'output quando voglio perché dipende dall'input. L'output può diventare garbage quando l'input sarà garbage.

Scenario 4. Scenario 2 + Scenario 3. I due diventano garbage contemporaneamente e devono deallocarsi nello stesso istante.

Scenario 5. La funzione mantiene un puntatore all'output.

[Un esempio sono il pattern Singleton oppure la memoizzazione].

L'output non è garbage finché non lo è per tutti gli output precedenti e quell'input non verrà mai più richiesto.

3.1. Tecniche di gestione della memoria

1. Non c'è (tipo C).

2. Si usa una gestione di garbage collection automatica. Il linguaggio fa un'approssimazione e rimuove la memoria quando viene contrassegnata come garbage. Ad ogni modo vi è un'euristica di garbage detection. Ma questo si somma all'intervento del programmatore anche grazie ai weak pointer: va a dire quali puntatori, nell'euristiche di garbage detection, hanno o meno senso.
3. Il programmatore gestisce quello che sostiene essere invarianti e dunque il compilatore aggiunge la parte di gestione della memoria senza usare euristiche automatiche.

3.1.1. Reference Counting

L'euristica qui è quella di controllare quanti puntatori ci sono a un dato boxed.

0 puntatori entranti a dato boxed \Rightarrow Il dato è garbage

Il dato è garbage \nRightarrow 0 puntatori entranti a dato boxed

In ogni cella è presente la tripla (tag, numero di celle, reference counter). L'ultimo è l'intero che conta i puntatori entranti ed è > 0 . Qualora divenisse 0 deallocherebbe la memoria.

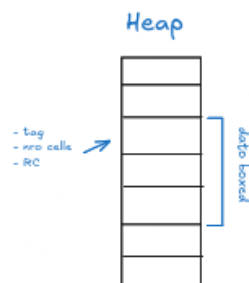


Figura 1: Heap per RC

Una **radice** è una cella di memoria sempre accessibile al programma. Lo stack e i registri lo sono: si può sempre accedere ad essi. L'heap è accessibile solo se ho un puntatore ad esso. Ci sono vari scenari in cui si aumenta il numero dei registri.

Nell'immagine qui sotto quando il registro cancella la reference a quella cella, allora ricorsivamente cancella tutte quelle a seguito perché segnate come garbage (poiché hanno tutte RC = 0).

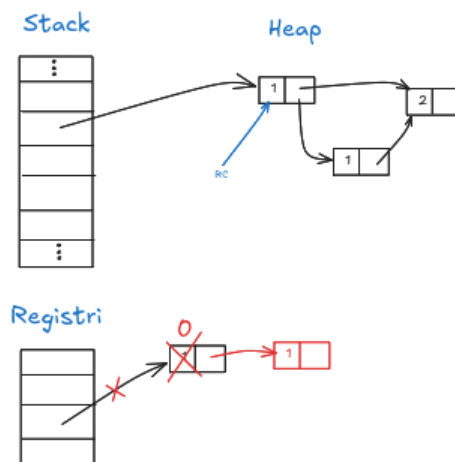


Figura 2: Esempio di RC

Però vi sono situazioni in cui vi è una specie di ricorsione in cui il RC non viene azzerato e dunque i dati nell'heap rimangono anche se dovrebbero essere considerati garbage perché non più accessibili.

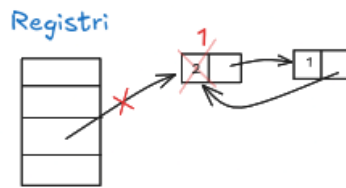


Figura 3: Esempio in cui il RC non si azzerava correttamente

- **Allocazione**

La memoria è frammentata \Rightarrow serve algoritmo di gestione del tool di aree di memoria libera (Best fit, First fit, Worst fit, ...). È costosa in fatto di spazio e tempo.

Si alloca un puntatore per una certa dimensione + 1, quest'ultimo perché viene esplicitata la cella per la RC. Allocata una nuova cella, essa avrà $RC = 1$.

```
1 p = alloc(size+1);
2 (*p)[0] = 1;
```

- **Copia di un puntatore**

Ogni volta che abbiamo un assegnamento il compilatore deve sapere se la variabile è un puntatore: se non lo è allora copia i bit, sennò deve fare la copia e deve anche incrementare di 1 il RC.

```
1 if q != nil {
2   (*q)[0]--;
3   if (*q)[0] == 0 {
4     dealloc(q);
5   }
6 }
7 q = p;
8 (*p)[0]++;
```

- **Deallocazione**

La deallocazione di un puntatore **p** è ricorsiva perché va a decrementare il RC di quello a cui punta e così via.

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

```

10
11   free(p);
12 }

```

In un punto qualsiasi del codice posso ritrovarmi l'esecuzione di una funzione lineare grande quanto tutto il programma. Non vi è un bound significativo.

Il costo è $O(n)$ con n numero di passi del programma. In altri modi è detto unbounded in Θ .

Usando una skip list extra possiamo non deallocare tutta la lista nello stesso momento. Ad esempio, (1) abbiamo una lista sopra; (2) cancelliamo il primo elemento della lista iniziale e il primo elemento della lista rossa, questo cancellerà tutti gli elementi della lista fino al primo elemento che ha un altro puntatore; (3) la lista rossa non ha lo stesso riserva perché abbiamo un ulteriore puntatore che si sposta al secondo elemento della lista rossa.

In questo esempio la lista rossa punta ai $(50 \times i)$ -esimi elementi.

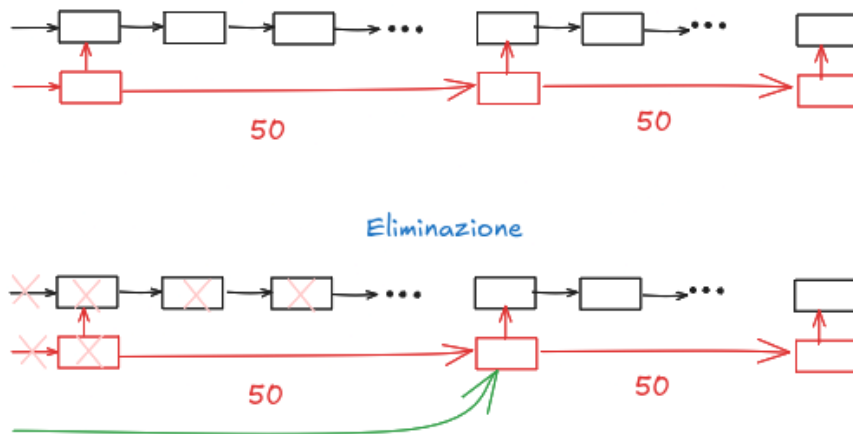


Figura 4: Esempio con skip list

Non è una buona idea usare RC in caso di programmi real time perché la mitigazione diviene complessa.

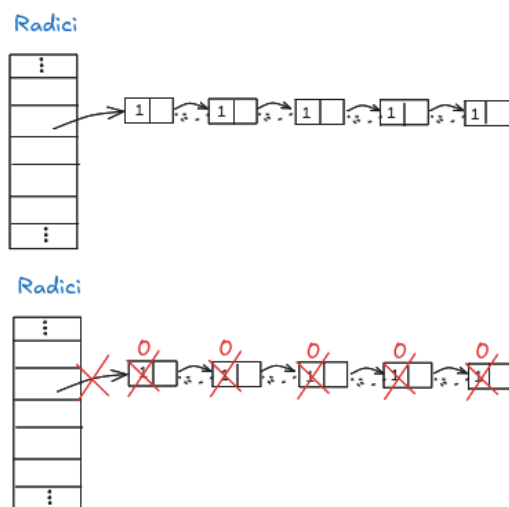


Figura 5: Lista bi-linkata con puntatori strong e weak

Il puntatore in avanti è usato per la lista, ma quello all'indietro è di convenienza. Dunque vi è una distinzione di puntatori, strong (in avanti) vs weak (indietro). Non ha senso avere il

weak pointer quando lo strong pointer viene eliminato. Dal punto di vista di GC basata su RC abbiamo che RC è uguale al numero di puntatori strong entranti.

Mettiamoci nel caso in cui p sia un puntatore strong e q un puntatore weak che però punta alla stessa cella di p . Una volta che p viene deallocato, q punterà a qualcosa di deallocato. Ci serve un meccanismo tale che il puntatore weak venga automaticamente deallocato!

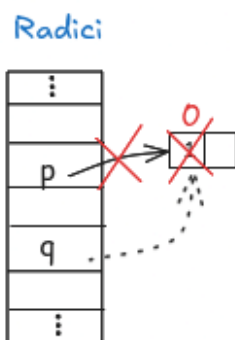


Figura 6: Esempio di due puntatori diversi che puntano alla stessa cella

Il RC comporta ad un rallentamento del codice globalmente facendo delle operazioni di allocazione costose. Anche un semplice assegnamento diviene più complesso perché deve gestire la tipologia e in caso fare deallocazioni (e decrementi).

3.1.2. Mark & sweep

L'euristica usata è:

Un dato non è raggiungibile dalle radici \Rightarrow il dato è garbage

Il dato è garbage \Rightarrow Un dato non è raggiungibile dalle radici

Fase 1. Si marca con dei bit tutto ciò che è raggiungibile.

Fase 2. Si fa deframmentazione. Si prendono tutte le celle marcate come «ancora vive» e le si deframmenta da un'altra parte.

Tutto ciò che non spostato lo considero deallocato.

Il runtime del linguaggio chiede tutta la memoria all'OS e la divide in 2 parti segnando la prima come «memoria da usare».

1. Fa tutte le allocazioni nella memoria come «memoria da usare».
2. In fase di deframmentazione sposta i dati nell'altra.
3. Considera come «memoria da usare» quella in cui ha spostato i dati.
4. Reitera dal punto 2.

Implementare l'algoritmo in due passate è folle, perché si reitera su tutta la memoria anche quando non vi è niente da spostare.

I linguaggi di programmazione che tendono ad allocare molti dati (linguaggi funzionali) usano questa tecnica. Si ha anche un vantaggio di cache per la località dei dati.

Non è necessario allocare due heap enormi: in caso di necessità posso ingrandire l'heap nella fase di passaggio al secondo heap. Ho comunque dello spreco.

Nell'heap vuoto tengo un **Heap Pointer** (HP). I dati che non vengono allocati nell'heap vuoto sono quelli che non hanno nessun riferimento da nessuna radice. Quando l'heap di lavoro riduce di molto i dati, comunque esso non viene ridimensionato. L'heap nuovo avrà una efficienza maggiore perché usando l'HP si allocheranno i nuovi valori come se fosse uno stack.

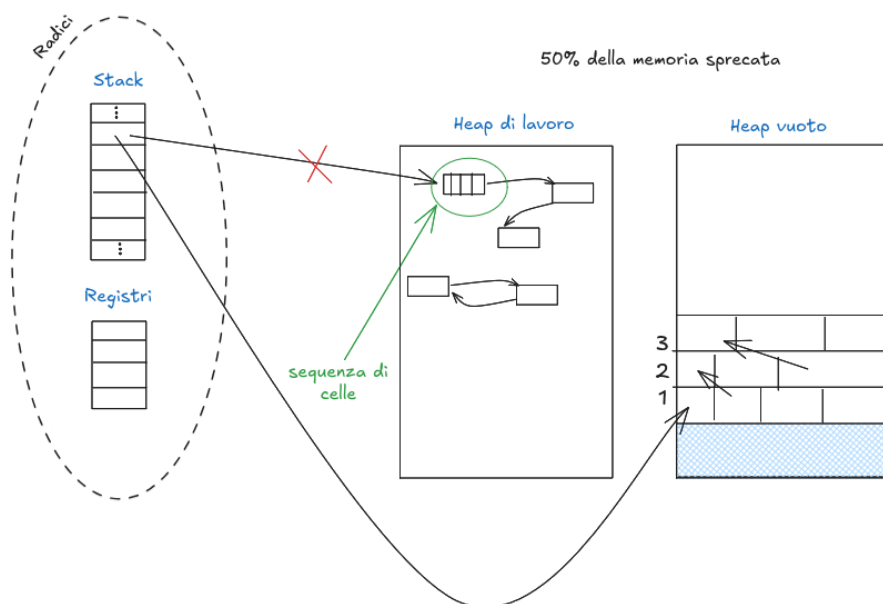


Figura 7: Mark & sweep

Ma cosa succede quando vi è un ciclo? Magari due radici puntano alla stessa cella. Bisogna preservare lo sharing: se una cella è già stata vista bisogna segnare un puntatore alla nuova cella già creata.

Dopo aver letto i dati ottenuti da quella cella va ad allocare il nuovo dato ed inserisce un puntatore che va dalla prima del vecchio dato alla nuova posizione: in questo modo sappiamo che quella cella è stata già allocata.

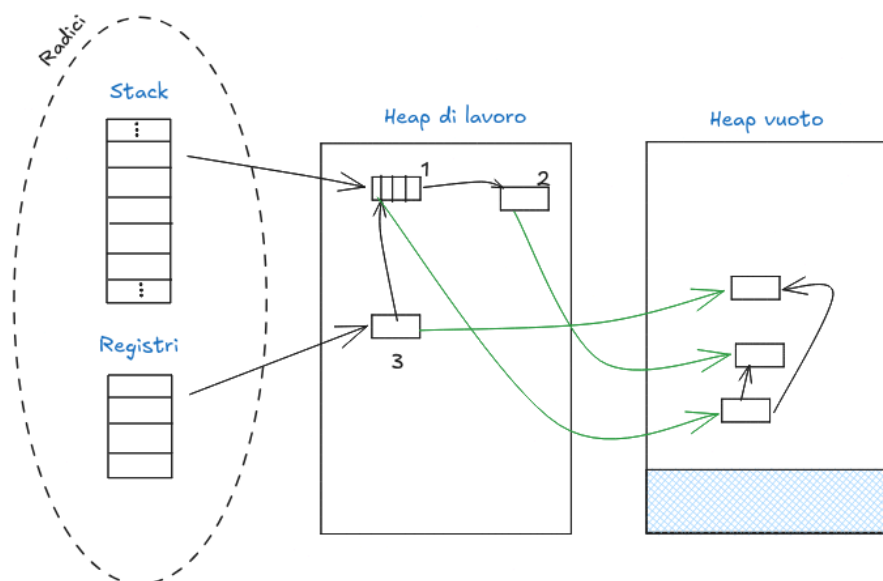


Figura 8: Fix di Mark & sweep che preserva lo sharing

In Figura 8 vi è un fix che preserva lo sharing e non diverge in caso di cicli. Una volta spostato tutto i puntatori di colore verde vengono eliminati. Non ho memoria di allocazione. È lineare

nel numero dei dati vivi dell'heap (lineare il numero di archi ma essi sono campi quindi sono lineari); idem per le radici (si inizia con un ciclo for di esso). Meglio avere strutture come le code di priorità per gestire gli heap.

In somma: spazio $O(0)$, in tempo $O(\# \text{ radici} + \# \text{ dati vivi})$.

Il costo sopra viene fatto ad ogni passata: il numero dei dati vivi può essere molto elevato.

Vi sono diverse tecniche per far scattare l'algoritmo in base alla tipologia del programma: magari se I/O bound oppure in base alla grandezza dello stack.

Nel caso di puntatori vi sono operazioni che si possono svolgere:

- Dereferenziazione;
- Somma scalare all'interno dei blocchi;
- Sottrazione scalare all'interno dei blocchi;
- L'uguaglianza/disuguaglianza va bene perché abbiamo preservato lo sharing;

E no:

- $<, >, \leq, \geq$ perché in caso si siano spostati i dati, tali relazioni possono non preservarsi.
- Hash. Il puntatore non può essere né la chiave dell'hash table né la chiave dell'albero.

Nel caso di somma e sottrazione siamo ok perché sono dati boxed. In C si possono fare operazioni fino al **n+1** blocco.

Si potrebbe mettere un ID univoco al dato boxed.

Non si possono usare i puntatori come chiavi nei dizionari perché potrebbero cambiare valore ad ogni iterazione.

Il RC va bene per qualsiasi operazione mentre il Mark&Sweep no.

3.1.3. Mark & sweep generazionale

L'ipotesi generazionale è un'ipotesi statistica teorizzata. Un dato x può essere:

Tipo 1. Destinato ad essere reachable molto a lungo.

Tipo 2. Destinato a diventare unreachable velocemente.

Nei programmi non vi sono dati intermedi a questi due tipi. I dati che rimangono a lungo spesso sono in strutture dati globali che rimangono attivi per tutta la durata del codice; i dati temporanei durano una qualche chiamata d'istruzione.

Ad esempio, un compilatore costruisce la symbol table molto velocemente con tanta roba. Successivamente esso crea il grafo delle dipendenze per allocare le variabili ai registri, che poi elimina una volta dopo; crea il grafo per minimizzare il caricamento e scaricamento dei dati nei registri.

Per i dati che presumo siano facilmente unreachable mi conviene tenere Mark&Sweep perché l'algoritmo è lineare nei dati vivi: un dato che muore velocemente mi permette di non fare troppi spostamenti fra heap. Molti linguaggi usano due generazioni (dati che muoiono in fretta e dati che muoiono quasi mai) ma si può estendere.

Qui l'**area young in uso** è il corrispettivo dell'heap di lavoro. L'**area young non in uso** è il corrispettivo dell'heap vuoto. L'**area old** (oppure **area major**) è sempre in uso. I dati che sono del tipo 1 vengono allocati nell'area old. Gli altri fanno continuo GC.

Come faccio a sapere di che tipo è il dato? Si usano euristiche come l'algoritmo Second Chance (anche detto LRU): se in un doppio tempo non muore allora presumo non morirà più e dunque lo sposto in area old. Come sapere quante volte ho spostato un dato? Potrei farlo con un contatore ma è quasi inutile perché posso usare un secondo puntatore che funge da marker per vedere i dati a cui è stata data una chance.

Nella **minor collection** faccio una deframmentazione della memoria. I dati che ho sotto HP nel momento in cui finisco una minor sono dati che sono già stati spostati una volta (quindi a cui è stata data una seconda chance). Tutti i nuovi dati allocati nell'heap incrementano HP ma non HWM.

I dati sotto l'High Watermark (HWM) si spostano in area old; gli altri in area young.

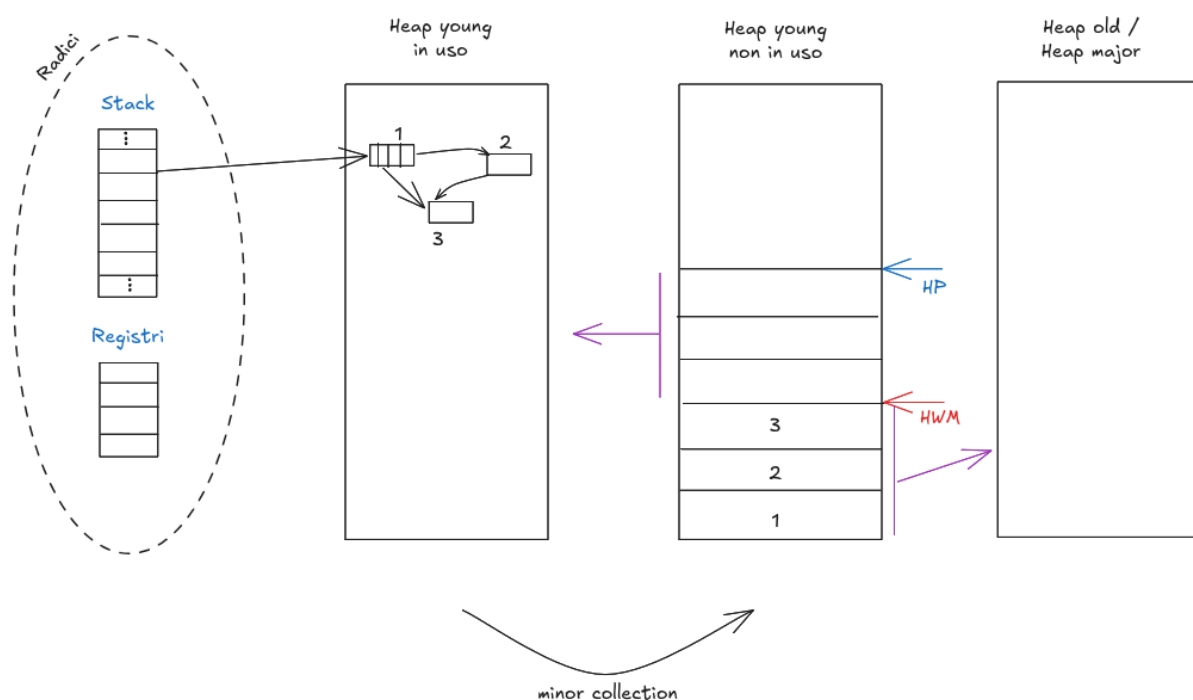


Figura 9: Mark & sweep generazione

Quando il major è pieno e tutti e 3 hanno la stessa dimensione, faccio una passata del major usandolo come se fosse il young in uso. Oppure posso scambiare i dati nel major un po' alla volta per spostare in basso quelli in uso, però anche se più semplice è più lento.

Nella pratica l'area old è molto grande e si usano degli algoritmi che permettono meno spreco di memoria.

Vi sono radici che potrebbero puntare all'heap major, ma questo non incide nel costo. Vi sono anche puntatori dall'area young a quella old, ma non devo seguirli perché sennò non si avrebbe una distinzione fra le due aree. Quindi il costo è computazionale SOLO nell'heap young. Il costo computazione è dunque:

$$O(\# \text{ radici} + \# \text{ dati young vivi})$$

Se ho immutabilità allora non ho archi da old a young. Se così non fosse ci sarebbe il rischio di deallocare tutti i nodi in young. Se il dato è nell'heap old che punta ad uno young vuol dire che tale valore è stato allocato sopra l'HWM, quindi un dato che è stato allocato in memoria young riceve un puntatore da un dato che è allocato in memoria old (perché allocato prima). IMPOSSIBILE.

In OCaml ho dati nello heap mutabili, quindi è possibile mutare un campo in old per puntare a young ma in questo caso non si può sapere se il dato in young è ancora attivo. In questi casi, aggiungo alle radici una lista di celle dello heap old: se non fosse una radice contrassegnerebbe il valore nello young come garbage. Assegnamento di un puntatore diventa un'operazione complessa per update della lista delle radici. Questi in genere sono parecchio meno performanti.

L'overhead che mi dice «quando vuoi assegnare i dati devi fermarti, fare controlli, aggiornare strutture dati e poi ripartire» prende il nome di **write barrier**. Frammento di codice che viene eseguito prima di ogni operazione di storage. Garantisce che le invarianti generazionali vengano rispettate.

Per fare interagire un codice che usa questo sistema di GC ad un altro, bisogna mettere un binding (allineare convenzioni di chiamate, tipi di dato) e sincronizzare i gestori della memoria. Questo è il motivo per cui si preferisce passare dal C che automaticamente non gestisce la memoria.

La sincronizzazione tra linguaggi è una fonte di garbage.

I binding sono una terza fonte di radici.

Blocco dei dati in memoria: il C può accedere al mio dato e quindi non posso spostarlo. Lo sposto un'ultima volta prima di darlo al C così evita di avere tutto sto giro di garbage.

Un dato può essere reachable solo a partire da strutture dati del C.

Il modo in cui si scrive una funzione aiuta il GC a capire quando un dato è garbage. Il GC influisce anche di un 30% dell'esecuzione del programma.

4. Funzioni di prima classe e chiusure

Essere un'entità di prima classe significa che può essere l'argomento o il risultato di una funzione, può essere definita dentro una funzione e in generale può essere memorizzata in una struttura. In un linguaggio tipato bisogna descrivere il suo tipo.

In C vi sono i puntatori a funzioni che descrivono cosa quella funzione si aspetta di input e cosa in output ma non possiamo dire che il C sia un linguaggio in cui le funzioni sono entità di prima classe. La descrizione non è condizione sufficiente per dire che un linguaggio ha HOF. Benché esistono puntatori a funzione non è possibile definire funzioni dentro funzioni, questo classifica il linguaggio come non funzionale.

Però, in Pascal, vi sono funzioni dentro funzioni ma non c'è il tipo «funzione».

Invece, in Haskell, le funzioni hanno un tipo e possono essere definite dentro altre funzioni ed essere il risultato di altre funzioni ma anche essere memorizzate in una struttura.

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

 Haskell

La freccia separa il dominio (quello in input) dal codominio (quello che la funzione produce in output).

Haskell fa inferenza di tipi per `aux` ma volendo si può esplicitare come `aux :: Int -> Int -> Int -> Int`. L'associatività della freccia è a destra. È buona norma, anche a fine di documentazione, esplicitare il tipo alle funzioni globali.

Per fare un test se un linguaggio è funzionale o meno si può fare il test di composizione funzionale. $(f \circ g)(x) = f(g(x))$

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

 Haskell

Quindi la composizione funzionale di `f g` è una funzione che ritorna un'altra funzione.

La sintassi dell'applicazione di un argomento è più leggera possibile, mettendo uno di fianco all'altro. L'operazione di applicazione di funzione è associativa a sinistra.

```
1 f(x) = f x
2
3 f(g(x)) = f (g x)
```

Questo test per linguaggio C viene fatto facendo un'estensione del C stesso come:

```
1 int (*)(int) compose(int (*)(int), int (*)(int)) {
2   int aux(int x) {
3     return f(g(x));
4   }
5
6   return aux;
7 }
```

 C++


```

8
9  int main() {
10     int (*plus2)(int) = compose(succ, succ);
11     printf("%d", plus2(1));
12     return 0;
13 }

```

Questo non va bene perché alla chiamata di `plus2` viene chiamato `aux` ma gli slot che contengono `f` e `g` non esistono più.

Una funzione è una **computazione ritardata** che può accedere a nomi definiti all'esterno del suo corpo ma che potrebbero non esistere più nel momento in cui il corpo viene eseguito.

In Pascal si può definire una funzione dentro una funzione ma non restituirle come risultato: se una funzione accede a nomi definiti all'esterno del suo corpo, deve trattarsi di variabili globali o di variabili locali ancora esistenti. Quando una funzione locale viene chiamata la funzione più grande non è ancora terminata: quindi l'AR di `E` è ancora nella pila e quindi si trova l'argomento `x`.

```

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;

```

pascal

4.1. Chiusura

Quando chiamo una funzione bisogna ripristinare l'ambiente che la funzione vedeva nel luogo e nel momento in cui la funzione è stata definita. Quindi la funzione è in un ambiente in cui ha tutto ciò di cui ha bisogno.

Questo si chiama chiusura: struttura composta che contiene il codice della funzione + il valore di tutti i nomi non locali a cui la funzione fa riferimento. La chiusura viene allocata nell'heap del programma in modo da poter sopravvivere anche alla terminazione delle funzioni che hanno causato la creazione della chiusura.

```

1  add = \x -> \y -> x + y
2  plus1 = add 1

```

Haskell

Quindi `plus1` fa riferimento alla funzione `\y -> x + y` per un `x = 1`.


4.1.1. Currying

Modo di definire funzioni con argomenti multipli come una cascata di funzioni ad argomento singolo.

In pratica una funzione a due argomenti è vista come una funzione ad un argomento che ritorna una funzione con un argomento.

Avere un linguaggio ad un solo argomento permette di avere un linguaggio semplice. Il currying permette di definire funzioni specializzando altre funzioni.


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

 Haskell

Ad esempio, la funzione successore può essere definita come chiusura di `add 1` e quindi applicarla parzialmente.

Preso ad esempio il quicksort:

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

 Haskell


Qui il `:` in `x : xs` è il cons, il quale separa la testa della lista (un elemento) alla coda (una lista). Al filter viene applicato un predicato: funzione che restituisce true o false in base alla condizione esposta su ogni elemento della lista. In questo caso `filter (<= x) xs` è zucchero sintattico per dire che sto applicando parzialmente la funzione `<= x` in cui il secondo elemento è congelato a `x`, quindi tutti gli elementi di `xs` saranno applicati a `<= x`. Quindi specializzo la funzione in modo da divenire «essere \leq ad x ».

Inoltre permette di scrivere codice più compatto. `f x y` è diversa da `f (x,y)` e risparmia pure 3 lessemi.

4.1.2. Incapsulamento

L'idea è di simulare la definizione di un oggetto (nel senso di OOP) che si comporti come un contatore. Si realizza in maniera funzionale. Serve una locazione di memoria in cui memorizzare il valore del contatore, un metodo per leggere il valore, un metodo per incrementare e un ultimo metodo per resettare.

```
1 let make_counter () =
2   let c = ref 0 in
3   (fun () -> !c),
4   (fun () -> c := !c + 1),
5   (fun () -> c := 0)
6
7 let get, inc, reset = make_counter()
```


 OCaml

Sono tre chiusure: funzioni che fanno riferimento alla variabile contatore `c`. Con questo ho realizzato una forma d'incapsulamento in cui possiamo accedere al valore solo mediante queste funzioni. Il `ref` permette di creare valori nell'heap mutabili, ma bisogna accedere al valore esplicitamente come `!c`.

4.1.3. Laziness

Facendo lo zip di una lista con la sua coda mi permette di ottenere una lista di coppie di elementi consecutivi. Per verificare se la lista è ordinata mi basta applicare la funzione `<=` a tutti gli elementi della lista di coppie.

```
1 zip f [] _ = []
2 zip f _ [] = []
3 zip f (x : xs) (y : ys) = f x y : zip f xs ys
4
5 sorted xs = and (zip (<=) xs (tail xs))
```

 Haskell


Ma la funzione `tail` è una funzione parziale. Viene valutato all'occorrenza, ma non viene valutato nel caso in cui `xs` è la lista vuota perché `zif`, nel caso in cui il secondo argomento è una lista vuota, non valuta il terzo argomento.

Per ritardare la valutazione degli argomenti, Haskell, crea chiusure in modo da ricordarsi il valore dei nomi non locali che vengono usati nelle sotto-espressioni in modo tale che quando `zif` deve valutare dei nomi in sospenso, essi hanno tutte le informazioni che gli servono per procedere con la valutazione.

Il fatto di non poter capire quale sia l'ordine di valutazione delle espressioni è un bel problema.

In un linguaggio lazy come Haskell non c'è bisogno di creare costrutti cortocircuitati perché la definizione dell'`and` è:

```
1 (&&) False _ = False
2 (&&) _ x = x
3
4 and [] = True
5 and (x : xs) = x && and xs
```

 Haskell

4.1.4. Comportamento in Java

Preso

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

 Java

`x` è un campo dell'oggetto. Java usa gli oggetti per rappresentare chiusure dove i campi degli oggetti sono i campi che contengono i valori dei nomi non locali cui la lambda espressione fa riferimento.

Le variabili sono immutabili. L'idea di copiare il valore delle variabili dentro le chiusure ci sta, ma Java permette di fare riferimenti a variabili non locali che sono `final`. Se non viene definita `final` ma non viene comunque mai alterata possiamo dire che `x` è effectively final.

Chiaro che se invece di usare un `int x` avessi un riferimento ad un oggetto e lo cambiassi, in quel caso la chiusura perderebbe di significato anche se valida.

4.2. Oggetti vs chiusure


Negli oggetti abbiamo dei campi mutabili e riferimenti a metodi. I metodi hanno un parametro implicito (`self` o `this`) con riferimento all'oggetto ricevente.

Nelle chiusure le funzioni hanno al loro interno riferimenti a valori non locali usati per il calcolo. I valori sono immutabili.

4.2.1. Comportamento in C

Vogliamo rappresentare la lambda espressione qui di seguito in C.

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

 Haskell

Qui noi potremmo avere due funzioni `f1 = \x -> \y -> x + y` e `f2 = \y -> x + y` la quale però è più complessa della prima perché deve gestire anche il valore di `x`.

```

1  typedef void* value;
2
3  typedef struct closure {
4      /* il valore di x viene recuperato accedendo alla chiusura nel primo parametro
       */
5      value (*func)(struct closure*, value);
6
7      /* non occupa spazio ma quando si alloca nell'heap il `data` prende un valore
       specificato da `lambda()` */
8      value data[0];
9  } closure;
10
11 value to_value(intptr_t n) {
12     return (value) n;
13 }
14
15 intptr_t to_int(value v) {
16     return (intptr_t) v;
17 }
18
19 /* funzione ausiliaria usata ogni volta che bisogna chiamare una chiusura */
20 /* n è il numero di valori associati che vanno in `closure.data` */
21 closure* lambda(value (*func)(closure*, value), int n) {
22     closure* cls = (closure*) malloc(sizeof(closure) + n * sizeof(value));
23     cls->func = func;
24     return cls;
25 }
26
27 value apply(value f, value arg) {
28     closure* cls = (closure*) f;
29     return cls->func(cls, arg);
30 }
31
32 value f2(closure* cls, value y) {
33     return to_value(to_int(cls->data[0]) + to_int(y));
34 }
35
36 /* una volta conosciuto il valore di x bisogna creare una chiusura per modellare
   f2 */
37 value f1(closure* cls, value x) {
38     /* conosciamo x ma vogliamo creare una chiusura che sia a conoscenza di x */
39     closure* c = lambda(f2, 1);
40
41     /* copiamo il valore di x dentro la chiusura che modella la funzione interna
       */
42     c->data[0] = x;

```

```

43
44     return (value) c;
45 }
46
47 /* ((f1 2) 3) */
48 int main() {
49     /* si chiama lambda(f1, 0) perché f1 non usa nessun valore non locale */
50     int res = to_int(apply(apply(lambda(f1, 0), to_value(2)), to_value(3)));
51 }

```

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.

5. Classi, interfacce e trait

5.1. Linguaggi object-based

Nei linguaggi object-based è possibile aggiungere o rimuovere campi e metodi da un oggetto. Non c'è uno schema fisso classe che definisce i campi e i metodi.

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

A livello implementativo i metodi dell'oggetto non sono altro che puntatori a funzioni che prendono come primo parametro il riferimento all'oggetto, quindi molto simile al concetto di chiusura.

```
1 obj = struct {
2   x = 2
3   set = set'
4   double = double'
5 }
6
7 set'(self, n) = self.x <- n
8 double'(self) = self.set(self, 2 * self.x)
9
10 obj.double()
```

Il significato del metodo `double` dipende dallo stato dell'oggetto in un particolare momento. Funziona correttamente con l'invariante che il metodo `set` assegna il valore di `n` al campo `x`. Il metodo `set` può essere cambiato runtime rompendo il funzionamento di `double`.

Concepire un concetto di analisi statica è complessa perché il tipaggio in questi casi è difficile da definire.

Definire un concetto di correttezza diviene complesso quando l'implementazione di un metodo cambia a runtime.

5.2. Linguaggi class-based

L'oggetto viene creato in una famiglia ben definita. L'insieme dei metodi è fissato una volta per tutte.

```
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()
```

Il tipo di un oggetto non può cambiare in maniera arbitraria, quindi è possibile ben definire il tipaggio, e quindi anche un'analisi statica.

I metodi sono campi di una virtual table: unica per tutti gli oggetti istanza di una certa classe e può essere precalcolata ancor prima che il programma vada in esecuzione.

Dunque la struttura ha tutti i campi e un puntatore alla virtual table in cui vi è una entry per ogni metodo della classe.

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

5.3. Ereditarietà

Meccanismo linguistico che permette di creare classi specifiche a partire da classi più generiche. Definisce in maniera incrementale nuove classi, ereditando tutti i campi e metodi di una classe aggiungendo nuovi campi e metodi.

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

La sua implementazione è naturale:

```
1  struct D {
2      vtab : D_virtual_table
3      x    : int
4      y    : int
5  }
6
7
8  struct D_virtual_table {
9      set : int -> void = set'
10     double : void -> void = double''
```

```

11  get : void -> int = get'
12  }
13
14
15  set'(self, n) = self.x <- n
16  double''(self) = double'(self); self.y <- 2 * self.y

```

La gerarchia che viene è dunque una relazione del tipo «X is-a Y» ma non dice nulla su cosa un oggetto è davvero in grado di fare. Potrebbero esserci casi in cui si definiscono metodi non completamente polimorfi o non ben generalizzati perché ci potrebbero essere elementi capaci di fare qualcosa che non sono presenti nella gerarchia sotto forma di albero definita con l'ereditarietà.

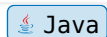
Non c'è un'indipendenza totale dalla implementazione della classe base. Non c'è bisogno di fare overriding di tutti i metodi nella classe derivata perché c'è il late binding (o dinamico) che capisce automaticamente che bisogna invocare i metodi definiti nell'oggetto reale (se è della classe derivata piuttosto che nella classe base).

Chi eredita il codice potrebbe semplicemente fare cut-and-paste, facendo rimanere sempre dei legami molto forti tra le classi che dipendono da come le classi sono realizzate internamente.

```

1  class Stack<T> {
2    ...
3  }
4
5  class SizedStack<T> extends Stack<T> {
6    ...
7  }

```



Relazione di sottotipo e principio di sostituzione mi permettono di usare `SizedStack` dove nel codice è atteso un `Stack`.

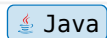
5.4. Composizione

Si intende sviluppare oggetti che all'interno hanno campi con riferimento ad altri oggetti.

```

1  class Stack<T> {
2    ...
3  }
4
5  class SizedStack<T> {
6    Stack<T> s;
7  }

```



In questo modo si è forzati esplicitamente a definire tutti i metodi usati. Però ho comunque il vantaggio di non dover riscrivere il codice della classe che prima era usata per base: le modifiche alla classe `Stack` non comportano modifiche da fare nella nuova classe `SizedStack`.

C'è riuso del codice ma non più un principio di sostituzione perché non c'è più una relazione fra le due classi.

Non si dovrebbe usare esplicitamente il campo `s`: deve essere necessariamente privato per evitare disallineamenti.

5.5. Interfacce

È più un concetto di «cosa un oggetto sa fare» piuttosto che «cosa un oggetto è».

Mi permette di definire un insieme di metodi che devono essere implementati per realizzare il polimorfismo.

Non mi impone una gerarchia ad albero. Ci sono classi in regioni disgiunte nella gerarchia ma che possono implementare la stessa interfaccia.

5.6. Trait

Elenco di operazioni per un'entità. In letteratura si trovano anche col nome di interfacce o di mixin.

In Go un metodo è una funzione definita su un argomento chiamato receiver. Un metodo come `o.m(a1, ..., an)` è lo stesso di fare `m(o, a1, ..., an)`. Avere il receiver non vuol dire avere late binding, è solo una variante sintattica, quasi come se fosse zucchero sintattico. Mettere in evidenza il receiver ha l'effetto di rendere molto più facile avere un IDE che suggerisce i metodi a disposizione (in questo esempio si sa che la struct `o` ha a disposizione il metodo `m`). I metodi possono essere definiti solo su tipi in cui il receiver è definito nel pacchetto corrente.

In Go viene chiamato «interfaccia» ed è un insieme di signature di metodi. Un tipo `T` implementa un'interfaccia `I` se definisce tutti i metodi elencati in `I`: non si esplicita una forma del tipo `T implements I`.

Questo molto in linea col duck typing: se cammina e si comporta come una papera, allora è una papera.

Ad esempio, nella seguente interfaccia:

```
1 type Measurable interface {
2     Area()      float64
3     Perimeter() float64
4 }
```

 Go

parlo proprio di metodi: non si va ad esplicitare il receiver, ma so che se implemento un'interfaccia allora chiamerò questi metodi da una struct. Questo perché ogni receiver ha un tipo specifico.

Si può usare il nome dell'interfaccia come tipo preso in una funzione:

```
1 func PrintMeasure(o Measurable) {
2     fmt.Println(o.Area()) // non può sapere staticamente quale metodo chiamare
                           // senza eseguirlo
3     fmt.Println(o.Perimeter())
4 }
```

 Go

Il compilatore Go silenziosamente capisce se l'argomento passato a `PrintMeasure` è un tipo che implementa le due funzioni presenti in `Measurable`.

Quindi, come in Java, un'interfaccia è un nuovo tipo di dato che rappresenta ogni tipo di dato che implementa l'interfaccia.

A differenza di Java, non devo esplicitare quali entità implementano i metodi dell'interfaccia.

I metodi dell'interfaccia, spesso, accedono a dati della struttura, quindi il compilatore deve sapere qual è la struttura dati a cui si accede quando invoca un metodo specificando il tipo interfaccia.

Un valore di tipo interfaccia `I` è una coppia `(v, p)` dove `v` è l'oggetto (campi ai dati + puntatori ai metodi implementati; è un riferimento al valore concreto che implementa l'interfaccia) e `p` è un puntatore a una tabella dei metodi (simile a una vtable) che contiene le implementazioni dei metodi dell'interfaccia per il tipo concreto di `v`. Ogni volta che si crea un'implementazione a quella interfaccia il compilatore esplicita l'oggetto e i puntatori a dove sono collocati i metodi che ha implementato.

```
1  c := Circle{1}
2  PrintMeasure(c)
```

 Go

Se `Circle` implementa l'interfaccia `Measurable`, al momento della chiamata a `PrintMeasure`, viene creata una coppia `(c, p)`, dove `c` è il valore `Circle{1}` e `p` è la tabella dei metodi con i puntatori alle funzioni `Area` e `Perimeter` implementate da `Circle`.

Quindi, il valore passato a `PrintMeasure` è effettivamente questa coppia `(v, p)`, dove `v` è il valore concreto e `p` la virtual table associata all'interfaccia `Measurable` per `v`.

Si sfrutta il principio di località in fase di compilazione che delinea dove si trovano i metodi perché le implementazioni dei metodi di `Circle` si troveranno solo nello stesso package in cui è definito `Circle`.

L'ereditarietà che si trova nelle interfacce si limita ad estendere l'elenco dei metodi definiti altrove.

```
1  type MeasurableColorable interface {
2      Measurable
3      Color() color
4      SetColor(c color)
5  }
```

 Go

Medesima cosa si trova per le struct, ma è più inteso come composizione. Il campo dell'altra struttura è anonimo. Tutti i campi della struttura inclusa sono accessibili come campi della struttura che la include. I metodi già definiti per la struttura inclusa (come `Area` e `Perimeter`) sono generati anche per la struttura che la include.

```
1  type ColoredCircle struct {
2      Circle
3      c color
4  }
```

 Go

Nel caso di una chiusura definiamo una funzione che può accedere a dati diversi. Quindi singola funzione + pluralità dei dati.

Nel caso dei trait concentriamo l'attenzione su un dato per il quale disponiamo delle operazioni. Quindi singolo dato + pluralità dei metodi.

5.6.1. Trait in Rust

Rust non ha un supporto runtime, macchinari ausiliari come il garbage collector. Bisogna star attenti ad allocazioni, deallocazioni e segmentazioni varie di memoria. Si ha un sistema di tipi con gestione lineare: Rust è in grado, in parte, di capire se una certa entità viene duplicata o meno: il compilatore vede se una risorsa ha 1 unico owner. Non è lineare quando la stessa risorsa viene passata a 2+ funzioni distinte (magari due thread che fanno accesso alla stessa risorsa, dunque nella stessa regione di memoria). Per definizione di risorsa lineare: non ho bisogno di lock perché non vi possono essere situazioni di deadlock su quella risorsa.

Nasce come linguaggio di sistema ma si contrappone al C perché aggiunge delle funzionalità moderne ed avanzate.

Sono presenti i trait, un meccanismo che agisce lato tipi (programmatore definisce insieme di metodi dentro un trait) e lato implementativo (il compilatore deve tradurre in qualche modo il codice in un modo simile a Go). Non c'è quindi un chissà quale lavoro runtime.

In Go l'implementazione è implicita, in Rust no. Bisogna esplicitare che il trait è implementato per un certo dato. Bisogna esplicitare l'argomento di ogni receiver, usando `&self`.

Il sistema di tipi di Rust nasce con l'idea di polimorfismo parametrico. `T` di `print_measure` è un parametro di tipo. `Measurable` è usato come vincolo per `T`.

```
1  trait Measurable {
2      fn area(&self) -> f64;
3      fn perimeter(&self) -> f64;
4  }
5
6  struct Vector {
7      x: f64,
8      y: f64,
9  }
10
11 struct Circle {
12     radius: f64,
13 }
14
15 impl Vector {
16     // metodo NON di istanza; funzioni semplici.
17     fn origin() -> Vector {
18         Vector { x: 0.0, y: 0.0 }
19     }
20
21     // metodo di istanza, come il metodo con receiver in Go
22     fn modulus(&self) -> f64 {
23         (self.x * self.x + self.y * self.y).sqrt()
24     }
25 }
26
27 impl Measurable for Circle {
```

 Rust

```

28     fn area(&self) -> f64 {
29         std::f64::consts::PI * self.radius * self.radius
30     }
31
32     fn perimeter(&self) -> f64 {
33         std::f64::consts::PI * 2.0 * self.radius
34     }
35 }
36
37 fn print_measure<T: Measurable>(shape: T) {
38     println!("Area = {}", shape.area());
39     println!("Perimeter = {}", shape.perimeter());
40 }
41
42 fn main() {
43     let o = Vector::origin();
44     println!("{}", o.modulus());
45     let c = Circle { radius: 4.0 };
46     print_measure(c);
47 }

```


Si usa un nome esplicito `T: Measurable` perché sennò si perderebbe l'informazione diretta sul tipo che passiamo.

Possiamo creare tipi polimorfi con vincoli come, ad esempio, un metodo su un `Rectangle` che va a vedere se in realtà è un quadrato.

```

1  struct Rectangle<T> {
2      width: T,
3      height: T,
4  }
5
6  impl<T> Rectangle<T>
7  where
8      T: PartialEq + Display,
9  {
10     fn is_square(&self) -> bool {
11         println!("{}", self.width); // il tipo di width deve implementare Display
12         self.width == self.height // i due tipi devono implementare PartialEq
13     }
14 }

```

 Rust

In questo modo `is_square` è implementato nel caso il tipo passato al `Rectangle` è un tipo che implementa i trait `PartialEq` e `Display`.

Per implementare il trait `T` per il tipo `U` almeno uno tra i due deve essere definito nel file corrente. Questa flessibilità più grande rispetto a Go, porta a vedere due casi comuni:


1. Definisco un nuovo tipo di dato `U`? Si implementano tutti i trait `T` con cui `U` è compatibile.

2. Definisco un nuovo trait `T`? Implemento `T` per tutti i tipi di dato che esistono in questo momento nel file in cui ho definito `T`.

In Rust i trait possono avere un'implementazione di default, che in genere sono molto generiche perché non si ha conoscenza del tipo del receiver.

Esiste anche qui l'ereditarietà tra i trait, vista comunque come estensione di metodi:


```
1 trait Foo {
2     fn f1(&self);
3 }
4
5 trait Bar : Foo {
6     fn f2(&self);
7 }
```

 Rust

Le invocazioni di metodo avviene in base a diversi fattori:

- Tipo «concreto» (non trait), viene risolto **staticamente**. Il compilatore trova staticamente qual è l'invocazione da chiamare.

```
1 fn smaller(a: Circle, b: Circle) -> bool { a.area() < b.area() }
```

 Rust

- Tipo «generico» viene risolto staticamente grazie alla **monomorfizzazione**, il quale crea a tempo di compilazione tante versioni di `smaller`, una per ogni istanziazione del parametro di tipo `T` (come i template di C++).

```
1 fn smaller<T>(a: Rectangle<T>, b: Rectangle<T>) -> bool { a.width *
  a.height < b.width * b.height }
```

 Rust

- Tipo riferimento o trait viene risolto **dinamicamente**. Il compilatore non sa a compile-time quali sono le funzioni `area` e `perimeter` da chiamare. Il passaggio dell'oggetto avviene esplicitamente: il compilatore fa i controlli del caso sul fatto se l'entità implementa il trait in questione e poi genera, analogamente a Go, una coppia data da valore + puntatore a tabella dei metodi.

```
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: 2.0 };
8     print_measure(&c as &Measurable);
9 }
```

 Rust

6. Classi di tipi

Non ha nulla a che vedere con l'idea di classe nell'OOP.

La classe dei tipi viene in risposta al problema del conflitto fra l'inferenza di tipo (anche se qui è più consono parlare di type reconstruction) e l'overloading. Ad esempio, in ML inferiamo il tipo di `a = int` e `b = float`.

```
1 val a = 3*3
2 val b = 3.14 * 3.14
```

ml

Qui si usa lo stesso operatore di moltiplicazione (`*`) ma ha due significati diversi quando viene chiamato in `a` (perché sono interi) e quando in `b` (float).

Però in ML non è permesso fare una funzione per inferire il tipo, in uno dei punti il compilatore di ML dà errore di tipo. Questo errore è dato dal fatto che non sa se `square` abbia tipo `int -> int` oppure `float -> float`.

```
1 fun square x = x*x
2 val a = square 3
3 val b = square 3.14
```

ml

In OCaml, per evitare problemi di questo tipo, vengono usati operatori distinti con diversi nomi.

```
1 let a = 3 * 3
2 let b = 3.14 *. 3.14
```

OCaml

In realtà c'è una versione di operatore per float per ogni operazione. In questo modo tocca definire due versioni diverse di `square`.

```
1 let squareI x = x * x
2 let squareF x = x *. x
3 let a = squareI 3
4 let b = squareF 3.14
```

OCaml

In OCaml si ha un problema pure per 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

L'uguaglianza fatta tra `x` e `y`, in realtà è un problema perché la funzione `member` potrebbe essere usata su una lista di funzioni (o altri tipi per cui la nozione di uguaglianza non è definita). E in questo caso viene generata un'eccezione runtime.

In ML si ha una notazione di polimorfismo sulla variabile generica `a` in cui si esplicita che ha la nozione di uguaglianza (`"a`"); quando si usa `'a` si ha la medesima cosa totalmente arbitraria di OCaml:

```
1 fun member (x, []) = false
```

ml

```

2     member (x, h :: y) = x == h orelse member x ys
3 ;
4 val member : "a -> "a list -> bool = <fun>

```


Dunque `member` non può essere chiamata su una lista di funzioni (a prescindere, l'uguaglianza di funzioni cos'è? Due programmi possono avere lo stesso output ma avere un corpo diverso [indecidibilità]).

La soluzione di Haskell permette di dividere i tipi in classi (non per forza disgiunte), le quali classi sono definite dalle operazioni supportate da quei tipi.

```

1 (+)      :: Num a => a -> a -> a
2 negate   :: Num a => a -> a
3 (==)     :: Eq a  => a -> a -> Bool
4 (<=)     :: Ord a  => a -> a -> Bool
5 ...

```

 Haskell

Però `=>` può prendere un senso diverso da quello delle type classes, e questo ci porta a capire come Haskell sia sufficientemente descrittivo anche senza questa funzionalità, anzi, Haskell senza type classes è uguale ad Haskell con type classes. Cosa ben diversa da Rust e Go in cui i trait sono implementati nativamente.

6.1. Type checking/inference/reconstruction

Presa l'espressione qui di seguito

```
1 e :: t
```

si ha che `e` è l'input del programma e `t` è l'output del compilatore che dice se l'espressione è ben tipata. L'espressione `e` ha delle variabili, quindi dato un contesto Γ vediamo se l'espressione `e` ha tipo `t` come:

$$\Gamma \vdash e :: t$$

Il contesto Γ è un insieme di variabili con tipo, quindi potremmo vederlo anche come:

$$x_1 :: t_1, x_2 :: t_2, \dots \vdash e :: t$$

Il **type checking** si forma come: noto il contesto e noto il tipo dell'espressione, controllare se l'espressione scritta ha davvero il tipo richiesto t .

La **type inference** invece è un problema in cui assumo di sapere qual è il tipo delle variabili che possono comparire nel programma (conosco il programma ma non il tipo), dunque capire se l'espressione è ben tipata e qual è quello più adatto e generale.

La type inference è più difficile del type checking perché non fa «semplici» controlli.

Nella **type reconstruction** ho solo l'espressione e : il compilatore, avendo solo il programma, deve trovare il modo di dare i tipi alle variabili che compaiono dentro e e poi inferire il tipo t del programma.

Haskell e altri fanno type reconstruction perché al compilatore Haskell diamo il codice della funzione e lui è capace di inferire i tipi delle variabili e poi il tipo dell'intera espressione. Spesso però la type inference è usata anche per far riferimento alla type reconstruction.

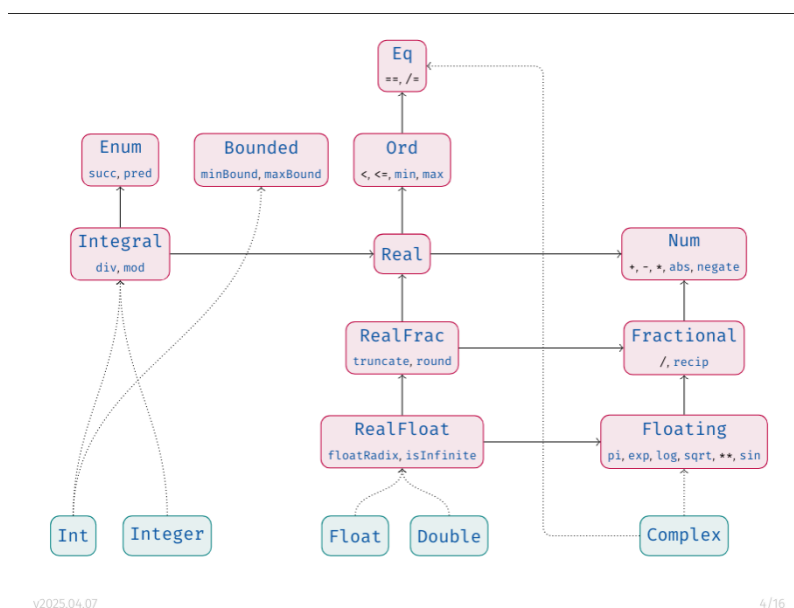
Ad esempio, in OCaml la `*` è disambiguata come operazione di moltiplicazione per interi e da lì il compilatore ricostruisce i tipi interi delle variabili. In Haskell non si disambigua l'operazione e dunque si rimane vago dicendo che l'operazione `*` ha senso per i tipi numerici. Infatti

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

Haskell

e quindi risponde molto generico dicendo che i due parametri sono di tipo `a` che appartiene alla classe di tipi `Num`.

Qui nella figura in seguito vi è un esempio di estensioni tra tipi di classi. Ad esempio, i numeri complessi hanno un concetto di uguaglianza (`Eq`) ma non di ordine totale (`Ord`).



v2025.04.07

4/16

Figura 10: Gerarchia di classi di tipi in Haskell

Una classe in Haskell viene definita come:

```
1 class Num a where
2   (+), (-), (*) :: a -> a -> a
3   negate, abs  :: a -> a
```

Haskell

`Num` è il nome della classe, `a` è un nome scelto dal programmatore che dà la genericità di tale classe. Poi si definiscono le signature.

Riprendendo l'esempio precedente, si può definire la funzione `square` in maniera più generica possibile, definendo un unico vincolo in cui il tipo in cui viene istanziata la variabile `a` deve appartenere a `Num`. Idem per `member`.

```
1 square :: Num a => a -> a
2 square = x * x
3
4 member :: Eq a => a -> [a] -> Bool
5 member _ [] = False
6 member x (y : ys) = x == y || member x ys
```

Haskell

Si può definire l'istanza di una classe in maniera esplicita come:


```

1 instance Num Int where
2     (+) = addInt
3     (-) = subInt
4     (*) = mulInt
5     negate = negateInt
6     abs = absInt

```

» Haskell

E si dice ad Haskell cosa usare quando vengono usate le funzioni. Ad esempio, quando due `Num` di tipo `Int` usano la funzione `(+)` si chiama la funzione `addInt`.

Non vi sono vincoli su dove istanziare i tipi per una determinata classe di tipo.

Vi sono due classi molto usati: `Show` (analogo al metodo `.toString()`) e `Read` (inverte quanto fatto in `show`).

```

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

```

» Haskell

In genere queste due funzioni coincidono perché si serializza quello che si può deserializzare.

Un esempio di tipo che non ha molto senso istanziare per `Show` sono le funzioni: vi sono troppe varie soluzioni diverse di interpretazione per «stampare una stringa per una funzione».

La funzione di `read` è parziale perché non si può sempre derivare un tipo da una stringa.

```

1 ghci> show 123
2 "123"
3 ghci> read "123"
4 *** Exception: Prelude.read: no parse
5 ghci> read "123" :: Int
6 123

```

» Haskell

Grazie alle classi anche le costanti sono molto generiche. Nel secondo caso non ci sono dubbi che sia frazionale ma non si sa se è float, double o altro.

```

1 ghci> :t 2412
2 2412 :: Num a => a
3 ghci> :t 2412.01
4 2412.01 :: Fractional a => a

```

» Haskell

Si può definire una sottoclasse `Ord` in modo tale che si dica «se un tipo è di classe `Ord` deve essere anche di `Eq`»:

```

1 class Eq a => Ord a where
2     compare :: a -> a -> Ordering
3     ...

```

» Haskell

Si possono anche definire implementazioni di default come i trait in Rust:

```

1 class Eq a where
2     (==) :: a -> a -> Bool
3     x == y = not (x /= y)

```

» Haskell

```
4 x /= y = not (x == y)
```

In realtà questi due sono inutili presi così, però si può definire solo uno tra `==` e `/=`.

Si possono definire delle istanze generiche: istanze in cui il tipo che dichiariamo essere istanza di una classe non è completamente definito. Ad esempio, il tipo delle liste di elementi di tipo `a` è di tipo `Eq`, o meglio, per ogni `a` che è istanza di `Eq`; cioè, per le liste deve esistere un'analogia nozione di uguaglianza degli elementi della lista.

```
1 instance Eq a => Eq [a] where
2   [] = []                = True
3   (x :: xs) == (y :: ys) = x == y && xs == ys
4   _ == _                = False
```

» Haskell

Grazie all'implementazione di default ho anche il concetto di disuguaglianza.

6.2. Costruttore di tipo

Un costruttore di tipo può essere visto come una funzione tale che se applicata ad un tipo produce un tipo. Ad esempio, la lista da solo non è un tipo, ma lista di interi è un tipo; lista da solo è un costruttore di tipo.

Un esempio di costruttore di tipo simile alle `Option` di Rust è il tipo `Maybe`.

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

» Haskell

Un esempio di definizione è:

```
1 class Foldable t where
2   length :: t a -> Int
3   sum    :: Num a => t a -> a
4
5 class Functor t where
6   fmap :: (a -> b) -> t a -> t b
```

» Haskell

Si esplicita che la funzione `length` ha un tipo `t a -> Int`; la funzione `sum` ha tipo `t a -> a` per ogni `a` della classe `Num`. `t` può essere visto come un contenitore che accetta elementi di tipo `a`. Ha senso chiedersi sempre la domanda «quanti elementi ci sono nel mio contenitore».

Un funtore è un'altra tipologia di contenitore: si applica una funzione a un tipo generico senza cambiare la struttura del tipo. La `fmap` applica la funzione che forniamo come primo argomento a tutti gli argomenti del contenitore dato in input. Applicare `fmap show (Just 67)` vuol dire ottenere comunque un qualcosa da cui siamo partiti, e infatti abbiamo in output `Just "67"`.

```

1 ghci> length Nothing
2 0
3 ghci> length (Just 67)
4 1
5 ghci> length [1, 2, 3, 4]
6 4
7 ghci> :t product
8 product :: (Foldable t, Num a) => t a -> a

```

» Haskell

In questo modo `length` viene usata su un tipo senza dover esplicitare quale funzione usare per un tale tipo. Lo usiamo anche nei casi in cui è sconveniente dare dei nomi diversi in base al contenitore: `length` e `sum` sono perfetti per un generico contenitore; se non fosse così dovremmo avere `sumList` per le liste e `sumMaybe` per i `Maybe` invece di un `sum` per entrambi.

Tra l'altro `product` ha due vincoli.

Il termine «foldable» sta per «ripiegabile», come combinare fra di loro gli elementi di un contenitore.

Una definizione per la deviazione standard $\mu(X) = \sqrt{E[(X - E[X])^2]}$ può essere:

```

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

```

» Haskell

Quindi per ogni tipo di contenitore `t` a patto che gli elementi di esso siano frazionari, dammi un `t a` e io ti ritorno un tipo `a`.

6.3. Interpretazioni delle classi di tipo

- Overloading

```
1 m :: C a => T
```

» Haskell

Esistono diverse versioni di `m` in base a come viene istanziata `a`.

- Polimorfismo limitato

Il modo in cui scegliamo come istanziare la variabile di `a` lo decidiamo noi. Per questo abbiamo il modo di vedere il polimorfismo con il quantificativo del «per ogni».

```
1 f :: a -> a
```

» Haskell

$\forall \alpha. \alpha \rightarrow \alpha$

dove un esempio è `id :: a -> a`.

Oppure stringendo il tipo di α ad un vincolo:

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

» Haskell

$\forall \alpha \in C. \alpha \rightarrow \alpha$

dove un esempio è `negate :: Num a => a -> a`.

- Trait

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

» Haskell

Un certo tipo è istanza di una classe, e questo si traduce nel far vedere ad Haskell l'implementazione dei metodi in base a suddetto tipo. Questo è molto simile a quanto fatto in Rust e Go, con l'unica differenza che non vi è un esplicito modo di dire che una struct implementa un'interfaccia.

Togliendo il sistema di type reconstruction che ha delle funzioni in più rispetto ad un classico linguaggio funzionale con type inference, possiamo implementare il codice qui di seguito:

```
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
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
```

» Haskell

Con un semplice record (quindi evitando l'uso esplicito di classi di tipo) si ha:

```
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
```

» Haskell

```

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

```

Dichiarare una classe vuol dire definire un tipo record che ha tanti campi quanti sono le operazioni nella classe; dichiarare che un certo tipo è istanza di una classe vuol dire definire un record che ha come tipo quello corrispondente al nome classe dove faccio vedere i nomi dei campi (cioè dei dizionari di funzioni).

La freccia `=>` che separa contesto e tipo diviene una semplice `->` e quindi ha il significato della funzione, dunque `square` diviene una funzione a due parametri, dove il primo parametro è il nome del record `Num` e quindi un dizionario che specifica cosa voglia dire sommare/moltiplicare/negare valori di tipo `a`. Poi si ha `mul numD` perché dal dizionario `numD` si estrae la funzione `mul`.

Nel modello standard di Haskell, il dizionario è passato esplicitamente come argomento della funzione, non incapsulato in ogni valore.

In Rust/Go tutte le volte che passo un argomento ad un metodo e il metodo si aspetta un oggetto di tipo trait, esso viene incapsulato in una coppia col valore dell'argomento e il valore del trait.

Due o più argomenti di un metodo vengono impacchettati in diverse coppie con stesso dizionario se gli argomenti hanno stesso tipo. In Haskell il dizionario e il valore dell'argomento sono tenuti separati così da passare il dizionario una volta sola invece di tante volte quanti sono gli argomenti dello stesso tipo.

7. Monadi

Una monade è un costruttore di tipo `M` e una coppia di funzioni `return` e `bind`. Si può pensare anche come tipo astratto di azioni.

Prende il concetto della branca della matematica chiamata teoria delle categorie.

L'idea è quella di avere una strategia di valutazione call-by-name: l'argomento di una funzione viene valutato (creando una chiusura) se e quando il corpo della funzione necessita il valore di tale argomento. Cosa vista al corso SCP. Questo è un aspetto del linguaggio chiamato laziness.


Il linguaggio deve essere puro: non deve avere effetti collaterali, effetti che nell'esecuzione del programma non concernano la valutazione dell'espressione ma altra roba del tipo stampe IO, scritture di file, connessioni al DB, eccezioni, modifica di aree di memoria. Chiaro che un linguaggio puro al 100% è un po' inutile, perché non mi permetterebbe nemmeno di fare stampe. In un linguaggio puro il concetto di sequenza che magari si ha nei linguaggi imperativi grazie al semi-colon `;` è assente.

In un linguaggio lazy è difficile capire qual è l'ordine di valutazione di un programma. Prendiamo ad esempio una funzione a più argomenti: in un linguaggio tradizionale tutti gli argomenti vengono valutati in un punto e successivamente viene eseguito il corpo della funzione (se la valutazione degli argomenti causano effetti collaterali si sa dove tali errori sono stati generati, rendendo più semplice la collocazione di effetti collaterali); diviene difficile in un linguaggio lazy mettere effetti collaterali perché non si ha coscienza di se e quando gli argomenti verranno valutati.

La maggior parte dei linguaggi puri sono anche lazy. In questa sezione vediamo come un linguaggio puro con effetti monadici è visto come il miglior modo per scrivere dei programmi.


Si ha trasparenza referenziale: se `a = b` allora posso usare uno al posto dell'altro perché so che non causeranno effetti collaterali. Ad esempio,

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

 Haskell

il compilatore può ottimizzare il codice, se non ci sono effetti collaterali, in una cosa del tipo

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

 Haskell

ma se `g` avesse avuto effetti collaterali li avremmo visti 1 volta nel primo codice e 2 nel secondo; se `b` avesse avuto effetti collaterali a causa di `h` non li avremmo visti nel secondo caso. Questo cambio si può effettuare qualora `g` e `h` sono funzioni pure.

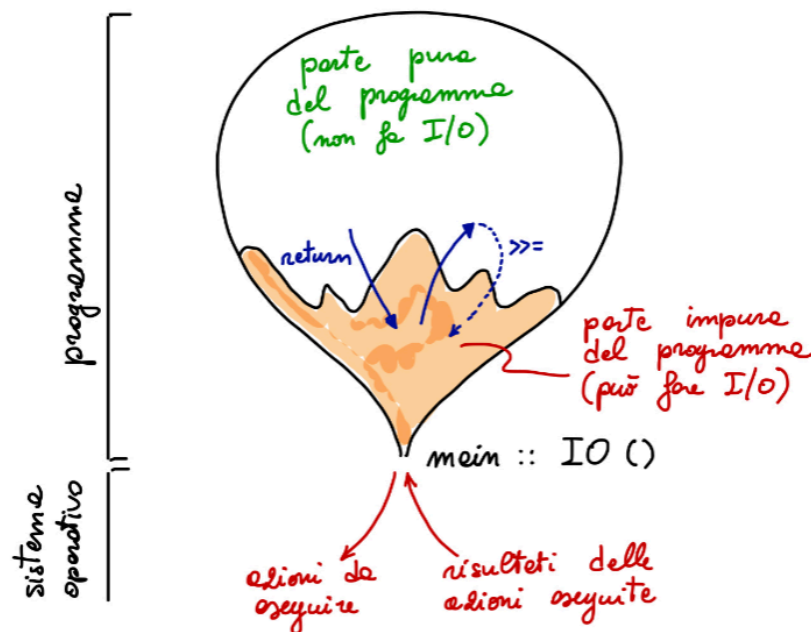
Si possono scrivere programmi modulari in maniera migliore perché i programmi vengano valutati in base alle esigenze. In un linguaggio tradizionale bisogna mescolare la parte di esecuzione del programma che ci serve (tipo la generazione di un numero primo) + la parte che interrompe l'esecuzione (tipo un break al ciclo che genera i numeri primi); nei linguaggi lazy si può creare una generica funzione che calcola infiniti numeri primi ma esso viene calcolato on-demand grazie ad un programma che richiede solo i primi n numeri primi e quindi totalmente indipendenti.

La normalizzazione permette al programma di terminare qualora esso è pensato per terminare; la laziness assicura che il programma termini. Un esempio è `const 1 undefined` perché il secondo argomento non viene mai valutato. Ma questo è anche fonte di imprevedibilità.

Il problema è che la valutazione lazy contribuisce ai leak di spazio perché va ad astrarre il concetto di lifetime degli oggetti: un programmatore non può predire quali strutture rimarranno vive a lungo.

Se il codice è puro allora è anche semplice da parallelizzare.

Quando si lavora con una monade un programma non fa cose ma crea delle azioni che, se eseguite, hanno effetti collaterali. Nel codice del programma si concretizza l'idea delle azioni che, se eseguite, avranno come effetto qualcosa. Quando si programma con le monadi si ha un programma fatto da una parte pura (quella che fa il succo del programma, come il calcolo dei numeri primi) e da una parte che produce azioni. Quindi i programmi sono puri ma passano il cosa fare al sistema operativo.



v2025.04.29

5/37

Figura 11: Struttura di un programma con monadi

7.1. Esempio: valutatore di espressioni

Un esempio di programma che valuta espressioni.

```
1 data Expr = Const Int | Add Expr Expr | Sub Expr Expr | Div Expr Expr
2
3 eval :: Expr -> Int
4 eval (Const n) = n
5 eval (Add t s) = eval t + eval s
6 eval (Sub t s) = eval t - eval s
7 eval (Div t s) = eval t `div` eval s
```

Haskell

La divisione ha però un effetto parziale perché dovremmo gestire anche la divisione per zero. Questo si può fare cambiando il tipo di ritorno di `eval` in modo da trattare i `Maybe`.

```
1  eval :: Expr -> Maybe Int
2  eval (Const n) = Just n
3  eval (Add t s) =
4      case eval t of
5          Nothing -> Nothing -- errore si propaga, quindi se t non va bene, a
                               ricorsione tutto il resto non va bene
6          Just m -> case eval s of
7              Nothing -> Nothing
8              Just n -> Just (m + n)
9  eval (Sub t s) =
10     case eval t of
11         Nothing -> Nothing
12         Just m -> case eval s of
13             Nothing -> Nothing
14             Just n -> Just (m - n)
15  eval (Div t s) =
16     case eval t of
17         Nothing -> Nothing
18         Just m -> case eval s of
19             Nothing -> Nothing
20             Just 0 -> Nothing
21             Just n -> Just (m `div` n)
```

Abbiamo aumentato di molto la complessità del codice, rimanendo puri. `Maybe Int` rappresenta adesso un modo di esprimere una computazione che può avere successo o può fallire. `eval` dunque produce un'azione che produce un risultato `Just ...` o un errore `Nothing`, e questa è una computazione pura perché abbiamo il risultato senza effetti collaterali.

Ora, proviamo a scrivere in maniera pura un modo per contare il numero delle operazioni di divisione.

```
1  type Counter a = Int -> (a, Int)
2
3  eval :: Expr -> Counter Int
4  eval (Const n) x = (n, x)
5  eval (Add t s) x =
6      let (m, y) = eval t x -- y è il valore del contatore attuale dopo aver
                               valutato t
7          (n, z) = eval s y in
8          (m + n, z)
9  eval (Sub t s) x =
10     let (m, y) = eval t x
11         (n, z) = eval s y in
12         (m - n, z)
13  eval (Div t s) x =
```



```

14     let (m, y) = eval t x
15         (n, z) = eval s y in
16         (m `div` n, z + 1)

```

Ma quindi diventa una funzione che prende un valore e un intero usato come accumulatore. Ad ogni `eval` si ritorna un valore e l'accumulatore incrementato di 1 nel caso della divisione. Con questa modifica si è persa completamente la logica del programma.

Essere una computazione pura dipende dal contesto. Si compongono azioni: nel primo caso si ha che se la prima fallisce allora lo fa tutto e così via; nel secondo si ha la modifica del contatore, quindi si ha propagazione del risultato alle azioni successive.

Per mettere tutto insieme bisogna cambiare

```

1 eval :: Expr -> Int

```

[Haskell](#)

in

```

1 eval :: Expr -> M int

```

[Haskell](#)

che però è impura e `M` può essere la monade `Maybe` oppure `Counter`.

In `M` occorrono due operazioni nuove:

- Return

```

1 return :: a -> M a

```

[Haskell](#)

La funzione `return` permette di creare una funzione pura della monade che non ha effetti collaterali e permette di restituire `a`.

- Bind

```

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

```

[Haskell](#)

Il primo argomento `M a` è la prima computazione della monade, che può produrre un valore di tipo `a` (nel caso di `Maybe` potrebbe fallire). Il secondo argomento `(a -> M b)` è una funzione perché la seconda computazione può aver bisogno di conoscere il valore prodotto della computazione precedente: infatti il dominio `a` coincide col tipo della computazione precedente. La computazione composta ottenuta è di tipo `M b`.

Viene usato per comporre insieme due operazioni che possono avere effetti collaterali. Se un'azione fallisce e cerco di comporla con qualcosa dopo, anche quella dopo lo farà. In genere quando compongo due operazioni, la seconda dipende dal risultato della prima.

Nel caso di divisione per zero si ha:

```

1 return :: a -> Maybe a
2 return = Just
3 -- equivalente a
4 -- return a = Just a
5
6 (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
7 (>>=) Nothing _ = Nothing

```

[Haskell](#)

```

8  (>>=) (Just a) f = f a
9
10 abort :: Maybe a
11 abort = Nothing
12 -- visto come un alias dell'altro valore di Maybe, però non è quanto usato nella
   stdlib
13
14 evalM :: Expr -> Maybe Int
15 evalM (Const n) = return n
16 evalM (Div t s) =
17     evalM t >>= \m ->
18     evalM s >>= \n ->
19         if n == 0 then abort
20         else return (m `div` n)

```

Chiaro che se `evalM` fallisse per `t` o `s` non avremmo i risultati `m` e/o `n`. Il corpo della lambda `\m` si estende più possibile a destra dell'espressione dopo la `->`; idem per `\n` che si estende fino alla fine. Nell'esempio non vi sono le parentesi perché non c'è da disambiguare nulla.

Mentre per il counter:

```

1  type Counter a = Int -> (a, Int) -- dunque il valore di ritorno è una
   funzione
2
3  return :: a -> Counter a
4  return a = \x -> (a, x) -- è una computazione pura nella monade Counter: non si
   altera il contatore
5  -- equivalente a
6  -- return a x = (a, x)
7
8  (>>=) :: Counter a -> (a -> Counter b) -> Counter b
9  (>>=) m f = \x -> let (a, y) = m x in f a y
10 -- dunque abbiamo il contatore x che viene applicato alla prima computazione m
11 -- e poi il valore nuovo (y) viene passato alla seconda computazione f
12
13 tick :: Counter () -- operazione che eseguiamo ogni qualvolta vorremo
   incrementare di uno il contatore
14 tick = \x -> ((), x + 1)
15
16 evalM :: Expr -> Counter Int
17 evalM (Const n) = return n
18 evalM (Div t s) =
19     evalM t >>= \m ->
20     evalM s >>= \n ->
21     tick >>= \() ->
22     return (m `div` n)

```

7.2. Monoide

Monade deriva da monoide, una struttura che gode di proprietà associativa:

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

e questa è l'operazione che noi abbiamo chiamato **bind**, capace di comporre tra di loro computazioni e dunque effetti collaterali generati da essi.

Si ha anche un elemento neutro:

$$a \cdot e = e \cdot a = a$$

Per una computazione pura (senza effetti collaterali) la legge di neutralità di e ci dice come «comporre un'operazione a con una computazione senza effetti collaterali e è lo stesso di prendere solo l'operazione a ».

Una monade non è al 100% un monoide perché, nel caso di `bind`, bisogna trasferire informazioni dalla prima alla seconda computazione. Vi è quindi questo sbilanciamento che non ci permette di ragionare proprio come al monoide.

7.3. Leggi di una monade

- 1° legge di neutralità

```
1 return a >>= f
```

[Haskell](#)

è equivalente a

```
1 f a
```

[Haskell](#)

- 2° legge di neutralità

```
1 m >>= return
```

[Haskell](#)

è equivalente a

```
1 m
```

[Haskell](#)

Dunque `return` non aggiunge nessun effetto collaterale e si comporta come elemento neutro.

- Associatività

```
1 (m >>= \a -> n) >>= f
```

[Haskell](#)

è equivalente a

```
1 m >>= (\a -> n >>= f)
```

[Haskell](#)

Ma questo solo se `a` non è libero in `f`, perché sennò cattureremmo l'occorrenza libera di `a` dentro `f` in virtù di `\a`.

Queste leggi dovrebbero essere dimostrate per ogni monade, perché nello ristrutturare il codice, il compilatore, potrebbe però ottimizzare il codice usando suddette leggi.

7.3.1. Dimostrazione per `Maybe`

Presa

```
1 return :: a -> Maybe a
```

[Haskell](#)

```

2 return = Just
3
4 (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
5 (>>=) Nothing _ = Nothing
6 (>>=) (Just a) f = f a

```

Si vede come le proprietà vengano dimostrate:

- 1° neutralità

```

1 return a >>= f
2 = Just a >>= f
3 = f a

```

» Haskell

- 2° neutralità

```

1 m >>= return
2
3 -- caso m = Nothing
4
5 = Nothing
6 = m
7
8 -- caso m = Just a
9
10 = return a
11 = Just a
12 = m

```

» Haskell

- Associatività

```

1 (m >>= \a -> n) >>= f
2
3 -- caso m = Nothing
4
5 = (Nothing >>= \a -> n) >>= f
6 = Nothing >>= f
7 = Nothing
8
9 -- caso m = Just a
10
11 = (Just a >>= \a -> n) >>= f
12 = ((\a -> n) a) >>= f
13 = n >>= f

```

» Haskell

```

1 m >>= (\a -> n >>= f)
2
3 -- caso m = Nothing
4

```

» Haskell

```

5   = Nothing >=> (\a -> n >=> f)
6   = Nothing
7
8   -- caso m = Just a
9
10  = Just a >=> (\a -> n >=> f)
11  = (\a -> n >=> f) a
12  = n >=> f

```

quindi viene dimostrata la legge per transitività dell'uguaglianza.

7.3.2. Dimostrazione per `Counter`


Bisogna fare attenzione all'uso della variabile `a` libera all'interno della funzione `f`.

Presa

```

1  return :: a -> Counter a
2  return a = \x -> (a, x)
3
4  (>=>) :: Counter a -> (a -> Counter b) -> Counter b
5  (>=>) m f = \x -> let (a, y) = m x in f a y

```


 Haskell

- 2° legge di neutralità

```

1  m >=> return
2  = \x -> let (a, y) = m x in return a y
3  = \x -> let (a, y) = m x in (a, y)
4  = \x -> m x
5  = m

```

 Haskell


Usando $\lambda x.Mx \equiv M$ se $x \notin M$.

- Associatività

```

1  (m >=> \a -> n) >=> f
2  = (\x -> let (a, y) = m x in (\a -> n) a y) >=> f
3  = (\x -> let (a, y) = m x in n y) >=> f
4  = \x -> let (b, z) = (\x -> let (a, y) = m x in n y) x in f
5  = \x -> let (b, z) = let (a, y) = m x in n y in f b z


```

 Haskell

```

1  m >=> (\a -> n >=> f)
2  = \x -> let (a, y) = m x in (\a -> n >=> f) a y
3  = \x -> let (a, y) = m x in (n >=> f) y
4  = \x -> let (a, y) = m x in (\y -> let (b, z) = n y in f b z)
5  = \x -> let (a, y) = m x in let (b, z) = n y in f b z

```

 Haskell

7.4. Classe `Monad`

È una classe non di tipi. `m` non è un tipo, perché `Maybe` e `Counter` non sono dei tipi da soli, bensì dei costruttori di tipo. Dunque si rappresenta una classe di costruttori di tipo.

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

» Haskell

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

`Monad` è un caso particolare di una classe molto più generica `Applicative` che a sua volta è un caso particolare di una classe più generica `Functor`.

Quindi se `m` è un `Applicative`, allora `m` è una monade nel momento in cui sono definite le due funzioni `return` e `bind`.

7.5. Operatore and-then `>>`

Ci sono dei casi in cui, quando compongo due azioni di una monade, il risultato della prima computazione non ci interessa. Dunque alla seconda computazione non interessa il risultato prodotto dalla prima ma solo la composizione degli effetti collaterali è importante.

Ad esempio, nel `tick` non viene usato il valore prodotto nella computazione precedente.

```
1 evalM :: Expr -> Counter Int
2 evalM (Const n) = return n
3 evalM (Div t s) =
4   evalM t >=> \m ->
5   evalM s >=> \n ->
6   tick >=> \() ->
7   return (m `div` n)
```

» Haskell

In questo caso è comodo definire un operatore ausiliario «and-then»: esegui l'azione `as` e applica il risultato alla funzione che butta via il valore del suo argomento e produce l'azione `bs` a prescindere dal valore restituito dalla prima.

```
1 (>>) :: Counter a -> Counter b -> Counter b
2 (>>) as bs = as >=> const bs
```

» Haskell

Si può generalizzare l'effetto di and-then ovunque nella libreria standard come

```
1 (>>) :: Monad m => m a -> m b -> m b
2 (>>) as bs = as >=> const bs
```

» Haskell

La terza legge del monoide può essere espressa usando and-then come

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

» Haskell

però in generale serve leggere i valori della computazione, quindi serve `bind`.

Ora, la definizione dell'`eval` diviene qualcosa del tipo:

```
1 evalN :: Expr -> Counter Int
2 evalN (Const n) = return n
3 evalN (Div t s) =
4   evalN t >=> \m ->
5   evalN s >=> \n ->
6   tick >>
```

» Haskell


```
7     return (m `div` n)
```

7.6. Notazione **do**

È semplice zucchero sintattico che permette di scrivere codice monadico senza usare funzioni lambda. Si può usare **do** per enfatizzare il fatto che le azioni che si fanno, oltre a produrre un risultato, hanno effetti collaterali.


Nell'esempio del **Maybe** :

```
1 evalM :: Expr -> Maybe Int
2 evalM (Const n) = return n
3 evalM (Div t s) =
4     evalM t >>= \m ->
5     evalM s >>= \n ->
6         if n == 0 then abort
7         else return (m `div` n)
```

 Haskell

Diviene

```
1 evalM :: Expr -> Maybe Int
2 evalM (Const n) = return n
3 evalM (Div t s) = do
4     m <- evalM t
5     n <- evalM s
6     if n == 0 then abort
7     else return (m `div` n)
```

 Haskell

Il compilatore espande il blocco dopo il **do** come il blocco sopra, quindi è visto come semplice zucchero sintattico.

$$\text{do } E_1, E_2 \Rightarrow E_1 \gg E_2$$


$$\text{do } x \leftarrow E_1, E_2 \Rightarrow E_1 \gg= \lambda x. E_2$$

7.7. Monadi composte

Prendendo l'esempio di monade che gestisce la divisione per zero e monade che gestisce il contatore delle divisioni, è possibile comporle in una sola in maniere diverse:


- Tradizionale, perché vi è il contatore ma anche il caso d'uso in cui il valore è **Nothing**. Se pensiamo al contatore come una variabile globale, possiamo vedere come «l'effetto» dell'incremento rimanga qualora dovessimo avere un'eccezione e aver come ritorno un **Nothing**.

```
1 eval :: Expr -> Int -> (Maybe Int, Int)
```

 Haskell

- Transazionale, in cui **eval** è una funzione a cui passiamo il valore corrente del contatore e se va tutto bene abbiamo **Just(valore, contatore)** sennò un semplice **Nothing**. In questo caso il contatore non viene ritornato qualora dovessimo avere un'eccezione e aver come ritorno un **Nothing**.

```
1 eval :: Expr -> Int -> Maybe (Int, Int)
```

 Haskell

Il problema è che in base a quale dei due metodi viene scelto, la semantica del programma cambia.

Invece di definire una monade per rappresentare computazioni che possono avere «certi effetti», definiamo una trasformazione di monade che aggiunge «certi effetti» a una monade già esistente.

Nella pratica non si definisce la monade `Maybe` in maniera isolata ma diciamo cosa vuol dire aggiungere gli effetti di esso (quindi di fallimento) ad un'altra monade. Idem per la monade `Counter` per cui aggiungiamo l'effetto di incremento di contatore.

Storicamente i monad transformer si definiscono col nome della monade con suffisso `T`. Il codice seguente non funziona perché in Haskell vi sono delle limitazioni agli alias di tipi. Un monad transformer `t` prende in input una monade `m` e restituisce in output una monade `t m`.

```
1  type MaybeT m a = m (Maybe a) -- monade di partenza in cui il
   risultato non è a ma Maybe a
2  -- quindi le computazioni di m possono produrre un valore o possono fallire e
   non produrre niente
3
4  -- return e bind sono funzioni overloaded, e il loro risultato dipende dal tipo.
   Quindi non sono definiti return e bind in maniera ricorsiva.
5  instance Monad m => Monad (MaybeT m) where
6      -- devo descrivere una computazione pura sulla monade MaybeT
7      return = return . Just
8      -- equivalente a
9      -- return a = return (Just a)
10
11     -- il bind trovato dentro è quello della monade m, non quello del MaybeT
12     (>=>) m f = m >=> \x ->
13         case x of
14             Nothing -> return Nothing -- mi serve qualcosa m (Maybe a) e dunque uso
               return
15             Just a -> f a
16
17 abort :: Monad m => MaybeT m a
18 abort = return Nothing
```

Provando a fare il `MaybeT` col counter si ha

```
1  type MaybeCounter = MaybeT Counter
2
3  evalM :: Expr -> MaybeCounter Int
4  evalM (Const n) = return n
5  evalM (Div t s) =
6      evalM t >=> \m ->
7      evalM s >=> \n ->
8      tick >=> \() -> -- ERRORE!
9      if n == 0 then abort
10     else return (m 'div' n)
```


Non c'è nulla di male a comporre azioni della monade `MaybeCounter` con altre di `MaybeCounter` e quindi il `bind` va, però il `tick` non va perché è un'azione della monade interna.

7.7.1. Lifting di operazioni

Quando usiamo `MaybeT` per trasformare una monade `m` in una monade `MaybeT m`, vorremmo poter trasformare computazioni di `m` (che non possono fallire) in computazioni di `MaybeT m` che non falliscono.

```
1 class MonadT t where
2   lift :: Monad m => m a -> t m a
```

» Haskell

`t` è un monad transformer se fornisce un metodo `lift` capace di elevare (promuovere) operazioni della monade originaria che produce `m a` ad azioni della monade trasformata che produce valori di tipo `a`.

```
1 instance MonadT MaybeT where
2   lift m = m >>= (return . Just)
```

» Haskell

Nel caso di `MaybeT` la `lift` è semplice: `m` è un'azione della monade originaria, si esegue l'operazione `m` e si fa `bind` e, dato che l'operazione non è fallita, si usa il `return` della monade originaria per produrre l'operazione pura grazie a `Just`. Quindi si promuovono azioni di una monade `m` (come `Counter`) in azioni di una monade `MaybeT` (come `MaybeCounter`).

Quindi la versione del valutatore che combina le monadi `Maybe` e `Counter`:

```
1 evalM :: Expr -> MaybeCounter Int
2 evalM (Const n) = return n
3 evalM (Div t s) =
4   evalM t >>= \m ->
5   evalM s >>= \n ->
6   lift tick >>= \() -> -- OK!
7   if n == 0 then abort -- fa fallire la monade trasformata
8   else return (m `div` n)
```

» Haskell

Il problema è che Haskell non accetta alias parziali di tipo.

```
1 type MaybeT m a = m (Maybe a)
2 instance Monad m => Monad (MaybeT m) where
```

» Haskell

Il codice sopra dovrebbe essere quello di definire un nuovo tipo, in cui `MaybeT` è un record in cui unico campo è il tipo che avremmo voluto usare, dunque creando un livello di indirezione mettendo un record banale (perché ha un solo campo) attorno al tipo che ci interessa davvero. `MaybeT` e `unMaybeT` sono una l'inversa dell'altra.

```
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
```

» Haskell

```

8     Nothing -> return Nothing
9     Just a -> unMaybeT (f a)

```

Equivalentemente per la trasformata di monade `CounterT`:

```

1 type CounterT m a = Int -> m (a, Int)
2
3 instance Monad m => Monad (CounterT m) where
4     return a = \x -> return (a, x)
5     (>=) m f = \x -> m x >= \ (a, y) -> f a y
6
7 tick :: Monad m => CounterT m ()
8 tick = \x -> return ((), x + 1)

```

» Haskell

La quale forma corretta è in realtà:

```

1 newtype CounterT m a = CounterT { unCounterT :: Int -> m (a, Int) }
2
3 instance Monad m => Monad (CounterT m) where
4     return a = CounterT $ \x -> return (a, x)
5     (>=) m f = CounterT $ \x ->
6         unCounterT m x >= \ (a, y) -> unCounterT (f a) y
7
8 tick :: Monad m => CounterT m ()
9 tick = CounterT $ \x -> return ((), x + 1)

```

» Haskell

7.8. Monade identità

Monade che rappresenta operazioni pure banale `Id`. Da essa possiamo ottenere ogni combinazione come pila di trasformazioni che partono da `Id`. Su essa applichiamo il monad transformer `MaybeT` per avere `Maybe`.

```

1 type Id a = a
2 instance Monad Id where
3     -- funzione che inietta il valore è semplicemente la funzione identità
4     return = id
5
6     -- il bind è la semplice applicazione funzionale
7     (>=) a f = f a

```

» Haskell

Dunque possiamo definire la monade `Maybe` come:

```

1 type Maybe a = MaybeT Identity a
2 -- MaybeT Identity a = Identity (Maybe a) = Maybe a

```

» Haskell

8. Rust

Un linguaggio è di sistema se il compilato può girare senza nessun supporto di runtime o da parte di un OS. Tipicamente viene usato per scrivere proprio OS oppure bare-metal (senza un OS ma è solo quel software che gira).

Come backend si usa LLVM: una libreria che fa le ottimizzazioni per le varie macchine.

Ha un po' di runtime (ecco perché è complicato integrarlo nel kernel Linux) ma non ha system thread (si definiscono quindi appoggiandosi su OS o in user space) o garbage collection.

Si basa su **zero cost abstraction**: il codice che si produce corrisponde a quello che si avrebbe nella maniera più semplice possibile con C e inoltre se si introducesse un'astrazione non aumenterebbe il costo dell'esecuzione. Tutto il sistema di ownership è imposto a compile-time.

Garantisce sicurezza in memoria: niente memory leak (memoria allocata ma mai liberata) o deallocazione doppia (tentativo di deallocazione di memoria già deallocata in passato) o puntatori dandling (puntatore a memoria non accessibile; magari deallocata o out-of-scope) o data race (operazioni di scrittura nella stessa locazione di memoria da parte di operazioni che vengono eseguite in diversi thread in maniera concorrente).

Inoltre il sistema di tipi + smart pointer minimizzano i problemi della concorrenza. Non li rimuovono perché c'è una complessità intrinseca che bisogna gestire comunque.

Vi sono due modi per la gestione della memoria.

Il primo si basa sul concetto che in informatica le risorse (socket, file pointer) non sono duplicabili: qui nasce il concetto di ownership (tipo Resource Acquisition is Initialization (RAII)) con borrowing in r/w. Questo funziona automaticamente senza costi a runtime ma il sistema di tipo (approssimazione decidibile di qualcosa non decidibile) potrebbe non funzionare egregiamente.

Il secondo prende l'idea che ha C++ degli smart pointer e li implementa coi trait: in questo modo abbiamo una forma di puntatori che però rispetta le invarianti. In questo caso è il programmatore che dice quali invarianti ha in mente da dover rispettare in parte anche a runtime.

8.1. Ownership

Ogni cella di memoria sullo heap ha un solo owner, il responsabile per la sua deallocazione. Diciamo che sullo stack la memoria lavora in push/pop tranne quando ci sono le chiusure, quindi non serve un responsabile per deallocazione.

- Quando una cella viene allocata sullo heap restituisce un puntatore che viene assegnato ad una variabile sullo stack. In questo caso la variabile che è sullo stack è l'owner del dato sullo heap.
- Quando il puntatore viene assegnato ad un'altra cella sullo heap (come nelle liste linkate) è la cella sullo heap che contiene il puntatore ad essere l'owner. Se `A -> B` allora `A` è l'owner di `B`.

Quando un owner viene deallocato, tutte le celle possedute da esso vengono deallocate.

- Se l'owner è nello stack viene deallocata quando viene fatta POP dello SF. Prima di fare il pop viene fatta la free nello heap.
- Se l'owner è una cella dello heap si vede se essa è owner e punta ad altre celle: in tal caso dealloca tutte.

Quando si chiude un blocco si considera quella riga come il punto in cui si deallocano tutte le variabili per cui le variabili dichiarate dentro sono owner.

Una cella sullo heap ha sempre uno e uno solo owner. Non posso avere due owner perché sennò il compilatore inserirebbe due blocchi per la deallocazione. Per un compilatore il problema dell'aliasing è indecidibile: non si sa se due puntatori puntano alla stessa cosa. Un assegnamento o chiamata di funzione fa un **move** di ownership: una variabile che perde l'ownership viene invalidata e non più utilizzabile.

```
1 fn main() {
2     let x = 4; // x è un dato unboxed, quindi sullo stack
3     let s = String::from("ciao"); // s sullo stack punta a una stringa nello heap,
    è una risorsa
4     let y = x;
5     // decommentando la prossima linea e commentando la successiva
6     // s perde l'ownership della stringa
7     //let t = s;
8     let t = String::from("ciao");
9     println!("x = {}, y = {}", x, y);
10    // la prossima riga è errata se s non ha più l'ownership
11    println!("s' = {}, t = {}", s, t);
12
13    // qui il compilatore crea codice di deallocazione per t, s. Questi due sono
    owner e lo capisco dal sistema di tipi
14 }
```

Preso ad esempio

```
1 fn main() {
2     let s1 = gives_ownership(); // presa ownership
3     let s2 = String::from("hello"); // presa ownership
4     let s3 = takes_and_gives_back(s2); // s2 perde ownership; s3 la prende
5 } // le stringhe puntate da s1 e s3 sono deallocate
6
7 fn gives_ownership() -> String {
8     let some_string = String::from("hello"); // allocazione
9     some_string // trasferimento di ownership
10 }
11
12 fn takes_and_gives_back(a_string: String) -> String { // presa ownership
13     a_string // trasferimento di ownership
14 }
```

Si vede che `gives_ownership()` ritorna la `String some_string`: questo vuol dire copiarlo nello SF del chiamante e quindi trasferire l'ownership; quando c'è la chiusura del blocco `some_string` non è più un owner perché l'ownership di `String("hello")` è stato trasferito dal chiamante alla return. Se il dato è boxed il tipo di ritorno esplicita il trasferimento di ownership.

Alla fine di `main` viene inserita la free di `s3` e di `s1`.

Se `s2` non fosse stata inizializzata avremmo avuto un errore di compilazione.

8.2. Reference

Meccanismo che permette di effettuare dei prestiti. Finché lavoriamo in sola lettura sono una comodità; quando vi è una restrizione di dati accessibili in lettura o scrittura divengono una necessità. Un owner può prestare la stessa variabile in lettura a più entità, anche in parallelo: una volta restituiti tutti posso deallocare il dato.

Il meccanismo dell'ownership risponde alla domanda *chi può deallocare il dato*.

Il meccanismo della reference risponde alla domanda *chi può accedere al dato*.

`&x` è una reference a(l contenuto di) `x`. `&mut x` è una reference a(l contenuto di) `x` che permette di modificarne il contenuto.

Se `x` ha tipo `T`, `&x` ha tipo `&T` e `&mut x` ha tipo `&mut T`.

8.3. Borrowing

Prendere una reference di una variabile implica fare **borrowing** delle variabili.

Non vi sono data races:


- Se una variabile è borrowed mutably, nessun altro borrow è possibile e l'owner è frozen (= non può accedere alla variabile fino a quando il borrowing termina).
- Se l'ownership è mutabile (variabile che posso accedere in scrittura) e la variabile viene borrowed (solo in lettura), l'owner è frozen (= non può modificare la variabile fino a quando il borrowing termina).

In sunto:

- Se la variabile è immutabile non ho nessun problema.
- Se la variabile è mutabile:
 1. Tutte le borrow fanno trasformare l'owner in frozen;
 2. Posso avere infinite borrow in lettura;
 3. Posso avere una sola borrow in scrittura.

Il compilatore capisce che il borrow termina quando finisce lo scope della variabile che ha ricevuto il prestito.

```
1  fn main() {
2      let x = 4;
3      let y = &x; // not-real borrowing perché x è unboxed
4      let t = String::from("ciao"); // prende ownership immutabile
5      let s = &t; // borrows immutabile
6      println!("x = {}, y = {}", x, y);
7      println!("s = {}, t = {}", s, t);
8  } // fine del borrowing (s è out-of-scope) e fine di ownership (t è out-of-
    scope) e la stringa è deallocata
9
10 fn main() {
11     let mut x = 4;
12     let y = &x; // x diviene 'frozen'
13     x = 5; // warning: assegnamento alla borrowed `x`
14 }
```

 Rust

```

15
16 fn main() {
17     let mut x = 4;
18     { let y = &x; } // ok: il borrow finisce alla fine del blocco interno
19     // y è deallocata
20     x = 5;
21 }
22
23 fn main() {
24     let x = 4;
25     let z = &mut x; // error: non si può fare borrow immutabile ad una variabile
    segnata come mutabile. type error
26 }
27
28 fn main() {
29     let mut x = 4;
30     let y = &x;
31     let z = &mut x; // error: non si può fare borrow mutabile perché già è preso
    come immutabile
32     // si avrebbe errore solo se si usasse y, a causa della laziness
33 }
34
35 fn main() {
36     let mut x = 4;
37     let y = &mut x;
38     let z = &mut x; // error: si può fare mutabile borrowing usa volta sola
39 }
40
41 fn increment(x: &mut i32) {
42     *x = *x + 1;
43 }
44
45 fn main() {
46     let mut x = 4; // x ha mutabile ownership
47     increment(&mut x); // mutabile borrows
48     x = x + 1; // x si può riassegnare grazie a sta roba
49     println!("x = {}", x); // x = 6
50 }

```

8.4. Lifetime

Indica il momento simbolico quando la cella verrà deallocata dall'owner. Non corrisponde allo scope perché l'ownership può essere trasferita anche mediante chiamate a funzione.

Nel caso di reference vi sono due lifetime: uno di quando il dato muore e uno di quando la reference muore. Il lifetime della reference è strettamente più piccolo del lifetime al dato: prima tolgo la reference e poi dealloco il dato.


I lifetime e le relazioni fra di essi vengono fatte a compile-time ma non sono statici. La soluzione è l'uso di lifetime simbolici, ovvero non in forma di durata in secondi ma in termini di variabili $\alpha, \beta, \gamma, \delta, \dots$. Non ci interessa quanto tempo impieghi ma le dipendenze fra di loro (e.g. $\alpha > \beta$). `'a : 'b` significa che `'a` termina dopo `'b`.

C'è una costante di tipo lifetime chiamata `'static`: corrisponde allo stesso lifetime del C, ovvero vivo fino al termine del programma.

Le reference possono essere tipate con il lifetime. e.g. `&'a i32` è una reference a un `i32` che muore a lifetime `'a`.


Vi è elisione: il compilatore riesce a capire i lifetime da solo quando può.

```
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
8 } // error: il lifetime di s finisce qui ma dovrebbe finire a 'a
```

 Rust

La funzione `dangle` è polimorfa perché per ogni `'a` restituisce una reference ad una stringa che muore al tempo `'a`. È il chiamante della funzione che definisce quando morirà il valore. Poiché il borrow deve terminare prima della deallocazione vuol dire che `'a` è più piccolo del drop, ma non si può sapere perché è un `'a` arbitrario.

```
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 x = 4;
8     let y = 3;
9     let z;
10    z = max(&x, &y);
11    println!("max = {}", z);
12 } // i lifetime finiscono in ordine inverso di dichiarazione
```

 Rust

L'ordine delle variabili è importante per riconoscere le variabili morte che possono essere rilasciate all'interno dello stack.

A livello di tipo devo informare il compilatore, attraverso i lifetime, se un puntatore riferisce ad una zona di memoria che non è più accessibile. Tutte le strutture dati che hanno puntatori devono avere il concetto di lifetime.

Ad esempio, nel codice seguente se non avessi esplicitato il lifetime avrei avuto un dangling pointer:

```
1 #[derive(Debug)]
```

 Rust

```

2 struct ImportantExcerpt<'a> {
3     part: &'a str,
4 }
5
6 fn main() {
7     // la stringa "Call me Ishmael. Some years ago..." finisce già nel data
    segment.
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     println!("{i:?}");
14 }


```

Oppure, grazie ai lifetime possono esplicitare che un tipo generico `T` muoia esattamente ad un lifetime `'a`. Grazie a `<'a, T: 'a>` si esplicita il lifetime del tipo. Il generico `T` deve risolvere il predicato che tutte le reference devono morire dentro un lifetime `'a`. Si usa la sintassi `T: 'a` perché il vincolo di tipo usato ricorda molto le classi di tipo.

```

1 struct Ref<'a, T: 'a>(&'a T);
2
3 fn print_ref<'a, T>(t: &'a T)
4 where
5     T: std::fmt::Debug + 'a
6 {
7     println!("`print_ref`: t is {:?}", t);
8 }
9
10 fn main() {
11     let x = 4;
12     let y = &x;
13     print_ref(y);
14 }

```

 Rust

8.5. Slice

È uno smartpointer per fare borrowing mutabile o meno di una parte di una struttura. Gli esempi classici sono `&str` e `&[T]`: il primo è una slice per una stringa e la seconda ad una array.

Strutturalmente sono un puntatore all'oggetto + numero di byte + capacità residua.

```

1 // proprio a causa delle slice il compilatore vuole una dimensione
    fissata per gli array
2 let xs: [i32; 5] = [1, 2, 3, 4, 5];
3
4 foo(&xs[2..4]);

```

 Rust

8.6. Chiusure

All'interno di uno scope annidato (non globale) vi è un codice che accede a del codice esterno.


Le chiusure catturano le variabili libere all'interno della chiusura, che è una struttura dati capace di accedere a tali variabili esterne.

Esistono diversi tipi di chiusure in Rust, classificate in base a come catturano le variabili dallo scope esterno.

- Trasferimento

La chiusura con move trasferisce la ownership delle variabili catturate dallo scope esterno alla chiusura. Questo è utile, ad esempio, quando si vuole passare la chiusura ad un altro thread.


```
1 move |params| {  
2     ...  
3 }
```

 Rust

- Borrowing

Si definisce se mutabile, immutabile o altrimenti. Si possono chiaramente esplicitare le lifetime.

```
1 |params| {  
2     ...  
3 }
```

 Rust

È possibile fare serie di chiusure annidate per controllare le diverse tipologie di chiusura.

In base alla tipologia di chiusura scelta si usa un trait diverso: `Fn`, `FnOnce`, `FnMut`.

8.7. Smart pointer

Strutture dati in user space che implementano dei trait. Vengono implementati uno o più trait in cui vi sono funzioni per deferenziare il puntatore, per deallocare la memoria quando si perde l'ownership, uno per il borrowing, uno per la copia, uno per la copia per differente thread e così via. In questo modo è possibile definire smart pointer che impongono delle invarianti runtime, quindi con costo runtime a differenza dei puntatori semplici. Ogni operazione che vogliamo fare su un puntatore ha un trait per fare ciò. Uno smartpointer che ha diversi di questi trait diviene indistinguibile al programmatore se un puntatore o reference.


L'uso di esso è quello di forzare determinate politiche di gestione della memoria oppure imporre invarianti nella gestione della memoria che non sarebbero catturate dal sistema di tipi.

Grazie ad essi, infatti, è possibile fare check e lock a runtime.

8.7.1. Box

L'unico che non ha un costo computazionale significativo. Sono i dati boxed, quelli che si trovano nello heap.

```
1 fn main() {  
2     let x = 5 // allocata sullo stack  
3     let b = Box::new(5) // 5 allocato nello heap  
4     // b è uno smart pointer  
5     // allocato sullo stack  
6 }
```

 Rust

```

7   println!("{x}, {b}") // dereferencing di b automatico
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

```

L'invariante di **Box** è che vi sarà 1 solo puntatore all'elemento nello heap (quindi garantisce che la deallocazione è sicura perché singola e anche l'accesso agli accessi concorrenti dato che vi è un solo puntatore entrante).


Viene usato negli stessi casi in cui vengono usati i dati boxed in C o Erlang. Ad esempio, allocare grandi dati conviene farlo nello heap.

L'uso, invece, necessario è quando si fa uso di ADT ricorsivi.

```

1  enum List {
2      Cons(i32, List),
3      Nil,
4  }

```


 Rust

Il compilatore non sa a compile-time quant'è grande la cella per la lista. La soluzione corretta è quella che fa uso di Box:

```

1  enum List {
2      Cons(i32, Box<List>),
3      Nil,
4  }
5  use List::{Cons, Nil};
6  fn main() {
7      let list = Cons(1, // cella allocata sullo stack
8                      Box::new(Cons(2, // cella allocata sullo heap
9                                    Box::new(Cons(3, // cella allocata sullo heap
10                                                Box::new(Nil))))));
11 } // quando list va out-of-scope, tutte le celle vengono deallocate

```

 Rust

Box<T> consente accesso in sola lettura se è immutabile, e lettura/scrittura se è mutabile.

8.7.2. Reference counter (Rc)

Con esso possiamo fare sharing di sotto liste. L'ownership è distribuita e quindi lascio la struttura dati quando i riferimenti ad essi finiscono, un po' anche come funziona il garbage collector in molti linguaggi.

Si contano dunque i puntatori entranti, decrementando tale contatore qualora vengano rilasciati.

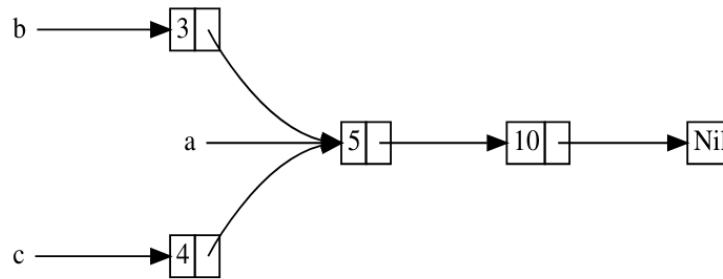


Figura 12: `Rc<T>` smart pointer

Infatti, il codice seguente è errato:

```

1  enum List {
2      Cons(i32, Box<List>),
3      Nil,
4  }
5  use List::{Cons, Nil};
6  // Codice errato (non compila): cosa succedrebbe se b andasse
7  // out of scope e c no?
8  fn main() {
9      let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
10     let b = Cons(3, Box::new(a));
11     let c = Cons(4, Box::new(a));
12 }

```

Rust

E dovrebbe essere:

```

1  enum List {
2      Cons(i32, Rc<List>),
3      Nil,
4  }
5
6  use List::{Cons, Nil};
7  use std::rc::Rc;
8  // Codice corretto:
9  // - Rc::new alloca spazio nello heap per un dato + il suo contatore
10 // di riferimenti
11 // - clone incrementa il reference counter
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

```

Rust

Questa tipologia di reference counter non ha un controllo da parte del compilatore perché gira a runtime.

Rust garantisce che un dato possa avere al più una reference mutabile oppure un numero arbitrario di reference in sola lettura (no data races!).

È considerato puntatore forte.

`Rc<T>` permette solamente reference di sola lettura.

Ha un problema di reference in caso di cicli.

8.7.3. RefCell

Se voglio mutare un valore mentre ho solo un riferimento immutabile alla struttura, posso usare `RefCell<T>`. Consente la mutabilità in tempo di esecuzione, aggirando il borrow checker statico di Rust.

`RefCell<T>` non alloca dinamicamente, quindi può essere usato per valori allocati sullo stack. Utilizza un contatore interno per garantire che vi sia al massimo un prestito mutabile oppure più prestiti immutabili. A differenza delle primitive di sincronizzazione (come Mutex), non usa lock: funziona in contesti a thread singolo.

Il controllo avviene a runtime: se l'invariante dei prestiti viene violato, il programma va in panic. Per accedere al valore interno si usano i metodi:

- `.borrow()` per ottenere un riferimento immutabile (`Ref<T>`).
- `.borrow_mut()` per ottenere un riferimento mutabile (`RefMut<T>`).

```
1  enum List {
2      Cons(Rc<RefCell<i32>>, Rc<List>),
3      Nil,
4  }
5
6  fn main() {
7      let value = Rc::new(RefCell::new(5));
8      let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));
9      let b = Cons(Rc::new(RefCell::new(6)), Rc::clone(&a));
10     let c = Cons(Rc::new(RefCell::new(10)), Rc::clone(&a));
11     *value.borrow_mut() += 10; // per mutare la cella ne acquisisco un lock in
    scrittura
12     println!("a after = {:?}", a);
13     println!("b after = {:?}", b);
14     println!("c after = {:?}", c);
15 }
```

Le teste delle liste sono condivise (grazie a `Rc`) ma anche modificabili (grazie a `RefCell`).

8.7.4. Weak

Puntatori deboli, usati come soluzione al problema dei cicli in `Rc`. Non hanno l'owner dei dati a cui puntano. Non contribuiscono alle reference counts - a differenza degli `Rc` - quindi non hanno un impatto sulla vita dei dati a cui puntano.

I weak (puntatori deboli) accompagnano sempre gli Rc (puntatori forti). Si usano:

```
1  upgrade(r Weak<T>) -> Option<Rc<T>>;
2  downgrade(r Rc<T>) -> Weak<T>;
```

I puntatori weak non garantiscono che il valore sia ancora presente, ecco perché l'upgrade ritorna una `Option`. Qualora il valore fosse una `None` sapremmo che il valore ha avuto un `drop`.

La gestione dei weak si trova in tutti i linguaggi che implementano una forma di garbage collection.

8.7.5. Mutex

Ha un metodo bloccante `lock()` e viene usato per gli accessi concorrenti. Il lock è inteso come semaforo: per avere accesso al dato si chiama esplicitamente il metodo, il quale rimane bloccato qualora non fosse disponibile.

8.7.6. Arc

È un reference counting atomico che, quando accede al numero per incrementare/decrementare il valore di counting, entra in una sezione critica.

9. GADT

Si traducono in «tipi di dati algebrici generalizzati». Permette di definire tipi aggiungendo dei vincoli nel loro tipo di ritorno. Estende gli ADT aggiungendo:

1. vincoli sul tipo dei parametri che possono cambiare in base al valore del costruttore;
2. il quantificatore «forall».

I tipi di dati algebrici (ADT) sono l'ultimo costrutto che è emerso in qualsiasi linguaggi di programmazione (tranne Go): elenca un numero di forme per un dato e ogni dato ha un certo numero di argomenti.


Gli ADT permettono di semplificare il codice non rinunciando ai sistemi di tipi. I costrutti sono implementati in user-space (ad esempio la printf classica di C non è in user-space). È zero-cost abstraction: non si introduce nessun overhead a runtime perché avviene solo in fase di tipaggio ma alla fine il codice generato è simile a quello di prima.

Vi sono situazioni in cui si scrive del codice pulito con sola business logic usando un linguaggio di programmazione non tipato che col sistema di tipi usuale non è tipabile.

Vogliamo rappresentare con una struttura dati gli alberi di sintassi astratta delle espressioni di un linguaggio di programmazione.

In Python possiamo usare la dataclass che genera del codice boilerplate.

```
1  from __future__ import annotations
2  from dataclasses import dataclass
3
4  @dataclass
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 # Unione disgiunta di quanto visto sopra
```

 Python

```

28 Expr = Num | String | Mult | Concat | Eq
29
30 def eval(e):
31     match e:
32         case Num(n):
33             return n
34         case String(s):
35             return s
36         case Mult(e1, e2):
37             return eval(e1) * eval(e2)
38         case Concat(e1, e2):
39             return eval(e1) + eval(e2)
40         case Eq(e1, e2):
41             return eval(e1) == eval(e2)
42
43 good = Eq(Mult(Num(3), Num(4)), Num(5))
44 bad  = Eq(Concat(String("ciao"), Num(2)), Num(3)) # crash

```


L'interprete di Python non ha comunque check sui tipi.

Non si può esplicitare cosa ritorna `eval` perché vi sono invarianti diverse di ritorno in base a cosa fa match:

```

1 def eval(e: Expr) -> int | str | bool:
2     ...

```

 Python


non funziona comunque perché non sono supportati tutti i casi per le operazioni `*`, `+`, `==`.

Invece, prendiamo lo stesso ADT ma scritto in OCaml:

```

1 type expr =
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 (* res è un tipo di dato monadico; vi sono dei costruttori di tipo per i
16    risultati *)
17 let rec eval : expr -> res = function
18   | Num n -> I n

```

 OCaml

```

19 | Mult (e1, e2) -> (
20     match (eval e1, eval e2) with I n1, I n2 -> I (n1 * n2) | _, _ -> E)
21 | And (e1, e2) -> (
22     match (eval e1, eval e2) with B b1, B b2 -> B (b1 && b2) | _, _ -> E)
23 | Eq (e1, e2) -> (
24     match (eval e1, eval e2) with
25     | I n1, I n2 -> B (n1 == n2)
26     | B b1, B b2 -> B (b1 == b2)
27     | _, _ -> E)
28
29 let good = Eq (Mult (Num 3, Num 4), Num 5)
30 let good = Eq (And (Bool true, Num 4), Num 5)

```

Aver definito un unico tipo, `expr`, mi permette di non aver duplicazione di codice che invece se avessi definito tipi per ogni combinazione avrei avuto.

Il GADT permette di spiegare l'invariante al compilatore andando a cercare di rappresentare dei predicanti sulla struttura. Ad esempio, un predicato su un file descriptor può essere «essere di scrittura» o «essere di lettura».

Si parametrizza su una variabile il tipo dell'espressione. Il parametro di tipo `'a` viene usato per codificare i predicati interessati, come «rappresentare un intero» o «rappresentare un booleano». Quindi le `int-expr` sono quelle ben formate che rappresentano interi. Vi possono essere insiemi vuoti, come le `string-expr`. Altro modo di vedere cosa è, una volta definito il grosso insieme delle `expr`, si ha prendendo i sotto insiemi anche non disgiunti.

Il tipo di ritorno deve avere lo stesso tipo del costruttore e stesso numero di parametri. I vincoli ad ogni costruttore vengono catturati dal pattern-matching. Se il costruttore ha delle variabili esistenziali, vengono generate dei nuovi tipi astratti locali che non escono dallo scope del ramo in cui si trovano.

Con

```

1  (* costruttore di tipo; 'a è totalmente ininfluente, si potrebbe
   mettere anche l'underscore _ *)
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  (* 'b non è quantificata da nessuna parte, quindi è come vista "forall 'b" *)
10 (* legare la 'b all'interno dell'expr sarebbe considerato un "exists 'b", che è
    un po' inutile in questo contesto *)

```

Avremo un codice che neppure compila per colpa della variabile `bad`.

Nei linguaggi con GADT o if-then-else di Kotlin il compilatore può riconoscere determinate forme del codice (es. il pattern match per i GADT, certe guardie negli if-then-else di Kotlin), ricordando comunque che il tipaggio è una proprietà indecidibile. Quando la forma viene

riconosciuta, il compilatore introduce equazioni fra tipi, cioè scopre se un tipo è uguale ad un altro tipo. Le equazioni vengono risolte potendo determinare il cambio del tipo di variabili, tipi di ritorno etc.

In Kotlin solo le variabili cambiano tipo (il tipo di ritorno no). L'unica variazione di tipo può essere lungo la gerarchia di ereditarietà oppure da tipo nullable a tipo non nullable: si distinguono i puntatori null da quelli che non possono esserlo, la deferenza viene fatta per forza mediante un if-then-else e vede se il ramo in cui si trova è quella in cui si ha già fatto il controllo se è null.

Dunque abbiamo:

```
1  (*
2  è un vincolo al tipo in cui si dice per un qualche 'a che decide il
3  compilatore, quindi non si può usare semplicemente:
4
5  let rec eval : 'a expr -> 'a
6
7
8  Btw, il tipo dell'output dipende dal tipo dell'argomento.
9  *)
10 let rec eval : type a. a expr -> a = function
11   (* Il tipo dell'input è          a expr
12      Il tipo da dare in output è    a
13   *)
14   | Num n ->
15     (* Sono nel caso Num. Num è dichiarato come Num: int -> int expr, quindi
16        n: int
17        l'input (Num n) ha tipo int expr
18        Quindi:
19        a expr = int expr
20        Per iniettività dei costruttori di GADTs:
21        a = int
22        Quindi:
23        il tipo di ritorno che era "a" ora è int.
24     *)
25     n
26   | Bool b -> b
27   | Mult (e1, e2) ->
28     (* Sono nel caso Mult. Mult è dichiarato come Mult: int expr * int expr ->
29        int expr, quindi
29        e1: int expr
30        e2: int expr
31        l'input (Mult (e1, e2)) ha tipo int expr
32        Quindi:
33        a expr = int expr
34        Per iniettività dei costruttori di GADTs:
35        a = int
```

```

36      Quindi:
37          il tipo di ritorno che era "a" ora è int.
38      Inoltre:
39          eval ha tipo 'a expr -> 'a
40      Quindi:
41          eval e1 ha tpo int
42          eval e2 ha tpo int
43      *)
44      eval e1 * eval e2
45 | And (e1, e2) -> eval e1 && eval e2
46 | Eq (e1, e2) ->
47     (* Sono nel caso Eq. Eq è dichiarato come Eq: forall 'b. 'b expr * 'b expr
    -> bool expr
48     Sia 'b un tipo ignoto ma fissato
49     e1: 'b expr
50     e2: 'b expr
51     l'input (Eq (e1, e2)) ha tipo bool expr
52     Quindi:
53     a expr = bool expr
54     Per iniettività dei costruttori di GADTs:
55     a = bool
56     Quindi:
57     il tipo di ritorno che era "a" ora è bool.
58     Inoltre:
59     eval ha tipo 'a expr -> 'a
60     Quindi:
61     eval e1 ha tpo 'b
62     eval e2 ha tpo 'b
63     *)
64     eval e1 == eval e2
65
66 let good = Eq (Mult (Num 3, Num 4), Num 5)
67 let good2 = Eq (And (Bool true, Bool true), Bool true)

```

Non solo si catturano le invarianti su una struttura dati ma inoltre l'output di `eval` dipende dal valore dell'argomento. Normalmente nei linguaggi di programmazione il valore a runtime non può determinare a compile-time il tipo della funzione; al massimo si fa runtime ma vi sono crash possibili. La `printf` invece lo fa: il valore del primo argomento determina il valore di ritorno della funzione dopo il primo argomento.

In questo caso, si tiene una versione di `printf` come

```

1 let rec to_string : type a. a spec -> a -> string =
2   ...

```



Qui la `to_string` prende la `spec` e il tipo da stampare `a`, ma il linguaggio non tiene a runtime il tipo, ma semplici sequenze di bit. Il primo argomento che passo è quello che usa per interpretare i bit (se intero, float, coppie, ...). Quando ho bisogno di ragionare in base al tipo


pago un leggero overhead perché si guarda a runtime. Un linguaggio che tiene sempre coppie runtime di `valore <-> tipo di un dato` è inefficiente rispetto a questa versione in OCaml.

9.1. Uguaglianza

Come detto, a runtime non vi sono informazioni di tipo. Quindi l'idea di uguaglianza è come può essere quella usata nel parser di un compilatore.

Qualcosa del tipo

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

 OCaml


non compila perché non mantenendo informazioni sui tipi runtime non capisce se `e2` è dello stesso tipo di `e1`, riconoscendola invece come di tipo `a expr`.

Bisogna quindi controllare se hanno lo stesso tipo.

Se il sistema di tipo è abbastanza espressivo certi tipi corrispondono a degli enunciati matematici e un dato di un certo tipo corrisponde alla dimostrazione di un certo tipo. Se questa cosa tipa allora vuol dire che la dimostrazione è corretta e dunque, se ho una funzione che prende una dimostrazione (un dato di un certo tipo dunque) vuol dire informare il compilatore che una certa cosa è vera.

Quindi definisco una funzione `same_type` che ritorna una dimostrazione che due certi valori hanno stesso tipo. La risposta alla domanda «quando due cose sono uguali» viene data, ad esempio, come: **l'uguaglianza è il più piccolo predicato riflessivo**, ovvero, voglio il più piccolo predicato che mi dica che `x = x` e nient'altro più. Dunque mi serve un solo tipo che mi dia la proprietà riflessiva.

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

 OCaml

Quindi l'unica possibilità che due tipi siano uguali è che la dimostrazione sia `Refl`.

- Dimostrazione di riflessività

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

 OCaml

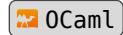
- Dimostrazione di simmetria

```
1 let symm : type a b. (a, b) eq -> (b, a) eq = function  
2   | Refl ->  
3     (* l'input Ref è tipato con forall 'c. ('c, 'c) eq  
4       con 'c tipo fissato.  
5       Quindi  
6         (a, b) eq = ('c, 'c) eq  
7       Quindi  
8         a = 'c = b  
9       Quindi output  
10        (b, a) eq ovvero ('c', 'c) eq *)  
11     Refl
```

 OCaml

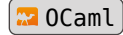
- Dimostrazione di transitività

```
1 let trans : type a b c. (a, b) eq -> (b, c) eq -> (a, c) eq =
2   fun x y ->
3     match x, y with
4       | Refl, Refl -> Refl
```



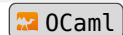
Quindi `same_type` adesso è una dimostrazione, non più un booleano:

```
1 (* None se non hanno stesso tipo ; Some Refl se hanno stesso tipo *)
2 let same_type : type a b. a expr -> b expr -> (a expr, b expr) eq option =
3   fun e1 e2 ->
4     match e1, e2 with
5       | Num _, Num _ -> Some Refl,
6       | Num _, Mult _ -> Some Refl,
7       | Bool _, Bool _ -> Some Refl,
8       (* etc. etc. *)
9       | _, _ -> None
```



Si deve procedere con il cast di tipo:

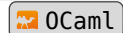
```
1 let cast : type a b. (a, b) eq -> a -> b = function
2   Refl ->
3     (* Poiché Refl: ('c, 'c) eq
4       si ha (a, b) eq = ('c, 'c) eq
5       quindi a = b = 'c
6       e il tipo di ritorno a -> b diventa 'c -> 'c *)
7   fun x -> x
```



L'operazione di **cast** non altera la memoria assolutamente, sennò si chiamerebbe promotion. Vuol dire cambiare il tipo di dato associato ad un dato ma non cambia runtime niente.

E dunque, alla fine, la funzione di ugaglianza è:

```
1 let equal : type a b. a expr -> b expr -> bool =
2   fun e1 e2 ->
3     match same_type e1 e2 with
4       | Some p -> e1 = cast (symm p) e2
5       | None -> false
```



Il compilatore non implementa davvero un pattern-matching nel caso di cast perché la funzione prende un input che non viene mai considerato. Il `cast` diviene dunque una funzione inline e la `same_type` ha la `Some Refl` che però è sempre `Refl` quindi la `option` può essere vista come booleano. Il pattern-matching su booleano è un if-then-else quindi l'ottimizzazione può essere vista come nel caso di una semplice `e1 = e2`.