Erlang

Tipi di dato

Come in tutti i linguaggi di programmazione, i dati si dividono in **predefiniti** o **definiti dall'utente**, ed **atomici** o **non atomici**.

In generale, ogni linguaggio di programmazione mette a disposizione alcuni tipi di dato predefiniti e, a seconda del linguaggio, offre la possibilità di definirne di nuovi.

I nuovi tipi possono essere semplici alias per tipi esistenti, oppure strutture che estendono tipi già presenti (non più semplici alias in questo caso).

Esistono anche tipi di dato più complessi, come i tipi algebrici, che descrivono le possibili forme che un dato può assumere.

Essendo Erlang un linguaggio dinamicamente tipato, non esistono dichiarazioni esplicite di nuovi tipi di dato. Non è presente un costrutto sintattico dedicato alla definizione di tipi. L'utente crea nuove tipologie di dato semplicemente utilizzando i valori in modo coerente.

Un esempio sono i valori booleani, che in Erlang sono rappresentati dagli atomi `true` e `false`.

Per quanto riguarda la distinzione tra dati atomici e non atomici, i tipi atomici sono quelli che non contengono altri dati al loro interno.

Un esempio di dato atomico è un numero, mentre un esempio di dato non atomico è una lista, che contiene al suo interno altri elementi.

Tra i dati **atomici**, in Erlang troviamo:

- **Numeri interi**, sui quali è possibile eseguire le comuni operazioni matematiche.
  È importante notare che gli operatori di confronto hanno alcune particolarità sintattiche: mentre l'operatore maggiore o uguale mantiene la forma standard (`>=`), l'operatore minore o uguale diventa `=<` per distinguerlo dalla forma di una freccia (essendo Erlang simile al Prolog, le frecce hanno un significato particolare).
  Altri operatori di confronto importanti sono l'uguaglianza stretta (`=:=`) e la disuguaglianza stretta (`=/=`).
- **Numeri in virgola mobile** (floating point), che quando combinati con numeri interi provocano la conversione implicita del risultato in floating point. È importante notare che il confronto tra un numero intero e uno floating point con `=:=` restituisce `false` (ad esempio, `5.0 =:= 5` è `false`), poiché rappresentano sequenze di bit differenti. Per un test di uguaglianza meno rigoroso, che considera equivalenti valori numericamente uguali indipendentemente dal tipo, si possono usare gli operatori `==` o `/=`.
- **PID** (Process IDentifier), ottenibili chiamando la funzione `self()`, che identificano univocamente i processi.
- **Reference**, ottenibili chiamando la funzione `make_ref()`. Una reference è un valore probabilisticamente unico, progettato per essere diverso da tutte le reference generate in precedenza. Non dovrebbe esistere un algoritmo in grado di prevedere la prossima reference che verrà generata.

- **Porte**. Nel modello ad attori di Erlang, quando è necessario interagire con entità esterne che non sono attori, è possibile avvolgerle in una specie di attore intermediario che permette di comunicare con esse utilizzando i meccanismi di invio e ricezione di messaggi tipici del linguaggio.
  Questi attori speciali, che fanno da wrapper a entità esterne, non possiedono tutte le caratteristiche degli attori normali. Ad esempio, non seguono il principio "Let it fail" di Erlang, che normalmente termina tutti gli attori associati a un attore che fallisce.
  Quando viene creato questo tipo di attore speciale, gli viene assegnata una porta anziché un PID.
- **Atomi**, che si scrivono normalmente con lettere minuscole. L'idea è che un programma utilizzi un numero limitato di atomi, che verranno rappresentati in memoria come sequenze di bit efficienti.
  È possibile racchiudere un atomo contenente spazi tra apici singoli (ad esempio, `'hello world'`). Da notare che `'ciao' =:= ciao` restituisce `true`, poiché sono considerati lo stesso atomo.
- **Caratteri**, anche se in realtà Erlang non ha un tipo carattere. Per invece rappresentare le stringhe viene utilizzata una lista di caratteri. Erlang controlla ogni lista se al suo interno ha valori che rientrano nei valori ASCII dei caratteri.

Passando ai dati **non atomici**, Erlang offre:

- **Tuple**. Si definiscono tra parentesi graffe, con elementi separati da virgole. Un esempio di tupla è `{4, {ciao, 2.0}, true}`. Esiste anche la tupla vuota `{}`, utilizzabile quando non si desidera restituire alcun valore significativo.
- **Liste**. Una lista può essere vuota (`[]`), oppure ha una *testa* (primo elemento) e una *coda* (una lista contenente tutti gli altri elementi).
  La testa può essere un valore qualsiasi, mentre la coda è a sua volta una lista.
  Un esempio di lista può essere scritto come `[ 2 | [ 3 | [ 4 | [ ] ] ] ]`, che rappresenta la struttura fondamentale. Per comodità è possibile utilizzare la sintassi semplificata `[2, 3, 4]`, ma concettualmente la lista è sempre composta da una testa e una coda.

  Essendo Erlang un linguaggio dinamicamente tipato, non ci sono garanzie che la coda sia effettivamente una lista. Quando la coda non è una lista, si parla di *lista impropria*, sulla quale non è possibile applicare le normali operazioni previste per le liste.

  Le operazioni predefinite sulle liste includono il calcolo della lunghezza, la concatenazione (`[2, 3] ++ [4, 5]` restituisce `[2, 3, 4, 5]`) e la sottrazione (`[2, 3, 4, 5] - [4, 2]` restituisce `[3, 5]`). La sottrazione segue una logica insiemistica, quindi in casi come `[2, 3, 4, 5] - [4, 2] - [4]`, il risultato sarà `[3, 4, 5]` e non `[3, 5]`.

  Un'altra potente caratteristica delle liste è la **list comprehension**. Un esempio:
  `[ {X, Y + 1} || X <- [1, 2, 3], {Y, _} <- [{4, 5}, {6, 7}] ].`
  Questa espressione restituisce:
  `[ {1, 5}, {1, 7}, {2, 5}, {2, 7}, {3, 5}, {3, 7} ].`

  Concettualmente, è come se ci fossero dei cicli for annidati che estraggono valori per X e Y. È anche possibile aggiungere filtri, ad esempio:

`[ {X, Y + 1} || X <- [1, 2, 3], {Y, _} <- [{4, 5}, {6, 7}], X + Y < 6 ].`
Questa espressione restituisce solamente `[ {1, 5} ]`, poiché solo la coppia $\{1, 4\}$ soddisfa la condizione $X + Y < 6$.

- **Bit strings**. Erlang offre la possibilità di accedere alla rappresentazione binaria di qualsiasi dato, permettendo di manipolare e analizzare sequenze di bit tramite pattern matching.
  Un esempio: `N = 16#7A5.` definisce un numero in base 16.
  Per accedere alla sua rappresentazione in bit, possiamo usare la sintassi:
  `« R:4, G:4, B:4 » = « N:12 ».`
  A questo punto, accedendo a R, G o B otterremo le rispettive sequenze di bit (nell'ordine: 7, 10 e 5).

  Questa funzionalità è particolarmente utile quando si lavora con pacchetti di rete, file binari o per interagire con dispositivi a basso livello, consentendo un controllo preciso sulle sequenze di bit.

Rimangono infine le **funzioni**, note anche come **chiusure**. Una caratteristica fondamentale dei linguaggi funzionali è che le funzioni sono oggetti di prima classe, manipolabili come qualsiasi altro valore. Una funzione può essere passata come argomento, restituita come risultato, inserita in strutture dati e così via.

In Erlang esistono diverse sintassi per definire funzioni. La prima forma ha la struttura: `nome_funzione(lista_argomenti) -> corpo ... end.` Questa sintassi può essere utilizzata nei file da compilare, ma non direttamente nella shell interattiva.

Una sintassi utilizzabile ovunque impiega la parola chiave **fun**, ad esempio:
`fun (lista_argomenti) -> ... end.` Questa è la sintassi per creare una funzione anonima.

È possibile definire funzioni annidate all'interno di altre funzioni. Le funzioni interne hanno accesso alle variabili definite nello scope più esterno (chiusura lessicale).

Un esempio: `G = fun (X) -> fun (Y) -> X + Y end end.`

Eseguendo `H = G(2).` e poi `H(3).`, otterremo il valore `5`. La variabile X, con valore 2, è stata "catturata" nella chiusura restituita da G.

Per dichiarare una funzione ricorsiva, si può usare la sintassi: `fun G(N) -> N * G(N) end.`
Il nome G è visibile solo all'interno della funzione stessa e non può essere richiamato dall'esterno.

In generale, le funzioni in Erlang utilizzano il pattern matching per selezionare diverse implementazioni in base all'input ricevuto, come accade anche con il costrutto *receive* per la gestione dei messaggi.

Tutti i linguaggi funzionali moderni permettono di definire funzioni per **casi**, consentendo di scrivere algoritmi in modo conciso e comprensibile, riducendo significativamente la quantità di codice.

Un esempio di funzione definita per casi può essere:
`fun ({N, 2}) -> N; ({ciao, N, M}) -> N + M; ([_, _, {X, Y}]) -> X + Y end.`
Qui il simbolo `_` indica un pattern che corrisponde a qualsiasi valore, il quale viene ignorato (non gli viene assegnato un nome).

Questa è una funzione definita tramite *pattern matching*. In base all'input fornito, verrà eseguito il ramo corrispondente al pattern che corrisponde. Se viene fornito un input che non corrisponde a nessuno dei pattern definiti, verrà sollevata un'eccezione.

È inoltre possibile utilizzare delle **guardie** per aggiungere condizioni aggiuntive. Dopo il pattern, attraverso la parola chiave `when`, si possono specificare condizioni che devono essere soddisfatte. Ad esempio:
`fun ({N, 2}) when N > 2 -> N; ({ciao, N, M}) -> N + M; ([_, _, {X, Y}]) -> X + Y end.`

Quando più pattern possono corrispondere all'input, l'ordine di valutazione è **sequenziale** dall'alto verso il basso.

Un aspetto importante delle guardie in Erlang è che il linguaggio si assicura che la loro valutazione non produca effetti collaterali, come l'invio di messaggi o la creazione di nuovi processi. Molti linguaggi moderni non effettuano questo controllo.

In Erlang, le guardie possono contenere solo combinazioni di funzioni predefinite chiamate **BIF** (Built-In Functions).
Questa restrizione rende il linguaggio delle guardie meno espressivo, il che può diventare problematico in casi complessi, come quando si desidera impedire l'attivazione di uno specifico caso in base a condizioni elaborate.

Rappresentazione dei dati a run-time

A livello di codice macchina non esistono i tipi di dato. Tutti i dati sono sequenze di bit, in genere multipli di byte o di word, e le operazioni aritmetico-logiche della CPU manipolano questi bit senza attribuire loro un significato semantico.

Il programmatore, quando scrive o legge un dato in memoria, lo interpreta in una determinata maniera. Di conseguenza, la stessa sequenza di bit nello stesso linguaggio di programmazione potrebbe rappresentare, a seconda del contesto, un carattere ASCII, un numero intero, un valore in virgola mobile, un puntatore, e a basso livello questa distinzione è completamente invisibile.

Sono le funzioni che associano un'interpretazione al dato e il programmatore deve utilizzarle in modo coerente. Se, ad esempio, ho scritto una word impostando i bit con l'intenzione di rappresentare un numero intero, ma successivamente la utilizzo come un puntatore, il risultato sarà inevitabilmente errato.

Per questo motivo è stato introdotto il concetto di **tipo**, che permette di controllare che il codice scritto dal programmatore si comporti correttamente.

Il sistema di tipi è un'analisi modulare statica effettuata a compile-time per garantire certe proprietà del codice a run-time, proprietà che normalmente sarebbero indecidibili.

Tipicamente, il sistema di tipi verifica che l'interpretazione del dato (dei bit) al momento della scrittura sia coerente con quella al momento della lettura. Molte funzioni implementate nei linguaggi di programmazione prevedono una specifica interpretazione del dato, e in quanto tali sono funzioni **monomorfe**. Monomorfo significa che queste funzioni operano correttamente solo se il dato in input ha esattamente una determinata interpretazione.

Mentre molte operazioni richiedono una specifica interpretazione del dato per avere senso (ad esempio, la concatenazione di stringhe richiede che i bit in input rappresentino stringhe, e la somma di interi richiede che i bit rappresentino numeri interi), esistono alcune operazioni che, dal punto di vista logico, possono essere implementate su qualunque tipo di dato perché non dipendono dall'interpretazione dei dati. Queste operazioni sono:

- Allocare
- Deallocare
- Spostare
- Copiare

Questo concetto si estende anche al passaggio di una funzione come input a un'altra funzione, poiché in questo caso si sta semplicemente copiando l'indirizzo della funzione nel punto in cui la funzione ricevente si aspetta di trovarlo, sia nei registri sia sullo stack.

Una funzione che non è legata a una particolare interpretazione o tipo diventa una funzione polimorfa, cioè può ricevere dati di qualunque tipo, interpretandoli liberamente pur mantenendo la propria semantica.

Un esempio di funzione polimorfa è il seguente (scritto in linguaggio OCaml):
`let swap (x, y) = (y, x);;` In questo caso `x` e `y` possono essere qualunque tipo di dato.

Questa forma di polimorfismo prende il nome di **polimorfismo uniforme**, in quanto è uniforme rispetto all'interpretazione dei dati. In altri linguaggi assume nomi diversi, come *Generics* in Java e *Template* in C++, dove però il polimorfismo deve essere dichiarato esplicitamente.

In Erlang non esiste un controllo a priori delle interpretazioni. In generale, i linguaggi di programmazione non tipati si presentano come linguaggi con polimorfismo uniforme implicito.

Come menzionato in precedenza, le operazioni che non richiedono la conoscenza del tipo di dato sono allocazione, deallocazione, spostamento e copia. Tuttavia, per implementare queste quattro operazioni è necessario conoscere la *lunghezza del dato*. La funzione `swap()` vista in precedenza cambia implementazione al variare della lunghezza dei dati da scambiare.

Per risolvere questo problema è stato introdotto un concetto di **tipi** (distinto da quello precedente), il cui unico scopo è misurare la dimensione del dato.

Quando si dichiara una funzione come swap in modo monomorfo, specificando esplicitamente i tipi (ad esempio, scambiare uno short int con un long int), il compilatore può generare il codice appropriato con le istruzioni assembly corrette per quei tipi specifici.

Ma cosa accade in caso di polimorfismo uniforme? In tal caso, il codice deve poter operare su qualunque tipo di dato, permettendo di scambiare, ad esempio, un intero con una stringa o una stringa con un valore in virgola mobile. Questo rappresenta una sfida implementativa: come può il codice generato gestire lo spostamento in memoria di quantità di dati di dimensioni differenti utilizzando le istruzioni assembly appropriate? Questo problema è comune a tutti i linguaggi con polimorfismo uniforme e a quelli non tipati, dove è possibile passare qualsiasi tipo di dato in qualsiasi contesto.

Come si implementano, quindi, funzioni polimorfe in grado di gestire dati di dimensioni differenti? Esistono tre tecniche principali.

Monomorfizzazione (C++, Rust, ...)
Questa tecnica impone i seguenti vincoli:

- Il linguaggio deve essere necessariamente tipato.
- Dato un programma, deve essere possibile calcolare un insieme *finito* di tipi sui quali ogni funzione opererà.

Un esempio di programma che non rispetta la seconda condizione può essere (in pseudo sintassi Erlang):

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

Questo esempio illustra la **ricorsione polimorfa** (polymorphic recursion), dove i tipi cambiano ad ogni chiamata ricorsiva, violando il secondo vincolo. Un sistema di tipi standard non permetterebbe di dichiarare questa funzione.

L'implementazione della monomorfizzazione consiste nel compilare la funzione polimorfa una volta per ciascuna combinazione di tipi su cui verrà utilizzata.

I vantaggi di questo approccio sono:

- Non vengono imposti vincoli sulla rappresentazione dei dati.
- Si possono applicare ottimizzazioni specifiche per ciascun tipo di dato.

Gli svantaggi sono:

- È limitato ai casi in cui i vincoli sono soddisfatti.
- Comporta tempi di compilazione maggiori.
- Aumenta la dimensione dell'eseguibile, non solo per la duplicazione del codice, ma anche perché ogni istanza della funzione riceve un nuovo nome composto dal nome originale più i tipi specifici, attraverso un processo chiamato **name mangling**.

Rappresentazione uniforme dei dati (Erlang, OCaml, Haskell, Java?, ...)
L'idea fondamentale di questo approccio è rappresentare tutti i dati con una word, la cui dimensione dipende dall'architettura del processore. Una word corrisponde alla dimensione necessaria per contenere un puntatore in memoria, garantendo così di poter memorizzare almeno un indirizzo.

I tipi di dati che occupano meno bit di una word sprecano spazio. Questi sono chiamati **value types** o **unboxed**.

I tipi di dati di dimensione maggiore di una word vengono allocati sullo *Heap* e sono rappresentati tramite un puntatore. Questi sono chiamati **reference types** o **boxed**.

I vantaggi di questo approccio sono:

- Tempi di compilazione ridotti e dimensione dell'eseguibile contenuta.

Gli svantaggi sono:

- Introduce indirezioni, con conseguente riduzione dell'efficienza dovuta ai continui accessi ai dati.

Un esempio è l'albero binario di ricerca utilizzato durante il corso:

```
Tree K V ::= {leaf, K, V} | {node, Tree K V, K, Tree K V}
```

con un dato di esempio `T = {node, {leaf, 4, true}, 5, {leaf, 6, false}}`

Questo dato non può essere contenuto in una singola word, quindi si accede a T tramite un puntatore allo Heap. Anche le strutture *leaf* non entrano in una singola word, quindi a loro volta punteranno ad altre sequenze di bit che rappresentano i valori/atomi in esse contenuti. Questa è la stessa logica utilizzata in C per implementare strutture dati come liste, alberi o costrutti simili.

"Alla C" (Rust in casi residuali)
In C, per dichiarare una funzione polimorfa, si accede ai dati in input/output tramite puntatori. Il puntatore viene dichiarato di tipo `void*` (ignorando l'informazione sul tipo di dato puntato). La funzione riceve in input coppie composte da un puntatore (`void*`) e dalla dimensione del dato (`size_t`).

Questo approccio non presenta particolari vantaggi. Gli svantaggi sono:

- A run-time è necessario preservare e passare esplicitamente la dimensione dei dati.
- Richiede intervento manuale da parte del programmatore (in C).
- È inefficiente, poiché il codice contiene cicli che operano sulle dimensioni dei dati.

Gestione della memoria

In Erlang è presente un **garbage collector automatico**, che si occupa di recuperare la memoria quando non viene più utilizzata.

Per fare ciò, il garbage collector esamina i dati in uso nel programma e deve determinare quali sequenze di bit rappresentano puntatori a aree di memoria utilizzate e quali no. La stessa sequenza di bit potrebbe essere interpretata come un puntatore o come un valore di altro tipo.

Si pone quindi il problema di distinguere se una word è stata concepita come un puntatore o come un altro tipo di dato.

Per risolvere questa problematica, in linguaggi come Erlang, OCaml, Haskell, e altri che non utilizzano la monomorfizzazione, si impiegano dei **tag**, utilizzando un bit della word per effettuare questa distinzione.

Quale bit viene scelto per fare questa distinzione? Vediamo alcune possibilità, con i relativi pro e contro:

- **Primo bit (bit più significativo)**: non è una soluzione praticabile, poiché impedirebbe di indirizzare il 50% (superiore o inferiore, in base al valore scelto tra `0` e `1`) della memoria.
- **Ultimo bit (bit meno significativo)**: anche in questo caso si perde l'accesso al 50% della memoria, ma a celle alterne. Questo causa una frammentazione della memoria, poiché si sprecano word quando un dato allocato sullo Heap termina in posizione pari, costringendo il dato successivo a iniziare due celle dopo.

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

  È comunque una soluzione più accettabile rispetto all'utilizzo del bit più significativo.

  Un aspetto importante da notare è che per indicare i puntatori si utilizza il valore `0` nell'ultimo bit, non il valore `1`. Questo perché, usando `1`, si andrebbe ad accedere a indirizzi non allineati in memoria, mentre gli indirizzi che terminano con `0` possono essere allineati.

  Un dato è considerato allineato quando il suo indirizzo di memoria è un multiplo della sua dimensione. Ad esempio, un dato di 4 byte è allineato quando il suo indirizzo è divisibile per 4.

  Un ulteriore problema riguarda l'implementazione delle operazioni aritmetico-logiche. I numeri vengono rappresentati all'interno del payload, quindi ad esempio `00000001` rappresenta il valore `0`, non `1`.
  Questo comporta difficoltà nell'implementazione delle operazioni, che richiedono controlli aggiuntivi per gestire questa caratteristica. L'implementazione delle operazioni aritmetico-logiche diventa così più costosa e complica l'interazione con altri linguaggi che non hanno questa problematica.

Se un linguaggio di programmazione, come Erlang:

- Ammette tipi di dato diversi.
- Permette confronti tra valori di tipi diversi, aspettandosi `false` come risultato.

Allora non è possibile riutilizzare le stesse sequenze di bit per rappresentare tipi di dato differenti.

Ad esempio, in OCaml/Haskell non vale la seconda assunzione, ed è quindi possibile riutilizzare le stesse sequenze.

Per distinguere i diversi tipi di dato, si possono adottare diverse strategie in base alla natura del dato (boxed o unboxed):

- Caso boxed:
  - Il dato è formato da word consecutive sullo Heap.

  - Si aggiunge una prima word che contiene un tag per distinguere i diversi tipi di dato.

    Nota: la word con il tag generalmente contiene anche la dimensione del dato sullo Heap, informazione utile per la garbage collection.

- Caso unboxed:
  - Si utilizzano i bit meno significativi per memorizzare il tag.
  - Si impiegano tag a lunghezza variabile, in base alla dimensione del payload da salvare. Nessun tag deve essere suffisso di un altro.

    Osservazione: per avere più spazio disponibile per il payload di un tipo specifico, gli si assegna un tag corto. Al contrario, per tipi con payload più piccoli si possono utilizzare tag più lunghi.

Rappresentazione dei payload

Come vengono rappresentate le sequenze di bit per i diversi tag?
Per quanto riguarda interi, floating point e altri tipi numerici, viene utilizzata essenzialmente la rappresentazione standard (sebbene con un numero inferiore di bit), permettendo così di lavorare con le istruzioni aritmetico-logiche, tenendo sempre in considerazione il bit di controllo che influenza la gestione dei dati.

Per i tipi di dati specifici (PID, Porte, ecc.) è il compilatore che determina come organizzare la sequenza di bit in modo appropriato.

Il discorso è diverso per la rappresentazione degli atomi.
Gli atomi sono un tipo di dato in cui l'unica proprietà rilevante è la loro identità. L'obiettivo principale è garantire che due atomi distinti vengano rappresentati da sequenze di bit diverse. Non possiedono altre proprietà: si vuole semplicemente associare a ciascun atomo una sequenza di bit univoca.

Nei linguaggi di programmazione che supportano gli atomi esistono due modalità di rappresentazione differenti:

- Linguaggi in cui l'insieme degli atomi è completamente noto a compile-time. L'unicità degli atomi rispetto all'insieme dei tipi è garantita (ogni atomo appartiene a uno e un solo tipo). Un esempio sono gli Algebraic Data Types in OCaml/Haskell e altri linguaggi funzionali.

  In questo caso, vengono assegnate rappresentazioni sequenziali agli atomi, eventualmente riutilizzandole per tipi diversi.

- Linguaggi in cui gli atomi possono comparire senza essere dichiarati preventivamente. Questo può avvenire:
  - Al momento del linking (Erlang, Prolog, OCaml). Un esempio tipico è quando due librerie A e B utilizzano ciascuna il proprio insieme di atomi. Al momento del linking è necessario effettuare l'unione dei due insiemi di atomi.

  - A run-time (Erlang). Ad esempio, quando un attore riceve messaggi da altri nodi potrebbe ricevere tipi di dati non conosciuti in precedenza.

In Erlang, Prolog, OCaml e linguaggi simili, atomi con lo stesso nome utilizzati da attori, moduli o librerie distinte **devono essere identificati** correttamente.

Si pone quindi una sfida durante la compilazione o l'interpretazione: associare a ogni atomo una sequenza di bit in modo da rendere possibile il linking e il message passing rispettando l'identificazione univoca degli atomi.

Alcune possibili soluzioni a questo problema sono:

- Utilizzare per ogni atomo il suo valore hash (come in OCaml).

  Questa soluzione presenta alcuni inconvenienti:

  - Possibili conflitti di hash, rilevabili in fase di compilazione o di linkaggio. In questi casi è necessario modificare il codice, ad esempio di una libreria.
  - Non è particolarmente efficiente. Per implementare un case/switch occorre ricorrere a una sequenza di if-then-else.

- Utilizzare sequenze consecutive di bit man mano che si incontrano nuovi atomi, mantenendo tabelle di associazione (nome atomo - sequenza di bit) che vengono scambiate tra i diversi nodi la prima volta che un messaggio contenente un nuovo atomo viene trasmesso (come in Erlang).

  Questa soluzione presenta alcuni svantaggi:

  - Richiede la gestione di diverse tabelle di traduzione.
  - Necessita di tradurre ogni messaggio scambiato fra nodi diversi per ri-mappare le sequenze di bit degli atomi.

  Quest'ultimo approccio è l'unico, tra i due presentati, che garantisce sempre la corretta funzionalità del sistema.

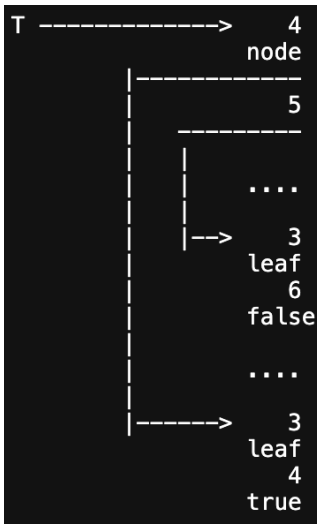
Pattern matching

Riprendiamo per chiarezza l'esempio già illustrato in precedenza:

```
Tree K V ::= {leaf, K, V} | {node, Tree K V, K, Tree K V}
```

con un'ipotetica struttura `T = {node, {leaf, 4, true}, 5, {leaf, 6, false}}`

`T` è un tipo di dato *boxed*, troppo esteso per essere contenuto in una singola word. Una sua possibile rappresentazione grafica può essere:



Per ogni tipo di dato boxed, il primo valore indica la dimensione della struttura dati.

Consideriamo ora una funzione che utilizza il pattern matching:

```
f({_, T1, 5, {leaf, K, V}}) -> {node, T1, 7, {leaf, K, V}}
```

Come viene compilato questo pattern?

Prendiamo come riferimento la seguente funzione:

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

dove `pattern` rappresenta il pattern della funzione,
`p` è il dato (equivalente a una word),
e `[| .,.  |]` è una funzione eseguita a tempo di compilazione (che genera codice).

Esaminiamo i possibili casi di questa funzione:

- `[| _, p |] =` : non produce alcun codice.
- `[| X, p |] = word X = p;`, viene quindi assegnata una word per il dato `p`.

  X è una nuova variabile. È importante notare che in Erlang, se si utilizza una variabile X già dichiarata o assegnata in un pattern, si intende il **valore** della variabile stessa.
- `[| k, p |] = if(p != k) return -1;`, dove `k` è una costante di un tipo atomico.
- ```
  [| P1 = P2, p |] =
      [| P1, p[1] |]
      [| P2, p[2] |]
  ```
- ```
  [| P1, ..., Pn, p |] =

      if(is_unboxed(p)) return -1;
      if(p[0].length != n) return -1;
      [| P1, p[1] |]
      ...
      [| Pn, p[n] |]
  ```

Riprendendo la struttura T precedente

```
f({_, T1, 5, {leaf, K, V}}) -> {node, T1, 7, {leaf, K, V}}
```

il codice compilato sarà:

```
if(is_unboxed(T)) return -1;
if(T[0].length != 4) return -1;
word T1 = T[2];
if(5 != T[3]) return -1;
if(is_unboxed(T[4])) return -1;
if(T[4][0].length != 3) return -1;
if(leaf != T[4][1]) return -1;
word K = T[4][2];
word V = T[4][3];
```

Il costo computazionale del pattern matching è **O(n)** sia in spazio che in tempo, dove n rappresenta la lunghezza del pattern.

Il costo computazionale dell'allocazione del risultato è **O(n)** sia in spazio che in tempo, dove n è la lunghezza dell'espressione utilizzata nell'output.

In pratica, questi costi sono **O(1)** a run-time poiché n **non** è un parametro dell'input a run-time.

Ogni variabile che ha catturato un tipo di dato boxed ha generato **sharing**, condividendo il dato attraverso il suo puntatore. Questo meccanismo non causa problemi se e solo se il linguaggio di programmazione non consente mutabilità dello Heap.

Nei linguaggi che supportano nativamente pattern matching profondi (Erlang/OCaml/Haskell), durante la compilazione delle dichiarazioni di funzioni **non** si compilano i pattern uno alla volta come spiegato finora.

Un esempio può essere:

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

Vogliamo ora compilare questo codice in modo efficiente.

La semantica del linguaggio stabilisce che viene utilizzato il primo pattern in ordine che corrisponde all'input. Tuttavia, questo approccio non è ottimale dal punto di vista computazionale.

Si consideri l'invocazione di `f({node, {node, .., 3, ..}, 5, {node, .., .., ..} })`

Dopo aver testato il primo pattern, se dimenticassimo tutto e passassimo a testare il secondo e poi il terzo, ripeteremmo operazioni già eseguite.

Non possiamo permutare i pattern per ottenere un albero di decisioni più efficiente, perché altrimenti non manterremmo l'ordine corretto di valutazione.

Per risolvere questo problema si utilizza un automa a stati finiti modificato, dove ogni stato dell'automa codifica l'informazione già acquisita sull'input.

Con questo approccio, la complessità diventa **sub-lineare** rispetto alla somma delle lunghezze dei pattern.

Ricorsione

Il principio fondamentale è che a run-time viene mantenuto uno **Stack** di **Record di Attivazione** per ogni chiamata di funzione. Per convenzione, lo Stack cresce verso il basso.
Un RA (record di attivazione) contiene:

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

Il costo in spazio e in tempo di una chiamata di funzione è **O(1)**.

Ipotizziamo uno scenario in cui si stia eseguendo una funzione `f()` e viene invocata una funzione `g()`. Il record di attivazione di `f()` deve contenere **tutta (e sola)** l'informazione necessaria per **completare** l'esecuzione di `f()` dopo che `g()` sarà terminata.
In questo caso, non tutte le informazioni contenute nel record di attivazione di `f()` sono necessarie dopo la chiamata di `g()`.

Un esempio di questo caso può essere:

```
f(int x, int y, int p) {
    int z, w;
    ...
    g();
    p = y * z;
    return p + p;
}
```

Il record di attivazione di `f` contiene:

```
*w
z
p
y
*x
REGISTRI
RETURN VALUE
RETURN ADDRESS
```

dove le variabili contrassegnate con * indicano variabili che non sono più necessarie dopo un certo punto dell'esecuzione, come si evince dal codice della funzione.

L'ottimizzazione consiste nel rimuovere le variabili non più necessarie dallo stack. Prima di invocare `g()`, lo stack pointer viene decrementato per liberare lo spazio occupato da `w`, ormai non più necessaria.

Inoltre, mediante un'analisi statica, è possibile determinare che sia più conveniente posizionare `x` subito dopo `w` nel record di attivazione, in modo da rilasciare anche lo spazio occupato da `x`.

Prolog, ad esempio, implementa sistematicamente queste ottimizzazioni.

Esaminiamo ora un caso limite di questo approccio, definito **Tail Calls** (chiamate di coda).

Definizione: una chiamata a `g()` all'interno di `f()` è considerata di coda se e solo se:

- L'unica istruzione eseguita **dopo** la chiamata a `g()` è un'istruzione di return.
- Il valore restituito dalla funzione `f()` è **esattamente** il valore restituito dalla funzione `g()`.

Quando una chiamata è identificata come chiamata di coda, viene compilata come:

```
pop(registri, variabili locali);
push(parametri attuali della g);
JUMP(codice della g);
```

La prima parte del record di attivazione della funzione `g()` coincide con la prima parte del record di attivazione della funzione `f()`.

Una chiamata di coda, quando è implementata la **Tail Call Optimization**, ha un costo di **O(1)** in tempo e **"O(0)"** in spazio (poiché viene riutilizzato il record di attivazione).

Un'ulteriore definizione, più diffusa ma meno precisa, è la seguente: una funzione `f()` si definisce **Tail Ricorsiva** se e solo se tutte le sue chiamate ricorsive sono chiamate di coda.

Eccezioni

Fino ad ora, uno **stack frame** è sempre stato equivalente ad un **record di attivazione**. Con l'introduzione delle eccezioni, uno stack frame può corrispondere a un record di attivazione oppure ad un **record try catch**.

All'interno di un record try catch troviamo un **puntatore al codice catch** ed un **tag "record try catch"**.

Un record di attivazione, invece, contiene i campi standard discussi in precedenza e un **tag "record di attivazione"**.

Cosa accade quando viene eseguita una throw(E)? Ecco una possibile implementazione in pseudo-codice:

```
throw(E) {
 finished = false;
 while(!finished){
    while(Stack[0].TAG != "record try-catch") Stack.pop();

    addr = Stack[0].indirizzo_codice_catch
    Stack.pop();
    case CALL addr(E) of
      {catched, V} -> finished = true; V
      not_catched -> nothing
 }
}
```

Nota: a basso livello, il codice dopo il catch analizza le varie eccezioni e:

- Se è in grado di gestirla, restituisce `{catched, E}` per un qualche valore `E`, oppure esegue un'altra `throw()` (in tal caso andrebbe modificato lo pseudo-codice precedente).
- Se non è in grado di gestirla, restituisce `not_catched`.

Osserviamo ora un esempio di codice Erlang:

```
cc(Bal) ->
 try
  receive
    print -> io:format("Il balance è ~p ~n", [Bal]), cc(Bal) ;
                                                     ^^^^^^^
                                                chiamata non di coda
    {put, N} -> cc(Bal+N) ;
    {tax, N} -> cc(Bal / N) ;
    {get, PID} -> PID ! Bal, cc(Bal) ;
    exit -> ok
  catch
   error:_ -> cc(Bal)              % gestisco la divisione per zero
              ^^^^^^
          chiamata di coda
  end
 end.
```

In questo esempio, l'obiettivo è gestire la divisione per zero che può verificarsi durante il pattern matching sulla clausola `tax`.

È necessario quindi aggiornare la definizione di chiamata di funzione di coda.

Una chiamata di funzione è considerata di coda quando:

- L'unica istruzione eseguita **dopo** la chiamata a `g()` è un'istruzione di return.
- Il valore restituito dalla funzione `f()` è **esattamente** il valore restituito dalla funzione `g()`.
- Non è contenuta all'interno della parte valutativa di un blocco `try ... catch` (ovvero non è protetta).

Vediamo ora come aggiornare il codice in una versione tail-ricorsiva:

```
cc(Bal) ->
  case
   try
    receive
      print -> io:format("Il balance è ~p ~n", [Bal]), {recur, Bal} ;
      {put, N} -> {recur, Bal+N} ;
      {tax, N} -> {recur, Bal / N} ;
      {get, PID} -> PID ! Bal, {recur, Bal} ;
      exit -> {result, ok}
   catch
    error:_ -> {recur, Bal}         % gestisco la divisione per zero
   end
  of
   {result, R} -> R ;
   {recur, Bal} -> cc(Bal) ;        % tail ricorsiva!
 end.
```

In questo modo attiviamo l'ottimizzazione della tail-ricorsione.

Una possibile riscrittura della struttura del codice (zucchero sintattico) può essere:

```
try
  E                                 % codice protetto
of
  p1 -> c1 ;                        % codice NON protetto
  ...                               % che fa match con il risultato di E
  pn -> cn ;
catch
 .... ->  ....                      % codice non protetto
```

Applicando questa struttura al codice precedente otteniamo:

```
cc(Bal) ->
 try
  receive
    print -> io:format("Il balance è ~p ~n", [Bal]), {recur, Bal} ;
    {put, N} -> {recur, Bal+N} ;
    {tax, N} -> {recur, Bal / N} ;
    {get, PID} -> PID ! Bal, {recur, Bal} ;
    exit -> {result, ok}
 of
  {result, R} -> R ;
  {recur, Bal} -> cc(Bal) ;          % tail ricorsiva!
 catch
  error:_ -> {recur, Bal}            % gestisco la divisione per zero
 end.
```

È importante notare che Erlang non è un linguaggio completamente puro.

Un **linguaggio di programmazione puro** è un linguaggio che aderisce rigorosamente a un paradigma di programmazione specifico senza deviazioni. Generalmente, il termine si riferisce ai linguaggi funzionali puri, che rispettano il modello della programmazione funzionale senza effetti collaterali. Le caratteristiche principali sono:

- **Assenza di effetti collaterali**: le funzioni non modificano lo stato globale o variabili esterne. Il risultato di una funzione dipende esclusivamente dai suoi input.
- **Trasparenza referenziale**: una funzione restituisce sempre lo stesso output per gli stessi input, indipendentemente dal contesto di esecuzione.
- **Immutabilità dei dati**: le variabili non possono essere modificate dopo la loro assegnazione. Si utilizzano strutture dati immutabili.
- **Lazy evaluation**: le espressioni vengono valutate solo quando necessario, migliorando efficienza e modularità.
- **Funzioni di prima classe e di ordine superiore**: le funzioni possono essere passate come parametri e restituite come valori.
- **Composizione funzionale**: le funzioni possono essere combinate per creare nuove funzioni senza necessità di strutture di controllo imperative.

Un esempio paradigmatico di linguaggio puro è *Haskell*.

Erlang permette l'accesso al file system tramite specifici costrutti. Naturalmente, una volta aperto un file, sarà necessario chiuderlo al termine del suo utilizzo. Un'eccezione potrebbe interrompere un'esecuzione corretta e causare problemi.

Un possibile codice per la gestione di un file può essere:

```
FD = open(...);
try
 ....                                % throw(exc2)
 close(FD)
catch
 exc1 -> close(FD), 4
 _:_ = R -> close(FD), throw(R)    % CODICE NON DEL TUTTO CORRETTO
end
```

Per risolvere questa problematica è necessario un nuovo costrutto.

Come in altri linguaggi di programmazione, esiste il costrutto `finally` (in Erlang denominato `after`).

Riprendendo la struttura precedente e aggiungendo il nuovo costrutto `after`:

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

Una possibile implementazione in pseudo-codice può essere:

```
case
 try
  {return, try E of F catch G}      % tutte le chiamate sono protette
 catch                              % e quindi non di coda
  Exc -> {exception, Exc}
 end
of
 {return, V} -> C, V ;
 {exception, Exc} -> C, throw(Exc)
end.
```

È importante sottolineare che quando si utilizza il costrutto `after`, si perde completamente l'ottimizzazione delle chiamate di coda.

In Erlang esistono tre tipologie di eccezioni:

- `throw()`: corrisponde al passaggio al *control operator*. Le eccezioni lanciate con `throw` non sono concepite come errori, ma semplicemente come meccanismo per il passaggio di controllo.
- `exit()`: interrompe l'esecuzione dell'attore. Viene utilizzata per implementare la filosofia di Erlang *Let it fail*. Queste eccezioni non dovrebbero essere catturate con `catch`.
- *Errori non risolvibili*: in questi casi non ha senso far ripartire l'attore, a causa di codice non implementato correttamente. In altri linguaggi equivalgono ad errori di compilazione. In queste situazioni viene fornito lo *stack trace* dell'errore.

Esistono inoltre altri tre costrutti considerati deprecati o di utilizzo limitato:

- `(catch throw(pippo))`: deprecato, utilizzato prima dell'introduzione del nuovo costrutto try catch descritto in precedenza. Non consente di distinguere il tipo di eccezione.
- `if`: forma semplificata di pattern matching, dove è possibile utilizzare esclusivamente *guardie*.
- `= (uguale)`: esegue un pattern matching tra la parte sinistra e quella destra. Quando i due pattern corrispondono si parla di *pattern irrefutabile*. Se viene utilizzato un pattern irrefutabile ma il match fallisce, viene generato un errore.

Esempio di Parallelizzazione (qsort)

Si vuole implementare una funzione che implementa l'algoritmo QuickSort (differente dal solito qsort studiato, questa è una versione funzionale non in-place).
Viene utilizzata la list comprehension per dividere la lista in due parti.
Viene quindi richiamata la funzione in maniera ricorsiva per riordinare le due liste ottenute.
Infine le liste vengono concatenate, ottenendo la lista ordinata.

```
qsort([]) -> [] ;
qsort([H|L]) ->
    L1 = [ X || X <- L, X =< H ],
    L2 = [ X || X <- L, X > H ],
    qsort(L1) ++ [ H | qsort(L2) ].
```

Questa versione ha un approccio sequenziale al problema, eseguendo tutta la computazione su un singolo attore.

Successivamente si può osservare una prima versione per parallelizzare il processo di sorting della lista.
Viene divisa la lista come in precedenza, per poi spawnare un nuovo attore che si occuperà di eseguire la funzione in maniera ricorsiva sulla parte destra della lista.
A sua volta l'attore spawnerà nuovi attori per completare l'esecuzione.
Alla fine vengono concatenate le diverse liste per ottenere la lista ordinata.

```
psort([]) -> [] ;
psort([H|L]) ->
    L1 = [ X || X <- L, X =< H ],
    L2 = [ X || X <- L, X > H ],
    SELF = self(),
    REF = make_ref(),
    spawn(fun() -> SELF ! {REF, psort(L2)} end),
    psort(L1) ++ [ H | receive {REF, SL2} -> SL2 end].
```

L'attore spawnato invia il messaggio contenete la lista ordinata al padre. Il padre resta in attesa della lista. Il REF viene usato per il pattern matching della msg queue.

Ora osserviamo una seconda versione del psort, che prende in input della funzione un numero N che limita il numero attori creati.
Il valore di N viene decrementato ad ogni chiamata ricorsiva, in modo che man mano che vengono spawnati nuovi attori il valore di N tenderà ad arrivare a 0, eseguendo quindi l'algoritmo sequenziale.

```
psort2(0, L) -> qsort(L) ;
psort2(_, L) when length(L) =< 10 -> qsort(L) ;
psort2(N, [H|L]) ->
    L1 = [ X || X <- L, X =< H ],
    L2 = [ X || X <- L, X > H ],
    SELF = self(),
    REF = make_ref(),
    spawn(fun() -> SELF ! {REF, psort2(N - 1, L2)} end),
    SL1 = psort2(N - 1, L1),
    SL2 = receive {REF, RES} -> RES end,
    SL1 ++ [ H | SL2].
```

Se N = 0, si effettua un qsort sequenziale per evitare overhead.
Se la lista è troppo piccola, si effettua un qsort sequenziale per evitare overhead.
L'attore spawnato invia il messaggio contenete la lista ordinata al padre. Il padre resta in attesa della lista.
Il REF viene usato per il pattern matching della msg queue.

Per concludere si può osservare una terza versione, con l'idea di base di fare eseguire dei job ad una pool di worker.
I worker ottengono i diversi job da uno scheduler, per poi inviare il risultato della propria esecuzione direttamente all'attore che lo ha richiesto allo scheduler.
Viene quindi concatenato il risultato con tutti i risultati ottenuti dai worker.

Nota: questo codice non funziona, va in deadlock. Il deadlock viene causato dal fatto che tutti i worker restano in attesa dei risultati da calcolare dagli altri worker, fino al non avere più worker disponibili. Questa problematica si può risolvere controllando il numero di worker disponibile prima di continuare. Se il numero è vicino al numero di worker disponibili, si ripiega sull'algoritmo sequenziale. (Per ulteriori dettagli vedere Virtuale anno precedente).

```
jsort([]) -> [] ;
jsort(L) when length(L) =< 10 -> qsort(L) ;
jsort([H|L]) ->
    L1 = [ X || X <- L, X =< H ],
    L2 = [ X || X <- L, X > H ],
    SELF = self(),
    REF = make_ref(),
    Job = fun () -> jsort(L2) end,
    scheduler ! {require, SELF, REF, Job},
    SL1 = jsort(L1),
    SL2 = receive {REF, RES} -> RES end,
    SL1 ++ [ H | SL2 ].
```

Se la lista è troppo piccola, si effettua un qsort sequenziale per evitare overhead.
L'atomo scheduler, grazie al costrutto register(), punta al PID dell'attore scheduler.

Implementazione dell'attore scheduler, il quale ha lo scopo di distribuire i diversi job ai vari

worker. Lo scheduler può ricevere nuovi job dagli attori (salvandoli nella propria lista di jobs interna) o ricevere richieste GET da parte dei diversi worker, i quali richiedono un nuovo job da eseguire.

```
scheduler(JOBS) ->
    receive
        {require, _, _, _} = JOB -> scheduler([JOB|JOBS]) ;
        {get, REF, WORKER} when JOBS =/= [] ->
            [JOB|L] = JOBS,
            WORKER ! {REF, JOB},
            scheduler(L)
    end.
```

Viene inserita una guardia che controlla se la lista di jobs è vuota. Se vuota non viene fatto pattern matching.
Viene divisa la lista JOBS in JOB (primo elemento) e L (il resto della lista) tramite pattern matching refutabile.
Al worker viene quindi inviato il primo elemento (JOB).
Lo scheduler viene eseguito ricorsivamente sul resto della lista (L).

Implementazione dell'attore worker, il quale ha lo scopo di eseguire jobs ricevuti dallo scheduler e di inviare il risultato di queste esecuzioni all'attore che ha effettuato la richiesta, senza passare dallo scheduler. Per comunicare con lo scheduler viene utilizzato il costrutto imperativo register(), il quale permette di utilizzare l'atomo "scheduler" per comunicare con l'attore designato all'esecuzione dello scheduler.

```
worker() ->
    REF = make_ref(),
    scheduler ! {get, REF, self()},
    receive
        {REF, {require, PID, REF2, F}} ->
            PID ! {REF2, F()},
            worker()
    end.
```

Il worker richiede nuovi job allo scheduler.
Il worker resta in attesa di nuovi job dallo scheduler. Quando il pattern fa match viene restituito il risultato all'attore che ha inviato inizialmente la richiesta di job allo scheduler.
Il worker richiama ricorsivamente se stesso per richiedere nuovi job allo scheduler.

Inoltre, sono presenti altre due funzioni: `benchmark()` e `main()`.

Benchmark è una funzione che permette di calcolare il tempo di esecuzione impiegato dalla funzione.

```
benchmark(F, L) ->
    T = [ timer:tc(?MODULE, F, L) || _ <- lists:seq(1, 10) ],
    lists:sum([ X || {X, _} <- T ]) / (1000 * length(T)).
```

Main è il main dell'attore. Viene generata una lista di numeri random utilizzata per testare i diversi approcci implementati. Vengono quindi stampati i tempi (in ms) di esecuzione per ogni funzione, andando a forzare una garbage collection dopo ogni esecuzione, per evitare possibili influenze sul tempo di esecuzione.

```
main() ->
    L = [ rand:uniform(10000) || _ <- lists:seq(1, 10000) ],
    io:format("Sequenziale: ~p~n", [ benchmark(qsort, [L]) ]),
    erlang:garbage_collect(),

    io:format("Psort: ~p~n", [ benchmark(psort, [L]) ]),
    erlang:garbage_collect(),

    io:format("Psort2(0): ~p~n", [ benchmark(psort2, [0, L]) ]),
    erlang:garbage_collect(),
    io:format("Psort2(1): ~p~n", [ benchmark(psort2, [1, L]) ]),
    erlang:garbage_collect(),
    io:format("Psort2(2): ~p~n", [ benchmark(psort2, [2, L]) ]),
    erlang:garbage_collect(),
    io:format("Psort2(3): ~p~n", [ benchmark(psort2, [3, L]) ]),
    erlang:garbage_collect(),
    io:format("Psort2(5): ~p~n", [ benchmark(psort2, [5, L]) ]),
    erlang:garbage_collect(),
    io:format("Psort2(8): ~p~n", [ benchmark(psort2, [8, L]) ]),
    erlang:garbage_collect(),
    io:format("Psort2(12): ~p~n", [ benchmark(psort2, [12, L]) ]),
    erlang:garbage_collect(),

    SCHED = spawn(?MODULE, scheduler, [[]]),
    register(scheduler, SCHED),
    [ spawn(?MODULE, worker, []) || _ <- lists:seq(1, 24) ],
    io:format("Jsort: ~p~n", [ benchmark(jsort, [L]) ]),
    unregister(scheduler),
    erlang:garbage_collect().
```

Viene utilizzato il **costrutto imperativo register()** per registrare un nome (atomo) rispetto al PID di un attore.
Questo permette di avere localmente una sorta di DNS per l'attore. Alla fine dell'esecuzione questa registrazione viene annullata tramite il costrutto **unregister()**.
Vengono inoltre spawnati 24 worker.
Nota: se si prova a registrare due volte lo stesso attore viene sollevata una exception.

Alcune considerazioni finali:
I risultati ci permettono di concludere che l'implementazione base di psort sia la versione più lenta, anche rispetto all'implementazione sequenziale. Questo è dovuto a due cause principali:

- Overhead varii, causati ad esempio da liste troppo piccole.
- Troppi attori rispetto ai core della propria CPU, causando un numero di context-switch elevato.

Bisogna quindi gestire al meglio la parallelizzazione, altrimenti si ottengono risultati peggiori rispetto all'approccio sequenziale.

La seconda versione di psort, che tiene in considerazione il numero di core del processore per gestire al meglio la parallelizzazione, ottiene risultati migliori rispetto al psort base. Va però usato il valore giusto di N, altrimenti i tempi vanno man mano a salire fino ad arrivare ai risultati del psort base.

Nel nostro esempio, vengono osservati i risultati con valori di N differenti, notando che con $N = 8$ si ottengono i risultati migliori mentre con $N = 12$ si peggiora il tempo di esecuzione precedentemente ottenuto con $N = 8$. Con $N = 0$, i risultati ottenuti sono simili a quelli dell'implementazione sequenziale (il costo dovrebbe essere uguale a quello sequenziale, ma a causa di overhead è leggermente superiore).

La terza versione, con pool di worker e scheduler, non è funzionante nella sua implementazione vista in precedenza. Maggiori dettagli sul perchè sono scritti nei commenti della funzione stessa. Teoricamente il tempo di esecuzione di questa versione dovrebbe essere simile a quello della psort con ottimizzazione N.

Esempio di Hot Code Swap

L'Hot Code Swap permette di modificare il codice di un attore senza avere tempi di down. Bisogna quindi modificare il codice dell'attore senza modificare il suo PID.

La Beam riesce a mantenere fino a due versioni compilate dello stesso modulo (quella attiva e quella alla quale vogliamo passare).

Un possibile esempio può essere il seguente:
Dato un attore che implementa una funzione `loop()` molto semplice:

```
loop(N, M) ->
   io:format("Versione 1: ~p, ~p~n", [N, M]),
   receive
       {add1, X} -> loop(N + X, M) ;
       {add2, X} -> loop(N, M + X) ;
       upgrade -> ?MODULE:new_loop(N, M)
   end.
```

ed una funzione `new_loop()`, la quale permette di effettuare l'hot code swap:

```
new_loop(N, M) -> loop(N, M).
```

La seguente funzione, insieme al pattern `upgrade` presente nella funzione `loop()` il quale richiama la funzione attraverso il costrutto `?MODULE`, permette di richiamare il nuovo codice compilato anche successivamente dalla Beam (l'ultima versione del modulo).

Per fare il passaggio tra le due versioni, è il programmatore che attivamente richiama il pattern `upgrade` che effettuerà il passaggio tra le due versioni.

Vogliamo quindi ora modificare il codice di questa funzione, senza però avere tempi di down. Una possibile versione intermedia (1.5), la quale riceve i dati anche in maniera diversa oltre a quello presente nella prima versione, può essere:

```
loop({N, M}) ->
   io:format("Versione 1.5: ~p, ~p~n", [N, M]),
   receive
       {add1, X} -> loop({N + X, M}) ;
       {add2, X} -> loop({N, M + X}) ;
       {add, 1, X} -> loop({N + X, M}) ;
       {add, 2, X} -> loop({N, M + X}) ;
       upgrade -> ?MODULE:new_loop({N, M})
   end.
```

con conseguenti funzioni `new_loop()`:

```
new_loop(N, M) -> loop({N, M}).         % Invocata dal vecchio codice

new_loop({N, M}) -> loop({N, M}).       % Invocata dal nuovo codice
```

Nota: in questo caso sono presenti due funzioni `new_loop()`, una che viene invocata dal modulo da sostituire, l'altra che verrà utilizzata per passare alla versione successiva.

Una versione 2, la quale esegue completamente la transizione eliminando la gestione dei pattern presenti nella prima versione, può essere:

```
loop({N, M}) ->
    io:format("Versione 2: ~p, ~p~n", [N, M]),
    receive
        {add, 1, X} -> loop({N + X, M}) ;
        {add, 2, X} -> loop({N, M + X}) ;
        upgrade -> ?MODULE:new_loop({N, M})
    end.
```

con conseguente funzione `new_loop()`:

```
new_loop({N, M}) -> loop({N, M}).
```

In questo modo si è passati da una prima versione che gestiva un tipo di dato ad una seconda versione che ne gestisce uno completamente diverso.

Esempio di Link e Monitoring

In Erlang sono presenti due primitive per sfruttare al massimo i meccanismi di collaborazione tra attori. Queste due primitive sono **link** e **monitor**.

Vediamo ora alcuni esempi di codice per la primitiva **link**:

```
sleep(N) -> receive after N -> ok end.

make_link(SHELL, 0) ->
    io:format("0: send to SHELL ~p~n", [SHELL]),
    % _ = 1 / 0,   % Volendo far sollevare exception terminando l'attore
    SHELL ! self(),
    sleep(200000),
    io:format("0: Termino ~n"),
    ok;

make_link(SHELL, N) ->
    process_flag(trap_exit, true),
    spawn_link(?MODULE, make_link, [SHELL, N - 1]),
    sleep(6000),
    receive
        MSG -> io:format("~p: Ho ricevuto ~p~n", [N, MSG])
    after 1000 -> ok
    end,
    io:format("~p: Termino ~n", [N]).

% Creo una catena di 10 attori ognuno linkato al precedente
test() ->
    spawn(?MODULE, make_link, [self(), 10]),

    receive
        PID ->
            io:format("Shell ha ricevuto da ~p~n", [PID]),
            exit(PID, kill),
            ok
    end.
```

Quando viene utilizzata la primitiva link vengono collegati due attori. In caso di terminazione di un attore, tutti gli attori linkati verranno terminati a loro volta.

A livello di Beam i due attori si inviano messaggi nascosti per comunicare tra loro in caso di terminazione.

La funzione `sleep()` permette di mandare l'attore in uno stato di timeout che dura N ms.

Attraverso la funzione `exit()` possiamo terminare un attore tramite il suo PID fornendo una *reason*. Sono presenti diverse reason, come `normal`, `killed` e `kill`, ognuna con livelli di importanza differenti.

Il costrutto `link(PID)` permette quindi di linkare due attori. La link è

- **Bidirezionale**: se viene terminato uno dei due attori linkato, anche l'altro verrà terminato.
- **Idempotente**: eseguire due volte la stessa operazione non cambia nulla.
- **Transitiva**.

E' inoltre presente una primitiva opposta `unlink(PID)` per spezzare il collegamento tra i due attori.

All'interno dell'esempio di codice non è presente `link(PID)` ma `spawn_link()`, questo perché permette di evitare race condition di alcun tipo (interruzione del codice tra la spawn ed il link degli attori).

Nel codice è inoltre presente il comando **imperativo** `process_flag(trap_exit, true)`. Questo comando permette di evitare il meccanismo di terminazione a catena.
In realtà se tramite `exit()` inviamo la reason `kill`, questo controllo viene superato e terminato l'attore. In questo caso il primo attore che riceve `kill` viene terminato, ma agli attori linkati verrà inviata la reason `normal` (bloccabile dal `process_flag()`).

Un'altro possibile fallimento dell'attore può avvenire a causa di *exception* (divisione per 0 nel codice). In questo caso l'attore solleva l'eccezione e viene terminato, facendo terminare a catena gli attori linkati.

Un possibile esempio di uso della link può essere: dati due attori, il corrente ed il PID, il corrente vuole richiedere un servizio a PID dal quale vuole una risposta.

```
sync_call(REQUEST, PID, TIMEOUT) ->
    REF = make_ref(),
    link(PID),
    PID ! {REF, REQUEST},

    RES =
        receive
            {REF, ANSWER} -> ANSWER
        after TIMEOUT -> exit(timeout)
        end,

    unlink(PID),
    RES.
```

`after TIMEOUT -> exit(timeout)` permette di terminare i due attori dopo un determinato tempo di attesa.

Oltre a link, esiste una seconda primitiva chiamata **monitor**. A differenza del link, attraverso il monitor abbiamo un collegamento più debole tra attori. Può essere utile per osservare il comportamento di un altro attore. Il monitor è:

- **Unidirezionale**: un attore monitora un altro attore e viene informato se l'altro attore termina.
- **Non idempotente**: se monitoro due volte devo anche de-monitorare due volte.
- **Non transitiva**.

Due esempi di utilizzo di monitor possono essere:

- Un browser che monitora (non linka) un server web.
- Vengono utilizzate due librerie `B` e `C`. Sia `B` che `C` usano una libreria `A`. Tutte le librerie sono implementate come attori. In questo caso se viene terminato `B` verrebbe terminato indirettamente anche `A` terminando a catena anche `C`. In questo caso link non è adatto e va utilizzato monitor.

Come per link, è presente anche il comando `spawn_monitor()` per evitari possibili race condition.

Volendo confrontare link e monitor (unlink e demonitor):

- Link e monitor prendono il PID dell'altro attore.
- Link non ritorna nulla, monitor ritorna un handle.
- Unlink prende il PID, demonitor prende l'hanlde ed il PID.
- `trap_link` (il meccanismo di `process_flag()`) funziona solo con il link.
- I messaggi legati al monitoring sono sempre in user space e non vengono propagati.

Inoltre, all'interno di sistemi distribuiti è possibile fare il monitoring di un Nodo.

Esempio di Distribuzione in Erlang

Come detto in precedenza, è possibile lanciare più Beam sulla stessa macchina. Queste Beam vengono identificate come diversi nodi sulla stessa macchina.

Per poter comunicare all'esterno bisogna lanciare `erl` con la flag `-sname nome` (short name, in alcuni casi conviene usare soltanto `-name`).

Tramite il comando `nodes()` si possono osservare tutti i nodi collegati con la propria shell. Inizialmente questo comando restituisce una lista vuota.

Per collegare due nodi si usa il comando `net_adm:ping(nome_attore@nome_dispositivo)`, il quale risponde con `pong` in caso di collegamento andato a buon fine, `peng` viceversa.
Il collegamento tra due nodi è transitivo.

Un ulteriore comando utile è `flush()`, il quale permette di stampare sulla shell tutti i messaggi presente nella message queue.

Vediamo ora un esempio di scenario di comunicazione tra due nodi. Su un nodo registriamo, tramite il costrutto imperativo `register()`, il PID del nodo chiamandolo `shell`.

Il secondo nodo potrà quindi comunicare con l'altro nodo utilizzando il comando
`{shell, nome_altro_attore@nome_altro_dispositivo} ! {ciao, self()}`

L'altro attore, eseguendo il comando `flush()`, osserverà il messaggio inviato.

Nota: il PID visibile su un attore è diverso dal PID visibile eseguendo il comando `self()`.

E' possibile inoltre effettuare spawn di attori su un altro nodo.

Per quanto riguarda la sicurezza tra gli attori, Erlang non garantisce alcun tipo di sicurezza built-in. I messaggi vengono inviati in chiaro tra gli attori.
E' presente una chiave che viene scambiata durante la comunicazione tra diversi cluster (presente nel file .erlang.cookie), ma anche la chiave viene inviata in chiaro (quindi facilmente intercettabile).

Esempio di Migrazione di codice tra due attori

Erlang di base non permette di migrare attori tra due nodi, ma è possibile tramite alcuni pattern spostare codice tra due nodi. Questo può essere utile per bilanciare il carico tra diversi nodi o per la prossimità geografica rispetto ai dati.

Un possibile esempio di migrazione può essere:

```
migrate(NODE, F) ->
 PID = spawn(NODE, F),
 Forward =
  fun Forward() ->
   receive
    {comeback, F} -> F() ;
    Msg -> PID ! Msg, Forward()
   end
  end,
 Forward().

server(Dati) ->
 receive
   msg1 -> server(Dati + 1) ;
   {migrate, NODE} ->
     migrate(NODE, fun () -> server(Dati) end)
 end.
```

Lo scopo di questo codice è quello di trasferire (migrare) il codice da un nodo all'altro.

Viene quindi spawnato un nuovo attore, al quale viene passato il codice che si vuole trasferire.

Per far funzionare questo sistema, l'attore che trasferisce il codice sul nuovo attore deve agire da man-in-the-middle (tramite la funzione ricorsiva `Forward()`).

Viene inoltre inserita la possibilità di effettuare il comeback all'attore precedente.

Quando trasferiamo funzioni tra due attori, il Byte Code inerente a quella funzione deve essere trasferito da una Beam all'altra.