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 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.