

Domande

Domanda

Definizione di paradigma ad attori

Risposta

Il **paradigma ad attori** è una tecnica fondamentale per sviluppare **programmi distribuiti**. Questi programmi eseguono **concorrentemente con thread** che possono trovarsi distribuiti in rete su diverse macchine. Il paradigma ad attori è considerato il **paradigma più efficace mai sviluppato** per implementare sistemi distribuiti.

Al centro di questo approccio vi è il concetto stesso di **attore**. Un attore è un'unità di computazione che si distingue per tre componenti principali:

- **PID (Process Identifier)**: Un identificatore univoco per l'attore. Funge da **nome logico** necessario per poter comunicare con l'attore, nascondendone la sua posizione fisica all'interno del sistema. Solo chi conosce il nome logico può comunicare con esso. I PID identificano i processi in modo univoco.
- **Mailbox**: Una coda dedicata alla ricezione dei messaggi. Ogni messaggio inviato a un attore viene depositato nella sua mailbox.
- **Behaviour**: Una funzione che gestisce l'elaborazione dei messaggi ricevuti. Questa funzione trasforma i messaggi in una lista di azioni da eseguire e definisce il **nuovo comportamento** dell'attore. Il comportamento di un attore può cambiare completamente dopo la ricezione e l'elaborazione di un messaggio. Le azioni possono includere computazioni interne. Il behavior deve specificare le azioni da compiere in risposta a specifici messaggi e, dopo aver ricevuto un messaggio, deve acquisire un nuovo behavior.

L'esecuzione nel paradigma ad attori è differente dalla programmazione multithreading tradizionale. Gli attori non sono costantemente impegnati in calcoli; quando ricevono un messaggio, eseguono una breve serie di operazioni, possono generare altri messaggi, e poi **tornano in stato di attesa**.

Questo modello è **ideale per configurazioni con un numero elevatissimo di attori**, anche se solo una piccola parte è attiva contemporaneamente. Questo si contrappone al multithreading, dove un numero eccessivo di thread può rallentare significativamente il sistema a causa dei costi associati a ciascun thread.

La comunicazione tra attori avviene tramite lo **scambio di messaggi asincrono**. L'ordine di ricezione dei messaggi in uno scenario distribuito potrebbe non essere sequenziale.

Oggi, quando si parla di programmazione ad attori, si fa riferimento essenzialmente a **Erlang**.

In Erlang, il principio è che **"tutto è un attore"**, inclusa la shell interattiva.

Domanda

Differenze fra il paradigma ad attori e quello ad agenti

Risposta

Il **paradigma ad attori** è una tecnica fondamentale per lo sviluppo di **programmi distribuiti**. Questi programmi si caratterizzano per l'esecuzione **concorrente** con thread che possono essere distribuiti in rete su macchine diverse. Viene definito come il **paradigma più efficace mai sviluppato** per l'implementazione di sistemi distribuiti. Quando oggi si parla di programmazione ad attori, ci si riferisce essenzialmente al modello implementato da Erlang.

Al centro del paradigma vi è il concetto di **attore**. Un attore è un'unità di computazione caratterizzata da tre componenti principali:

- **PID (Process Identifier)**: Un identificatore univoco per l'attore. Funge da **nome logico** necessario per poter comunicare con l'attore, nascondendone la sua posizione fisica all'interno del sistema. Solo chi conosce il nome logico può comunicare con esso. I PID identificano i processi in modo univoco.
- **Mailbox**: Una coda dedicata alla ricezione dei messaggi. Ogni messaggio inviato a un attore viene depositato nella sua mailbox.
- **Behaviour**: Una funzione che gestisce l'elaborazione dei messaggi ricevuti. Questa funzione trasforma i messaggi in una lista di azioni da eseguire e definisce il **nuovo comportamento** dell'attore. Il comportamento di un attore può cambiare completamente dopo la ricezione e l'elaborazione di un messaggio. Le azioni possono includere computazioni interne. Il behavior deve specificare le azioni da compiere in risposta a specifici messaggi e, dopo aver ricevuto un messaggio, deve acquisire un nuovo behavior.

L'esecuzione nel paradigma ad attori si basa sullo **scambio di messaggi asincrono**. Non esiste sincronizzazione diretta tra gli attori. Quando si invia un messaggio, non si sa se o quando verrà ricevuto.

Un attore corrisponde a un **singolo thread** di computazione. Non possono esistere thread multipli all'interno di un singolo attore. Gli attori **non condividono risorse**, né stato né memoria, anche se eseguiti sulla stessa macchina. Collaborano esclusivamente tramite lo scambio di messaggi.

La programmazione ad attori è un caso particolare di programmazione reattiva o event-driven. È progettata per scenari con un **numero massiccio di attori** (≥ 100.000), come nell'esempio di WhatsApp. Non tutti gli attori sono attivi contemporaneamente; un attore, dopo aver ricevuto ed elaborato un messaggio ed eseguito le azioni conseguenti (come inviare altri messaggi), **torna in stato di attesa**. Questo la rende ideale per configurazioni

con un numero elevatissimo di attori, dove solo una piccola parte è attiva, a differenza del multithreading tradizionale dove ogni thread è costoso.

Un aspetto fondamentale del paradigma è la tolleranza ai guasti tramite il principio "**Let it fail!**". Questo principio prevede che, in presenza di errori, un processo (attore) e tutti quelli strettamente connessi ad esso vengano terminati. Un **supervisore** si occupa di riavviare i processi interrotti. L'idea è quella di strutturare la computazione gerarchicamente. Un attore padre può supervisionare i suoi figli e riavviarli se falliscono. Nei linguaggi funzionali privi di mutazione come Erlang, il padre può ricreare rapidamente (in tempo $O(1)$) lo stato precedente dei figli perché il suo stato non è stato alterato dal fallimento, possedendo ancora lo stato che aveva al momento della creazione dei figli.

In Erlang, "**tutto è un attore**", inclusa la shell interattiva. Erlang è un linguaggio funzionale, dinamicamente tipato (non tipato) e con variabili immutabili. Erlang non nacque originariamente come linguaggio per attori, ma riscoprì questo paradigma successivamente. La Beam (la macchina virtuale di Erlang) gestisce l'esecuzione degli attori. Gli attori possono essere eseguiti in modo trasparente sullo stesso nodo, macchina virtuale, core o CPU. Erlang permette anche di sviluppare sistemi distribuiti con la possibilità di aggiornare il codice durante l'esecuzione (**Hot Code Swap**) senza interrompere il programma. Esistono anche meccanismi di **Linking** che collegano attori in modo che il fallimento di uno termini anche gli altri collegati, e la possibilità di comunicare tra **nodi** distribuiti. La sicurezza tra attori in Erlang base non è garantita, e i messaggi vengono inviati in chiaro. È possibile attuare la **migrazione di codice** tra attori su nodi diversi tramite specifici pattern, utile per bilanciare carico o prossimità geografica.

Mentre gli attori sono un modello relativamente semplice e ben definito di concorrenza basato sulla ricezione e elaborazione di messaggi, il termine "agente" è più ampio e spesso implica caratteristiche aggiuntive rispetto a un semplice attore.

Alcune differenze comuni tra il **paradigma ad attori** e il **paradigma ad agenti** possono essere:

- Gli **attori** sono tipicamente *reattivi*. Attendono la ricezione di un messaggio nella loro mailbox e poi eseguono il loro comportamento in risposta a quel messaggio. Gli **agenti**, pur potendo anch'essi essere reattivi, sono spesso definiti anche come *proattivi* o *autonomi*. Possono prendere decisioni proprie, perseguire obiettivi senza essere esplicitamente istruiti da un messaggio specifico, e iniziare azioni di propria iniziativa basandosi sul loro stato interno o sulla percezione dell'ambiente.
- Gli **agenti** spesso percepiscono ed agiscono su un *ambiente* condiviso (non necessariamente memoria condivisa in senso tradizionale, ma un contesto in cui gli agenti coesistono e interagiscono). Possono avere "sensori" per percepire l'ambiente e "attuatori" per modificarlo. Gli **attori**, invece, interagiscono principalmente solo con altri attori inviando e ricevendo messaggi.
- **Comunicazione:** Entrambi utilizzano lo scambio di messaggi. Tuttavia, la comunicazione tra **agenti** può essere più sofisticata e basata su "speech acts" (atti

linguistici), dove il messaggio non è solo dato, ma ha anche un'intenzione associata (es. chiedere, offrire, rifiutare).

In sintesi, l'Actor model è un modello fondamentale per la programmazione concorrente e distribuita, basato su unità isolate che comunicano tramite messaggi.

Il concetto di Agente è più ampio e aggiunge spesso caratteristiche come autonomia, proattività, interazione con l'ambiente e, a volte, mobilità o intelligenza, posizionandosi a un livello di astrazione superiore.

Domanda

Punti di forza e di debolezza del paradigma ad attori

Risposta

La programmazione ad attori è una tecnica fondamentale per lo sviluppo di **programmi distribuiti**, caratterizzati dall'esecuzione concorrente con thread potenzialmente distribuiti in rete su macchine diverse. Viene considerato il **paradigma più efficace mai sviluppato** per l'implementazione di sistemi distribuiti. Linguaggi come Erlang ed Elixir aderiscono completamente a questo modello.

Al centro del paradigma vi sono gli **attori**, unità di computazione isolate. Ogni attore possiede un **PID** (identificatore univoco e nome logico per la comunicazione), una **Mailbox** (coda per i messaggi in arrivo) e un **Behaviour** (una funzione che processa i messaggi e definisce il nuovo stato/comportamento dell'attore). Gli attori **non condividono risorse** (stato o memoria) e collaborano esclusivamente tramite lo **scambio di messaggi asincrono**.

Punti di Forza del Paradigma ad Attori:

1. **Efficacia per Sistemi Distribuiti e Concorrenti:** È considerato il paradigma più efficace per implementare sistemi distribuiti, gestendo l'esecuzione concorrente su thread che possono trovarsi su macchine diverse.
2. **Modello Concettuale Semplice e Chiaro:** La definizione degli attori tramite PID, Mailbox e Behaviour fornisce un modello chiaro e comprensibile per la costruzione di sistemi concorrenti.
3. **Isolamento dello Stato:** Gli attori non condividono memoria o stato. Questo isolamento evita intrinsecamente molti problemi classici della programmazione concorrente basata su memoria condivisa, come le race condition.
4. **Scalabilità Elevata:** Il paradigma è progettato per gestire un **numero massiccio di attori** (≥ 100.000). Poiché gli attori sono inattivi quando non ricevono messaggi, è efficiente in scenari con alta concorrenza dove solo una frazione degli attori è attiva in un dato momento. WhatsApp viene citato come esempio di sistema che beneficia di questa scalabilità.
5. **Tolleranza ai Guasti ("Let it fail!"):** Un principio fondamentale è la gestione degli errori tramite la terminazione dei processi (attori) falliti e di quelli strettamente collegati, con

riavvio affidato a processi supervisor. Questa strategia è particolarmente efficace in linguaggi funzionali come Erlang, dove lo stato del supervisore non viene modificato dal fallimento del figlio, permettendo una ricreazione rapida dello stato del figlio. La computazione viene strutturata gerarchicamente per facilitare questa supervisione.

6. **Comunicazione Asincrona:** L'unica forma di comunicazione è lo scambio asincrono di messaggi. Questo favorisce il disaccoppiamento tra gli attori.
7. **Pattern Matching:** Nei linguaggi che supportano nativamente il paradigma (come Erlang), il pattern matching nel costrutto `receive-end` permette una gestione efficiente e dichiarativa dei messaggi in arrivo. Questa tecnica si estende anche alla definizione generale delle funzioni.
8. **Integrazione con Programmazione Funzionale (in Erlang):** L'implementazione del paradigma ad attori in un linguaggio funzionale puro come Erlang beneficia di caratteristiche come l'immutabilità dei dati, che è cruciale per l'efficienza del meccanismo "Let it fail!". L'ottimizzazione delle chiamate di coda (Tail Call Optimization) rende la ricorsione (usata per definire il behavior degli attori) efficiente come un ciclo.

Punti di Debolezza del Paradigma ad Attori:

1. **Problemi con Approcci Ibridi:** L'utilizzo di librerie per implementare il paradigma ad attori in linguaggi mainstream (come Akka in Java/Scala) non è sempre consigliabile e può essere problematico. Questo perché combinare costrutti da paradigmi diversi tende a indebolire le garanzie offerte dai singoli paradigmi, come la garanzia di immutabilità dei dati.
2. **Gestione dell'Ordine dei Messaggi:** Lo scambio di messaggi asincrono non garantisce la ricezione sequenziale dei messaggi in uno scenario distribuito. Se l'ordine di elaborazione dei messaggi è critico, deve essere gestito esplicitamente a livello di comportamento dell'attore.
3. **Overhead e Necessità di Gestione Attenta:** La creazione e gestione di un numero molto elevato di attori può introdurre overhead, specialmente per attori con compiti molto piccoli o se il numero di attori attivi supera la capacità dei core CPU, portando a elevati costi di context-switch. Una parallelizzazione non gestita correttamente può risultare in prestazioni peggiori rispetto a un approccio sequenziale.
4. **Interazione con Entità Esterne Limitate:** Le interazioni con entità esterne al modello ad attori, come driver o risorse non-attore, devono avvenire tramite attori speciali identificati da "Porte" anziché PID. Queste Porte non seguono completamente i principi degli attori normali, come il "Let it fail!".
5. **Sicurezza Base dei Messaggi:** Nei linguaggi base che implementano il paradigma (come Erlang base), i messaggi tra attori vengono inviati in chiaro.
6. **La Tolleranza ai Guasti non è una Panacea:** Il principio "Let it fail!" implica la terminazione degli attori falliti e collegati. Se i problemi persistono dopo ripetuti tentativi di riavvio, anche i supervisor superiori nella gerarchia falliranno. Non è una garanzia contro *ogni* tipo di fallimento e richiede un'attenta progettazione della gerarchia di supervisione.

Domanda

Erlang: genesi, storia, obiettivi, punti di forza

Risposta

Erlang è un linguaggio di programmazione che è stato creato nel 1976. Inizialmente, non nacque con il paradigma ad attori in mente, ma lo scoprì successivamente alla sua creazione. Oggi, quando si parla di programmazione ad attori, si fa riferimento essenzialmente a Erlang. Erlang ha una sintassi che condivide elementi con il linguaggio di programmazione logica Prolog. È un linguaggio funzionale e, cosa piuttosto rara tra i linguaggi funzionali moderni, è non tipato (dinamicamente tipato). Può essere considerato relativamente più vicino al basso livello rispetto ad altri linguaggi simili, quasi come "l'assembly dei linguaggi di programmazione funzionali". Elixir è un linguaggio più recente implementato sulla stessa macchina virtuale di Erlang (Beam), ispirandosi alla sintassi di Ruby e arricchendo Erlang con metaprogrammazione e macro igieniche.

Gli obiettivi principali alla base dello sviluppo di Erlang e dell'adozione del paradigma ad attori sono profondamente legati alla necessità di costruire sistemi robusti e scalabili:

- Sviluppare **programmi distribuiti**, ovvero programmi che eseguono concorrentemente con thread potenzialmente distribuiti in rete su macchine diverse.
- Implementare **sistemi distribuiti** in modo efficace.
- Progettare sistemi per essere eseguiti per **periodi estremamente lunghi**.
- Gestire la **concorrenza massiva** (≥ 100.000 attori).
- Consentire l'**aggiornamento del codice durante l'esecuzione** senza interrompere il programma (Hot Code Swap).
- Fornire una soluzione efficace per la **gestione dei guasti**.

Il paradigma ad attori, particolarmente nella sua implementazione in Erlang, presenta numerosi punti di forza che lo rendono ideale per i sistemi distribuiti e concorrenti:

1. **Efficacia per Sistemi Distribuiti e Concorrenti:** È considerato il paradigma più efficace per implementare sistemi distribuiti, gestendo l'esecuzione concorrente su thread che possono trovarsi su macchine diverse.
2. **Modello Concettuale Semplice e Chiaro:** La definizione degli attori tramite PID, Mailbox e Behaviour fornisce un modello chiaro e comprensibile per la costruzione di sistemi concorrenti.
3. **Isolamento dello Stato:** Gli attori non condividono memoria o stato. Questo isolamento evita intrinsecamente molti problemi classici della programmazione concorrente basata su memoria condivisa, come le race condition.
4. **Scalabilità Elevata:** Il paradigma è progettato per gestire un **numero massiccio di attori** (≥ 100.000). Poiché gli attori sono inattivi quando non ricevono messaggi, è efficiente in scenari con alta concorrenza dove solo una frazione degli attori è attiva in un

dato momento. WhatsApp viene citato come esempio di sistema che beneficia di questa scalabilità.

5. **Tolleranza ai Guasti ("Let it fail!")**: Un principio fondamentale è la gestione degli errori tramite la terminazione dei processi (attori) falliti e di quelli strettamente collegati, con riavvio affidato a processi supervisor. Questa strategia è particolarmente efficace in linguaggi funzionali come Erlang, dove lo stato del supervisore non viene modificato dal fallimento del figlio, permettendo una ricreazione rapida dello stato del figlio. La computazione viene strutturata gerarchicamente per facilitare questa supervisione.
6. **Comunicazione Asincrona**: L'unica forma di comunicazione è lo scambio asincrono di messaggi. Questo favorisce il disaccoppiamento tra gli attori.
7. **Pattern Matching**: Nei linguaggi che supportano nativamente il paradigma (come Erlang), il pattern matching nel costrutto `receive-end` permette una gestione efficiente e dichiarativa dei messaggi in arrivo. Questa tecnica si estende anche alla definizione generale delle funzioni.
8. **Integrazione con Programmazione Funzionale (in Erlang)**: L'implementazione del paradigma ad attori in un linguaggio funzionale puro come Erlang beneficia di caratteristiche come l'immutabilità dei dati, che è cruciale per l'efficienza del meccanismo "Let it fail!". L'ottimizzazione delle chiamate di coda (Tail Call Optimization) rende la ricorsione (usata per definire il behavior degli attori) efficiente come un ciclo.

Presenta anche dei punti di debolezza:

1. **Problemi con Approcci Ibridi**: L'utilizzo di librerie per implementare il paradigma ad attori in linguaggi mainstream (come Akka in Java/Scala) non è sempre consigliabile e può essere problematico. Questo perché combinare costrutti da paradigmi diversi tende a indebolire le garanzie offerte dai singoli paradigmi, come la garanzia di immutabilità dei dati.
2. **Gestione dell'Ordine dei Messaggi**: Lo scambio di messaggi asincrono non garantisce la ricezione sequenziale dei messaggi in uno scenario distribuito. Se l'ordine di elaborazione dei messaggi è critico, deve essere gestito esplicitamente a livello di comportamento dell'attore.
3. **Overhead e Necessità di Gestione Attenta**: La creazione e gestione di un numero molto elevato di attori può introdurre overhead, specialmente per attori con compiti molto piccoli o se il numero di attori attivi supera la capacità dei core CPU, portando a elevati costi di context-switch. Una parallelizzazione non gestita correttamente può risultare in prestazioni peggiori rispetto a un approccio sequenziale.
4. **Interazione con Entità Esterne Limitate**: Le interazioni con entità esterne al modello ad attori, come driver o risorse non-attore, devono avvenire tramite attori speciali identificati da "Porte" anziché PID. Queste Porte non seguono completamente i principi degli attori normali, come il "Let it fail!".
5. **Sicurezza Base dei Messaggi**: Nei linguaggi base che implementano il paradigma (come Erlang base), i messaggi tra attori vengono inviati in chiaro.

6. **La Tolleranza ai Guasti non è una Panacea:** Il principio "Let it fail!" implica la terminazione degli attori falliti e collegati. Se i problemi persistono dopo ripetuti tentativi di riavvio, anche i supervisor superiori nella gerarchia falliranno. Non è una garanzia contro *ogni* tipo di fallimento e richiede un'attenta progettazione della gerarchia di supervisione.

Domanda

Definizione di paradigma funzionale

Risposta

Il paradigma di programmazione funzionale vede il **programma come un'espressione**. Questo approccio si contrappone a paradigmi come quello imperativo, dove il programma è visto come una sequenza di azioni che modificano lo stato del programma.

Le caratteristiche fondamentali e i punti chiave del paradigma funzionale sono:

1. **Centralità delle Funzioni:** Nel calcolo lambda, da cui derivano molti concetti dei linguaggi funzionali moderni, "tutto è una funzione". Le funzioni sono le entità principali per costruire i programmi.
2. **Funzioni di Prima Classe e di Ordine Superiore:** Nei linguaggi funzionali, le funzioni sono tendenzialmente entità di prima classe. Ciò significa che possono essere:
 - Argomenti di altre funzioni.
 - Risultati di altre funzioni.
 - Definite all'interno di altre funzioni.
 - Assegnate a variabili.
 - Memorizzate in strutture dati. Le funzioni di ordine superiore sono quelle che prendono una o più funzioni come argomenti o restituiscono una funzione come risultato.
3. **Chiusure (Closures):** Un concetto strettamente legato alle funzioni di prima classe e annidate è quello delle chiusure. Una chiusura "impacchetta" il codice della funzione insieme ai valori delle variabili libere (variabili definite nello scope esterno alla funzione ma utilizzate al suo interno). Questo assicura che la funzione possa essere invocata correttamente anche in un contesto diverso da quello in cui è stata definita.
4. **Purezza e Assenza di Side Effects:** Una caratteristica distintiva (soprattutto nei linguaggi funzionali "puri" come Haskell) è l'assenza o la limitazione degli effetti collaterali (side effects). Un side effect si verifica quando una funzione modifica uno stato al di fuori del suo scope locale o interagisce con il mondo esterno (es. I/O, rete). L'assenza di side effects porta alla **trasparenza referenziale**: un'espressione può essere sostituita con il suo valore senza cambiare il comportamento del programma. Questo semplifica la comprensione del codice e facilita la parallelizzazione.
5. **Immutabilità dei Dati:** Nelle lingue funzionali (come Erlang, tranne in costrutti specifici), le variabili, una volta definite, non possono essere modificate. Si comportano essenzialmente come costanti. Le strutture dati sono tipicamente immutabili.

6. **Ricorsione:** La ricorsione è un meccanismo fondamentale per l'iterazione e la definizione del comportamento, ad esempio degli attori in Erlang. L'ottimizzazione delle chiamate di coda (Tail Call Optimization - TCO) è una tecnica comune nei compilatori funzionali per rendere la ricorsione efficiente in termini di spazio.
7. **Pattern Matching:** Il pattern matching è un costrutto potente utilizzato per definire funzioni per casi, basandosi sulla struttura dell'input. Permette di scrivere algoritmi in modo conciso e leggibile. In Erlang, viene ampiamente utilizzato per la gestione dei messaggi ricevuti dagli attori nel costrutto `receive-end`.
8. **Sistemi di Tipi (spesso forti):** Molti linguaggi funzionali moderni sono fortemente tipati e offrono sistemi di tipi sofisticati (statici) che garantiscono la correttezza del codice a compile-time, prevenendo errori a runtime. Concetti come l'inferenza di tipi, le classi di tipo (in Haskell) o i GADT (Generalized Algebraic Data Types) aiutano a gestire la complessità e a esprimere proprietà sui dati e sulle funzioni. Anche se Erlang è un'eccezione notevole per essere dinamicamente tipato, questo non è rappresentativo della maggior parte dei linguaggi funzionali moderni.
9. **Wholemeal Programming:** Approccio che si concentra sull'operare su intere strutture dati (come liste) piuttosto che manipolare singoli elementi passo dopo passo, spesso tramite funzioni predefinite o componibili.

Mentre l'imperativo si basa sul concetto di "stato" e sulla modifica di variabili, il funzionale si concentra sull'**applicazione di funzioni** e sulla composizione di espressioni.

La ripetizione in un linguaggio imperativo si ottiene tipicamente con cicli che modificano lo stato (es. contatori, elementi di array), mentre in un linguaggio funzionale si usa la ricorsione o funzioni su strutture dati (come `map`, `filter`, `fold`).

Domanda

Punti di forza e di debolezza del paradigma funzionale

Risposta

Il paradigma funzionale concepisce il **programma come un'espressione**, focalizzandosi sulla valutazione di funzioni piuttosto che su una sequenza di azioni che modificano uno stato globale, come accade nel paradigma imperativo.

Punti di Forza del Paradigma Funzionale:

1. **Purezza e Assenza/Limitazione degli Effetti Collaterali (Side Effects):** Una caratteristica fondamentale, soprattutto nei linguaggi funzionali "puri", è l'assenza o la forte limitazione degli effetti collaterali. Le funzioni pure restituiscono sempre lo stesso output per gli stessi input, indipendentemente dal contesto di esecuzione. Questo rende il codice più facile da ragionare e da testare.
2. **Trasparenza Referenziale:** Direttamente correlata alla purezza, implica che un'espressione può essere sostituita con il suo valore senza cambiare il comportamento

del programma. Questo principio facilita le ottimizzazioni del compilatore (come l'inlining).

3. **Immutabilità dei Dati:** Le variabili, una volta definite, non possono essere modificate; si comportano come costanti. Le strutture dati sono tipicamente immutabili. L'assenza di mutazione dei dati è una garanzia fondamentale che semplifica la gestione della concorrenza e del parallelismo, poiché non ci sono stati condivisi da proteggere.
4. **Funzioni di Prima Classe e di Ordine Superiore:** Le funzioni sono trattate come qualsiasi altro dato: possono essere passate come argomenti, restituite come valori, assegnate a variabili e memorizzate in strutture dati. Le funzioni di ordine superiore (che operano su altre funzioni) permettono l'astrazione su pattern computazionali comuni e abilitano tecniche come il "currying".
5. **Chiusure (Closures):** Consentono a una funzione definita all'interno di un altro ambito di "catturare" i valori delle variabili libere definite nell'ambito esterno, garantendo che la funzione possa essere eseguita correttamente anche fuori dal suo contesto di definizione originale. L'immutabilità nei linguaggi funzionali puri evita ambiguità semantiche legate alla modifica di queste variabili catturate.
6. **Wholemeal Programming:** L'approccio di operare su intere strutture dati (come liste) tramite funzioni predefinite o componibili, piuttosto che manipolare i singoli elementi. Questo porta a codice più conciso e astratto.
7. **Pattern Matching:** Un potente costrutto sintattico che permette di definire funzioni o gestire flussi di controllo in base alla struttura dei dati in input. Rende il codice più leggibile e conciso, specialmente per la gestione di tipi di dati complessi o per l'elaborazione di messaggi. Le "guardie" permettono di aggiungere condizioni booleane ai pattern senza effetti collaterali.
8. **Ricorsione:** Meccanismo fondamentale per l'iterazione. Supportata spesso da ottimizzazioni come la Tail Call Optimization per gestirla efficientemente in termini di stack space.
9. **Sistemi di Tipi Forti (nella maggior parte dei linguaggi moderni):** Molti linguaggi funzionali moderni (ma non Erlang) utilizzano sistemi di tipi statici e sofisticati che garantiscono la correttezza del codice a compile-time, catturando invarianti e prevenendo errori a runtime. Concetti come l'inferenza di tipi e i GADT (Generalized Algebraic Data Types) aumentano l'espressività e la capacità di modellare domini complessi.
10. **Facilità di Parallelizzazione/Concorrenza:** L'immutabilità e l'assenza di stati condivisi rendono molto più semplice scrivere codice concorrente corretto. Erlang è un esempio eccellente di come questi principi supportino sistemi distribuiti e tolleranti ai guasti.
11. **Potenziale nella Gestione degli Effetti Collaterali (tramite Monadi):** Sebbene la purezza sia una sfida per le interazioni con il mondo esterno, le monadi offrono un modo strutturato e matematicamente definito per gestire effetti collaterali (I/O, errori, stato) in linguaggi altrimenti puri, senza sacrificare completamente la purezza del core logico.

Punti di Debolezza del Paradigma Funzionale:

1. **Gestione Diretta degli Effetti Collaterali:** Nei linguaggi puramente funzionali, le operazioni che modificano lo stato esterno (come I/O, accesso a file, rete) non possono essere espresse direttamente in funzioni pure. Questo rende il linguaggio "abbastanza inutile" nella sua forma più basilare. Le monadi risolvono questo problema, ma aggiungono un livello di astrazione e complessità. Implementare compiti con effetti collaterali senza monadi può portare a codice "stravolto" e difficile da seguire.
2. **Complessità di Concetti Avanzati:** Concetti come le monadi o i GADT sono potenti ma richiedono una comprensione profonda e possono presentare una curva di apprendimento ripida per chi proviene da paradigmi diversi. Anche la notazione (come la notazione `do` in Haskell o le trasformazioni di monadi) può essere inizialmente complessa.
3. **Prevedibilità dell'Esecuzione (in linguaggi Lazy):** In linguaggi funzionali con valutazione pigra (lazy evaluation) come Haskell, l'ordine in cui le espressioni vengono valutate non è sempre immediato da dedurre guardando il codice, il che può portare a comportamenti inaspettati.
4. **Prestazioni per Alcuni Task:** Sebbene l'immutabilità aiuti la concorrenza, può comportare la creazione frequente di nuove strutture dati per simulare le modifiche. Implementazioni basate sulla rappresentazione uniforme dei dati (come in Erlang/Haskell) possono introdurre indirezioni che riducono l'efficienza a causa dei continui accessi alla memoria. La gestione della memoria (Garbage Collection) nei linguaggi che allocano frequentemente presenta sfide (come i riferimenti ciclici o il costo di Mark & Sweep con molta memoria viva), sebbene questi non siano problemi esclusivi del paradigma funzionale.

Domanda

Definizione di funzioni higher-order

Risposta

In un linguaggio di programmazione funzionale, le **funzioni sono considerate entità di prima classe**. Questo significa che le funzioni possono essere manipolate e utilizzate come qualsiasi altro valore o dato all'interno del linguaggio.

Specificamente, essere un'entità di prima classe implica che una funzione può:

- Essere passata come **argomento** ad altre funzioni.
- Essere restituita come **risultato** da altre funzioni.
- Essere definita **all'interno** di altre funzioni.
- Essere assegnata a una **variabile**.
- Essere memorizzata in una **struttura dati** (come array, liste, alberi, ecc.).

Le **funzioni di ordine superiore** sono quelle funzioni che sfruttano questa capacità di essere entità di prima classe. In particolare, una funzione è definita di ordine superiore se

accetta una o più funzioni come argomenti oppure restituisce una funzione come proprio risultato.

Questo concetto è centrale nel λ calcolo, dove "tutto è una funzione" e si parla esplicitamente di "funzioni di prima classe e di ordine superiore". Linguaggi come Haskell sono citati come esempi di linguaggi funzionali in cui le funzioni hanno un tipo, possono essere definite dentro altre funzioni e possono essere risultati di altre funzioni. Anche in Erlang le funzioni sono oggetti di prima classe e possono essere passate come argomento, restituite come risultato, inserite in strutture dati e definite in modo annidato.

Diversi esempi pratici di funzioni di ordine superiore sono:

- La **composizione funzionale** $(f \circ g)(x) = f(g(x))$.
Un esempio in Haskell $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$ mostra una funzione `.` che prende due funzioni $(b \rightarrow c)$ e $(a \rightarrow b)$ come argomenti e restituisce una nuova funzione $(a \rightarrow c)$. Questo è un esempio classico di funzione di ordine superiore.
- Funzioni come `fmap` (`fmap :: (a -> b) -> t a -> t b`), che prende una funzione $(a \rightarrow b)$ come primo argomento. Questo è un metodo comune nelle classi di tipo come `Functor` per applicare una funzione a elementi contenuti in una struttura dati.

L'utilizzo delle funzioni di ordine superiore, abilitato dalla natura di prima classe delle funzioni, permette tecniche potenti come il currying, che trasforma una funzione a più argomenti in una sequenza di funzioni a singolo argomento, e la creazione di chiusure, che impacchettano il codice della funzione insieme ai valori delle variabili libere dal loro ambiente di definizione. Queste capacità consentono un alto livello di astrazione e modularità nella programmazione funzionale.

Domanda

Definizione di oggetti di prima classe

Risposta

In un linguaggio di programmazione, un'entità (che sia un dato o, come vedremo, in particolare una funzione) è detta di **prima classe** se può essere trattata **come qualsiasi altro valore o dato all'interno del linguaggio**. Questo significa che per un'entità di prima classe valgono le stesse regole e possibilità di manipolazione che si applicano ai tipi di dati fondamentali come numeri o stringhe.

Essere un'entità di prima classe implica che tale entità può:

- Essere passata come **argomento** ad altre funzioni.
- Essere restituita come **risultato** da altre funzioni.
- Essere definita **all'interno** di altre funzioni (funzioni annidate).
- Essere assegnata a una **variabile**.

- Essere memorizzata in una **struttura dati** (come array, liste, alberi, ecc.).

Questo concetto è profondamente radicato nel **λ calcolo**, dove "tutto è una funzione" e le funzioni sono naturalmente considerate entità di prima classe e di ordine superiore fin dagli anni '30.

I linguaggi funzionali tendenzialmente supportano le funzioni come entità di prima classe. **Haskell** è citato come esempio di linguaggio funzionale in cui le funzioni hanno un tipo, possono essere definite annidate e restituite come risultati. Allo stesso modo, in **Erlang** le funzioni sono esplicitamente definite come "oggetti di prima classe, manipolabili come qualsiasi altro valore". Possono essere passate come argomento, restituite, inserite in strutture dati e definite annidate. La definizione del "behavior" di un attore in Erlang è essa stessa una funzione, e la possibilità che questo behavior cambi mostra la flessibilità con cui le funzioni sono trattate. Anche il "pattern matching" in Erlang può essere utilizzato per selezionare diverse implementazioni di una funzione in base all'input ricevuto, trattando la funzione in modo molto dinamico.

Il concetto di entità di prima classe non si limita strettamente alle sole funzioni. Ad esempio, in linguaggi con supporto per gli effetti algebrici, viene introdotto il concetto di "fibra", che rappresenta un frammento di stack distaccato ed è anch'esso un **tipo di dato di prima classe** su cui è possibile invocare operazioni.

La capacità di trattare le funzioni come entità di prima classe è un abilitatore fondamentale per altre potenti caratteristiche dei linguaggi, in particolare nel paradigma funzionale:

- **Funzioni di ordine superiore (Higher-Order Functions):** Una funzione è di ordine superiore se accetta altre funzioni come argomenti o restituisce una funzione come risultato. Questa capacità deriva direttamente dal fatto che le funzioni sono di prima classe. La composizione funzionale $(f \circ g)$ è un esempio classico che dimostra l'uso di funzioni di ordine superiore.
- **Chiusure (Closures):** Le chiusure (o "funzioni") sono pacchetti che combinano il codice della funzione con i valori delle "variabili libere" (quelle non definite localmente) catturate dall'ambiente in cui la funzione è stata definita. Poiché una funzione di prima classe può essere passata in contesti o tempi diversi da quello di definizione, la chiusura assicura che essa abbia accesso ai dati di cui ha bisogno. In linguaggi funzionali puri con immutabilità, la semantica delle chiusure è più chiara perché i valori catturati non possono essere modificati dopo la definizione.
- **Currying:** Molti linguaggi funzionali implementano funzioni "a più argomenti" come "cascate" di funzioni a singolo argomento, che è reso possibile dalla natura di prima classe delle funzioni e dall'uso delle chiusure.
- **Valutazione pigra (Lazy Evaluation):** Le chiusure possono essere usate per "impacchettare" espressioni che devono essere valutate solo se necessario, in linea con il concetto di valutazione pigra.

È utile confrontare questo concetto con linguaggi che non trattano le funzioni come entità pienamente di prima classe.

In **C**, esistono puntatori a funzioni, ma non è possibile definire funzioni annidate.

In **Pascal**, si possono definire funzioni annidate, ma manca un vero e proprio tipo funzione. Linguaggi come Java, Python e C++ non sono pienamente funzionali; modellano le funzioni in modi diversi (spesso come oggetti) e utilizzano "zucchero sintattico" per le espressioni lambda, ma le integrazioni sono state aggiunte a posteriori.

Domanda

Immutabilità: vantaggi e svantaggi

Risposta

L'immutabilità, in un linguaggio di programmazione, si riferisce alla proprietà per cui le **variabili, una volta definite, non possono essere modificate**. Esse si comportano essenzialmente come costanti. Di conseguenza, le **strutture dati sono considerate immutabili**.

Questo concetto è centrale nei linguaggi funzionali puri, dove non esiste la mutabilità delle strutture dati e non c'è assegnamento diretto. In tali linguaggi, qualsiasi operazione che in un paradigma imperativo comporterebbe una modifica in loco, in un linguaggio funzionale puro implica la creazione di una nuova versione del dato o della struttura dati, lasciando invariata quella originale.

Vantaggi dell'Immutabilità:

1. **Garanzie sulla Manipolazione dei Dati:** L'assenza di mutabilità garantisce che il dato non possa essere modificato da altre parti del programma. Questa garanzia è fondamentale per sapere che certe parti del dato rimangono immutate, il che è cruciale per ottenere algoritmi con una determinata complessità computazionale asintotica nei linguaggi funzionali.
2. **Semplificazione della Gestione dello Stato:** In sistemi come gli attori in Erlang, i linguaggi funzionali privi di mutazione garantiscono che lo stato di un attore padre non venga alterato dai suoi attori figli. Poiché la memoria non viene mai modificata ma si crea sempre nuova memoria, il padre mantiene esattamente lo stato in cui si trovava quando ha creato i figli. Questo semplifica notevolmente la gestione dei guasti, permettendo al padre di ricreare i figli con lo stato corretto in tempo $O(1)$.
3. **Chiarezza Semantica delle Chiusure:** In un linguaggio funzionale puro, dove non c'è assegnamento, la semantica delle chiusure (che catturano i valori delle variabili libere) risulta più chiara perché i valori catturati non possono essere modificati dopo la definizione della chiusura. Si contrappone a linguaggi impuri dove la modifica di variabili libere catturate può rendere la semantica oscura.
4. **Vantaggio Aggiuntivo (Concurrency/Parallelismo):** L'immutabilità semplifica enormemente la programmazione concorrente e parallela. Dato che lo stato condiviso tra thread o processi non può essere modificato, non esistono "data race" o condizioni di

competizione legate alla modifica concorrente dello stesso dato. Questo elimina la necessità di molti meccanismi di sincronizzazione complessi (lock, mutex, semafori), rendendo il codice più sicuro e più facile da parallelizzare correttamente.

Svantaggi dell'Immutabilità:

1. **Gestione Complessa degli Effetti Collaterali:** I linguaggi puramente funzionali, a causa della loro immutabilità e dell'assenza di effetti collaterali, non possono gestire direttamente operazioni come la stampa a schermo, le connessioni di rete, o la gestione di eccezioni e contatori in modo semplice e diretto. L'implementazione di tali "effetti" senza l'uso di costrutti specifici (come le Monadi) stravolge la struttura del programma, e la logica principale si perde nella gestione dell'effetto. Le Monadi offrono un modo strutturato per gestire questi effetti, ma il problema di fondo (l'incapacità di avere effetti diretti a causa della purezza/immutabilità) persiste.
2. **Svantaggio Aggiuntivo (Potenziale Overhead di Copia):** Anche se la gestione della memoria è spesso ottimizzata nei linguaggi funzionali (ad esempio, tramite strutture dati persistenti che condividono parti della memoria), in alcuni casi la creazione di nuove copie per ogni modifica può comportare un overhead di memoria e tempo rispetto a modifiche in loco in un linguaggio imperativo. Questo dipende fortemente dalla specifica implementazione del linguaggio e dalle strutture dati utilizzate.
3. **Costi di Allocazione e Gestione della Memoria:** Poiché la modifica di dati immutabili richiede la creazione di nuove strutture, i linguaggi funzionali tendono ad allocare memoria molto frequentemente. Questo richiede algoritmi di gestione della memoria e garbage collection più sofisticati, come il Mark & Sweep, potenzialmente generazionale. Questi algoritmi possono avere costi computazionali e presentare complessità implementative (ad esempio, nella gestione dei puntatori o nella preservazione dello sharing). L'approccio basato sulla rappresentazione uniforme dei dati (utilizzato in alcuni linguaggi funzionali come Erlang, che interseca con l'immutabilità) può introdurre indirezioni, con conseguente riduzione dell'efficienza.
4. **Indebolimento delle Garanzie con Paradigmi Misti:** Introdurre costrutti imperativi che permettono la mutazione dei dati in un linguaggio multi-paradigma indebolisce le garanzie fornite dalla pura immutabilità, come la garanzia che il dato non possa essere modificato da altri.

Domanda

Strictness vs lazyness

Risposta

Nei linguaggi di programmazione, la strategia di valutazione si riferisce al modo in cui gli argomenti di una funzione o le espressioni vengono valutati. Le due strategie principali sono la **valutazione stretta (strict evaluation)** e la **valutazione pigra (lazy evaluation)**.

Valutazione Pigra (Lazy Evaluation):

La valutazione pigra è una caratteristica di alcuni linguaggi funzionali, come **Haskell**. In un

linguaggio lazy:

- Le espressioni vengono valutate **solo quando necessario**.
- Gli argomenti di una funzione vengono valutati **al massimo una volta** e solo se necessario.
- Le chiusure possono essere utilizzate per "impacchettare" espressioni che non devono essere valutate subito, ma solo se necessario, allineandosi con il concetto di valutazione pigra.

La valutazione pigra è strettamente correlata alla **purezza** e all'**immutabilità** dei dati. La purezza implica la **trasparenza referenziale**, il che significa che un'espressione può essere sostituita dal suo valore senza cambiarne il comportamento. Questo è possibile solo in un linguaggio puro dove l'esecuzione di un'espressione non causa effetti collaterali visibili all'esterno. L'immutabilità dei dati, dove le variabili non possono essere modificate dopo l'assegnazione, rafforza questa purezza.

Vantaggi della Valutazione Pigna (e Purezza associata):

1. **Trasparenza Referenziale e Ottimizzazioni:** La purezza garantisce la trasparenza referenziale, permettendo al compilatore di effettuare ottimizzazioni aggressive come l'inlining.
2. **Modularità:** Permette di dividere il programma in moduli richiamati solo se strettamente necessario, migliorando la modularità.
3. **Normalizzazione:** Alcune espressioni con parti non definite o potenzialmente fallibili possono comunque essere evaluate se la parte "pigra" non viene mai richiesta (es. `const 1 undefined`, dove `undefined` non viene valutato).
4. **Parallelismo:** Nei linguaggi pigri e puri, è più semplice parallelizzare l'esecuzione.
5. **Efficienza Potenziale:** Le espressioni vengono valutate al massimo una volta e solo se necessario, potenzialmente migliorando l'efficienza evitando computazioni non richieste.

Svantaggi della Valutazione Pigna (e Purezza associata):

1. **Difficoltà nella Gestione degli Effetti Collaterali:** La purezza rende difficile o impossibile gestire direttamente operazioni con effetti collaterali, come la stampa a schermo, connessioni di rete, gestione di eccezioni o contatori. L'implementazione di tali "effetti" in un linguaggio puro stravolge la struttura del programma. La logica principale si perde nella gestione dell'effetto.
2. **Imprevedibilità dell'Ordine di Esecuzione:** Se un linguaggio puramente lazy *consentisse* (attraverso costrutti speciali) effetti collaterali, l'ordine in cui questi effetti si verificano potrebbe essere imprevedibile, rendendo difficile ragionare sul comportamento del programma.
3. **Complessità per Effetti Collaterali:** Richiede l'introduzione di costrutti specifici, come le **Monadi**, per gestire in modo strutturato le computazioni con effetti collaterali. Anche con

le monadi, la gestione di effetti multipli o combinati può presentare problematiche concettuali e ingegneristiche.

Valutazione Stretta (Strict Evaluation):

La valutazione stretta è la strategia di valutazione predefinita nella maggior parte dei linguaggi di programmazione (imperativi e molti funzionali non puri). In un linguaggio stretto:

- Gli argomenti di una funzione vengono valutati **completamente prima** che la funzione stessa venga invocata.
- Le espressioni vengono generalmente valutate non appena incontrate, a meno che non siano all'interno di costrutti speciali che introducono forme limitate di pigrizia (come gli short-circuiting operator `&&` e `||` in molti linguaggi).

Vantaggi della Valutazione Stretta:

1. **Gestione Diretta degli Effetti Collaterali:** Le operazioni con effetti collaterali (I/O, eccezioni, modifiche di stato) sono integrate naturalmente nel flusso di esecuzione. È facile stampare a schermo, scrivere su file, ecc., in modo sequenziale e prevedibile.
2. **Ordine di Esecuzione Prevedibile:** L'ordine in cui le espressioni vengono valutate è solitamente ben definito (spesso da sinistra a destra o basato su precedenza degli operatori), semplificando il ragionamento sul flusso del programma, specialmente in presenza di effetti collaterali.
3. **Implementazione Runtime Potenzialmente Più Semplice:** Generalmente non richiede meccanismi complessi per gestire espressioni non ancora valutate ("thunks"), portando a un runtime più semplice per l'esecuzione delle chiamate a funzione.

Svantaggi della Valutazione Stretta:

1. **Potenziale per Computazioni Inutili:** Gli argomenti vengono valutati anche se la funzione potrebbe non utilizzarli, sprecando potenzialmente risorse computazionali.
2. **Complessità nella Programmazione Concorrente/Parallela:** L'intercalarsi di effetti collaterali e la mutabilità dei dati (spesso associata alla valutazione stretta nei paradigmi imperativi) rendono molto più complesso scrivere codice concorrente o parallelo sicuro, richiedendo meccanismi di sincronizzazione espliciti (lock, mutex) per evitare "data race". I side effect sono spesso causa di conflitti tra attività parallele.
3. **Meno Spazio per Alcune Ottimizzazioni:** La presenza di effetti collaterali e la mancanza di trasparenza referenziale limitano le ottimizzazioni che il compilatore può effettuare, come l'inlining o la rimozione di codice morto, se non è certo che non alterino il comportamento del programma.
4. **Difficoltà di Astrazione in Alcuni Casi:** Costruire computazioni complesse combinando funzioni più semplici può essere complicato se le funzioni intermedie hanno effetti collaterali che devono essere gestiti in un ordine preciso.

Domanda

Risposta

Nei linguaggi funzionali come **Erlang**, **OCaml** e **Haskell**, che supportano **polimorfismo uniforme implicito**, si adotta una strategia di rappresentazione dei dati a **run-time** che consente di gestire valori di qualsiasi tipo senza richiedere una dichiarazione esplicita di tipo (come invece accade nei **template** di C++ o nei **generics** di Java).

Questa strategia si basa su un modello di **rappresentazione uniforme dei dati con tagging**.

Rappresentazione uniforme dei dati

L'idea centrale è rappresentare **tutti i dati con una word**, cioè un'unità di memoria la cui dimensione dipende dall'architettura del processore (tipicamente 32 o 64 bit), e che è in grado di contenere un puntatore.

- **Dati piccoli** (come interi piccoli o booleani) possono essere rappresentati direttamente nella word stessa: questi sono detti **unboxed** o **value types**.
- **Dati complessi o grandi** (come tuple, liste, alberi, stringhe lunghe, funzioni) vengono memorizzati nello **Heap** e sono rappresentati nella word come **puntatori**: questi sono detti **boxed** o **reference types**.

Per distinguere i vari tipi di dato, viene usato un **tag** inserito all'interno della word stessa (ad esempio nel bit meno significativo), che permette al run-time di riconoscere se una word rappresenta un valore diretto, un puntatore o un altro tipo specifico.

Viene scelto il bit meno significativo perchè:

- Se fosse stato scelto il 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.

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).

Tagging e struttura dei dati

- Il **bit di tag** consente di distinguere tra valori e puntatori, e in generale tra le diverse classi di dato.

- I dati **boxed** (cioè quelli nello heap) iniziano con una **word di header** che contiene informazioni strutturali come:
 - Il **tag** del tipo (es. tupla, lista, funzione...)
 - La **dimensione** della struttura
 - (eventualmente) metadati per la garbage collection

Questa struttura è fondamentale per gestire la memoria dinamica, la condivisione dei dati tra funzioni, e il comportamento efficiente del garbage collector.

Esempi di tipi di dato e rappresentazione

- **Atomi**: identificatori simbolici (es. `ok`, `error`) rappresentati da un ID o puntatore a una tabella globale di simboli.
- **Numeri**: interi o float, unboxed se piccoli, altrimenti boxed.
- **Tuple**: allocate sullo heap, con header e valori.
- **Liste**: strutture ricorsive (testa e coda), ogni elemento è boxed.
- **Bit string**: sequenze binarie, allocate sullo heap.
- **Chiusure (funzioni)**: rappresentate come strutture contenenti codice e ambiente (variabili libere), anch'esse allocate sullo heap.

Efficienza ed effetti collaterali

- Questo modello introduce **indirezioni** (accesso tramite puntatori), con conseguente **overhead di accesso** alla memoria.
- Le operazioni aritmetico-logiche devono **tener conto del bit di tag**, aggiungendo complessità al codice macchina.
- Non è possibile interpretare liberamente le sequenze di bit a run-time, quindi la coerenza tra scrittura e lettura dei dati è fondamentale.

Sistema di tipi e controllo statico

A livello di codice macchina, i **tipi non esistono**: tutto è una sequenza di bit. Il **linguaggio di programmazione** introduce i tipi per:

- Garantire coerenza tra l'**interpretazione alla scrittura** e alla **lettura** dei dati.
- Prevenire errori semantici (es. interpretare un numero come un puntatore).
- Consentire al compilatore di **verificare staticamente** alcune proprietà del programma.

Questo controllo viene eseguito tramite il **sistema di tipi**, che agisce **a compile-time** ed è un'analisi statica, modulare, e conservativa: assicura che operazioni e interpretazioni siano compatibili anche se, in generale, il problema è indecidibile a run-time.

Polimorfismo uniforme e sue implicazioni

In questi linguaggi è presente il **polimorfismo uniforme**, che consente di scrivere funzioni che operano su **dati di qualunque tipo** senza dipendere dalla loro rappresentazione.

Esempio in OCaml:

```
let swap (x, y) = (y, x);;
```

Questa funzione funziona su qualsiasi coppia di valori, indipendentemente dal tipo. Questo tipo di polimorfismo:

- È **implicito** (non serve dichiararlo)
- Richiede che i dati abbiano una **rappresentazione uniforme**
- Sposta la complessità sulla rappresentazione a run-time

Alternative alla rappresentazione uniforme

Nel testo vengono anche descritte altre strategie usate in contesti diversi:

Monomorfizzazione (C++, Rust)

Ogni funzione polimorfa viene compilata una volta per ogni tipo effettivamente usato (es. `swap<int>`, `swap<string>`).

Vantaggi: ottimizzazione specifica per tipo, nessuna indirezione.

Svantaggi: aumento del tempo di compilazione e della dimensione dell'eseguibile (name mangling).

“Alla C” – void*

- Dati gestiti con puntatori `void*` e dimensione `size_t`.
- Nessuna informazione di tipo a run-time.
- Funzioni devono operare manualmente su byte, con perdita di sicurezza.
- Più flessibile ma inefficiente e priva di controllo statico.

Gestione della memoria nei linguaggi funzionali

La rappresentazione uniforme, combinata con l'**immutabilità** dei dati e l'uso intensivo dello heap, rende la **Garbage Collection (GC)** una componente fondamentale:

- Tecniche comuni: **Mark & Sweep**, **Generazionale**, **Reference Counting** (meno usato).
- Il **tag** nelle strutture boxed aiuta la GC a riconoscere cosa deallocare e quanto spazio occupa.
- I dati effimeri (come strutture intermedie) vengono raccolti rapidamente nelle generazioni giovani.

Domanda

La garbage collection nei linguaggi funzionali

Risposta

Nello stile di programmazione funzionale puro, dove l'**immutabilità dei dati** è una proprietà fondamentale, si ha la garanzia che un dato non possa essere modificato dopo la sua creazione. Questo porta alla necessità di creare nuove strutture dati ogni volta che si

"modifica" qualcosa, generando potenzialmente molti oggetti di breve durata.

Questo pattern di **frequente allocazione** rende i linguaggi funzionali particolarmente adatti all'uso di algoritmi di Garbage Collection come il **Mark & Sweep**.

Sono quindi presenti diverse problematiche relative alla gestione della memoria, come la formazione di **garbage** (memoria non più utile ma non deallocata), i **dangling pointers** (puntatori a memoria non allocata), il **riutilizzo improprio di memoria deallocata** e la **frammentazione della memoria**.

Le tecniche di gestione della memoria si dividono principalmente in tre categorie:

- **Nessuna gestione automatica**, affidata completamente al programmatore (es. C).
- **Garbage Collection automatica**, dove il runtime del linguaggio cerca di identificare e deallocare il garbage. Questa si basa su euristiche di "garbage detection".
- **Gestione esplicita da parte del programmatore**, dove il programmatore descrive al compilatore come gestire l'allocazione/deallocazione (es. Rust, C++).

Nell'ambito della Garbage Collection automatica, esistono due tecniche principali:

1. **Reference Counting (Conteggio dei Riferimenti):**

- Si basa sull'euristica che un dato (generalmente allocato sullo Heap, definito "boxed") con **zero puntatori entranti** è considerato garbage.
- Richiede di tenere traccia del numero di puntatori per ogni dato boxed, spesso memorizzando un **Reference Counter (RC)** nella prima cella del dato stesso.
- Un limite importante è la gestione delle **strutture dati cicliche**, dove i riferimenti reciproci impediscono al contatore di raggiungere zero, causando **memory leak**. Questo problema può essere mitigato distinguendo tra **strong pointer** (conteggiati) e **weak pointer** (non conteggiati).
- La memoria risulta **frammentata**, richiedendo algoritmi per la gestione degli spazi liberi.
- Il costo computazionale della **copia di un puntatore** (che richiede l'aggiornamento dei contatori) può essere significativo e potenzialmente illimitato.

2. **Mark & Sweep:**

- Si basa sull'euristica che un dato **non raggiungibile dalle "radici"** (come stack e registri) è considerato garbage.
- Comprende due fasi concettuali: **Mark**, che identifica gli oggetti raggiungibili a partire dalle radici, e **Sweep**, che compatta gli oggetti marcati e dealloca quelli non marcati. Viene implementato come un'unica operazione efficiente.
- Una strategia implementativa prevede l'uso di **due Heap**, uno di lavoro e uno vuoto, scambiandone i ruoli dopo ogni ciclo di GC. Questo permette di **evitare la frammentazione** nel nuovo Heap.
- A differenza del Reference Counting, il Mark & Sweep gestisce **automaticamente le strutture dati cicliche**.

- Un problema nella versione base era la perdita dello **sharing** (condivisione) dei dati nel nuovo Heap, risolto salvando un puntatore al nuovo dato nella cella originale.
- Il costo computazionale in tempo è **$O(n + m)$** (n = radici, m = dati vivi), che può essere elevato se ci sono molti dati vivi.
- Presenta problemi con **confronti** ($<$, $>$, etc.) e **funzioni hash** che usano i puntatori, poiché i dati vengono spostati in memoria. Questo richiede l'aggiunta di un **ID univoco** ai dati boxed come chiave per i dizionari, introducendo un'indirezione aggiuntiva.
- **Mark & Sweep Generazionale:**
 - Una variante per migliorare l'efficienza quando ci sono molti dati vivi.
 - Si basa sull'**ipotesi generazionale**: i dati tendono a diventare irraggiungibili rapidamente o a rimanere raggiunti a lungo.
 - Utilizza **tre Heap**: due "Young" e uno "Old" (o "Major"). I dati giovani vengono gestiti principalmente tra i due Heap Young (Minor Collection), mentre i dati che sopravvivono più cicli vengono spostati nell'Heap Old.
 - Il costo in tempo è ridotto a **$O(n + m')$** , dove m' è il numero di dati nei soli Heap Young, escludendo i dati presumibilmente di lunga vita nello Heap Old dalla scansione frequente.
 - Nei linguaggi con **dati mutabili**, è necessario gestire il caso in cui una cella nell'Heap Old punti a un dato nell'Heap Young, aggiungendo queste celle alle radici e introducendo un **Write Barrier** (overhead sull'assegnamento di puntatori).

L'immutabilità dei dati tipica dei linguaggi funzionali come Erlang ha un impatto positivo sul GC: quando una variabile cattura un dato boxed, si crea **sharing** tramite puntatore, ma questo non causa problemi perché il dato puntato non può essere modificato. Come menzionato in, la necessità del Write Barrier nel Generational Mark & Sweep sorge specificamente nei linguaggi con **dati mutabili**, suggerendo che l'immutabilità possa semplificare o eliminare questo overhead in linguaggi funzionali puri.

Domanda

Pattern, pattern matching e guardie

Risposta

Il **Pattern Matching** è una potente caratteristica presente in molti linguaggi di programmazione, in particolare quelli funzionali. Esso consente di specificare diverse implementazioni di una funzione o diverse azioni da intraprendere in base alla **forma** (struttura) dell'input ricevuto. Non si tratta solo di controllare l'uguaglianza di un valore, ma di destrutturare dati complessi ed estrarne componenti, associandoli a variabili locali. Ad esempio, in Erlang, il pattern matching è una caratteristica fondamentale utilizzata in vari contesti:

1. **Definizione di Funzioni:** Le funzioni possono essere definite per casi, con implementazioni diverse per pattern di input differenti. Quando una funzione viene chiamata, l'input viene confrontato con i pattern definiti in ordine sequenziale dall'alto verso il basso, e viene eseguito il codice associato al primo pattern che corrisponde.
2. **Ricezione di Messaggi:** Nel modello ad attori di Erlang, gli attori ricevono messaggi tramite il costrutto `receive-end`. I messaggi nella mailbox vengono confrontati con una serie di pattern definiti all'interno del blocco `receive`, sempre in ordine. Il primo pattern che corrisponde al messaggio in cima alla coda viene utilizzato, il messaggio viene rimosso, e il codice associato viene eseguito.
3. **Assegnamento (= operator):** L'operatore `=` in Erlang non è un semplice assegnamento come nei linguaggi imperativi, ma esegue un pattern matching tra la parte sinistra e la parte destra. Se il pattern a sinistra corrisponde al valore a destra, le variabili nel pattern vengono associate ai valori corrispondenti. Se il pattern a sinistra è un **pattern irrefutabile** (che corrisponde sempre a qualsiasi valore, come una singola variabile `X` o un wildcard `_`), funziona come un assegnamento o una semplice associazione. Se il pattern è **refutabile** (come un pattern complesso tipo `{A, B}` o `[H|T]`) e il match fallisce, viene generato un errore.

I **Pattern** sono descrizioni della forma attesa per un dato. Possono includere:

- **Valori letterali:** Come numeri interi, floating point, atomi. Un pattern `5` corrisponde solo al valore `5`.
- **Atomi:** Utilizzati spesso per distinguere tipi diversi di messaggi o costruttori di dati. Un pattern `print` corrisponde solo all'atomo `print`.
- **Variabili:** Denotate con una lettera maiuscola (es. `N`, `PID`, `T1`, `K`, `V`, `H`, `L`, `X`, `Y`). Quando un pattern corrisponde, la variabile viene **associata** al valore corrispondente nel dato. In linguaggi funzionali come Erlang, queste variabili sono immutabili; non si può riassegnare un nuovo valore a una variabile dopo che le è stato associato un valore tramite pattern matching.
- **Wildcard (`_`):** Un pattern che corrisponde a qualsiasi valore, ma che non associa il valore a una variabile. Utile quando si vuole ignorare una parte della struttura.
- **Strutture dati:** Come tuple (`{...}`) o liste (`[...]`, `[H|L]`). I pattern possono essere annidati, permettendo di fare il match su strutture complesse. Un pattern `[H|L]` fa il match con una lista non vuota, associando il primo elemento a `H` e il resto della lista a `L`. Un pattern `{_, T1, 5, {leaf, K, V}}` matches a 4-elemento tuple where the third element is 5, and the fourth element is a tuple starting with the atom `leaf`, binding other components to variables.
- **Bit Strings:** Erlang permette pattern matching anche su sequenze di bit.

Le **Guardie** offrono un ulteriore livello di controllo. Dopo che un pattern ha fatto match strutturale, è possibile aggiungere una condizione logica usando la parola chiave `when`. Il ramo di codice viene eseguito solo se sia il pattern che la guardia corrispondono/sono vere.

A livello implementativo, il compilatore traduce il pattern matching in una serie di controlli sulla struttura dei dati e assegnamenti di variabili. Per pattern matching su dati complessi e potenzialmente annidati, vengono generati accessi diretti ai componenti del dato (es. `T.length`, `T`, `T`). Se ci sono molti pattern per la stessa funzione o blocco `receive`, il compilatore utilizza tecniche basate su automi a stati finiti per compilare i pattern in modo efficiente, evitando ripetizioni di controlli e ottenendo una complessità sub-lineare rispetto alla somma delle lunghezze dei pattern, anziché la naive sequenziale $O(n)$. Per un singolo pattern, il costo runtime è $O(n)$ dove n è la lunghezza del pattern, ma poiché il pattern è fisso a compile-time, questo è considerato $O(1)$ a runtime.

Domanda

Hot-swap del codice in Erlang: come invocarlo e come è implementato

Risposta

L'**Hot Code Swap** è una caratteristica fondamentale del linguaggio Erlang, che permette di **caricare a runtime una nuova versione del codice senza interrompere l'esecuzione del programma**. Questa funzionalità è essenziale per applicazioni che richiedono runtime "infiniti" e tempi di inattività minimi.

L'hot code swap in Erlang non avviene automaticamente, ma deve essere **attivamente richiamato dal programmatore**. Tipicamente, questo viene fatto definendo nel "behavior" dell'attore un pattern di messaggi dedicato all'aggiornamento.

1. Si prepara una nuova versione del modulo Erlang. Questo modulo conterrà la logica aggiornata per il comportamento dell'attore.
2. Nell'implementazione corrente dell'attore (nella sua funzione ricorsiva che definisce il "behavior"), si include un caso di pattern matching nel blocco `receive-end` specifico per gestire la richiesta di aggiornamento.
3. Quando un attore riceve un messaggio che corrisponde a questo pattern (ad esempio, un messaggio con l'atomo `upgrade`), il codice associato a questo pattern viene eseguito.
4. All'interno di questo codice, l'attore effettua una **chiamata esplicita alla nuova versione della sua funzione di behavior**. Questo viene fatto utilizzando il costrutto `?MODULE:new_loop(...)` nell'esempio fornito. Il macro `?MODULE` assicura che la chiamata si rivolga alla versione più recente del modulo caricata nella Beam.
5. Lo **stato corrente** dell'attore (che nei linguaggi funzionali come Erlang è tipicamente rappresentato dagli argomenti della funzione ricorsiva di behavior) deve essere **passato esplicitamente** come argomenti alla nuova funzione di behavior. Questo garantisce che il nuovo codice riprenda l'esecuzione dallo stato corretto.
6. La nuova funzione di behavior deve quindi richiamare la funzione ricorsiva di loop nella nuova versione del modulo, utilizzando lo stato ricevuto.

Questo approccio consente di passare da una versione del codice a un'altra, anche se gestiscono tipi di dati completamente diversi, purché lo stato venga trasferito correttamente.

L'implementazione dell'hot code swap si basa sulle capacità della **macchina virtuale di Erlang, chiamata Beam**.

1. La Beam è in grado di **mantenere in memoria fino a due versioni compilate** dello stesso modulo contemporaneamente: la versione attualmente attiva e la nuova versione a cui si desidera passare.
2. Quando una nuova versione di un modulo viene compilata e caricata la Beam la rende disponibile.
3. Il passaggio effettivo alla nuova versione avviene nel momento in cui un processo Erlang (un attore) **esegue una chiamata a una funzione nella nuova versione del modulo**. Utilizzare il macro `?MODULE` garantisce che la chiamata dinamica si rivolga all'ultima versione caricata dalla Beam.
4. Poiché gli attori Erlang sono solitamente implementati tramite **funzioni ricorsive** (spesso in forma tail-recursive per beneficiare dell'ottimizzazione della Tail Call Optimization), la chiamata alla nuova versione del behavior function (`?MODULE:new_loop`) sostituisce il contesto di esecuzione corrente dell'attore con quello della nuova funzione nel nuovo modulo.
5. A livello di implementazione, questo implica che la Beam gestisce i riferimenti alle diverse versioni del codice dei moduli e indirizza le chiamate alle funzioni appropriate in base alla versione attiva per quel processo. Il codice del modulo da trasferire tra nodi distribuiti (un concetto correlato ma diverso dall'hot-swap *all'interno* dello stesso nodo) richiede il trasferimento del Byte Code tra le istanze della Beam. L'hot-swap locale sfrutta il fatto che la Beam già possiede le multiple versioni caricate in memoria.

Domanda

Chiamate di funzione di coda, funzioni tail ricorsive e relativa ottimizzazione

Risposta

In Erlang, come in molti linguaggi funzionali, le funzioni possono essere implementate tramite **ricorsione**. A runtime, per ogni chiamata di funzione, la macchina virtuale (Beam) mantiene uno **Stack di Record di Attivazione (RA)**. Un RA contiene informazioni necessarie per l'esecuzione della funzione, tra cui variabili locali, parametri, valori dei registri salvati, indirizzo del valore di ritorno e indirizzo di ritorno.

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

Tuttavia, non tutte le informazioni nel record di attivazione della funzione chiamante potrebbero essere necessarie dopo che la funzione chiamata ha terminato l'esecuzione. È possibile applicare ottimizzazioni per rimuovere le variabili non più necessarie dallo stack prima di invocare un'altra funzione.

Un caso limite e particolarmente importante di questa ottimizzazione è la gestione delle **Chiamate di Coda (Tail Calls)**. Una chiamata a una funzione `g()` all'interno di una funzione `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()`.
- Non è contenuta all'interno della parte valutativa di un blocco `try ... catch` (ovvero non è protetta).

Quando una chiamata viene identificata come chiamata di coda, il compilatore di Erlang esegue l'**Ottimizzazione delle Chiamate di Coda (Tail Call Optimization - TCO)**. Invece di spingere un nuovo record di attivazione per la funzione chiamata (`g()`), il compilatore sostituisce il record di attivazione corrente della funzione chiamante (`f()`) con quello della funzione chiamata.

La sequenza di istruzioni compilate per una chiamata di coda ottimizzata è:

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

Il vantaggio fondamentale della TCO è che una chiamata di coda implementata in questo modo ha un costo di **$O(1)$ in tempo e " $O(0)$ " in spazio**, poiché non viene allocato un nuovo record di attivazione sullo stack. Questo è particolarmente importante per le funzioni ricorsive.

Una funzione `f()` si definisce **Tail Ricorsiva** se e solo se tutte le sue chiamate ricorsive sono chiamate di coda. Questa è una definizione molto diffusa e pertinente nel contesto della programmazione funzionale.

L'ottimizzazione della tail recursion è cruciale in Erlang, specialmente per l'implementazione del **behavior degli attori**. Il behavior di un attore è tipicamente implementato come una funzione ricorsiva che attende e processa messaggi. Se questa funzione ricorsiva non fosse tail recursive, ogni messaggio ricevuto e processato causerebbe la crescita dello stack, portando inevitabilmente a uno stack overflow per attori di lunga durata. Rendendo la funzione di behavior tail ricorsiva, lo stack non cresce e l'attore può vivere indefinitamente processando messaggi.

L'esempio fornito nel materiale illustra questo concetto con la funzione `loop()` e la sua evoluzione. La funzione `loop` (e le sue versioni `loop` in 1.5 e 2) definisce il comportamento di un attore che riceve messaggi tramite un costrutto `receive-end`. Al ricevimento di un messaggio, viene eseguita l'azione corrispondente e poi viene effettuata una chiamata ricorsiva a `loop()` (o alla nuova versione `?MODULE:new_loop` durante l'hot-swap) per continuare a processare i messaggi futuri.

Domanda

Costrutti tipici dei linguaggi di programmazione funzionale in sintassi Erlang e loro uso: pattern matching, case analysis, local function declaration (sia nella versione ricorsiva che in quella non ricorsiva), list comprehension

Risposta

Erlang è un **linguaggio di programmazione funzionale**, noto per aver ridefinito la programmazione ad attori. Si distingue per essere dinamicamente tipato e relativamente vicino al basso livello. Molti dei suoi costrutti riflettono il paradigma funzionale e sono fondamentali per la sua architettura, in particolare quella basata sugli attori.

Ecco alcuni costrutti funzionali tipici e il loro uso in Erlang:

1. Pattern Matching

- **Descrizione:** Il Pattern Matching è un meccanismo fondamentale in Erlang. Permette di confrontare una descrizione (un pattern) con un dato in input. Se il dato corrisponde al pattern, le variabili nel pattern vengono legate ai valori corrispondenti nel dato. È un modo conciso e potente per destrutturare dati e controllare il flusso di esecuzione.
- **Uso:**
 - **Nella ricezione di messaggi (receive-end):** Un attore processa i messaggi nella sua mailbox confrontandoli con una serie di pattern definiti all'interno di un blocco `receive-end`. Il primo pattern che corrisponde a un messaggio estratto dalla coda viene eseguito.
 - *Sintassi Esempio (dal conto corrente):*

```
receive
    print -> io:format("Il balance e' ~p ~n", [Bal]),
    cc(Bal);
    {put, N} -> cc(Bal + N);
    {get, PID} -> PID ! Bal, cc(Bal);
    exit -> ok
end.
```

In questo esempio, `print`, `{put, N}`, `{get, PID}` ed `exit` sono pattern. `{put, N}` lega la variabile `N` al secondo elemento della tupla ricevuta. `{get, PID}` lega `PID` al secondo elemento della tupla. I diversi casi di pattern matching sono separati da punto e virgola.

- **Nelle definizioni di funzione:** Le funzioni possono avere diverse implementazioni (clausole) selezionate in base al pattern matching sugli argomenti in input.
 - *Sintassi Esempio (funzione definita per casi):*

```

fun ({N, 2}) -> N;
    ({ciao, N, M}) -> N + M;
    ([_, _, {X, Y}]) -> X + Y
end.

```

Qui, la funzione anonima ha tre clausole, ognuna con un pattern diverso per l'input. Il simbolo `_` ignora un valore nel pattern.

- **Con le Guardie (when):** È possibile aggiungere condizioni aggiuntive a un pattern tramite la parola chiave `when` seguita da una guardia. Le guardie possono contenere solo combinazioni di Built-In Functions (BIFs) per evitare effetti collaterali.

- *Sintassi Esempio:*

```

fun ({N, 2}) when N > 2 -> N;
    ...
end.

```

Questo pattern corrisponde solo se l'input è `{N, 2}` e `N` è maggiore di 2.

- **Implementazione e Costo:** La compilazione del pattern matching è efficiente, spesso utilizzando un automa a stati finiti modificato per gestire l'ordine di valutazione (dall'alto verso il basso) e ottimizzare i test. Sebbene l'analisi di compilazione possa essere complessa, a runtime il costo pratico è $O(1)$ per pattern di lunghezza fissa (non dipendente dall'input a runtime).

2. Case Analysis (case construct)

- **Descrizione:** Il costrutto `case` è un'altra forma di pattern matching. Permette di valutare un'espressione e confrontare il suo risultato con una serie di pattern.
- **Uso:** Viene spesso utilizzato per gestire in modo strutturato i diversi risultati possibili di un'espressione, come nel caso della gestione delle eccezioni con `try`. È fondamentale per implementare correttamente la tail recursion in presenza di blocchi `try`.

- *Sintassi Esempio (dal conto corrente tail-ricorsivo):*

```

case try receive
    ...
    exit -> {result, ok}
    catch error:_ -> {recur, Bal}
end of
    {result, R} -> R ;
    {recur, Bal} -> cc(Bal) ; % tail ricorsiva!
end.

```

In questo esempio, il risultato del blocco `try receive ... end` viene confrontato con i pattern `{result, R}` e `{recur, Bal}`. Questo permette di separare la logica di gestione del risultato dalla chiamata ricorsiva, facilitando l'ottimizzazione della tail recursion.

3. Dichiarazione di Funzioni

- Le funzioni sono oggetti di prima classe in Erlang, possono essere passate come argomenti, restituite come risultati, ecc.. Erlang offre diverse sintassi per definirle:
- Funzioni Nominate (nei Moduli):** Le funzioni che costituiscono l'API di un modulo o l'implementazione interna vengono dichiarate con un nome all'interno di un file `.erl`.

- Sintassi Esempio:*

```
-module(hello).  
-export([cc/1]).  
  
cc(Bal) -> % Definizione della funzione cc con 1 argomento  
    receive  
        ... % behavior  
    end.
```

La direttiva `-module` dichiara il nome del modulo. `-export` rende la funzione `cc` (con arità 1, ovvero 1 argomento) accessibile dall'esterno del modulo. Ogni definizione di funzione (o clausola) termina con un punto.

- Funzioni Locali/Annidate e Anonime:** È possibile definire funzioni senza nome usando la parola chiave `fun`. Queste sono le "funzioni anonime" o "lambda". Possono essere annidate all'interno di altre funzioni e formano **chiusure** (closure), avendo accesso alle variabili del loro ambito di definizione.

- Sintassi Esempio (funzione anonima semplice):*

```
fun (Arg) -> Arg + 1 end.
```

- Sintassi Esempio (funzione anonima come chiusura):*

```
G = fun (X) -> fun (Y) -> X + Y end end.  
H = G(2). % H è una chiusura che "ricorda" X=2  
H(3).    % Valuta (2 + 3) -> 5
```

- Funzioni Ricorsive:** La ricorsione è il modo standard per implementare iterazione e loop nei linguaggi funzionali come Erlang. La definizione di un attore in Erlang è tipicamente una funzione ricorsiva che rappresenta il suo behavior.
- Sintassi Esempio (ricorsione nominata - Tail Recursive):*

```
cc(Bal) -> % Prima chiamata o ricorsiva
    receive
        {put, N} -> cc(Bal + N); % Chiamata ricorsiva
        ...
    end.
```

Questa funzione `cc` chiama se stessa alla fine di ogni ramo del `receive`, mantenendo lo stato (`Bal`).

- *Sintassi Esempio (ricorsione anonima):*

```
FunRicorsiva = fun F(N) -> N * F(N - 1) end. % F è visibile
solo all'interno
```

- **Ottimizzazione della Tail Recursion (TCO):** Le funzioni ricorsive in Erlang sono spesso scritte in forma **tail recursive** per beneficiare dell'Ottimizzazione delle Chiamate di Coda (TCO) implementata dalla Beam. Una chiamata è di coda se è l'ultima operazione eseguita prima del ritorno. La TCO sostituisce lo stack frame della funzione chiamante con quello della funzione chiamata, trasformando la ricorsione in un loop efficiente che non consuma stack. Questo è cruciale per gli attori che devono girare indefinitamente.

4. List Comprehension

- **Descrizione:** È una sintassi concisa e potente per creare nuove liste applicando trasformazioni e filtri agli elementi di una o più liste di input.
- **Uso:** Permette di esprimere operazioni su liste in modo dichiarativo, simile alla notazione matematica per gli insiemi.
- *Sintassi Esempio:*

```
[ Expression || Generator1, ..., GeneratorN, Filter1, ...,
FilterM ].
```

- *Esempio da Fonte:*

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

% Con filtro:
[ {X, Y + 1} || X <-, {Y, _} <- [{4, 5}, {6, 7}], X + Y < 6 ].
% Risultato: [ {1, 5} ]
```

Gli elementi `X <-` e `{Y, _} <- [{4, 5}, {6, 7}]` sono generatori. `X + Y < 6` è un filtro. `{X, Y + 1}` è l'espressione il cui risultato forma gli elementi della

nuova lista.

Domanda

Record vs tuple: differenze (o loro mancanza) a run-time, vantaggi e svantaggi, implementazione di record tramite macro

Risposta

Tuple:

- In Erlang, le **tuple** sono presentate come un tipo di dato non atomico.
- Sono definite tra parentesi graffe `{}` con gli elementi separati da virgole.
- Un esempio è `{4, {ciao, 2.0}, true}`. Esiste anche la tupla vuota `{}`.
- In Rust, alcune strutture (`struct`) che non hanno campi nominati, come `struct Ref<'a, T: 'a>(&'a T);` o `struct MyBox<T>(T);`, sono descritte come **tuple**.
- Negli esempi in OCaml, le coppie (`'a * 'b`) sono utilizzate come tipi per rappresentare dati strutturati, analoghe a tuple a due elementi.
- Le tuple possono essere utilizzate nel **pattern matching** per estrarre valori in base alla loro forma e posizione.

Record:

- Storicamente, il concetto di **record** è stato introdotto nel linguaggio COBOL per la gestione aziendale, rappresentando un tipo di dato strutturato.
- Linguaggi come Go utilizzano le `struct` (`type Vector struct { X, Y float64 }`) per definire strutture dati con campi nominati, che sono l'analogo dei record.
- Anche Rust usa le `struct` con campi nominati (`struct Circle { radius : f64, }`, `struct Rectangle<T>`) per definire strutture dati.
- In Python, l'uso del decoratore `@dataclass` permette di definire classi con campi nominati (`@dataclass class Num: n: int`), che funzionano come record.
- Nel contesto delle classi di tipo in Haskell, il compilatore, durante la traduzione, crea un "record" o "dizionario" contenente i metodi associati a un'istanza di classe di tipo. Questo è un altro uso del termine legato a un dettaglio implementativo del compilatore.

Differenze (o loro mancanza) a Runtime:

- I dati che superano la dimensione di una "word" (la dimensione nativa del processore, spesso sufficiente per un puntatore) vengono solitamente allocati sullo Heap e referenziati tramite puntatori. Sia tuple che strutture con campi nominati (record/struct) rientrano in questa categoria se sufficientemente grandi.
- I dati allocati sullo Heap possono includere metadati nella loro prima cella, come tag (per distinguere i tipi di dati) e la dimensione.
- L'accesso agli elementi interni di una struttura allocata in memoria avviene tramite offset rispetto all'indirizzo base della struttura. Per esempio, nel pattern matching di una

struttura boxed in OCaml, l'accesso avviene tramite indici che corrispondono a offset.

- Per le **tuple**, l'accesso è *posizionale*. A runtime, la posizione N corrisponde a un offset pre-calcolato rispetto all'inizio della tupla.
- Per i **record/struct**, l'accesso è *nominale*. A compile time, il nome del campo viene associato a un offset pre-calcolato. A runtime, l'accesso al campo con nome 'X' si traduce nell'accesso all'indirizzo base + offset di 'X'.
- Il "record di attivazione" è una struttura runtime distinta dalle tuple o dai record di dati utente, gestita sullo stack per il controllo del flusso e la gestione delle variabili locali durante l'esecuzione delle funzioni.
- Il "record" compilato per le classi di tipo in Haskell è un dizionario di puntatori a funzioni, passato a runtime dove necessario per risolvere chiamate polimorfe. Questo è un dettaglio implementativo del sistema di tipi e non una rappresentazione generale dei dati record definiti dall'utente.

Vantaggi e Svantaggi:

- **Tuple:**
 - **Vantaggi:** Sono semplici per raggruppare un piccolo numero di valori eterogenei. La loro struttura è implicita dalla posizione.
 - **Svantaggi:** L'accesso posizionale può rendere il codice meno leggibile, poiché il significato di un elemento dipende dalla sua posizione piuttosto che da un nome esplicito. Possono diventare difficili da gestire per un numero elevato di elementi. Non è un aspetto esplicitamente discusso nelle fonti, ma è una conoscenza generale.
- **Record (Structs con campi nominati):**
 - **Vantaggi:** Migliorano la leggibilità del codice dando nomi espliciti ai campi. Questo li rende più auto-documentanti. Sono adatti per modellare entità complesse con molte proprietà distinte. Permettono di codificare invarianti all'interno di strutture dati (specialmente in combinazione con sistemi di tipi avanzati come gli ADT generalizzati), consentendo controlli statici da parte del compilatore. Offrono una struttura fissa che facilita il ragionamento sulla correttezza (a differenza dei sistemi object-based dinamici).
 - **Svantaggi:** Richiedono una definizione esplicita della struttura prima dell'uso. Potrebbero essere più verbosi per raggruppamenti molto semplici e temporanei rispetto alle tuple. Non esplicitamente discusso nelle fonti come svantaggio.

Implementazione di Record tramite Macro:

Le **macro** possono essere utilizzate per ridurre la verbosità nella definizione dei record.

Sistemi di macro più potenti del `@dataclass` di Python (come quelli presenti in Rust con `derive` o sistemi di macro Lisp) possono non solo generare costruttori o metodi base, ma anche implementare automaticamente interfacce o trait basati sulla struttura del record, automatizzare la derivazione di funzionalità (es. serializzazione/deserializzazione) e ridurre drasticamente la quantità di codice boilerplate necessario per lavorare con record complessi.

Domanda

Macro igieniche vs non-igieniche: il problema della cattura dei nomi

Risposta

Le **macro igieniche** sono definite come un meccanismo che **garantisce che le variabili introdotte nella macro non interferiscano con quelle già esistenti nel codice in cui vengono espanso**. Il loro scopo è **assicurare l'unicità delle variabili, evitando problemi di scope e collisioni di nomi**. Questo significa che il compilatore o l'interprete si occupa di gestire i nomi delle variabili all'interno dell'espansione della macro in modo che non "catturino" o vengano "catturate" da variabili con lo stesso nome definite al di fuori della macro stessa.

Il **problema della cattura dei nomi** (name capture problem) è un inconveniente comune nei sistemi di macro che non sono "igienici". Si verifica quando un identificatore (nome di variabile, funzione, ecc.) all'interno della definizione della macro ha lo stesso nome di un identificatore nel codice in cui la macro viene espansa (utilizzata). A seconda di come il sistema di macro gestisce gli scope durante l'espansione, l'identificatore all'interno della macro potrebbe legarsi all'identificatore esterno (cattura) in modo non intenzionale, o l'identificatore esterno potrebbe essere oscurato dall'identificatore interno della macro. Questo può portare a comportamenti del programma inaspettati e difficili da diagnosticare.

Le **macro igieniche** risolvono questo problema assicurando che gli identificatori introdotti dalla macro (come `temp` nell'esempio `SWAP`) siano rinominati o gestiti in modo che non collidano con gli identificatori esistenti nel contesto di utilizzo.

Vantaggi e svantaggi:

- **Macro Igieniche:**
 - **Vantaggi:** Forniscono **maggiore prevedibilità e sicurezza**. Gli utenti delle macro possono utilizzarle senza preoccuparsi di potenziali collisioni di nomi con le loro variabili locali. Questo rende il codice che usa macro igieniche **più robusto** e meno soggetto a bug difficili da trovare legati all'espansione delle macro. Evitano **problemi di scope e collisioni di nomi**.
 - **Svantaggi:** Possono essere **più complesse da implementare** per chi sviluppa il sistema di macro. In rari casi, un programmatore potrebbe *desiderare* una certa interazione tra i nomi interni della macro e l'ambiente esterno (anche se questo è spesso un cattivo design di macro), e l'igiene potrebbe ostacolarlo (anche se sistemi di macro avanzati offrono modi per disattivare selettivamente l'igiene).
- **Macro Non-Igieniche:**
 - **Vantaggi:** Sono concettualmente **più semplici da implementare** (spesso sono semplici sostituzioni testuali, come nel preprocessore C).
 - **Svantaggi:** Soffrono del **problema della cattura dei nomi**, che può portare a **bug inaspettati** e complicati, rendendo il codice che utilizza tali macro meno affidabile e più difficile da debuggare. Richiedono all'utente della macro di conoscere i nomi

interni usati dalla macro per evitarli nel proprio codice, il che viola il principio di incapsulamento.

Domanda

Esempi di comandi implementabili tramite macro ma non tramite funzioni in un linguaggio di programmazione eager (call-by-value). Possibili workaround cambiando la sintassi e passando funzioni invece di espressioni.

Risposta

I linguaggi con valutazione eager, come la maggior parte dei linguaggi mainstream (diversi da quelli **lazy** come Haskell), **valutano gli argomenti di una funzione prima di chiamare la funzione stessa**. Le fonti menzionano la differenza tra linguaggi puri e impuri in relazione agli effetti collaterali e alla prevedibilità dell'ordine di esecuzione, che è legato alla valutazione eager. Le funzioni in linguaggi funzionali possono essere entità di prima classe, possono essere argomento o risultato di altre funzioni e possono essere definite localmente, catturando variabili libere per formare **chiusure**. Una chiusura impacchetta il codice della funzione con i valori delle sue variabili libere.

Il Problema: Differenze di Valutazione

Nei linguaggi eager, una funzione opera sempre su valori già calcolati. Una macro, invece, opera a un livello diverso (solitamente pre-compilazione o compilazione) e manipola il **codice sorgente** o l'**albero di sintassi astratta** prima che avvenga la valutazione vera e propria.

Questo fa emergere una classe di "comandi" che sono facili da implementare come macro ma problematici o impossibili da implementare come funzioni in un linguaggio eager: **quelli che richiedono un controllo sulla valutazione dei loro argomenti**.

Esempi di Comandi (Macro vs. Funzione Eager):

Il caso paradigmatico è un **costrutto condizionale** che *non* valuta tutti i suoi rami.

Consideriamo l'implementazione di un semplice IF condizionale:

- **Implementazione tramite Macro:** Una macro `IF(condizione, ramo_vero, ramo_falso)` può essere definita in modo che, basandosi sul risultato della valutazione di `condizione`, espanda solo `ramo_vero` o solo `ramo_falso`. L'altro ramo **non viene mai valutato**. (Questa operazione è a livello di manipolazione del codice).
- **Implementazione tramite Funzione Eager:** Una funzione `IF(condizione, ramo_vero, ramo_falso)` in un linguaggio eager **valuterebbe sempre tutti e tre gli argomenti** (`condizione`, `ramo_vero`, e `ramo_falso`) *prima* di entrare nel corpo della funzione. Se `ramo_vero` o `ramo_falso` contengono side effect (come stampe, modifiche di stato, accessi a file/rete) o operazioni costose, queste operazioni verrebbero eseguite indipendentemente dal fatto che la condizione sia vera o falsa.

Questo è il motivo per cui costrutti come `if-then-else`, `while`, `for`, e gli operatori booleani "short-circuit" (`&&`, `||` in molti linguaggi) sono tipicamente primitive del linguaggio o implementati a basso livello dal compilatore, piuttosto che librerie di funzioni standard.

Altri esempi che sfruttano la capacità della macro di manipolare il codice o ritardarne la valutazione includono:

- **Debugging Print Statements:** Macro come `DEBUG_PRINT(espressione)` che non solo stampano il valore di `espressione` ma anche il **codice sorgente** di `espressione` stessa (es. `DEBUG_PRINT(x + y)` stampa "`x + y = 10`"). Una funzione riceverebbe solo il valore `10`, non il codice sorgente.
- **Costrutti di Controllo Personalizzati:** Sebbene complessi, alcune macro permettono di definire DSL (Domain Specific Language) o estensioni sintattiche che alterano il flusso di controllo in modi non esprimibili con semplici chiamate a funzione in un contesto eager.
- **Accesso ai Nomi delle Variabili:** Alcune macro (spesso non igieniche, ma la capacità deriva dalla manipolazione del codice) possono accedere o generare nomi di variabili, cosa impossibile per una funzione che opera su valori.

Possibili Workaround (Cambiando la Sintassi e Passando Funzioni):

Per superare la limitazione della valutazione eager e implementare "comandi" che si comportano in modo simile ai costrutti con valutazione controllata (come il condizionale) usando solo funzioni, è necessario **modificare il modo in cui gli argomenti vengono passati**.

Invece di passare direttamente i valori o i risultati delle espressioni (`ramo_vero`, `ramo_falso`), si possono passare delle **funzioni** (o **chiusure**) che, se invocate, calcolano il valore dell'espressione desiderata. Questo sposta il controllo della valutazione dal chiamante eager alla funzione chiamata.

Tornando all'esempio `IF`:

- **Sintassi Originale (Fallisce con Funzione Eager):** `IF(condizione(), EspressioneVeraConSideEffect(), EspressioneFalsaConSideEffect())` -> Tutte e tre valutate.
- **Sintassi con Workaround (Passando Funzioni):** `IF_Func(condizione(), FunzioneCheIncapsulaVero, FunzioneCheIncapsulaFalso)`

L'implementazione della funzione `IF_Func` sarebbe:

```
// In un linguaggio eager con funzioni di prima classe e chiusure
funzione IF_Func(condizione, func_vero, func_falso):
  se condizione è vera:
    restituisci func_vero() // valuta la funzione solo qui
```

```
altrimenti:  
    restituisci func_falso() // valuta la funzione solo qui
```

`FunzioneCheIncapsulaVero` e `FunzioneCheIncapsulaFalso` sarebbero delle **chiusure** create nel punto di chiamata. Per esempio, `FunzioneCheIncapsulaVero` potrebbe essere `funzione() { restituisci EspressioneVeraConSideEffect() }`. Le fonti descrivono le chiusure come l'impacchettamento di codice e variabili libere e la loro utilità nell'incapsulare dati o ritardare la valutazione.

Questo approccio richiede un **cambiamento sintattico** dal punto di vista dell'utente: invece di scrivere l'espressione direttamente, l'utente deve "avvolgerla" in una funzione (spesso una lambda anonima).

Riepilogo del Workaround:

- **Cambio di sintassi:** Gli argomenti che non devono essere valutati eageremente (come i rami di un condizionale) devono essere passati come **funzioni (chiusure)** invece che come espressioni dirette o i loro valori.
- **Implementazione con funzioni:** La funzione che implementa il comando (es. `IF_Func`) riceve queste funzioni come argomenti e le **invoca esplicitamente** solo quando necessario, ottenendo così un comportamento simile alla valutazione controllata di una macro.

Questo workaround imita la **lazy evaluation** a un livello più grossolano, applicata esplicitamente dal programmatore attraverso la creazione e il passaggio di funzioni.

Domanda

Le eccezioni: il costrutto `try .. of .. catch .. after`. Attenzione alla semantica: cosa accade se un'eccezione viene sollevata nel `try`? Se viene sollevata in un ramo dopo l'`of`? In un ramo dentro al `catch`? Nell'`after`? Se viene sollevata sia nel `catch` che nel `try`? Etc.

Risposta

Le fonti introducono il costrutto `try E of F catch G after C end` in Erlang. Questo costrutto permette di gestire eccezioni (`catch G`), risultati di successo (`of F`) e azioni di pulizia o finalizzazione che devono essere eseguite indipendentemente dall'esito (`after C`). Erlang distingue diverse tipologie di eccezioni:

- `throw(E)` : utilizzato per trasferire il controllo in modo non locale e non necessariamente per segnalare un errore. Gli Algebraic Effects si basano su questo concetto.
- `exit(E)` : utilizzato per interrompere l'esecuzione di un attore, implementando la filosofia "Let it fail!". Le fonti indicano che queste eccezioni *non dovrebbero* essere catturate con `catch`.
- "Errori non risolvibili": corrispondono a errori di runtime (es. divisione per zero).

Il meccanismo di propagazione delle eccezioni tradizionali (`throw` nel confronto con gli effetti algebrici) implica che l'istruzione `throw` cerca un gestore risalendo lo stack di chiamate e rimuovendo i record di attivazione finché non ne trova uno (`Stack.pop()`).

Analizziamo ora la semantica del costrutto `try...of...catch...after` nei vari scenari basandoci sulle fonti e integrando con conoscenze esterne dove necessario. La struttura pseudocodice fornita per l'implementazione dell' `after` è particolarmente utile per comprendere il flusso: `case try {return, try E of F catch G} catch {Exc -> {exception, Exc}} end of {return, V} -> C, V ; {exception, Exc} -> C, throw(Exc) end`. Questa struttura implica che il blocco `try E of F catch G` è protetto, e il suo risultato (normale o eccezione catturata) viene gestito *dopo* che l'inner `try...catch` è terminato, e l' `after C` viene eseguito prima di propagare ulteriormente il risultato o l'eccezione.

Semantica in Diversi Scenari:

1. Eccezione sollevata nel `try E`:

- Il codice nel blocco `try E` viene eseguito.
- Se `E` solleva un'eccezione (es. un errore come divisione per zero o un `throw`), l'esecuzione di `E` si interrompe e il controllo passa al blocco `catch G`.
- L'eccezione sollevata viene confrontata con i pattern definiti nel blocco `catch G`.
- Se viene trovato un pattern corrispondente, il codice associato in `G` viene eseguito.
- Indipendentemente dal fatto che il codice in `G` gestisca completamente l'eccezione o sollevi una nuova eccezione/re-throw, il blocco `after C` viene **sempre** eseguito.
- Dopo l'esecuzione di `after C`, se il blocco `catch G` ha gestito l'eccezione e completato normalmente, il risultato del `catch` diventa il risultato dell'intero costrutto `try...end`. Se il blocco `catch G` ha sollevato una nuova eccezione o re-thrown quella originale, questa nuova eccezione viene propagata ulteriormente dopo l'esecuzione di `C`.
- Se l'eccezione sollevata in `try E` non corrisponde a nessun pattern nel `catch G`, l'eccezione non viene catturata da questo blocco `catch`. In questo caso, l'esecuzione del `try...of...catch` viene interrotta, il blocco `after C` viene comunque eseguito, e l'eccezione originale non catturata viene propagata all'esterno del costrutto `try...end`.

2. Eccezione sollevata in un ramo dopo l' `of F`:

- Il codice nel blocco `try E` viene eseguito **senza sollevare eccezioni**.
- Il valore di ritorno del blocco `try E` viene confrontato con i pattern definiti nel blocco `of F`.
- Viene eseguito il codice associato al pattern corrispondente in `F`.
- Se l'esecuzione di questo codice solleva un'eccezione, il controllo passa al blocco `catch G`. Le fonti indicano che sia `E` che i rami di `of` sono protetti dal `catch`.
- Da questo punto, la gestione è analoga al caso 1 (Eccezione sollevata nel `try E`): l'eccezione viene confrontata con i pattern di `G`, il blocco `after C` viene eseguito, e

il risultato (valore o eccezione propagata) dipende dall'esito del `catch G`.

3. Eccezione sollevata in un ramo dentro al `catch G`:

- Un'eccezione è stata sollevata in `try E` o in un ramo di `of F`, ed è stata catturata da un pattern in `catch G`.
- Il codice associato a quel pattern in `G` viene eseguito.
- Se questo codice solleva una **nuova** eccezione (o re-throw l'eccezione catturata), l'esecuzione del `catch G` si interrompe.
- Il blocco `after C` viene **sempre** eseguito.
- Dopo l'esecuzione di `after C`, la nuova eccezione sollevata nel `catch G` (o quella re-thrown) viene propagata all'esterno del costrutto `try...end`.

4. Eccezione sollevata nell' `after C`:

- Il blocco `try...of...catch` (sia il percorso di successo tramite `of` sia il percorso di gestione dell'eccezione tramite `catch`) è terminato.
- Il blocco `after C` viene eseguito.
- Se il codice nel blocco `after C` solleva un'eccezione, questa eccezione ha generalmente la priorità su qualsiasi risultato (normale o eccezione) che sarebbe stato propagato dal blocco `try...of...catch` precedente. L'eccezione sollevata in `after` si propaga **immediatamente** all'esterno del costrutto `try...end`. La pseudo-codice fornita non mostra esplicitamente questo caso (mostra `C` che precede `throw(Exc)`, ma non cosa succede se `C` stessa lancia un'eccezione), ma è la semantica standard dei blocchi `finally` in linguaggi simili. Il valore di ritorno di `C` viene comunque ignorato.

5. Eccezione sollevata sia nel `try E` che nel `catch G`:

- Un'eccezione `Exc1` viene sollevata in `try E`.
- Controllo passa al blocco `catch G`, e `Exc1` corrisponde a un pattern.
- Il codice associato in `G` inizia l'esecuzione.
- Durante l'esecuzione del codice in `G` (che sta gestendo `Exc1`), viene sollevata una **nuova** eccezione `Exc2` (o il codice in `G` re-throw un'eccezione, che per semplicità consideriamo `Exc2`).
- L'eccezione `Exc2` interrompe l'esecuzione di `G`. L'eccezione originale `Exc1` che veniva gestita viene scartata.
- Il blocco `after C` viene **sempre** eseguito.
- Dopo l'esecuzione di `after C`, l'eccezione `Exc2` sollevata in `G` viene propagata all'esterno del costrutto `try...end`.

6. Eccezione sollevata sia nel `try E` che nell' `after C`:

- Un'eccezione `Exc1` viene sollevata in `try E`.
- Controllo passa al blocco `catch G`, e `Exc1` corrisponde a un pattern.
- Il codice associato in `G` viene eseguito e **gestisce con successo** `Exc1` (ovvero, termina normalmente o restituisce un valore senza sollevare una nuova eccezione).
- Il blocco `after C` viene eseguito.
- Durante l'esecuzione del codice in `C`, viene sollevata una nuova eccezione `Exc2`.

- L'eccezione `Exc2` sollevata in `after C` si propaga **immediatamente** all'esterno del costrutto `try...end`, ignorando il fatto che `Exc1` sia stata gestita con successo.

7. Eccezione sollevata sia in un ramo dell' `of F` che nell' `after C` :

- Il blocco `try E` viene eseguito con successo.
- Il risultato di `E` corrisponde a un pattern in `of F`, e il codice associato viene eseguito.
- Durante l'esecuzione di questo codice in `F`, viene sollevata un'eccezione `Exc1`.
- Controllo passa al blocco `catch G` (poiché i rami di `of` sono protetti), e `Exc1` corrisponde a un pattern.
- Il codice associato in `G` viene eseguito e **gestisce con successo** `Exc1`.
- Il blocco `after C` viene eseguito.
- **Conoscenza Esterna:** Durante l'esecuzione del codice in `C`, viene sollevata una nuova eccezione `Exc2`.
- L'eccezione `Exc2` sollevata in `after C` si propaga **immediatamente** all'esterno del costrutto `try...end`, ignorando il fatto che `Exc1` sia stata gestita.

Domanda

Come vengono implementate le eccezioni? Come interferiscono con la tail-call optimization?

Risposta

Implementazione delle Eccezioni

Le eccezioni in Erlang influiscono sulla struttura dello stack di esecuzione. In un linguaggio con eccezioni, uno **stack frame** può corrispondere sia a un normale record di attivazione di una funzione, sia a un **record try catch**.

All'interno di un record try catch, troviamo un puntatore al codice del blocco `catch` e un tag che lo identifica come tale ("record try catch").

Un record di attivazione standard, invece, contiene i campi tipici (variabili locali, indirizzo di ritorno, ecc.) e un tag che lo identifica come "record di attivazione".

Quando viene sollevata un'eccezione, ad esempio tramite `throw(E)`, il sistema cerca un gestore percorrendo lo stack. Questo processo, mostrato in pseudo-codice, implica un ciclo che **rimuove (pop) gli stack frame** fino a quando non ne trova uno con il tag "record try-catch". Una volta trovato il record try-catch, il sistema recupera l'indirizzo del codice associato al blocco `catch`, rimuove anch'esso dallo stack e chiama il codice del gestore (`CALL addr(E)`). Il codice del `catch` analizza quindi l'eccezione per determinare se gestirla o meno.

Il costrutto `try E of F catch G after C end` in Erlang è progettato per gestire sia l'esecuzione normale (`of F`) che le eccezioni (`catch G`), garantendo al contempo l'esecuzione di codice di pulizia o finalizzazione (`after C`). Le fonti mostrano una possibile implementazione concettuale dell' `after`, che implica che il blocco `try E of F catch G`

viene eseguito in un contesto protetto, e il blocco `after C` viene eseguito **dopo** questo, indipendentemente dal suo esito (successo o eccezione gestita).

Interferenza con la Tail-Call Optimization (TCO)

La **Tail-Call Optimization (TCO)** è una tecnica di compilazione che trasforma le chiamate di coda (`tail calls`) in salti (`JUMP`), riutilizzando il record di attivazione corrente anziché crearne uno nuovo. Una chiamata a una funzione `g()` all'interno di una funzione `f()` è considerata di coda se l'unica istruzione eseguita dopo la chiamata a `g()` è un'istruzione di `return`, e il valore restituito da `f()` è esattamente il valore restituito da `g()`. Quando viene applicata la TCO, una chiamata di coda ha un costo computazionale di $O(1)$ in tempo e " $O(0)$ " in spazio sullo stack.

Le eccezioni **interferiscono** con la TCO. Le fonti specificano esplicitamente una condizione aggiuntiva per una chiamata di funzione affinché sia considerata di coda (e quindi ottimizzabile): **"Non è contenuta all'interno della parte valutativa di un blocco try ... catch (ovvero non è protetta)".**

Questo vincolo deriva direttamente dal meccanismo di gestione delle eccezioni descritto in precedenza. Se una chiamata all'interno di un blocco `try` fosse ottimizzata come chiamata di coda, il suo record di attivazione verrebbe riutilizzato e il precedente record di attivazione (potenzialmente un record `try catch` necessario per catturare un'eccezione) verrebbe rimosso dallo stack prematuramente. Se successivamente venisse sollevata un'eccezione all'interno della funzione chiamata, non ci sarebbe più un record `try catch` sullo stack per catturarla, violando la semantica del costrutto. Pertanto, il compilatore non può applicare la TCO a chiamate che si trovano all'interno della parte protetta di un blocco `try`.

Un altro punto di interferenza menzionato è l'uso del blocco `after`. Le fonti indicano che l'utilizzo del costrutto `try...of...catch...after` comporta la **perdita dell'ottimizzazione per le chiamate di coda**. Questo accade perché il blocco `after C` deve essere eseguito *dopo* il completamento del blocco `try E of F catch G`. Per garantire l'esecuzione di `C`, la struttura logica (riflessa anche nel pseudo-codice) richiede che il flusso di controllo ritorni in qualche modo al gestore dell'intero costrutto `try...end` *prima* di una potenziale chiamata ricorsiva di coda, rompendo così la condizione che la chiamata ricorsiva sia l'ultima istruzione eseguita.

Esempio di Superamento del Limite

Le fonti mostrano come sia possibile **recuperare la TCO** anche in presenza di codice che può sollevare eccezioni, strutturando il programma in modo che la chiamata ricorsiva avvenga *al di fuori* del blocco `try...catch`. Nell'esempio del conto corrente `cc`, il blocco `try receive ... catch ... end` viene utilizzato per gestire potenziali eccezioni (come la divisione per zero). Il risultato di questo blocco `try...catch` è un valore che indica l'azione successiva desiderata: `{result, R}` per terminare o `{recur, Bal}` per continuare ricorsivamente. Questo risultato viene poi gestito da un'istruzione `case` esterna al `try...catch`. La chiamata ricorsiva `cc(Bal)` avviene in un ramo di questa `case` esterna,

che non è protetta dal `try...catch`. In questo modo, la chiamata `cc(Bal)` diventa una chiamata di coda e l'ottimizzazione può essere applicata.

Domanda

Costrutti per la programmazione concorrente: send asincrona, receive asincrona, spawning di un nuovo processo, link/unlink, monitor/demonitor e intercettamento degli EXIT message per processi linked/monitored

Risposta

La concorrenza in Erlang si basa sul concetto di **attori (o processi Erlang)**, che sono entità di computazione indipendenti che non condividono memoria o stato, nemmeno quando eseguite sulla stessa macchina. L'unico modo per gli attori di comunicare è attraverso lo **scambio di messaggi asincroni**.

Ecco i costrutti principali descritti nelle fonti:

1. Spawning (Creazione di un Nuovo Processo):

- Per creare un nuovo attore (processo Erlang), si utilizza la funzione `spawn(NomeModulo, NomeFunzione, ListaArgomenti)`.
- Questo comando avvia l'esecuzione della funzione specificata nel modulo dato come un nuovo processo indipendente.
- Il comando `spawn` **restituisce il PID (Process Identifier)** del nuovo attore appena creato.
- Il PID è l'identificatore univoco dell'attore, fungendo da nome logico per raggiungerlo.
- Un attore corrisponde a un singolo thread di computazione; all'interno di un attore non possono esistere thread multipli tradizionali. I processi Erlang sono "green thread" leggeri gestiti dalla Beam VM, non thread del sistema operativo.

2. Send Asincrona (Invio di Messaggi):

- Gli attori comunicano inviando messaggi in modo **asincrono** ad altri attori di cui conoscono il PID.
- La sintassi per inviare un messaggio è `PID ! Messaggio`.
- L'invio è *esclusivamente* asincrono, il che significa che non esiste sincronizzazione diretta tra mittente e destinatario; il mittente non sa se o quando il messaggio verrà ricevuto.
- Questa è l'unica forma di comunicazione diretta tra attori in Erlang.
- I messaggi inviati vengono depositati nella **Mailbox** dell'attore destinatario, che è una coda per la ricezione dei messaggi.

3. Receive Asincrona (Ricezione di Messaggi):

- Gli attori ricevono ed elaborano i messaggi presenti nella loro mailbox utilizzando il costrutto `receive...end`.
- All'interno di `receive`, vengono specificati una serie di **pattern** che vengono confrontati con i messaggi presenti nella mailbox.

- Quando un messaggio nella mailbox corrisponde a un pattern, il codice associato a quel pattern viene eseguito, e il messaggio viene consumato dalla mailbox.
- Il pattern matching può estrarre valori dal messaggio associandoli a variabili.
- I diversi casi di pattern matching all'interno di `receive` sono separati da punto e virgola.
- Se nessun messaggio nella mailbox corrisponde a nessuno dei pattern specificati, l'attore entra in uno stato di attesa finché un messaggio che corrisponde non arriva o finché non scade un eventuale timeout specificato tramite una clausola `after`.

4. Link e Unlink:

- La primitiva `link(PID)` crea un collegamento tra l'attore corrente e l'attore identificato da `PID`.
- Il collegamento è **bidirezionale**: se uno dei due attori linkati termina (per qualsiasi ragione, inclusa un'eccezione non gestita), l'altro attore viene terminato a sua volta. Questo meccanismo è fondamentale per la strategia di tolleranza ai guasti "Let it fail!" di Erlang, spesso utilizzato per creare strutture gerarchiche di supervisione.
- Il link è anche **idempotente e transitivo**.
- La funzione `spawn_link()` combina l'operazione di `spawn` e `link` in un'unica chiamata atomica per evitare race condition.
- La primitiva `unlink(PID)` rimuove il collegamento tra due attori precedentemente linkati.

5. Monitor e Demonitor:

- La primitiva `monitor(PID)` crea un collegamento più debole rispetto a `link`. Restituisce un handle per l'istanza del monitoraggio.
- Il monitoraggio è **unidirezionale**: l'attore che monitora viene informato se l'attore monitorato termina, ma la terminazione dell'attore monitorato *non* causa la terminazione dell'attore che sta monitorando.
- Il monitoraggio non è idempotente e non è transitivo.
- È utile per osservare il comportamento di un altro attore senza creare una dipendenza forte che ne causi la terminazione in caso di fallimento, ad esempio quando diverse parti indipendenti del sistema dipendono da un servizio comune.
- La funzione `spawn_monitor()` combina `spawn` e `monitor` in modo atomico.
- La primitiva `demonitor(Handle)` rimuove un monitoraggio precedentemente stabilito, utilizzando l'handle restituito da `monitor`.

6. Intercettamento degli EXIT message (Trapping Exits):

- Normalmente, la terminazione di un processo linkato si propaga causando la terminazione degli attori linkati.
- Un attore può modificare questo comportamento impostando un flag di processo con la primitiva `process_flag(trap_exit, true)`.
- Quando questo flag è impostato su `true`, invece di terminare in caso di ricezione di un segnale di uscita da un processo linkato, l'attore **riceve il segnale come un normale messaggio nella sua mailbox**.

- Questo messaggio di uscita ha tipicamente la forma `{'EXIT', Pid, Reason}`, dove `Pid` è l'identificatore del processo terminato e `Reason` indica la causa della terminazione (es. `normal`, `noproc`, o la ragione di un'eccezione non gestita).
- Questo permette all'attore di gestire la terminazione del processo linkato in modo programmatico, ad esempio riavviandolo o intraprendendo altre azioni di recupero.
- Una notevole eccezione è la ragione `kill` per la funzione `exit(PID, kill)`, che bypassa sempre il `trap_exit` e causa la terminazione immediata dell'attore.

Domanda

Registrazione/pubblicazione del PID su un singolo nodo e in ambiente distribuito

Risposta

In Erlang, ogni processo (attore) è identificato da un **PID (Process Identifier)**, che è un identificatore univoco. Il PID funge da **nome logico** che indica come raggiungere l'attore. L'unico modo per comunicare con un attore, la cui posizione è nascosta dal linguaggio, è conoscere il suo nome logico (il PID). La funzione `spawn(NomeModulo, NomeFunzione, ListaArgomenti)` viene utilizzata per creare un nuovo attore e **restituisce il PID** dell'attore appena creato. La funzione `self()` restituisce il PID del processo corrente.

Gestione dei PID su un Singolo Nodo:

Su un singolo nodo (una singola istanza della Beam VM), i processi possono comunicare direttamente inviando messaggi al PID del destinatario tramite la sintassi `PID ! Messaggio`.

Tuttavia, i PID sono valori effimeri; cambiano ogni volta che un processo viene creato o, se termina e viene riavviato (ad esempio, da un supervisore secondo il principio "Let it fail!"), avrà un nuovo PID. Per fornire un modo più stabile per fare riferimento ai processi, in particolare quelli che forniscono servizi noti, è possibile **registrare** un processo con un nome atomico locale.

Le fonti menzionano il costrutto `register()` come un "costrutto imperativo" che consente di utilizzare un atomo per comunicare con l'attore designato. Ad esempio, nel codice di un worker, `scheduler ! {get, REF, self()}` invia un messaggio all'attore registrato con il nome atomico `scheduler`. Nell'esempio di comunicazione distribuita, un PID viene registrato con l'atomo `shell` su un nodo. Questo meccanismo permette ad altri processi sullo stesso nodo di riferirsi al processo registrato utilizzando il nome atomico anziché il suo PID corrente.

Gestione dei PID in un Ambiente Distribuito:

Erlang è progettato per la programmazione distribuita e può funzionare con più istanze della Beam VM (nodi) interconnesse. Per abilitare la comunicazione tra nodi, questi devono essere avviati con nomi specifici usando le flag `-sname nome` (short name) o `-name nome`. È possibile collegare i nodi tra loro, ad esempio usando `net_adm:ping(nome_attore@nome_dispositivo)`, e questa connessione è transitiva.

In un ambiente distribuito, la comunicazione con un processo che si trova su un nodo diverso avviene utilizzando una combinazione del nome atomico con cui il processo è registrato *su quel nodo* e il nome del nodo stesso. Le fonti mostrano un esempio di invio di un messaggio da un nodo all'attore registrato come `shell` su un altro nodo, utilizzando la sintassi

```
{shell, nome_altro_attore@nome_altro_dispositivo} ! {ciao, self()}.
```

Questo meccanismo implica che:

1. È necessario **registrare** il processo con un nome atomico **sul nodo dove risiede** se si vuole che altri processi (anche da altri nodi) lo raggiungano facilmente con un nome stabile.
2. Per inviare un messaggio a un processo registrato su un nodo remoto, si utilizza la tupla `{NomeRegistrato, NomeNodo}` come destinatario. Il runtime di Erlang si occupa di **instradare il messaggio** al nodo corretto basandosi sul `NomeNodo` e, una volta raggiunto quel nodo, di consegnarlo al processo registrato con `NomeRegistrato`. Sebbene i PID stessi possano apparire diversi tra nodi, il runtime è in grado di gestire correttamente l'indirizzamento.
3. È anche possibile **spawning di attori direttamente su un altro nodo**. Questo comando probabilmente restituisce un PID locale che il runtime sa come mappare e indirizzare verso il processo sul nodo remoto.

Domanda

Mobilità del codice: in quali situazioni migrare il codice verso un altro nodo è consigliabile? Come si può implementare in Erlang la migrazione del codice?

Risposta

Mobilità del Codice in Erlang (Migrazione di Attori)

In Erlang, la migrazione diretta degli attori (processi) tra nodi **non è una primitiva nativa del linguaggio**. Erlang di base non permette di migrare attori tra due nodi. Tuttavia, è possibile implementare meccanismi per **spostare il codice** e lo stato di un attore verso un altro nodo utilizzando i costrutti esistenti.

Situazioni in cui Migrare il Codice è Consigliabile:

Le fonti indicano due scenari principali in cui la migrazione del codice (o più precisamente, la ricreazione di un attore con il suo stato su un nodo diverso) può essere utile:

1. **Bilanciamento del carico:** Spostare attori da un nodo sovraccarico a un nodo meno impegnato può aiutare a distribuire il lavoro e migliorare le prestazioni complessive del sistema.
2. **Prossimità geografica rispetto ai dati:** Se un attore deve elaborare grandi quantità di dati residenti su un nodo specifico, può essere più efficiente spostare l'attore vicino ai dati piuttosto che trasferire continuamente i dati verso l'attore.

Come Implementare la Migrazione del Codice in Erlang:

Sebbene la migrazione automatica non sia integrata, è possibile implementarla manualmente seguendo determinati "pattern". L'approccio descritto si basa sulla creazione di un **nuovo attore sul nodo di destinazione** che esegue la stessa funzione (e gestisce lo stato) dell'attore originale, mentre l'attore originale funge temporaneamente da intermediario.

Ecco come viene illustrato l'esempio nelle fonti:

1. **Spawning Remoto:** Viene creata una nuova istanza dell'attore sul nodo di destinazione (`NODE`) utilizzando `spawn(NODE, F)`. `F` è la funzione che rappresenta il comportamento dell'attore e contiene lo stato (`Dati`).
2. **Funzione "Man-in-the-Middle" (`Forward`):** L'attore originale (quello che ha richiesto la migrazione) non termina immediatamente. Al contrario, inizia a eseguire una funzione ricorsiva, chiamata `Forward` nell'esempio.
3. **Inoltro dei Messaggi:** La funzione `Forward` riceve messaggi destinati all'attore originale e li inoltra (`PID ! Msg`) al nuovo attore creato sul nodo remoto. Questo garantisce che le comunicazioni con l'attore non vengano interrotte durante o dopo il processo di "migrazione".
4. **Gestione del "Comeback":** La funzione `Forward` include anche un pattern per gestire un messaggio `{comeback, F}`. Questo suggerisce la possibilità di implementare un meccanismo inverso per riportare l'esecuzione (e lo stato) all'attore originale o a un altro nodo, richiamando la funzione `F` localmente.
5. **Trasferimento del Byte Code:** Affinché questo meccanismo funzioni, il **Byte Code** relativo alla funzione dell'attore che si desidera migrare deve essere trasferito dalla Beam VM di origine a quella di destinazione.

Questo approccio permette di "spostare" l'esecuzione di una logica specifica e il suo stato associato a un altro nodo, offrendo una forma di mobilità del codice implementata a livello applicativo utilizzando le potenti primitive di comunicazione e spawning di Erlang. È importante notare che, a differenza di una vera migrazione di processo che sposterebbe lo stato esatto dell'attore in esecuzione (stack, mailbox, ecc.), questo pattern ricrea l'attore sul nodo di destinazione e richiede che lo stato venga passato come argomento alla funzione ricorsiva.

Domanda

Come stabilire una connessione fra più nodi Erlang e come registrare PID in modo da contattare attori in esecuzione su altri nodi.

Risposta

In Erlang, ogni programma concorrente è un **attore**. Gli attori comunicano esclusivamente tramite lo **scambio asincrono di messaggi**. Ogni attore è identificato in modo univoco da un **PID (Process Identifier)**, che funge da **nome logico** per raggiungerlo.

Stabilire una Connessione tra più Nodi Erlang:

Erlang è progettato per sviluppare sistemi distribuiti. Un'istanza della macchina virtuale Erlang (Beam) è chiamata **nodo**. Per consentire la comunicazione tra nodi distinti (che possono trovarsi sulla stessa macchina fisica o su macchine diverse in una rete), è necessario che questi nodi siano avviati con nomi specifici.

1. **Avvio dei Nodi con Nomi:** Quando si avvia l'ambiente Erlang (`erl`), è fondamentale utilizzare i flag di nome per identificare il nodo:

- `-sname NomeBreve` : Utilizzato per nomi brevi (senza dominio).
- `-name NomeLungo` : Utilizzato per nomi lunghi (generalmente nel formato `nome@host.dominio`). Le fonti suggeriscono che in alcuni casi sia più conveniente usare solo `-name` . Avviando `erl -sname mio_nodo` , il nodo avrà il nome `mio_nodo@nome_dispositivo` .

2. **Connessione tra Nodi:** Una volta che i nodi sono attivi e nominati, possono essere collegati:

- Si utilizza la funzione `net_adm:ping(NomeNodoRemoto)` . Ad esempio, da `mio_nodo` si potrebbe chiamare `net_adm:ping(altro_nodo@nome_altro_dispositivo)` .
- Se la connessione ha successo, la funzione restituisce `pong` . Se fallisce, restituisce `pong` .
- Una volta stabilita una connessione tra due nodi, questa connessione è **transitiva**. Ciò significa che se `NodoA` è connesso a `NodoB` e `NodoB` è connesso a `NodoC`, allora `NodoA` può comunicare con `NodoC` tramite `NodoB`.
- È possibile verificare l'elenco dei nodi attualmente collegati tramite il comando `nodes()` .

Registrazione e Contatto di Attori su altri Nodi:

Il PID di un attore è l'identificatore fondamentale. Quando si crea un attore con `spawn(NomeModulo, NomeFunzione, ListaArgomenti)` , la funzione restituisce il **PID** dell'attore appena creato. Questo PID è il "nome logico" per comunicare con quell'attore.

I PID, tuttavia, sono effimeri: cambiano se un processo termina e viene riavviato (ad esempio, per gestione dei guasti secondo il principio "Let it fail!"). Per fornire un modo più stabile per fare riferimento a processi noti, specialmente in un sistema distribuito, Erlang fornisce un meccanismo di **registrazione**.

1. **Registrazione del PID su un Nodo:** È possibile associare un **nome atomico** locale a un PID specifico su un nodo utilizzando il costrutto `register(AtomoNome, PID)` . Questo crea una sorta di "DNS locale" per l'attore su quel nodo. L'atomo (scritto in minuscolo) diventa un alias stabile per il PID corrente dell'attore registrato.

- Esempio nelle fonti: `register(scheduler, SCHED)` registra il PID contenuto nella variabile `SCHED` con il nome atomico `scheduler` .
- Il costrutto `register()` è imperativo.

- Non è possibile registrare due volte lo stesso atomo sullo stesso nodo; ciò solleverebbe un'eccezione.
- La registrazione può essere rimossa con `unregister(AtomoNome)`.

2. **Contattare Attori Registrati su Nodi Remoti:** Una volta che un attore su un nodo remoto è stato registrato con un nome atomico, è possibile inviargli messaggi da un altro nodo utilizzando una sintassi specifica:

- La sintassi per inviare un messaggio è `{NomeRegistrato, NomeNodoRemoto} ! Messaggio`.
- Il `NomeNodoRemoto` è il nome completo del nodo, nel formato `nome@nome_dispositivo`.
- Esempio dalle fonti: `{shell, nome_altro_attore@nome_altro_dispositivo} ! {ciao, self()}` invia il messaggio `{ciao, self()}` all'attore registrato come `shell` sul nodo specificato.
- Il runtime di Erlang gestisce l'instradamento del messaggio: identifica il nodo di destinazione basandosi sul `NomeNodoRemoto` e, una volta raggiunto quel nodo, utilizza il `NomeRegistrato` per trovare il PID dell'attore destinatario e consegnare il messaggio. Sebbene il PID possa apparire diverso su nodi diversi, il runtime sa sempre come indirizzare correttamente l'attore.

Domanda

Il modello di sicurezza di Erlang basato su token (e perché per lanciare due nodi su una stessa macchina non è necessario specificare alcun token)

Risposta

Modello di Sicurezza e il Token in Erlang Distribuito

Le fonti affrontano direttamente il tema della sicurezza tra attori (processi) in un sistema distribuito Erlang, affermando esplicitamente che **Erlang di base non garantisce alcun tipo di sicurezza built-in**. I messaggi scambiati tra gli attori vengono inviati **in chiaro**.

Esiste un meccanismo basato su una **chiave** (spesso chiamata "cookie" e presente nel file `.erlang.cookie`) che viene scambiata durante la comunicazione tra diversi cluster. Tuttavia, la fonte sottolinea che **anche questa chiave viene inviata in chiaro (quindi facilmente intercettabile)**.

Pertanto, il "modello di sicurezza" fornito da questo token è minimale e si limita a una forma di **autenticazione rudimentale tra nodi**: due nodi possono stabilire una connessione solo se condividono lo stesso cookie segreto. Se i cookie non corrispondono durante il processo di handshake iniziale, la connessione viene rifiutata. Questo impedisce a nodi non autorizzati (che non conoscono il cookie) di collegarsi a un cluster. Tuttavia, la trasmissione in chiaro del cookie stesso lo rende vulnerabile a intercettazioni.

Perché per Lanciare Due Nodi sulla Stessa Macchina non è Necessario Specificare Esplicitamente il Token?

Le fonti descrivono come avviare nodi con nomi (`-sname` o `-name`) e come collegarli utilizzando `net_adm:ping/1`. Affermano che è possibile lanciare più istanze della macchina virtuale Beam sulla stessa macchina fisica, e che queste vengono identificate come nodi diversi. Menzionano la chiave/cookie che viene scambiata "durante la comunicazione tra diversi cluster".

La ragione per cui questo accade è che il meccanismo del cookie è comunque **necessario** per l'autenticazione tra *qualsiasi* due nodi Erlang distinti che tentano di connettersi, indipendentemente dal fatto che si trovino sulla stessa macchina fisica o su macchine diverse in rete. Tuttavia, in un setup standard su una singola macchina, se i nodi vengono avviati dallo stesso utente, il runtime Erlang cerca automaticamente il file `.erlang.cookie` nella directory home dell'utente e utilizza quella chiave per entrambi i nodi. Di conseguenza, i nodi condividono già lo stesso cookie senza che l'utente debba specificarlo esplicitamente sulla riga di comando, consentendo la connessione. La frase nella fonte che menziona "diversi cluster" potrebbe portare a pensare che il cookie sia richiesto solo per reti geograficamente distribuite, ma in realtà è un requisito per l'autenticazione di base tra *ogni* coppia di nodi che si connettono. Se si avviano nodi con utenti diversi o da ambienti che non condividono lo stesso file `.erlang.cookie`, o se si vuole usare un cookie diverso da quello predefinito, è necessario specificarlo esplicitamente (ad esempio, con il flag `-cookie NomeCookie` all'avvio di `erl`).

Domanda

Comunicazione con librerie e processi esterni: il meccanismo basato su porte.

Risposta

Le fonti spiegano che in Erlang, l'unità fondamentale di computazione è l'**attore** (o processo). Gli attori comunicano tra loro esclusivamente tramite lo **scambio asincrono di messaggi**, identificandosi a vicenda tramite il loro **PID (Process Identifier)**, che funge da nome logico.

Tuttavia, il mondo esterno al runtime Erlang (Beam) non è composto da attori. Per interagire con entità esterne che non sono attori, come librerie scritte in altri linguaggi (ad esempio, C/C++) o processi del sistema operativo, Erlang fornisce un meccanismo apposito.

Il Meccanismo Basato su Porte:

Le fonti specificano che, nel modello ad attori di Erlang, quando è necessario interagire con entità esterne che **non sono attori**, è possibile "avvolgerle" in una sorta di **attore intermediario**. Questo attore speciale funge da **wrapper** per l'entità esterna.

1. **Identificazione:** A differenza degli attori "normali" che sono identificati da un PID, a questi attori speciali che incapsulano entità esterne viene assegnata una **porta** anziché un PID.
2. **Comunicazione tramite Messaggi:** L'interazione con queste entità esterne avviene comunque utilizzando i meccanismi standard di **invio e ricezione di messaggi** tipici del

linguaggio Erlang. L'attore Erlang comunica con la porta (l'intermediario), e la porta gestisce la comunicazione effettiva con l'entità esterna, traducendo i messaggi Erlang nel formato atteso dall'esterno e viceversa.

3. **Distanza dal Modello "Let it fail!":** Un aspetto cruciale evidenziato dalle fonti è che questi attori speciali che fungono da wrapper per entità esterne **non possiedono tutte le caratteristiche degli attori normali**. In particolare, essi **non seguono il principio "Let it fail!"**. Il principio "Let it fail!" è una strategia di gestione degli errori in Erlang dove, in caso di guasto, un processo e i suoi collegati (linkati o supervisionati) vengono terminati, e un supervisore si occupa eventualmente di riavviarli. Poiché l'entità esterna non è gestita dalla Beam, l'attore intermediario (la porta) non può applicare questa logica di riavvio o terminazione a catena allo stesso modo di un attore Erlang puro.

Le porte in Erlang sono comunemente utilizzate per interagire con programmi esterni eseguiti come processi separati ("External Port Programs"). La comunicazione avviene tipicamente tramite lo standard input e output del processo esterno. I messaggi Erlang vengono serializzati in un formato binario e inviati al processo esterno via standard input, e la porta Erlang corrispondente legge i messaggi serializzati dallo standard output del processo esterno. Questo richiede che il programma esterno sia scritto in modo da leggere e scrivere dati in questo formato specifico. Le porte possono essere utilizzate anche per interazioni più dirette con il sistema operativo o dispositivi, o per implementare funzionalità native che non sono convenienti o possibili da scrivere in Erlang puro. Poiché il runtime Erlang non controlla l'esecuzione del codice esterno, un fallimento nel processo esterno non può essere gestito con la stessa granularità o affidabilità del principio "Let it fail!" applicato agli attori Erlang. L'attore Erlang connesso alla porta riceverà una notifica di uscita se il processo esterno termina (ad esempio, `{'EXIT', Port, Reason}`), ma la responsabilità di gestire questa situazione (es. riavviare il processo esterno, loggare l'errore) ricade sul codice Erlang.

Domanda

Context switch, cooperative multi-tasking, preemptive multi-tasking

Risposta

Le fonti descrivono l'architettura di esecuzione di Erlang. La macchina virtuale di Erlang, chiamata Beam, viene eseguita tipicamente come un unico processo del sistema operativo, il quale può essere multi-threaded con un thread per core del processore. All'interno di questa macchina virtuale (la Beam), gli attori di Erlang sono implementati come **Language Thread**. Il sistema operativo non ha diretta consapevolezza di questi Language Thread; la loro schedulazione sui thread del kernel è gestita direttamente dalla macchina virtuale Erlang.

- **Context Switch (Cambio di Contesto):** Il contesto di un processo o thread include tutte le informazioni necessarie per riprendere la sua esecuzione (stato dei registri, program counter, ecc.). Un cambio di contesto si verifica quando il sistema (l'OS o, nel caso di

Erlang, la Beam) mette in pausa l'esecuzione di un thread e ne riprende un altro. Le fonti affermano esplicitamente che il **cambio di contesto per i Language Thread in Erlang è "estremamente leggero"**. Questo è un vantaggio chiave che consente a un sistema Erlang di gestire un numero massiccio di attori/Language Thread (centinaia di migliaia o persino milioni).

- **Cooperative Multi-tasking (Multi-tasking Cooperativo):** Le fonti indicano chiaramente che Erlang implementa un **"cambio di contesto collaborativo"**. Questo è il principio fondamentale del multi-tasking cooperativo. In un sistema cooperativo, un thread cede volontariamente il controllo del processore quando ha finito un compito o raggiunge un punto logico in cui è opportuno farlo (ad esempio, in attesa di un messaggio nella sua mailbox). La responsabilità della schedulazione e del cambio di contesto ricade sulla macchina virtuale (la Beam). Le fonti associano i Language Thread Erlang a questo modello, definendoli **"green thread"**, che utilizzano "risorse minime". Questo modello si adatta bene alla natura degli attori di Erlang, che tipicamente eseguono brevi operazioni in risposta a un messaggio e poi tornano in attesa.
- **Preemptive Multi-tasking (Multi-tasking Preemptive):** Il multi-tasking preemptive è il modello di schedulazione più comune nei sistemi operativi moderni. In questo modello, il sistema operativo (o lo scheduler) può interrompere l'esecuzione di un thread in qualsiasi momento, anche se questo non ha completato la sua operazione o non ha ceduto volontariamente il controllo. Il sistema operativo riprende il controllo e decide quale thread eseguire successivamente, basandosi su algoritmi di schedulazione e priorità. Il cambio di contesto in un sistema preemptive a livello di OS è generalmente più "pesante" o "costoso" rispetto al cambio di contesto leggero dei Language Thread/green thread, perché implica la gestione completa dello stato del thread a livello del kernel.

In sintesi, mentre il sistema operativo gestisce i thread del kernel (su cui si basa la Beam) in modo tipicamente preemptive (informazione esterna), la macchina virtuale Erlang (Beam) gestisce i suoi milioni di Language Thread (attori) utilizzando un modello di **multi-tasking cooperativo con cambi di contesto estremamente leggeri**. Questo approccio, dove gli attori cedono il controllo dopo brevi computazioni, permette a Erlang di scalare a un numero molto elevato di processi concorrenti su hardware limitato, una caratteristica fondamentale per la costruzione di sistemi massivamente concorrenti e distribuiti.

Domanda

Implementazione del preemptive multi-tasking in Erlang senza far ricorso a interrupt hardware

Risposta

La macchina virtuale di Erlang, chiamata Beam, viene eseguita tipicamente come un unico processo del sistema operativo, il quale può essere multi-threaded con un thread del sistema operativo per core del processore. All'interno di questa macchina virtuale (Beam), gli attori di Erlang sono implementati come **Language Thread**. Il sistema operativo non ha

diretta consapevolezza di questi Language Thread; la loro schedulazione sui thread del kernel è gestita direttamente dalla macchina virtuale Erlang.

Le fonti affermano esplicitamente che Erlang implementa un **"cambio di contesto collaborativo"** per i suoi Language Thread. Questo è il principio del multi-tasking cooperativo. In questo modello, i thread (in questo caso, i Language Thread/attori) cedono volontariamente il controllo del processore. Gli attori di Erlang sono progettati per eseguire brevemente una serie di operazioni in risposta a un messaggio, eventualmente generando altri messaggi, per poi tornare in stato di attesa. Questa natura di "breve esecuzione e attesa" si adatta bene a un modello cooperativo. Il cambio di contesto per questi Language Thread è descritto come **"estremamente leggero"**, richiedendo risorse minime.

Mentre le fonti descrivono la schedulazione degli attori come cooperativa, l'implementazione reale della Beam include un meccanismo per evitare che un singolo attore "monopolizzi" un thread del kernel eseguendo una computazione troppo lunga.

Questo meccanismo si basa sul **conteggio delle riduzioni (reduction counting)**. Ogni attore viene allocato un certo numero di "riduzioni" (passi di esecuzione di base). Quando un attore esaurisce le sue riduzioni assegnate prima di cedere il controllo (ad esempio, inviando un messaggio o entrando nello stato di attesa), lo scheduler della Beam lo può *sospendere* e passare l'esecuzione a un altro attore. Questo comportamento è una forma di *soft preemption* a livello della macchina virtuale (user-space), non basata su interrupt hardware o su time-slice fisse del sistema operativo, ma sulla "fatica" computazionale dell'attore stesso. Questo impedisce a un attore che esegue un calcolo pesante (diverso dal semplice attesa di messaggi) di bloccare la schedulazione degli altri attori su quel core.

Domanda

Implementazione di Erlang: strutture dati usate a run-time per rappresentare il set di processi in esecuzione e la loro memoria.

Risposta

In Erlang, la macchina virtuale chiamata **Beam** viene eseguita tipicamente come un **unico processo del sistema operativo**, che a sua volta può essere **multi-threaded** con un thread del sistema operativo per core del processore. Gli attori di Erlang, concettualmente centrali nel paradigma, sono implementati all'interno di questa macchina virtuale come **Language Thread**.

Il set di attori in esecuzione è gestito direttamente dalla macchina virtuale Erlang. Ogni attore è identificato univocamente da un **PID (Process Identifier)**. La Beam si occupa della loro **schedulazione** sui thread del kernel, con un bilanciamento automatico del carico. Sebbene le fonti non descrivano esplicitamente la struttura dati interna utilizzata dalla Beam per mantenere l'insieme degli attori in attesa di esecuzione, il concetto implica che lo scheduler della Beam mantenga una qualche forma di **coda o collezione** di contesti di Language Thread pronti o sospesi. L'esempio di pseudo-codice di uno scheduler basato su Algebraic Effects mostra l'idea di una "Queue" di "fibre" (che rappresentano contesti di

esecuzione/stack) gestita dallo scheduler a livello utente. Anche la possibilità di registrare attori con nomi (atomi) tramite `register/2` implica una struttura interna nella Beam per mappare nomi ai PID corrispondenti.

Per quanto riguarda la **memoria** associata agli attori:

1. Memoria per il singolo attore/Language Thread:

- Gli attori non condividono stato né memoria, anche quando eseguiti sulla stessa macchina. La comunicazione avviene esclusivamente tramite **scambio di messaggi asincrono** attraverso la loro **Mailbox**, che è una coda per la ricezione dei messaggi.
- Il cambio di contesto per i Language Thread è descritto come **"estremamente leggero"**, e questi attori sono definiti **"green thread"** che utilizzano risorse minime.
- L'impronta di memoria iniziale di un Language Thread è **estremamente contenuta**, circa 300 parole, consentendo di gestire centinaia di migliaia o milioni di attori.
- Contrariamente ai thread tradizionali che spesso hanno stack preallocati, gli attori Erlang iniziano senza uno stack preallocato. Lo **stack cresce dinamicamente** quando necessario, copiando il contenuto dello stack precedente in uno nuovo di dimensione raddoppiata. Le fonti menzionano che il costo computazionale di questa crescita è ammortizzato. La gestione dello stack è fondamentale per l'esecuzione delle funzioni e delle chiamate ricorsive.

2. Rappresentazione dei dati e gestione della memoria generale:

- A runtime, Erlang utilizza una **rappresentazione uniforme dei dati**. Tutti i dati sono rappresentati da una **word**. La dimensione di una word dipende dall'architettura del processore e è sufficiente a contenere almeno un puntatore.
- I tipi di dati che non rientrano in una word sono allocati sullo **Heap** e rappresentati da un **puntatore** (dati "boxed" o "reference types"). I dati che rientrano in una word possono essere rappresentati direttamente (dati "unboxed" o "value types").
- Per distinguere tra puntatori e altri tipi di dati all'interno di una word (necessario per la garbage collection), viene utilizzato un **tag**, tipicamente un bit meno significativo della word. Per i dati "boxed" allocati sull'Heap, una prima word contiene un tag per distinguere i tipi e spesso anche la dimensione del dato.
- Erlang dispone di un **garbage collector automatico** per recuperare la memoria non più utilizzata. Le fonti descrivono diverse tecniche di garbage collection, tra cui Reference Counting e Mark & Sweep (anche Generazionale), e associano queste tecniche e la necessità di tag/rappresentazione uniforme ai linguaggi (come Erlang, OCaml, Haskell) che non usano la monomorfizzazione per gestire il polimorfismo. L'uso di tag e la rappresentazione uniforme facilitano la gestione della memoria e il garbage collection in questi linguaggi.

In sintesi, a runtime, Erlang implementa gli attori come Language Thread leggeri con stack dinamici. La Beam gestisce la schedulazione di questi attori, mantenendo una collezione interna dei contesti di esecuzione. La memoria non è condivisa tra attori. I dati sono

rappresentati in modo uniforme (word-based con tag), con dati più grandi allocati sull'Heap, la cui gestione e deallocazione avviene tramite un garbage collector automatico.

Domanda

Erlang Term Storage (ETS)

Risposta

Sebbene i principi fondamentali di Erlang promuovano l'assenza di memoria condivisa tra attori come regola generale per la robustezza e la scalabilità, esistono casi d'uso all'interno di un singolo nodo Erlang (istanza della Beam) in cui l'accesso rapido e condiviso a grandi quantità di dati è necessario per motivi di performance, ad esempio per implementare cache, lookup tabelle o mantenere stato condiviso non critico per la tolleranza ai guasti a livello di attore individuale.

Per rispondere a queste esigenze, Erlang fornisce un meccanismo chiamato **Erlang Term Storage (ETS)**. L'ETS è un **sistema di database in-memory** ad alte prestazioni integrato direttamente nella macchina virtuale Beam. Permette a diversi attori all'interno dello stesso nodo Erlang di accedere e manipolare *dati condivisi e mutabili*.

Le caratteristiche principali dell'ETS (basate su conoscenza esterna) includono:

- **Archiviazione in Memoria:** I dati vengono memorizzati direttamente nella RAM del nodo Erlang, consentendo accessi molto veloci.
- **Accesso Condiviso:** A differenza della memoria privata di ciascun attore, le tabelle ETS possono essere condivise e accessibili da più attori contemporaneamente.
- **Mutabilità:** I dati nelle tabelle ETS sono mutabili, il che si discosta dal principio generale di immutabilità dei dati in Erlang, ma è necessario per un database.
- **Tipi di Tabelle:** ETS supporta diversi tipi di tabelle (come `set`, `ordered_set`, `bag`, `duplicate_bag`), ciascuno con regole diverse per l'indicizzazione, l'ordine dei dati e la gestione delle chiavi duplicate.
- **API Erlang:** ETS è accessibile tramite funzioni built-in (BIF) nel linguaggio Erlang, consentendo la creazione, l'inserimento, la cancellazione e la lettura di dati nelle tabelle.
- **Gestione:** Le tabelle ETS sono "possedute" da un attore. Se l'attore proprietario termina, la tabella viene automaticamente cancellata (a meno che non sia stata configurata diversamente).

L'utilizzo dell'ETS introduce la possibilità di gestire dati mutabili e condivisi, il che può portare a problematiche di concorrenza (come race conditions) se non gestito con attenzione a livello applicativo. Tuttavia, per scenari specifici dove le prestazioni sono cruciali e la logica di accesso può essere attentamente controllata (spesso incapsulata all'interno di un singolo attore "server" che gestisce l'accesso alla tabella ETS), offre un'alternativa molto più veloce rispetto alla comunicazione tramite messaggi per ogni accesso.

In conclusione, mentre le fonti descrivono accuratamente i principi di base di Erlang (inclusa l'assenza di memoria condivisa tra attori e l'uso di Mnesia come database distribuito), esse non coprono l'Erlang Term Storage (ETS), che è uno strumento per la gestione di dati **in-memory condivisi e mutabili all'interno di un singolo nodo Erlang**, utilizzato per ottimizzazioni di performance in situazioni specifiche.

Garbage Collection

Domanda

Definizione di garbage collection

Risposta

La Garbage Collection è una **tecnica automatica di gestione della memoria**. In un linguaggio di programmazione, uno dei problemi che sorgono nella gestione della memoria è la **formazione del Garbage**, ovvero la memoria non deallocata che non è più utile al programma. L'utilità di un dato è, a livello teorico, indecidibile.

L'obiettivo della Garbage Collection è quello di **recuperare la memoria quando non viene più utilizzata**. Il runtime del linguaggio, tramite un'**euristica di garbage detection**, cerca di approssimare l'utilità di un dato e **dealloca i dati che considera garbage**.

1. **Reference Counting**: Si basa sull'euristica che un dato allocato sullo Heap (detto "boxed") è considerato garbage se ha **0 puntatori entranti (strong pointer)**. Per implementarla, ogni dato boxed memorizza un contatore (Reference Counter). Questa tecnica ha problemi con le strutture dati cicliche.
2. **Mark & Sweep**: Si basa sull'euristica che un dato **non raggiungibile dalle "radici"** (come lo stack e i registri) è considerato garbage. L'algoritmo opera in due fasi concettuali: **Mark**, dove vengono marcati tutti gli oggetti raggiungibili dalle radici, e **Sweep**, dove vengono deallocati quelli non marcati. Una sua implementazione efficiente utilizza due Heap alternati (uno di Lavoro e uno Vuoto), spostando i dati vivi dall'Heap di Lavoro all'Heap Vuoto e ricompattandoli per evitare frammentazione. Questo approccio è spesso utilizzato in linguaggi che allocano molto frequentemente, **come i linguaggi funzionali**. Un problema di Mark & Sweep è l'impossibilità di usare i puntatori come chiavi nei dizionari a causa dello spostamento dei dati.
3. **Mark & Sweep Generazionale**: Una versione ottimizzata di Mark & Sweep basata sull'ipotesi generazionale, che suggerisce che i dati sono destinati a rimanere raggiungibili per molto tempo o a diventare irraggiungibili rapidamente. Divide la memoria in Heap Young (per dati più recenti) e Heap Old (per dati più "anziani"), concentrando le operazioni di garbage collection (Minor Collection) principalmente sugli Heap Young per ridurre il costo computazionale, dato che si suppone che la maggior parte del garbage sia generato tra i dati recenti. Richiede meccanismi aggiuntivi come il Write Barrier per gestire i puntatori dagli Heap Old agli Heap Young.

Le fonti menzionano esplicitamente che **Erlang dispone di un garbage collector automatico** e che i linguaggi funzionali che allocano frequentemente (come Erlang, OCaml, Haskell) usano spesso Mark & Sweep (per allocazioni frequenti). La rappresentazione uniforme dei dati con tag facilita il processo di garbage collection, aiutando il collector a distinguere tra puntatori e altri tipi di dati in una word.

Domanda

Vantaggi e svantaggi della garbage collection

Risposta

La **gestione automatica della memoria**, come la garbage collection, è una tecnica in cui il runtime del linguaggio cerca di **approssimare l'utilità di un dato** e si occupa di **deallocare i dati che considera garbage**. In un linguaggio con garbage collection automatica, il programmatore non gestisce esplicitamente la deallocazione della memoria. L'utilità di un dato è teoricamente indecidibile, quindi la GC si basa su euristiche. Linguaggi come Erlang, OCaml, Haskell utilizzano la gestione automatica della memoria con garbage collection. LISP è stato il primo linguaggio ad avere un garbage collector.

Vantaggi della Garbage Collection:

- **Deallocazione automatica della memoria:** Il sistema si occupa di recuperare la memoria non più utilizzata.
- **Riduzione degli errori del programmatore:** Evitando la gestione manuale, si prevengono problematiche comuni come la formazione di garbage (memoria non deallocata ma non più utile), i puntatori a memoria non allocata (dangling pointers), il riutilizzo improprio di memoria deallocata (accesso o ri-deallocazione) e la frammentazione della memoria. La gestione esplicita da parte del programmatore richiede una descrizione precisa degli invarianti di accesso ai dati.
- **Allocazione efficiente (con Mark & Sweep):** Dopo l'esecuzione dell'algoritmo Mark & Sweep (in alcune implementazioni), l'allocazione di nuovi dati può avvenire in modo molto efficiente, come se fosse uno stack.
- **Adatta a linguaggi con frequente allocazione:** Il Mark & Sweep è spesso utilizzato in linguaggi che allocano memoria molto frequentemente, come i linguaggi funzionali.

Svantaggi e problematiche associate alla Garbage Collection:

- **Euristiche non sempre ottimali:** Il garbage collector approssima l'utilità dei dati, e in alcuni casi l'euristica potrebbe non essere ottimale.
- **Problemi con strutture dati cicliche (Reference Counting):** L'euristica del Reference Counting (RC) non dealloca correttamente le strutture dati con riferimenti ciclici, portando a memory leaks. Questo richiede l'introduzione di meccanismi come i weak pointer.
- **Costo computazionale potenzialmente elevato:** Alcuni metodi di gestione della memoria, come il reference counting o il Mark & Sweep, possono comportare costi

elevati. Il reference counting può richiedere deallocazioni a cascata con costo $O(n)$, mentre il Mark & Sweep ha un costo $O(n + m)$, dove n è il numero di radici e m la quantità di dati vivi. In entrambi i casi, il tempo necessario per la raccolta può crescere significativamente in presenza di strutture complesse o grandi quantità di dati attivi.

- **Interferenza con interazione esterna:** La gestione automatica della memoria può complicare l'interazione con entità esterne o linguaggi diversi, poiché i puntatori mantenuti da altre parti del sistema diventano radici per il garbage collector. La sincronizzazione tra linguaggi diversi può richiedere attenzione.
- **Complessità implementativa (Tagging):** In linguaggi come Erlang, OCaml, Haskell che non usano la monomorfizzazione per il polimorfismo uniforme, è necessario distinguere i puntatori da altri tipi di dati utilizzando dei tag (spesso un bit della word). Questo può complicare l'implementazione delle operazioni aritmetico-logiche o causare frammentazione della memoria se si usa l'ultimo bit come tag.

Domanda

La tecnica del reference counting per l'implementazione di garbage collection

Risposta

Il **reference counting** è una delle **principali categorie di algoritmi per la garbage collection automatica**. Questa tecnica si basa su un'**euristica** per la rilevazione della memoria non più utile (garbage). L'euristica del reference counting stabilisce che **se un dato boxed ha 0 puntatori entranti, allora quel dato è garbage**.

Per implementare questa euristica, è necessario **tenere traccia, per ogni dato boxed, del numero di puntatori entranti**. A questo scopo, viene memorizzato un **Reference Counter (RC)**, un numero intero, nella prima cella del dato boxed, insieme al tag e alla dimensione del dato. Il valore del RC deve essere **maggiore di 0**; altrimenti, il dato viene considerato garbage e deallocato.

Un **problema significativo** del reference counting si presenta con le **strutture dati che contengono riferimenti ciclici** (ad esempio, una lista doppiamente linkata). In questi casi, i puntatori all'interno del ciclo impediscono al contatore di arrivare a 0, **impedendo la deallocazione** e creando **garbage** ogni volta che si perde il puntatore alla prima cella della struttura ciclica.

Per affrontare il problema delle strutture dati cicliche, viene introdotto il concetto di **weak pointer**, distinguendolo dai normali puntatori, ora definiti **strong pointer**. Nel reference counter **vengono conteggiati solo i puntatori strong**. Un dato è considerato vivo se ha almeno un puntatore strong entrante; altrimenti è considerato morto (i weak pointer non contano ai fini del reference counting). L'utilizzo dei weak pointer richiede una **gestione attenta** da parte del programmatore e **strutture dati aggiuntive** per verificare se il puntatore weak si riferisce ancora a un dato allocato.

Le **operazioni di allocazione e gestione dei dati** hanno costi specifici. Quando una cella viene allocata, il RC è inizializzato a 1. La **copia di un puntatore** (ad esempio $q = p$) richiede l'aggiornamento del valore RC: se q puntava a qualcosa, il RC di quell'oggetto viene decrementato (potenzialmente causando una deallocazione a cascata); il RC dell'oggetto puntato da p viene poi incrementato. La funzione di deallocazione (`dealloc`) in pseudo codice mostra una **deallocazione a cascata** che attraversa ricorsivamente le strutture dati per decrementare i RC degli elementi contenuti.

Il **costo computazionale** della copia di un puntatore e della conseguente deallocazione a cascata risulta proporzionale alla lunghezza della catena di deallocazioni, che può essere potenzialmente pari al numero totale di operazioni eseguite dal programma fino a quel momento. Questo costo in tempo è quindi **$O(n)$** , dove n rappresenta il numero di passi del programma, ed è definito come **unbounded**.

La memoria può subire **frammentazione**, richiedendo l'uso di algoritmi efficaci per la gestione dello spazio libero.

Domanda

Algoritmi mark & sweep per la garbage collection

Risposta

La gestione automatica della memoria, attuata tramite **Garbage Collection (GC)**, è una tecnica in cui il run-time del linguaggio cerca di determinare l'utilità di un dato e dealloca quelli che considera "garbage". Il **Mark & Sweep** è una delle **due principali categorie di algoritmi** per la garbage collection automatica. Si basa su un'**euristica** per la rilevazione della memoria non più utile, la quale afferma che **un dato non raggiungibile dalle radici viene considerato garbage**.

È importante definire le **radici**: sono celle di memoria sempre accessibili al programma, come lo **stack e i registri**.

L'algoritmo Mark & Sweep nella sua versione base si compone concettualmente di due fasi, sebbene venga implementato come un'unica operazione per ridurre il costo computazionale:

1. **Fase di Mark**: Questa fase **parte dalle radici e marca con dei bit tutti gli oggetti raggiungibili**. Tutto ciò che non viene marcato è considerato garbage.
2. **Fase di Sweep**: Questa fase **effettua una sorta di deframmentazione**. Tutte le celle marcate (ovvero i dati "vivi" raggiunti dalle radici) vengono **spostate e ricompattate**, mentre le celle non marcate vengono **deallocate**.

Una possibile implementazione di questo algoritmo, descritta nei documenti, utilizza **due Heap differenti**: uno **Heap di Lavoro** (in uso) e uno **Heap Vuoto** (non utilizzato).

- Durante l'esecuzione normale del programma, i dati vengono allocati nello Heap di Lavoro.
- Quando l'algoritmo di garbage collection si attiva, i dati vivi vengono spostati dallo Heap di Lavoro allo Heap Vuoto. L'Heap Vuoto viene trattato come uno Stack, con un Heap Pointer che sale man mano che vengono salvati nuovi dati, il che permette di **evitare la frammentazione** all'interno di questo nuovo Heap.
- Durante lo spostamento, per i dati boxed, l'algoritmo itera sui puntatori presenti nelle radici, segue i puntatori per localizzare il dato nello Heap di Lavoro e alloca una copia di questo dato nello Heap Vuoto. I puntatori all'interno del dato copiato vengono aggiornati per riflettere la nuova posizione.
- Una volta terminata l'iterazione sui grafi raggiungibili dalle radici, **tutti i dati non spostati** nello Heap di Lavoro originale **sono considerati garbage**.
- L'Heap di Lavoro originale viene quindi considerato vuoto (riportando il suo heap pointer all'inizio), e i **due Heap si scambiano di ruolo**.

Al termine dell'algoritmo, l'Heap che diventa il nuovo Heap di Lavoro contiene tutte le celle allocate in modo compattato, permettendo un'**allocazione molto efficiente** dei nuovi dati, simile a quella di uno stack.

Linguaggi che allocano molto frequentemente, come i **linguaggi funzionali** utilizzano spesso Mark & Sweep.

Tuttavia, la versione base presenta dei **problemi**:

- **Puntatori ciclici**: Potrebbero teoricamente portare ad allocazioni infinite, anche se l'euristica "non raggiungibile dalle radici" li considera garbage se non facenti parte di un grafo raggiungibile.
- **Perdita della condivisione (sharing)**: Le celle condivise potrebbero essere duplicate nel nuovo Heap. Per risolvere questo, nella prima cella del dato da copiare (nello Heap di Lavoro) viene salvato un puntatore al nuovo dato copiato nel secondo Heap. Questo indica che la cella è stata visitata e permette di ripristinare lo sharing.

Il **costo computazionale** della versione base è **$O(0)$ in spazio** (non richiede allocazione di memoria aggiuntiva per l'algoritmo stesso) e **$O(n + m)$ in tempo**, dove n è il numero delle radici e m è il numero di dati vivi. Questo costo in tempo non è ottimale se il numero di dati vivi (m) è elevato.

Un altro aspetto da considerare è la compatibilità dello spostamento dei dati con le operazioni sui puntatori. Mentre dereferenziazione, somma/sottrazione scalare all'interno dei blocchi e confronto di uguaglianza tra due puntatori non presentano problemi, le **operazioni di confronto ($<$, $>$, $<=$, $>=$) e le funzioni hash che usano il puntatore come chiave** sono **problematiche** perché il valore del puntatore cambia durante la compattazione. Questo implica che **non è possibile utilizzare i puntatori come chiavi nei dizionari** in modo affidabile. La soluzione suggerita è aggiungere un **ID univoco** al dato boxed da usare come chiave, il che comporta un costo aggiuntivo di indirizzione per leggerlo.

Per migliorare le prestazioni in presenza di molta memoria viva, viene introdotta una versione più avanzata: il **Mark & Sweep Generazionale**. Questa tecnica si basa sull'**ipotesi generazionale**: un dato è destinato a rimanere raggiungibile per un periodo molto lungo o a diventare irraggiungibile rapidamente, escludendo il caso intermedio.

L'algoritmo generazionale utilizza **tre Heap differenti**: uno **Heap Young in uso**, uno **Heap Young non in uso** e uno **Heap Old (Major)**. L'idea è spostare i dati destinati a rimanere vivi a lungo nello Heap Old. Le collezioni avvengono principalmente tra i due Heap Young, con un costo computazionale più efficiente.

- Le collezioni negli Heap Young prendono il nome di **Minor Collection**.
- Durante una Minor Collection, i dati vivi che si trovano sotto un certo puntatore (chiamato **High Water Mark**, indicando che sono sopravvissuti a una collezione precedente) nello Heap Young di origine vengono spostati nello Heap Old. Gli altri dati vivi vengono spostati nell'altro Heap Young.
- Lo Heap Old può raggiungere la saturazione e richiedere una **Major Collection** (deframmentazione al suo interno).
- Il costo computazionale in tempo del Generational GC è **$O(n + m)$** , dove n è il numero delle radici e m è il numero di dati **nello Heap Young**, risultando più efficiente perché esclude una parte dei dati.

Un caso particolare da considerare sono i **puntatori dallo Heap Old a dati nello Heap Young**, che possono verificarsi nei linguaggi con **dati mutabili**. Secondo l'euristica, un dato Young puntato solo da un dato Old "vivo" è comunque vivo. Per gestire questo, l'insieme delle radici viene esteso per includere le celle nello Heap Old che contengono tali puntatori. Ciò comporta un costo aggiuntivo per l'assegnamento di un puntatore (chiamato **Write Barrier**) per aggiornare la lista di queste celle radici.

Altre fonti di radici possono includere i binding con codice scritto in altri linguaggi; se un altro linguaggio mantiene un puntatore a un dato gestito dal GC, quel dato diventa una radice e non deve essere spostato.

Domanda

Gestione dei puntatori dallo heap young allo heap old

Risposta

Nell'algoritmo **Mark & Sweep Generazionale**, la memoria è divisa in più aree, tipicamente uno **Heap Young** (o "Minor") e uno **Heap Old** (o "Major"). Questa suddivisione si basa sull'**ipotesi generazionale**, che suggerisce che i dati o diventano irraggiungibili rapidamente o rimangono raggiungibili per un periodo molto lungo. L'obiettivo è spostare i dati che sopravvivono per un certo tempo nello Heap Old. La garbage collection (chiamata **Minor Collection**) avviene principalmente tra gli Heap Young, con un costo computazionale più efficiente rispetto a collezionare l'intero heap.

Tuttavia, può presentarsi un **caso particolare** che richiede attenzione: una cella situata nello **Heap Old** che contiene un **puntatore a un dato situato nello Heap Young**. Questo scenario può verificarsi in linguaggi di programmazione che supportano dati mutabili.

Secondo l'euristica fondamentale del Mark & Sweep, un dato è considerato garbage se non è raggiungibile dalle radici. Le radici includono inizialmente lo stack e i registri. In un GC generazionale, durante una Minor Collection (che si concentra sugli Heap Young), i dati nello Heap Young vengono marcati se sono raggiungibili dalle radici "tradizionali" (stack, registri) o da dati nello Heap Old che sono considerati "vivi".

Il problema sorge se un dato nello Heap Young è raggiungibile *solo* da un dato nello Heap Old che, a sua volta, è considerato vivo (ovvero, raggiungibile dalle radici). Se il GC non considerasse questo puntatore Old-to-Young, l'oggetto Young verrebbe erroneamente deallocato.

Per gestire questa situazione, l'insieme delle radici per la Minor Collection viene **esteso**. Vengono aggiunte al gruppo delle radici le celle nello Heap Old che contengono puntatori verso dati nello Heap Young.

Questo implica che il sistema di runtime deve in qualche modo tenere traccia di questi puntatori Old-to-Young. Ciò comporta un **costo aggiuntivo** per l'assegnamento di un puntatore (quando si modifica un campo di un oggetto Old in modo che punti a un oggetto Young). Questa operazione richiede di aggiornare la lista o il set di queste celle radici aggiuntive.

Questo overhead associato all'aggiornamento delle radici dovuto agli assegnamenti di puntatori dallo Heap Old allo Heap Young è chiamato **Write Barrier**.

In sintesi, per garantire la correttezza della garbage collection generazionale, i puntatori dagli oggetti Old agli oggetti Young vengono identificati e le loro sorgenti (le celle nello Heap Old) vengono trattate come radici aggiuntive durante le collezioni del Heap Young. La manutenzione di queste "radici" dinamiche introduce un costo aggiuntivo (il Write Barrier) sulle operazioni di scrittura dei puntatori.

Domanda

Gestione dei puntatori dallo heap old allo heap young quando necessaria (e sapere quando lo è)

Risposta

La **Garbage Collection (GC)** automatica, come il **Mark & Sweep**, si basa sull'euristica che **un dato non raggiungibile dalle radici viene considerato garbage**. Le radici sono celle di memoria sempre accessibili al programma, come lo stack e i registri.

Nel **Mark & Sweep Generazionale**, la memoria è organizzata in diverse aree per migliorare l'efficienza, tipicamente uno **Heap Young** (o Minor) e uno **Heap Old** (o Major). Questa divisione si basa sull'**ipotesi generazionale**, che suggerisce che i dati o diventano

rapidamente irraggiungibili o rimangono raggiungibili per lungo tempo, escludendo il caso intermedio. L'idea è spostare i dati che sopravvivono a diverse collezioni nello Heap Old. Le operazioni di GC più frequenti, dette **Minor Collection**, avvengono principalmente sugli Heap Young, riducendo il costo computazionale.

Il problema dei puntatori dallo Heap Old allo Heap Young:

Durante una Minor Collection, che si concentra sugli Heap Young, il GC deve determinare quali dati nello Heap Young sono ancora raggiungibili. Oltre alle radici tradizionali (stack, registri), un dato nello Heap Young potrebbe essere raggiungibile *solo* da una cella nello Heap Old. Se il GC ignorasse questi puntatori, l'oggetto Young a cui puntano verrebbe erroneamente deallocato, violando l'euristica fondamentale della reachability.

Quando è necessaria la gestione di questi puntatori:

La gestione specifica dei puntatori dallo Heap Old allo Heap Young **è necessaria solo in linguaggi di programmazione con dati mutabili**. Nei linguaggi che permettono la mutabilità, un oggetto già promosso nello Heap Old può essere modificato in modo da contenere un puntatore a un nuovo oggetto appena allocato nello Heap Young. Senza mutabilità, un oggetto Old, una volta promosso, non potrebbe modificare i propri campi per acquisire nuovi riferimenti a oggetti Young.

Come vengono gestiti:

Per gestire correttamente questi scenari nei linguaggi con mutabilità, l'insieme delle radici per la Minor Collection viene **esteso**. Vengono considerate radici aggiuntive le celle nello Heap Old che contengono puntatori a dati nello Heap Young. Durante una Minor Collection, il GC parte quindi non solo dalle radici tradizionali, ma anche da questa lista di "radici" dinamiche nello Heap Old, seguendo i puntatori per marcare i dati Young raggiungibili.

Meccanismo di implementazione: il Write Barrier

Per mantenere aggiornata questa lista di celle nello Heap Old che puntano a dati Young, viene introdotto un meccanismo chiamato **Write Barrier**. Ogni volta che viene eseguita un'operazione di assegnamento di un puntatore che modifica una cella nello Heap Old in modo che punti a un oggetto nello Heap Young, il Write Barrier intercetta questa operazione e aggiunge la cella Old modificata alla lista delle radici aggiuntive.

Questo comporta un **costo computazionale aggiuntivo** per l'assegnamento di puntatori, in quanto è necessario eseguire l'operazione di Write Barrier oltre all'assegnamento stesso.

Funzioni e chiusure

Domanda

Cosa significa che in un linguaggio di programmazione le funzioni sono entità di prima classe? Argomentare facendo esempi e contro-esempi.

Risposta

In un linguaggio di programmazione, affermare che le **funzioni sono entità di prima classe**

significa che possono essere trattate **come qualsiasi altro dato** o valore all'interno del linguaggio. Questo implica diverse capacità specifiche per le funzioni:

- Possono essere passate come **argomenti** ad altre funzioni.
- Possono essere **restituite come risultato** da altre funzioni.
- Possono essere **definite all'interno** di altre funzioni (funzioni annidate).
- Possono essere **assegnate a variabili**.
- Possono essere **memorizzate in strutture dati** (come array, liste, alberi, ecc.).

In generale, in un linguaggio tipato, un'entità di prima classe, inclusa una funzione, ha un tipo che ne descrive il dominio (input) e il codominio (output).

Questo concetto è fondamentale nei **linguaggi funzionali**, dove le funzioni sono al centro del paradigma.

Esempi di linguaggi con funzioni di prima classe:

- **Haskell:** È esplicitamente indicato come un linguaggio funzionale in cui le funzioni sono entità di prima classe. La libreria standard di Haskell include la funzione di **composizione funzionale** (`(.)`), che prende in input due funzioni e ne restituisce una terza come output, dimostrando la capacità di passare funzioni come argomenti e di restituirle.
- **Erlang:** Viene descritto come un linguaggio funzionale dove le funzioni sono oggetti di prima classe. È possibile definire **funzioni anonime** usando la sintassi `fun (lista_argomenti) -> ... end` e assegnarle a variabili. Le funzioni possono anche essere definite per casi, selezionando diverse implementazioni tramite pattern matching.
- **OCaml:** Anche se non esplicitamente definito "puro", OCaml è un linguaggio funzionale dove le funzioni sono entità di prima classe. Esempi come `make_counter` mostrano funzioni annidate che vengono restituite e assegnate a variabili (`get`, `inc`, `reset`).

Contro-esempi di linguaggi senza funzioni di prima classe (o non completamente):

- **C:** In C, esistono i **puntatori a funzioni**, che permettono di passare funzioni come argomenti o memorizzarle, ma **non è possibile definire funzioni all'interno di altre funzioni**. Quindi, non soddisfa appieno tutti i requisiti di un linguaggio con funzioni di prima classe complete.
- **Pascal:** In Pascal è possibile definire **funzioni annidate**, ma **non esiste un tipo funzione** che permetta di trattare le funzioni come valori, e **non è possibile restituire funzioni come risultato**. Per gestire l'accesso alle variabili esterne da parte delle funzioni annidate (che in Pascal non sono chiusure complete), viene utilizzato un meccanismo chiamato **puntatore di catena statica**.
- **Java, Python, C++:** Questi linguaggi sono considerati **multi-paradigma** e **non pienamente funzionali**. Hanno incorporato caratteristiche funzionali in un secondo momento. Spesso le funzioni vengono modellate come **oggetti** o tramite **interfacce funzionali** (come il `Comparator` in Java). Permettono l'uso di λ espressioni (spesso con

zucchero sintattico), che possono comportarsi in modo simile alle chiusure, ma l'implementazione nativa e il modo in cui vengono trattate non le rende "entità di prima classe" nello stesso senso dei linguaggi funzionali puri. Ad esempio, la valutazione di espressioni in Python può non sollevare errori sui tipi finché non si tenta l'esecuzione, a differenza di OCaml che usa un sistema di tipi più rigoroso.

La necessità delle Chiusure:

Quando le funzioni possono essere definite all'interno di altre funzioni e poi passate o restituite, sorge il problema della **visibilità delle variabili libere** (variabili utilizzate dalla funzione interna ma definite nello scope esterno). Poiché la funzione interna può essere invocata in un luogo e momento diversi rispetto a dove è stata definita, i dati esterni di cui ha bisogno potrebbero non esistere più nel momento dell'esecuzione.

Per risolvere questo problema, nei linguaggi con funzioni di prima classe complete, viene utilizzato il meccanismo delle **chiusure**. Una chiusura è un pacchetto che contiene il **codice della funzione** e i **valori delle sue variabili libere** nel momento in cui la funzione viene definita. Questi dati vengono copiati e "impacchettati" insieme al codice, garantendo che siano disponibili quando la chiusura viene eseguita.

Le chiusure hanno vari utilizzi, tra cui il **currying** (specializzazione di funzioni a più argomenti), l'**incapsulamento di dati sensibili**, e l'**impacchettamento di espressioni per valutazione pigra (lazy evaluation)**.

In sintesi, la capacità di trattare le funzioni come entità di prima classe (passarle, restituirle, assegnarle, annidarle e memorizzarle) è una caratteristica distintiva dei linguaggi funzionali e viene tipicamente implementata tramite il concetto di chiusura per gestire correttamente l'accesso alle variabili esterne.

Domanda

Che cos'è una chiusura?

Risposta

In un linguaggio di programmazione, una **chiusura** può essere definita come un pacchetto che contiene il **codice della funzione** e i **valori delle sue variabili libere**. Le variabili libere sono quelle variabili utilizzate dalla funzione ma definite nello scope più esterno (chiusura lessicale).

Le chiusure sono strettamente legate al concetto di **funzioni di prima classe**. Quando le funzioni possono essere trattate come valori, possono essere definite all'interno di altre funzioni, passate come argomenti, o restituite come risultati. Se una funzione definita internamente viene passata o restituita, potrebbe essere invocata in un luogo e in un momento diversi rispetto a dove è stata definita. In questo scenario, i dati di cui ha bisogno dallo scope esterno (le sue variabili libere) potrebbero non esistere più nel momento dell'esecuzione. La chiusura risolve questo problema copiando e "impacchettando" questi

dati insieme al codice della funzione, garantendo che siano disponibili quando la chiusura viene eseguita.

Esempi e contesti dall'interno delle fonti:

- Nei linguaggi come **Erlang**, dove le funzioni sono entità di prima classe, le funzioni annidate possono accedere alle variabili definite nello scope più esterno, un concetto chiamato chiusura lessicale. Un esempio dato è `G = fun (X) -> fun (Y) -> X + Y end end.`, dove la variabile `X` (con valore 2 nell'esempio di utilizzo) viene "catturata" nella chiusura restituita dalla funzione esterna `G`.
- In **OCaml**, che è un linguaggio funzionale, le chiusure sono utilizzate per incapsulare dati sensibili. L'esempio `make_counter` crea un riferimento modificabile `c` e restituisce tre funzioni (`get`, `inc`, `reset`) che operano su questo riferimento incapsulato nella loro chiusura.
- L'esistenza di chiusure nei linguaggi funzionali puri come Haskell e OCaml è implicita nella capacità di definire funzioni annidate che accedono all'ambiente circostante. In un linguaggio funzionale puro, il problema della mutabilità delle variabili libere all'interno delle chiusure non si presenta poiché non è permesso l'assegnamento.
- **Rust** supporta le chiusure. In Rust, la cattura delle variabili libere all'interno di una chiusura può avvenire tramite trasferimento di ownership (se la chiusura è di tipo `move`) o tramite borrowing (mutabile/immutabile).
- Le chiusure possono essere utilizzate per vari scopi, tra cui:
 - Il **currying** o specializzazione di funzioni a più argomenti.
 - L'**incapsulamento di dati sensibili** creati localmente da un'altra funzione.
 - L'impacchettamento di espressioni per la **valutazione pigra (lazy evaluation)**.

In sintesi, una chiusura è un costrutto che permette a una funzione di "ricordare" e accedere all'ambiente (variabili libere) in cui è stata creata, anche quando viene eseguita al di fuori di quell'ambiente. Questo è cruciale nei linguaggi con funzioni di prima classe per garantire la correttezza e supportare vari pattern di programmazione.

Domanda

Perché sono utili le chiusure?

Risposta

Una **chiusura** è un concetto fondamentale nei linguaggi che supportano le funzioni di prima classe. Come abbiamo definito, una chiusura è un pacchetto che contiene il **codice della funzione** e i **valori delle sue variabili libere**. Le variabili libere sono quelle utilizzate dalla funzione interna ma definite nello scope più esterno, quello in cui la funzione interna è stata creata.

L'utilità principale delle chiusure risiede nella loro capacità di "catturare" e preservare l'ambiente di definizione (lo scope lessicale) di una funzione, anche quando quella funzione viene eseguita in un contesto diverso o in un momento successivo rispetto alla sua

creazione. Questo è particolarmente importante nei linguaggi funzionali, dove le funzioni possono essere passate come argomenti, restituite come risultati, assegnate a variabili o memorizzate in strutture dati. Quando una funzione annidata viene "estratta" dal suo ambiente di definizione (ad esempio, restituita da una funzione esterna), ha comunque bisogno di accedere ai dati dello scope esterno che utilizza. La chiusura garantisce che questi dati siano disponibili al momento dell'esecuzione.

Esistono diversi **utilizzi specifici che dimostrano l'utilità delle chiusure**:

- 1. Currying e Specializzazione di Funzioni:** Le chiusure consentono di definire funzioni che restituiscono altre funzioni, catturando parte degli argomenti iniziali. Questo è il fondamento del currying, dove una funzione con più argomenti può essere vista come una cascata di funzioni a singolo argomento. Permette di "specializzare" una funzione fornendo solo alcuni argomenti, ottenendo una nuova funzione che "ricorda" gli argomenti forniti e attende i rimanenti. L'esempio in Haskell `add = \x -> \y -> x + y` e l'uso `let f = add 1` mostrano come `f` diventi una funzione che somma 1 al suo input, catturando il valore 1 nella sua chiusura. Similmente, l'esempio in Erlang `G = fun (X) -> fun (Y) -> X + Y end end.` mostra come la variabile `X` venga "catturata" nella chiusura della funzione interna, permettendo alla funzione restituita da `G(2)` di "ricordare" il valore 2 per le future invocazioni con `Y`.
- 2. Incapsulamento di Dati Sensibili:** Le chiusure possono essere utilizzate per creare funzioni che operano su dati privati e locali, incapsulandoli all'interno della chiusura stessa. Questo è un potente meccanismo di incapsulamento simile agli oggetti, ma basato sulle funzioni. L'esempio OCaml `make_counter` crea un riferimento mutabile `c` e restituisce funzioni (`get`, `inc`, `reset`) che possono accedere e modificare `c`, ma `c` stesso non è direttamente accessibile dall'esterno. È "incapsulato" nella chiusura di queste tre funzioni.
- 3. Valutazione Pigra (Lazy Evaluation):** Le chiusure sono utilizzate per "impacchettare" espressioni che non devono essere valutate immediatamente, ma solo quando necessario. Questo è un concetto chiave nei linguaggi lazy come Haskell. L'argomento di una funzione viene avvolto in una chiusura che contiene il codice dell'espressione e l'ambiente necessario per valutarla in seguito. Verrà valutato al massimo una volta, e solo se il suo valore è effettivamente richiesto dalla computazione. Questo migliora l'efficienza e la modularità, come visto negli operatori logici in Haskell che non valutano gli argomenti non necessari.
- 4. Gestione dello Stato in Linguaggi Puri/Funzionali:** Nei linguaggi funzionali puri, dove non c'è mutazione diretta dei dati, le chiusure che catturano valori immutabili sono semanticamente chiare. Anche in linguaggi funzionali che permettono una forma limitata di mutazione (come i riferimenti in OCaml usati nell'esempio del contatore), le chiusure forniscono un modo controllato per gestire lo stato locale. L'immutabilità nei linguaggi funzionali semplifica anche concetti come lo sharing dei dati e la gestione di attori padre-figlio, che indirettamente beneficiano del modello a chiusure che incapsula lo stato associato al "comportamento".

Domanda

Come sono realizzate le chiusure?

Risposta

Una chiusura è realizzata attraverso l'**impacchettamento** o la **combinazione** di due elementi: il **codice della funzione** stessa e i **valori (o riferimenti) delle sue variabili libere**.

Le **variabili libere** sono quelle variabili che vengono utilizzate dal corpo della funzione, ma che sono definite nello scope più esterno (lo scope lessicale) in cui la funzione è stata creata.

La ragione per cui questo impacchettamento è necessario è che una funzione in un linguaggio con funzioni di prima classe (cioè funzioni che possono essere passate come argomenti, restituite come risultati, assegnate a variabili, ecc.) può essere invocata in un luogo e in un tempo molto diversi da quelli in cui è stata definita. Senza la chiusura, i dati (le variabili libere) di cui la funzione ha bisogno dal suo ambiente di definizione originale potrebbero non esistere più al momento dell'esecuzione, poiché lo scope esterno potrebbe essere terminato. La chiusura garantisce che questi dati siano accessibili quando la funzione viene eseguita, "copiando e 'impacchettando' tali dati insieme al codice della funzione".

Le fonti offrono dettagli su come questa "cattura" o "impacchettamento" avviene in contesti specifici:

1. **Concetto Generale / Linguaggi Funzionali Puri:** L'idea base è copiare i valori delle variabili libere. In linguaggi funzionali puri, dove non esiste assegnamento (mutabilità delle variabili), la semantica della chiusura è più diretta. Le variabili libere catturate sono valori immutabili.
2. **Java:** Nei linguaggi che hanno aggiunto le chiusure (come le lambda expression in Java) in seguito, la cattura delle variabili libere avviene per copia del valore. C'è una restrizione importante: le variabili catturate devono essere `effectively final`. Questo significa che il loro valore non deve essere modificato dopo l'assegnazione iniziale. Questo è un modo per evitare semantiche oscure che sorgono se le variabili catturate venissero modificate dopo che la chiusura è stata creata. Tuttavia, se la variabile catturata è un riferimento a un oggetto mutabile, lo stato interno dell'oggetto a cui il riferimento punta *può* comunque essere modificato, il che può complicare la semantica. La chiusura contiene copie dei *valori* delle variabili libere.
3. **Rust:** In Rust, le chiusure supportano la cattura delle variabili libere. Il meccanismo di cattura è strettamente legato al sistema di ownership e borrowing di Rust. Le catture possono avvenire tramite:
 - **Trasferimento di ownership:** Se la chiusura è marcata come `move` (`(move |params| { body })`), la chiusura prende possesso delle variabili libere catturate.
 - **Borrowing (mutabile/immutabile):** Altrimenti, la chiusura prende in prestito (mutable o immutabile) le variabili libere. Questo approccio esplicito è basato sul

sistema dei tipi di Rust fa sì che esistano diversi *tipi* di chiusure, con trait distinti che ne descrivono il comportamento di cattura.

4. **Erlang:** Erlang, essendo un linguaggio funzionale con funzioni di prima classe, supporta la chiusura lessicale. L'esempio `G = fun (X) -> fun (Y) -> X + Y end end.` mostra che la variabile `X` (con il suo valore) viene "catturata" nella chiusura restituita dalla funzione esterna. Il modello funzionale di Erlang tende a favorire l'immutabilità.
5. **Relazione con gli Oggetti:** Il confronto tra oggetti e chiusure offre un'intuizione sulla loro struttura concettuale. Mentre gli oggetti contengono campi (dati) e riferimenti a metodi (funzioni), le chiusure contengono i valori delle variabili libere (dati) e un riferimento alla funzione. Un metodo di un oggetto ha un parametro implicito (`self / this`) che si riferisce all'oggetto stesso. In modo simile, la funzione all'interno di una chiusura può essere vista come avente un parametro implicito che si riferisce alla chiusura stessa, dove essa trova i valori delle variabili libere catturate.

Alternative (Non Chiusure): È interessante notare che alcuni linguaggi che consentono funzioni annidate ma non sono pienamente funzionali (come il Pascal) non realizzano l'accesso all'ambiente esterno tramite chiusure come definite sopra. Il Pascal, ad esempio, utilizza un **puntatore di catena statica** che collega il record di attivazione della funzione annidata al suo ambiente di definizione esterno, permettendo l'accesso alle variabili locali di quel contesto. Questo è un meccanismo diverso dalla chiusura che "impacchetta" i dati necessari. Il C, invece, non permette la definizione di funzioni annidate, quindi non ha bisogno di affrontare questo problema in quel modo.

In sintesi, la realizzazione di una chiusura si basa sul concetto fondamentale di **bundle** (pacchetto) che unisce il codice della funzione al suo ambiente di definizione catturato. Il modo in cui le variabili di questo ambiente vengono catturate (copia, riferimento, borrowing) e le regole associate (come `effectively final` in Java o i meccanismi di ownership in Rust) possono variare tra i diversi linguaggi.

Oggetti, classi e trait

Domanda

Il modello di programmazione a oggetti: varianti, nozioni fondamentali, implementazione, limiti.

Risposta

Nozioni Fondamentali del Modello a Oggetti:

- **Oggetti:** L'unità centrale del modello. Un oggetto incapsula al suo interno sia **dati** (spesso chiamati campi o stato) che **comportamento** (attraverso i metodi).
- **Messaggi/Metodi:** Gli oggetti interagiscono tra loro scambiandosi messaggi. L'invio di un messaggio a un oggetto corrisponde all'invocazione di un suo metodo. I metodi possono essere visti come funzioni che operano sullo stato dell'oggetto, spesso avendo un parametro implicito (`self` o `this`) che si riferisce all'oggetto stesso.

- **Modularizzazione:** Il paradigma nasce con l'idea di partizionare lo stato globale del programma in tanti oggetti, ciascuno responsabile della propria porzione di stato.

Varianti del Modello a Oggetti:

I linguaggi di programmazione a oggetti si dividono in due grandi famiglie:

- **Linguaggi Object-Based:**
 - Sono privi della nozione esplicita di **classe**.
 - Gli oggetti vengono creati al bisogno, definendo i loro campi e metodi direttamente sull'istanza.
 - È possibile aggiungere o rimuovere campi e metodi a un oggetto in maniera dinamica a *run-time*, senza uno schema fisso predefinito.
 - L'esempio più diffuso è JavaScript.
 - I metodi possono essere visti come campi dell'oggetto che puntano a funzioni, le quali ricevono l'oggetto stesso come argomento implicito (`self`). Questo li rende **concettualmente molto vicini alle chiusure**.
- **Linguaggi Class-Based:**
 - Il comportamento è più **disciplinato**.
 - Gli oggetti sono istanze di **classi**, che agiscono come *blueprint* o famiglie predefinite.
 - Una classe definisce in maniera **fissa** l'insieme dei campi e dei metodi che le sue istanze avranno.
 - Questo permette di ragionare più facilmente sulla **correttezza** del codice e di assegnare un **tipo statico** agli oggetti.
 - Esempi diffusi includono C++, Java, C#.
 - Concetti come l'**ereditarietà** (dove una classe può ereditare proprietà da una classe padre) e le **interfacce** (che definiscono un contratto sui metodi che una classe deve implementare) sono centrali in questa famiglia.

Implementazione:

- **Rappresentazione Interna:** Un oggetto può essere concettualmente visto come una struttura dati che contiene i valori dei suoi campi e riferimenti ai suoi metodi.
- **Metodi e Chiusure:** Come accennato, in linguaggi object-based, i metodi possono essere implementati come funzioni che "catturano" l'oggetto (`self`) al quale appartengono, in modo simile a come una chiusura cattura le sue variabili libere. Concettualmente, una chiusura è un pacchetto di codice e i valori delle variabili libere, mentre un oggetto è un pacchetto di campi (dati) e riferimenti a metodi (funzioni).
- **Dispatching dei Metodi (Late Binding):** Quando si invoca un metodo `o.m(...)`, il codice esatto da eseguire per `m` dipende dal tipo effettivo dell'oggetto `o` a *run-time*. Nei linguaggi class-based o con interfacce/trait (con polimorfismo dinamico di default), questo comporta una **ricerca dinamica** (late binding) dell'implementazione corretta. Ad esempio:

- In Go, un valore di tipo interfaccia (che è una forma di trait) è rappresentato come una coppia (v, p) , dove v è l'oggetto e p è una tabella dei metodi (`method table`) definita dall'interfaccia per il tipo di v . L'invocazione del metodo `o.Area()` (dove `o` ha un tipo interfaccia `Measurable`) recupera la tabella p da `o` e chiama la funzione corretta trovata in p .
- In linguaggi funzionali con **Type Classes** (come Haskell), concettualmente simili ai trait/interfacce, l'implementazione avviene spesso passando implicitamente un "dizionario" (record/struct) dei metodi come parametro aggiuntivo alle funzioni polimorfe.
- **Gestione della Memoria:** Gli oggetti, specialmente quelli allocati dinamicamente, risiedono tipicamente sullo Heap. Nei linguaggi OOP comuni come Java, la gestione della memoria è demandata a un **Garbage Collector**, che identifica e recupera la memoria non più raggiungibile.
- **Mutabilità:** Tipicamente, i campi degli oggetti sono mutabili, a differenza dei valori catturati nelle chiusure in linguaggi funzionali puri. La mutabilità in contesti concorrenti o distribuiti può complicare la gestione degli errori e la correttezza.

Limiti:

- **Linguaggi Object-Based:**
 - Difficoltà a ragionare sulla **correttezza** del codice: la possibilità di modificare metodi a run-time rende difficile garantire che altre parti del codice che dipendono da quei metodi funzionino correttamente.
 - Difficoltà nella **tipizzazione statica**: la struttura dinamica degli oggetti rende arduo per il compilatore tracciare i tipi.
 - Mancanza di incapsulamento rigoroso: "i metodi viaggiano insieme all'oggetto", permettendo a chi riceve un oggetto di alterarlo in modi non desiderati.
- **Linguaggi Class-Based (Ereditarietà):**
 - **Rigidità e Fragilità:** L'ereditarietà crea una forte dipendenza tra classi padre e figlie. Modifiche a una classe base possono "rompere" le sottoclassi ("fragile base class problem").
 - Può portare a duplicazione di codice se non usata con attenzione. La **composizione** è spesso vista come un'alternativa meno rigida.
- **OOP in generale (quando si basa su stato mutabile condiviso):**
 - La combinazione di mutabilità dei dati e condivisione tra oggetti può rendere difficile garantire determinate proprietà dell'output basandosi sulle proprietà dell'input. Introdurre costrutti imperativi (come la mutabilità dello stato degli oggetti) in un contesto con garanzie diverse (come l'immutabilità nei linguaggi funzionali) può indebolire tali garanzie.
 - In sistemi concorrenti o distribuiti, la gestione degli errori con stato mutabile condiviso è complessa e può portare a deadlock.

Contesto Multi-Paradigma:

È importante notare che la maggior parte dei linguaggi moderni sono **multi-paradigma**, incorporando costrutti da diversi modelli, inclusi elementi della programmazione funzionale (come funzioni di prima classe e chiusure). Tuttavia, combinare paradigmi può potenzialmente indebolire le garanzie specifiche di ciascun modello puro.

Domanda

Metodi in Go, cosa sono, come funzionano.

Risposta

I **metodi in Go** sono strettamente legati alle **strutture dati** e offrono un modo per associare comportamenti specifici a tipi definiti dall'utente, pur distanziandosi dal modello classico della programmazione a oggetti basata sulle classi.

Cosa sono i Metodi in Go:

- Un metodo è una **funzione in cui un argomento specifico, detto receiver, viene distinto dagli altri**.
- La sintassi per definire un metodo è diversa da quella di una normale funzione. Ad esempio, invece di `func Mod(v Vector) float64 { ... }`, si scrive `func (v Vector) Mod() float64 { ... }`, dove `(v Vector)` è il receiver.
- I metodi vengono **invocati con una sintassi `o.m(...)`** (simile a quella di Java), invece della tradizionale `m(o, ...)`.
- A **run-time non c'è alcuna differenza** tra l'esecuzione di funzioni e metodi. La sintassi con il receiver a sinistra è scelta per motivi di utilità, come l'auto-completamento, poiché il tipo è definito localmente.

Come Funzionano i Metodi in Go:

- **Receiver Type:** Il tipo del receiver (nell'esempio `Vector` in `(v Vector)`) deve essere **definito nello stesso file** in cui viene dichiarato il metodo. Questo limite può essere aggirato utilizzando **alias** per tipi preesistenti.
- **Receiver Value vs. Pointer:** Il receiver può essere una **copia del valore** (`(v Vector)`) o un **puntatore** ad esso (`(v *Vector)`).
 - Se il receiver è una **copia** (value receiver), le modifiche apportate al receiver all'interno del metodo si applicano solo a quella copia locale e non influenzano l'oggetto originale.
 - Se il receiver è un **puntatore** (pointer receiver), il metodo opera direttamente sull'oggetto originale, consentendo modifiche permanenti.
 - Go si differenzia da linguaggi come Java in cui gli oggetti sono sempre allocati nell'Heap e il passaggio di un oggetto a un metodo passa sempre un riferimento. In Go, se non si specifica il passaggio per riferimento (`*`), viene creata una copia locale del dato.

- **Invocazione con Sintassi Uniforme:** Indipendentemente dal fatto che il receiver sia un valore o un puntatore, la sintassi per accedere/modificare i campi e per l'invocazione del metodo è la stessa (`v.Mod()` , `v.Scale(10)`). Il compilatore gestisce automaticamente la dereferenziazione necessaria per i pointer receivers.
- **Relazione con le Interfacce (Trait):** I metodi sono fondamentali per il concetto di **interfaccia** in Go (che corrisponde al concetto di "trait" in altri linguaggi). Un'interfaccia definisce un **insieme di metodi**.
 - Un tipo `T` **implementa automaticamente un'interfaccia** `I` se definisce tutti i metodi elencati in `I` con la stessa firma. Questo meccanismo è noto come **Duck Typing**. Non è richiesta alcuna dichiarazione esplicita (`implements`) come in Java.
 - Questa implementazione automatica consente il **polimorfismo**. Una funzione che accetta un'interfaccia (`Measurable` nell'esempio) può ricevere qualsiasi tipo che implementi quella interfaccia (`Square` , `Circle`).
- **Implementazione delle Interfacce e Dispatching dei Metodi:** Quando un valore viene passato a una funzione che si aspetta un'interfaccia, viene creata una **coppia (dati, codice)**. Questa coppia, che rappresenta il valore di tipo interfaccia, contiene l'oggetto (`v`) e una **tabella dei metodi** (`p`) definita dall'interfaccia per il tipo di `v` .
 - L'invocazione di un metodo su un valore di tipo interfaccia (`o.Area()` dove `o` è `Measurable`) comporta un **dispatching dinamico** (late binding). Il codice esatto da eseguire (il puntatore alla funzione corretta) viene recuperato dalla tabella `p` all'interno della coppia (`v` , `p`) a tempo di esecuzione.
- **Confronto con altri Paradigmi/Linguaggi:** Go elimina le classi a favore delle interfacce (trait). Le interfacce in Go, concettualmente simili ai trait di Rust o alle Type Classes di Haskell, differiscono nell'implementazione del dispatching dei metodi: in Go e Rust ogni oggetto/trait object porta con sé il dizionario/tabella dei metodi, mentre in Haskell il dizionario della Type Class viene passato una volta sola.

In sintesi, i metodi in Go sono funzioni associate a tipi specifici tramite un receiver, che supportano il polimorfismo attraverso le interfacce e vengono implementati a runtime con una tabella di metodi dinamica.

Domanda (Esercizio)

Scrivi il metodo Go che implementa l'operazione X per la struttura dati S.

Risposta

Domanda

Cos'è il duck typing? Come si concretizza in un linguaggio come Go?

Risposta

Il duck typing è un meccanismo attraverso il quale un tipo **implementa automaticamente un'interfaccia** se definisce tutti i metodi elencati in quell'interfaccia con la stessa firma. Non

è richiesta alcuna "cerimonia sintattica", come una dichiarazione esplicita (`implements`). L'idea alla base è "Se cammina come un'anatra e starnazza come un'anatra, allora è un'anatra", applicata ai tipi e ai loro metodi.

Come si concretizza in un linguaggio come Go:

Go non utilizza il modello classico di programmazione a oggetti basato sulle classi e sull'ereditarietà di implementazione. Invece, si avvale del concetto di **interfaccia** (che nei documenti viene paragonato al concetto di "trait" presente in altri linguaggi) e di **metodi** associati alle strutture dati.

Ecco come il duck typing si manifesta in Go:

1. **Definizione di Interfaccia:** In Go, un'interfaccia definisce semplicemente un **insieme di firme di metodi**. Ad esempio, l'interfaccia `Measurable` nei documenti specifica che un tipo deve avere i metodi `Area()` e `Perimeter()`.
2. **Implementazione Automatica:** Un tipo concreto (come una `struct Square` o `Circle`) **implementa automaticamente** un'interfaccia se definisce *tutti* i metodi specificati dall'interfaccia. Non c'è bisogno di dichiarare esplicitamente che `Square` o `Circle` "implementano" `Measurable`. Il compilatore Go verifica questa corrispondenza in modo automatico.
3. **Polimorfismo:** Questa implementazione automatica consente il **polimorfismo**. Una funzione o una variabile che si aspetta un valore di tipo interfaccia può ricevere qualsiasi tipo concreto che soddisfi i requisiti (i metodi) dell'interfaccia. La funzione `PrintMeasure(o Measurable)` è un chiaro esempio: può accettare sia un `Square` che un `Circle` perché entrambi implementano l'interfaccia `Measurable` definendo i metodi richiesti.
4. **Implementazione a Run-time:** Quando un valore di un tipo concreto (come `Square{1}`) viene passato a una funzione che si aspetta un'interfaccia (`PrintMeasure`), viene creato un **valore di tipo interfaccia**. Questo valore è una **coppia (dati, codice)**, dove i "dati" sono l'oggetto concreto sottostante e il "codice" è una **tabella dei metodi** (simile a una *virtual table*) che contiene i puntatori alle implementazioni corrette dei metodi specificati dall'interfaccia per quel tipo concreto.
5. **Dispatch Dinamico:** Quando un metodo viene chiamato su un valore di tipo interfaccia (ad es. `o.Area()` dove `o` è `Measurable`), l'indirizzo esatto della funzione da eseguire (quale `Area` chiamare, se quella di `Square` o di `Circle`) viene recuperato a tempo di esecuzione dalla tabella dei metodi presente nella coppia (dati, codice). Questo è un esempio di **dispatch dinamico** o **late binding**.

In sintesi, il duck typing in Go permette di ottenere flessibilità e polimorfismo attraverso le interfacce e i metodi, senza la rigidità dell'ereditarietà di implementazione o la necessità di dichiarazioni esplicite, affidandosi al compilatore per verificare la compatibilità dei tipi basata sulla presenza dei metodi richiesti. L'ereditarietà in Go è limitata alla sola specifica delle interfacce o alla composizione di strutture dati.

Domanda

Interfacce in Go, cosa sono, come funzionano (implementazione), esempi.

Risposta

Cosa sono le Interfacce in Go:

- Go non adotta il tradizionale modello di programmazione a oggetti basato sulle classi come in Java o C++. Al suo posto, utilizza l'idea dei **trait**, che in Go vengono chiamati **interfacce**.
- Un'interfaccia è un **insieme di firme di metodi**.
- L'idea fondamentale è che **tutte le entità che implementano l'interfaccia supportano determinate operazioni**.

Come Funzionano (Implementazione) le Interfacce in Go:

Il meccanismo chiave per l'implementazione delle interfacce in Go è il **Duck Typing**.

- Un tipo concreto (come una `struct`) **implementa automaticamente un'interfaccia** se definisce **tutti i metodi** elencati in quell'interfaccia con la stessa firma.
- **Non è richiesta alcuna "cerimonia sintattica"** come la parola chiave `implements` presente in altri linguaggi. Il compilatore Go verifica la compatibilità basandosi solo sulla presenza e sulla firma dei metodi richiesti dall'interfaccia.
- Questa implementazione automatica consente il **polimorfismo**: una funzione che accetta un valore di tipo interfaccia può ricevere qualsiasi tipo concreto che implementi quell'interfaccia.

L'implementazione a run-time si basa su un concetto di **valore di tipo interfaccia**:

- Quando un valore di un tipo concreto che implementa un'interfaccia viene passato a una funzione che si aspetta tale interfaccia, viene creato un **valore di tipo interfaccia**.
- Questo valore di tipo interfaccia è concettualmente una **coppia (dati, codice)**.
 - `v` : rappresenta l'oggetto concreto sottostante (i *dati*).
 - `p` : rappresenta una **tabella dei metodi** (simile a una *virtual table* in altri linguaggi) specifica per il tipo concreto `v` nel contesto dei metodi definiti dall'interfaccia `I`.
- L'invocazione di un metodo su un valore di tipo interfaccia (ad esempio, chiamare `Area()` su un oggetto di tipo `Measurable`) comporta un **dispatching dinamico** (o *late binding*). A tempo di esecuzione, l'indirizzo esatto della funzione corretta (cioè, quale implementazione del metodo `Area` chiamare, se quella di `Square` o `Circle`) viene recuperato dalla tabella dei metodi `p` contenuta nella coppia `(v, p)`.
- Passare un valore di tipo `Square` a una funzione che accetta `Measurable` significa creare e passare la coppia `(s, p)`, dove `s` è il `Square` e `p` è la tabella con i puntatori ai metodi `Area` e `Perimeter` specifici per `Square`. Un discorso analogo vale per `Circle`.

Differenze e Relazioni:

- L'implementazione di Go, dove ogni oggetto/valore di interfaccia porta con sé la tabella dei metodi, si distingue da approcci come le Type Classes di Haskell, dove il "dizionario" dei metodi viene passato una volta sola alla funzione polimorfa.
- Le interfacce in Go possono **estendere** altre interfacce, ereditando le firme dei metodi. Questo tipo di ereditarietà è **solo a livello di specifica** dell'interfaccia, non di codice.
- La composizione di strutture è un meccanismo separato in Go per riutilizzare implementazioni, dove una struct può includere (anche anonimamente) un'altra struct, ereditandone campi e metodi. Questo non è direttamente l'implementazione delle interfacce, ma un altro strumento legato alla strutturazione del codice.

Esempi:

Gli esempi forniti nelle fonti chiariscono il concetto:

1. **Definizione di Tipi Concreti con Metodi:** Vengono definite le struct `Square` e `Circle`, entrambe con metodi `Area()` e `Perimeter()`:

```
type Square struct {
    side float64
}
func (o Square) Area() float64 { return o.side * o.side }
func (o Square) Perimeter() float64 { return 4 * o.side }

type Circle struct {
    radius float64
}
func (o Circle) Area() float64 { return pi * o.radius * o.radius }
func (o Circle) Perimeter() float64 { return 2 * pi * o.radius }
// pi sarebbe definito altrove, ad es. in "math"
```

2. **Definizione dell'Interfaccia:** Viene definita l'interfaccia `Measurable` che specifica i metodi `Area()` e `Perimeter()`:

```
type Measurable interface {
    Area() float64
    Perimeter() float64
}
```

Nota: `Square` e `Circle` implementano *automaticamente* `Measurable` perché definiscono entrambi i metodi specificati.

3. **Funzione Polimorfa che Accetta l'Interfaccia:** Viene definita una funzione `PrintMeasure` che accetta un tipo `Measurable`. Questa funzione può ricevere qualsiasi tipo concreto (come `Square` o `Circle`) che implementi l'interfaccia:

```
func PrintMeasure(o Measurable) {
    fmt.Println(o.Area())
    fmt.Println(o.Perimeter())
}
```

4. **Utilizzo nel main** : Nel `main`, vengono create istanze di `Square` e `Circle` e passate alla funzione `PrintMeasure`. Go gestisce automaticamente la creazione dei valori di tipo interfaccia e il dispatching dinamico:

```
func main() {
    s := Square{1}
    c := Circle{2}
    PrintMeasure(s) // Square -> Measurable
    PrintMeasure(c) // Circle -> Measurable
}
```

In conclusione, le interfacce in Go sono insiemi di firme di metodi utilizzate per definire comportamenti attesi. Grazie al duck typing, qualsiasi tipo che possieda i metodi richiesti implementa automaticamente l'interfaccia. A runtime, il polimorfismo è gestito tramite valori di interfaccia che contengono una tabella di puntatori ai metodi corretti del tipo concreto sottostante, consentendo il dispatching dinamico.

Domanda

C'è distinzione tra binding statico e binding dinamico in Go/Rust? Fare (contro-)esempi.

Risposta

Binding Statico (Compile-Time Dispatch):

Il binding statico (o *compile-time dispatch*) si verifica quando il compilatore è in grado di determinare esattamente quale implementazione di un metodo o funzione verrà chiamata già al momento della compilazione. Questo è il caso più comune e offre generalmente le migliori prestazioni in quanto non richiede overhead a runtime.

- **In Go**: Quando si invoca un metodo su un valore di un **tipo concreto** (cioè, non un tipo interfaccia), il binding è statico. Il compilatore sa esattamente il tipo dell'oggetto e quale funzione specifica (`func (T) Method(...)`) deve chiamare.
 - **Esempio**:

```
type Vector struct { X, Y float64 }
func (v Vector) Mod() float64 { ... }
func main() {
    v := Vector{3, 4}
    fmt.Println(v.Mod())
    // Binding statico: il compilatore sa che deve chiamare
```

```
Vector.Mod  
}
```

In questo caso, il compilatore sa che `v` è di tipo `Vector` e lega la chiamata `v.Mod()` direttamente all'implementazione `func (v Vector) Mod() float64`. Non c'è "late binding".

- **In Rust:** Rust favorisce fortemente il binding statico.
 - Quando si invoca un metodo su un valore di un **tipo concreto**, il binding è statico.
 - Quando si invoca un metodo su un valore di un **tipo generico con vincoli di trait**, il binding è comunque statico. Questo avviene grazie alla **monomorfizzazione**: il compilatore crea versioni separate del codice per ogni specifica combinazione di tipi con cui la funzione generica viene utilizzata.
- **Esempi:**

```
struct Circle { radius : f64, }  
impl Circle { fn area(&self) -> f64 { ... } }  
// Implementazione per Circle  
fn smaller(a : Circle, b : Circle) -> bool {  
    a.area() < b.area()  
    // Binding statico: a e b sono Circle concreti  
}  
  
struct Rectangle<T> { width : T, height : T, }  
impl <T: PartialOrd + std::ops::Mul<Output = T>> Rectangle<T> {  
    // Aggiunti vincoli per * e <  
    fn area(&self) -> T {  
        self.width * self.height  
        // Binding statico: monomorfizzazione crea codice  
        // specifico per T  
    }  
}  
fn compare_areas<T: PartialOrd + std::ops::Mul<Output = T>>(a :  
Rectangle<T>, b : Rectangle<T>) -> bool {  
    a.area() < b.area()  
    // Binding statico: monomorfizzazione per il tipo T effettivo  
}
```

Nella funzione `smaller`, il compilatore sa che `a` e `b` sono `Circle` e lega `area()` all'implementazione per `Circle`. Nella funzione `compare_areas`, anche se è generica `T`, il compilatore, tramite monomorfizzazione, genererà versioni concrete (es. per `Rectangle<f64>`), e il binding per `area()` sarà statico in ogni versione specializzata.

Binding Dinamico (Run-Time Dispatch):

Il binding dinamico (o *run-time dispatch* o *late binding*) si verifica quando l'esatta

implementazione di un metodo da chiamare non può essere determinata dal compilatore e viene decisa solo al momento dell'esecuzione del programma.

- **In Go:** Il binding dinamico è strettamente legato all'uso delle **interfacce**. Quando un metodo viene chiamato su un valore di un **tipo interfaccia**, il sistema Go determina a runtime quale metodo concreto chiamare basandosi sul tipo effettivo contenuto nel valore interfaccia. Un valore di tipo interfaccia è concettualmente una coppia (dati, codice), dove codice è una tabella di metodi (simile a una virtual table) che contiene i puntatori alle implementazioni corrette per il tipo concreto sottostante.
- **Esempio:**

```
type Measurable interface {
    Area() float64
    Perimeter() float64
}

type Square struct { side float64 }
func (o Square) Area() float64 { return o.side * o.side }
func (o Square) Perimeter() float64 { return 4 * o.side }

type Circle struct { radius float64 }
func (o Circle) Area() float64 { return math.Pi * o.radius *
o.radius }
func (o Circle) Perimeter() float64 { return 2 * math.Pi *
o.radius }

func PrintMeasure(o Measurable) {
    // Accetta un'interfaccia
    fmt.Println(o.Area())
    // Binding dinamico: quale Area() chiamare (Square o Circle)
    fmt.Println(o.Perimeter())
    // è deciso a runtime
}

func main() {
    s := Square{1}
    c := Circle{2}
    PrintMeasure(s)
    // s (Square) passato come Measurable
    PrintMeasure(c)
    // c (Circle) passato come Measurable
}
```

Quando `PrintMeasure(s)` viene chiamata, `s` (tipo `Square`) viene "in scatolata" in un valore di interfaccia `Measurable` contenente `s` e i puntatori ai metodi `Area` e `Perimeter` di `Square`. Quando `o.Area()` viene chiamato dentro `PrintMeasure`, il sistema Go usa la tabella dei metodi nel valore interfaccia per trovare e invocare

l'implementazione corretta (`Square.Area`). Lo stesso avviene per `c` (tipo `Circle`), che viene incasellata con i puntatori ai metodi di `Circle` .

- **In Rust:** Rust supporta il binding dinamico esplicitamente tramite i **trait objects**. Un trait object è tipicamente un puntatore (come `&Measurable` o `Box<dyn Measurable>`) che può puntare a qualsiasi tipo concreto che implementi il trait. Il programmatore deve indicare esplicitamente quando un oggetto deve essere trattato come un trait object per ottenere il dispatching dinamico.

- **Esempio:**

```
trait Measurable {
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}

struct Circle { radius : f64, }
impl Measurable for Circle { ... } // Implementazione per Circle

struct Square { side : f64, }
impl Measurable for Square { ... } // Implementazione per Square

fn print_measure(shape : &dyn Measurable) {
    // Accetta un trait object (reference a trait)
    println!("Area = {}", shape.area());
    // Binding dinamico: risolto a runtime
    println!("Perimeter = {}", shape.perimeter());
    // Binding dinamico: risolto a runtime
}

fn main() {
    let s = Square { side: 1.0 };
    let c = Circle { radius: 2.0 };
    print_measure(&s);
    // Passato riferimento a Square come trait object &dyn
    // Measurable
    print_measure(&c);
    // Passato riferimento a Circle come trait object &dyn
    // Measurable
}
```

La funzione `print_measure` accetta un riferimento a un trait object `&dyn Measurable` (il `dyn` è spesso implicito ma esplicita che si tratta di dispatch dinamico). Quando `print_measure(&s)` viene chiamata, un puntatore a `s` e una tabella dei metodi appropriata per `Square` nel contesto del trait `Measurable` vengono passati. La chiamata a `shape.area()` a runtime utilizza questa tabella per chiamare l'implementazione corretta per `Square` . Similmente accade per `&c` e `Circle` .

In Sintesi:

Sia Go che Rust distinguono tra binding statico e dinamico. Go utilizza il binding dinamico automaticamente quando si lavora con tipi interfaccia, mentre Rust richiede l'uso esplicito di **trait objects** (tipicamente riferimenti a trait) per abilitare il binding dinamico, preferendo altrimenti il binding statico (anche tramite monomorfizzazione per il polimorfismo generico)..

Domanda

Argomenti e variabili di tipo interfaccia in Go. Come sono implementati?

Risposta

Argomenti e Variabili di Tipo Interfaccia in Go

In Go, le interfacce sono un concetto fondamentale per ottenere il polimorfismo. Un'interfaccia è definita come **un insieme di tipi di metodi**. Qualsiasi tipo concreto che definisce tutti i metodi elencati in un'interfaccia (con gli stessi tipi) **implementa automaticamente tale interfaccia**, senza la necessità di una sintassi esplicita come `implements` in Java. Questo meccanismo è noto come **Duck Typing**.

Una variabile o un argomento di tipo interfaccia in Go può contenere valori di *qualsiasi tipo concreto* che implementi tale interfaccia.

- **Esempio di argomento di tipo interfaccia:** La funzione `PrintMeasure(o Measurable)` mostrata negli esempi accetta un argomento `o` di tipo interfaccia `Measurable`. Questa funzione può essere chiamata con un valore di tipo `Square` o `Circle`, poiché entrambi implementano l'interfaccia `Measurable`. Questa è una forma di **funzione polimorfa**.
- **Esempio di variabile di tipo interfaccia:** Se avessimo dichiarato una variabile `var m Measurable`, potremmo successivamente assegnarle un valore di tipo `Square` (`m = s`) o un valore di tipo `Circle` (`m = c`), purché `s` e `c` siano di tipi che implementano `Measurable`.

Implementazione dei Tipi Interfaccia in Go

L'aspetto cruciale, come descritto nei documenti, riguarda l'implementazione a runtime. Quando un metodo viene chiamato su un valore di tipo interfaccia, il compilatore non sa a priori quale implementazione specifica del metodo debba essere eseguita, perché il tipo concreto sottostante può variare durante l'esecuzione.

Per gestire questa dinamicità, un **valore di tipo interfaccia** in Go è implementato a runtime come una **coppia (dati, codice)**.

1. **dati (o `v` nell'esempio del testo):** Rappresenta il valore concreto dell'oggetto sottostante (ad esempio, un'istanza di `Square` o `Circle`).
2. **codice (o `p` nell'esempio del testo):** È una **tabella dei metodi** (simile a una virtual table) specifica per il tipo concreto contenuto in `dati`. Questa tabella contiene i

puntatori alle implementazioni corrette dei metodi definiti nell'interfaccia, relative a quel tipo concreto.

Quando viene invocato un metodo su un valore interfaccia (ad esempio, `o.Area()` all'interno di `PrintMeasure`), il sistema Go **a tempo di esecuzione** utilizza la tabella (`codice`) contenuta nella coppia interfaccia per trovare e chiamare l'implementazione corretta del metodo per il tipo effettivo dell'oggetto `dati`. Questo processo è una forma di **late binding** o **binding dinamico**, poiché l'indirizzo della funzione da chiamare viene determinato solo a runtime.

Quando si passa un valore di un tipo concreto (come `Square`) a una funzione che si aspetta un'interfaccia (come `Measurable`), il compilatore crea e passa questa coppia (`dati`, `codice`). Il `dati` sarà l'oggetto `Square`, e il `codice` sarà una tabella di puntatori ai metodi `Area` e `Perimeter` implementati per il tipo `Square`. Analogamente, per un valore di tipo `Circle`, verrà creata una coppia con l'oggetto `Circle` e la tabella dei metodi per `Circle`.

Questo meccanismo di binding dinamico tramite la coppia (`dati`, `codice`) e la tabella dei metodi è il modo in cui Go implementa il polimorfismo basato sulle interfacce. Linguaggi come Go e Rust (tramite i loro `trait object`) portano con sé questa tabella/dizionario di metodi per ogni oggetto/valore interfaccia che richiede dispatch dinamico.

Distinzione con Binding Statico (Compile-Time Dispatch)

È importante notare la distinzione con il binding statico (o *compile-time dispatch*), che avviene quando il compilatore conosce esattamente il tipo del ricevitore e può legare la chiamata al metodo specifico già in fase di compilazione. Questo accade quando si chiama un metodo direttamente su una variabile di un **tipo concreto** (non interfaccia), come nell'esempio iniziale di `v.Mod()` dove `v` è una `Vector` concreta. In questi casi, non c'è l'overhead del lookup nella tabella a runtime presente nel binding dinamico delle interfacce.

Domanda

Trait in Rust, cosa sono, come funzionano

Risposta

In Rust, il concetto di trait è fondamentale per definire comportamenti condivisi tra tipi diversi e per abilitare il polimorfismo.

- Un trait è definito come **un insieme di tipi di metodi**. È un concetto simile alle interfacce in Go o alle interfacce in Java.
- I trait sono utilizzati per implementare il **polimorfismo parametrico bounded** (vincolato), usando un meccanismo analogo ai templates/generics con vincoli, simile all'`extends` in Java per specificare che un tipo generico deve implementare una certa interfaccia.
- I trait possono avere tipi associati (`Associated Types`).

- È possibile definire **implementazioni di default** per i metodi all'interno di un trait. Quando un tipo implementa un trait con implementazioni di default, può scegliere di utilizzare l'implementazione predefinita o ridefinirla.
- I trait supportano l'ereditarietà: un trait può estendere un altro trait, includendo tutti i metodi definiti nel trait padre.

Come funzionano i Trait in Rust

Il funzionamento dei trait in Rust si basa su come vengono definiti, implementati e utilizzati come vincoli o trait objects.

1. **Definizione di un Trait:** Si definisce un trait specificando i metodi che i tipi che lo implementano devono fornire. La sintassi usa la parola chiave `trait`.

```
trait Measurable {  
    fn area(&self) -> f64;  
    // Metodo che accetta un riferimento immutabile al receiver  
    fn perimeter(&self) -> f64;  
    // Metodo che accetta un riferimento immutabile al receiver  
}
```

Nell'implementazione di un metodo in un trait, si usa `&self` (o `&mut self` o `self`) come segnaposto per l'argomento del receiver, poiché il tipo specifico non è noto nella definizione del trait.

2. **Implementazione di un Trait per un Tipo:** A differenza di Go, dove l'implementazione delle interfacce è automatica tramite **Duck Typing** (un tipo implementa un'interfaccia semplicemente definendone i metodi), Rust richiede una **sintassi esplicita** per dichiarare che un tipo implementa un certo trait.

```
struct Circle {  
    radius : f64,  
}  
  
impl Measurable for Circle {  
    // Implementazione esplicita  
    fn area(&self) -> f64 {  
        std::f64::consts::PI * self.radius * self.radius  
    }  
    fn perimeter(&self) -> f64 {  
        2 * std::f64::consts::PI * self.radius  
    }  
}
```

Per poter implementare un trait `T` per un tipo `U`, almeno uno tra `T` e `U` deve essere definito nello stesso file. Questo limita l'implementazione di trait esterni per tipi esterni, a

differenza di Go dove si può implementare un'interfaccia per un tipo definito altrove solo nel file dove si definiscono i metodi, talvolta usando tipi alias.

3. **Utilizzo dei Trait (Polimorfismo):** I trait vengono utilizzati principalmente in due modi per ottenere il polimorfismo: come vincoli per tipi generici (Static Dispatch) e come "Trait Objects" (Dynamic Dispatch).

- **Binding Statico (Compile-Time Dispatch):** Quando si definisce una funzione o una struttura dati generica vincolata da un trait, il compilatore genera codice specifico per ogni tipo concreto che soddisfa il vincolo e con cui la funzione/struttura viene utilizzata. Questo processo è chiamato **monomorfizzazione**.

```
// Funzione polimorfa con vincolo
fn print_measure<T : Measurable>(shape : T) {
// T deve implementare Measurable
    println!("Area = {}", shape.area());
    println!("Perimeter = {}", shape.perimeter());
}

// Tipo polimorfo con vincolo
struct Rectangle<T: PartialEq + Display> {
// T deve implementare PartialEq e Display
    width : T,
    height : T,
}
```

In questo caso, la chiamata al metodo (es. `shape.area()`) viene risolta in fase di compilazione (staticamente). Non c'è overhead a runtime dovuto al lookup del metodo, ma si possono avere tempi di compilazione maggiori e una dimensione maggiore dell'eseguibile a causa della duplicazione del codice per ogni istanza dei tipi.

- **Binding Dinamico (Run-time Dispatch):** È possibile utilizzare i trait per creare **Trait Objects**. Un trait object è un valore che può contenere qualsiasi tipo concreto che implementi un dato trait. Questo è tipicamente gestito tramite un puntatore o un riferimento a un trait object, come `&dyn Measurable` (il testo mostra `&Measurable`, che in Rust moderno è solitamente scritto `&dyn Measurable`).

```
// Funzione che accetta un riferimento a un trait object
fn print_measure_dynamic(shape : &dyn Measurable) {
    println!("Area = {}", shape.area());
    println!("Perimeter = {}", shape.perimeter());
}
```

Quando un metodo viene chiamato su un trait object (`shape.area()` in `print_measure_dynamic`), l'implementazione specifica del metodo non è nota in fase di compilazione. Il compilatore genera codice che, a tempo di esecuzione, utilizza una **tabella di metodi** (analoga a una virtual table o al concetto di coppia

"dati, codice" per le interfacce Go) associata al trait object per trovare e chiamare l'implementazione corretta per il tipo concreto sottostante. Questo è un esempio di **late binding** o binding dinamico. Il passaggio di un oggetto come trait object richiede un'indicazione esplicita da parte del programmatore. A differenza di linguaggi come Haskell, dove il "dizionario" dei metodi può essere passato una volta sola per tipo determinato dal compilatore, in Rust (e Go) l'oggetto (o il suo riferimento/trait object) porta con sé questo dizionario per ogni oggetto/valore che richiede dispatch dinamico.

In sintesi, i trait in Rust definiscono capacità e sono implementati esplicitamente dai tipi. Consentono sia un polimorfismo a binding statico (tramite monomorfizzazione di generici vincolati, efficiente ma con potenziale code bloat) sia un polimorfismo a binding dinamico (tramite trait objects, con l'overhead di un lookup a runtime ma maggiore flessibilità). Questo sistema offre un controllo granulare su dove e come viene gestito il polimorfismo e il dispatch dei metodi.

Domanda

Trait come vincoli di parametri di tipo: a cosa servono? Fare esempi.

Risposta

A Cosa Servono i Trait Come Vincoli di Parametri di Tipo in Rust

In Rust, i trait servono come vincoli per i parametri di tipo principalmente per abilitare il **polimorfismo parametrico bounded** (vincolato). Questo meccanismo è analogo all'uso di templates/generics con vincoli in altri linguaggi, come l'utilizzo di `extends` in Java per specificare che un tipo generico deve implementare una certa interfaccia.

Quando si definisce una funzione, una struttura dati o un'altra entità generica che opera su tipi sconosciuti `T`, l'uso di un trait come vincolo (`T: NomeTrait`) garantisce al compilatore che **il tipo concreto `T` che verrà utilizzato a runtime implementerà i metodi definiti in quel trait**. Questo permette di scrivere codice generico che può invocare in modo sicuro i metodi definiti nel trait sul parametro di tipo `T`, sapendo che tali metodi esisteranno per qualsiasi tipo concreto che soddisfi il vincolo.

In sintesi, usare i trait come vincoli serve a:

1. **Garantire la sicurezza dei tipi:** Il compilatore può verificare in fase di compilazione che solo tipi che implementano il trait specificato possano essere utilizzati con l'entità generica.
2. **Abilitare codice polimorfo sicuro e riutilizzabile:** Si può scrivere una singola implementazione di una funzione o struttura dati che funziona correttamente per una famiglia di tipi diversi, purché implementino il trait richiesto.
3. **Permettere il dispatch statico (Monomorfizzazione):** L'uso di trait come vincoli in funzioni e tipi generici porta tipicamente il compilatore a generare codice specifico per ogni tipo concreto con cui l'entità generica viene usata. Questo processo, chiamato

monomorfizzazione, evita l'overhead del binding dinamico a runtime, risultando in chiamate a metodo efficienti.

Come Funzionano i Trait Come Vincoli

Quando si definisce un'entità generica con un vincolo di trait, si utilizza la sintassi `NomeTipo<T: NomeTrait>` o `fn nome_funzione<T: NomeTrait>(param: T)`. Il compilatore, in fase di compilazione, analizza come l'entità generica viene utilizzata nel programma. Per ogni tipo concreto (es. `i32`, `String`, `Circle`) che viene passato come `T` e che implementa `NomeTrait`, il compilatore crea una versione specializzata dell'entità generica con quel tipo specifico al posto di `T`. Le chiamate ai metodi del trait all'interno del codice generico vengono così risolte in chiamate dirette alle implementazioni specifiche per quel tipo concreto.

Esempi di Trait Come Vincoli

1. **Funzione Polimorfa con Vincolo:** Viene presentata una funzione `print_measure` che accetta un argomento di tipo generico `T`. Il vincolo `T: Measurable` impone che `T` debba essere un tipo che implementa il trait `Measurable`:

```
trait Measurable {
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}

// ... implementazioni di Measurable per Square e Circle ...

fn print_measure<T : Measurable>(shape : T) {
    // T deve implementare Measurable
    println!("Area = {}", shape.area());
    println!("Perimeter = {}", shape.perimeter());
}
```

La funzione `print_measure` può quindi essere chiamata con un'istanza di `Square` o `Circle` (o qualsiasi altro tipo che implementi `Measurable`), perché il vincolo `T: Measurable` garantisce che i metodi `area()` e `perimeter()` siano disponibili su `shape`. Il compilatore genererà versioni di `print_measure` specifiche per `Square` e `Circle` (monomorfizzazione).

2. **Tipo Polimorfo con Vincoli Multipli:** Viene mostrata una struttura `Rectangle` definita con un parametro di tipo `T`. Questo parametro è vincolato a implementare i trait `PartialEq` e `Display`:

```
struct Rectangle<T> {
    width : T,
    height : T,
}

impl<T : PartialEq + Display> Rectangle<T> {
```

```
// T deve implementare PartialEq e Display
fn is_square(&self) -> bool {
    println!("Width = {}", self.width); // Richiede T: Display
    self.width == self.height // Richiede T: PartialEq
}
}
```

Qui, il tipo `T` deve soddisfare *entrambi* i vincoli. L'implementazione del metodo `is_square` sfrutta i metodi forniti da questi trait: `println!` (specificamente la macro `Debug` o `Display` usata con `{:?}` o `{}`) richiede che `T` implementi `Display`, e l'operatore di uguaglianza `==` richiede che `T` implementi `PartialEq`. Definendo questi vincoli, il compilatore sa che può utilizzare queste operazioni sul tipo `T` all'interno del blocco `impl`.

Questi esempi dimostrano come i trait, usati come vincoli, permettano di scrivere codice generico che opera su tipi diversi in modo sicuro ed efficiente, sfruttando il polimorfismo a tempo di compilazione (**monomorfizzazione**). Questo si distingue dall'uso di trait objects (`&dyn Trait`), che invece utilizzano il binding dinamico a runtime.

Si può anche fare un paragone con le Type Classes in Haskell, dove si usa una notazione simile (`NomeClasse => ...`) per imporre vincoli sui parametri di tipo (`a in Num a => a -> a`), garantendo la disponibilità di determinate operazioni (come `+`, `-`, `*`) per i tipi appartenenti alla classe. Sebbene i meccanismi interni di implementazione (dizionari passati una volta per tipo in Haskell vs. monomorfizzazione in Rust per i vincoli parametrici) differiscano, il concetto di base di usare una "classe di tipi" (trait/type class) come vincolo per il polimorfismo è analogo.

Domanda

Analogie e differenze tra oggetti, interfacce, trait, chiusure.

Risposta

I concetti di oggetti, interfacce, trait e chiusure rappresentano diversi approcci per organizzare codice e dati, spesso legati a paradigmi di programmazione differenti (orientata agli oggetti, funzionale) o a modi specifici di gestire il polimorfismo e l'incapsulamento in linguaggi moderni.

Oggetti

- Nel paradigma di **programmazione orientata agli oggetti**, i programmi sono visti come insiemi di oggetti che comunicano scambiandosi messaggi (attraverso metodi).
- Gli oggetti **incapsulano stato (campi) e comportamento (metodi)**.
- Nei linguaggi **class-based** (come C++, Java, C#), gli oggetti sono istanze di classi, che definiscono una struttura fissa per i campi e l'insieme dei metodi.
- Gli oggetti possono partecipare al **polimorfismo per sottotipo** (ad esempio, una

sottoclasse può essere usata dove ci si aspetta la superclasse).

- Nei linguaggi object-based (come Javascript), gli oggetti non derivano da classi e possono avere campi e metodi aggiunti dinamicamente.
- I **campi** degli oggetti sono tipicamente **mutabili**.
- I metodi degli oggetti in linguaggi class-based hanno un parametro implicito (`self` o `this`) che si riferisce all'oggetto ricevente.

Interfacce

- Le interfacce definiscono **"cosa sa fare" un determinato oggetto**.
- Specificano un **insieme di metodi** che le classi o i tipi che implementano l'interfaccia devono fornire.
- Permettono il **polimorfismo basato sul comportamento**, disaccoppiato dalla gerarchia di ereditarietà.
- In linguaggi come Java, una classe deve **dichiarare esplicitamente** di implementare un'interfaccia tramite la parola chiave `implements`.
- Non contengono stato (campi) né implementazione dei metodi, solo la loro firma.

Trait

- Il termine "Trait" viene usato in linguaggi come Rust. In Go, il concetto analogo viene chiamato **"interfaccia"**.
- Un trait (o interfaccia Go) è un **insieme di tipi di metodi**. Tutte le entità che implementano il trait/interfaccia supportano le operazioni definite.
- Definiscono **cosa un tipo può fare**, in modo simile alle interfacce.
- Rust utilizza i trait per il **polimorfismo parametrico bounded** (vincolato). Si usano i trait come vincoli sui parametri di tipo (`T: NomeTrait`) per garantire che il tipo `T` implementi i metodi del trait.
- Rust può usare i trait sia per il **dispatch statico (monomorfizzazione)**, generando codice specializzato per ogni tipo concreto, sia per il dispatch dinamico (con **trait objects**).
- In Go, l'implementazione di un'interfaccia (trait) è **implicita (Duck Typing)**: un tipo implementa un'interfaccia se definisce tutti i metodi richiesti, senza dichiarazione esplicita.
- In Go, un valore di tipo interfaccia è rappresentato internamente da una coppia (`dati`, `tabella dei metodi`), che consente il dispatch dinamico a runtime.
- A differenza delle interfacce, alcuni linguaggi (come Rust) permettono di definire **implementazioni di default** per i metodi nei trait. (Questa informazione non è esplicitamente nei PDF, ma è conoscenza comune di Rust e amplia il concetto).

Chiusure (Closures)

- Una chiusura è formata dal **codice di una funzione** più i **valori delle variabili libere** (quelle non definite all'interno della funzione stessa, ma accessibili dall'ambiente in cui è

stata creata).

- Nascono dalla possibilità di definire funzioni annidate o anonime che accedono a variabili definite nello scope esterno.
- Permettono a una funzione di essere invocata in un luogo e tempo diversi da dove è stata definita, **mantenendo l'accesso** ai dati dell'ambiente di definizione.
- Vengono utilizzate per **incapsulare** dati (le variabili libere) e comportamento (la funzione).
- Nei linguaggi funzionali puri, i valori delle variabili catturate dalla chiusura sono **immutabili**.
- Hanno un parametro implicito che si riferisce all'ambiente della chiusura dove trovare i valori delle variabili libere.

Analogie e Differenze

1. Polimorfismo e Comportamento:

- **Analogia:** Oggetti (tramite ereditarietà o interfaccia), Interfacce, e Trait sono tutti modi per ottenere il **polimorfismo basato sul comportamento**. Permettono di scrivere codice generico che opera su diversi tipi purché essi forniscano un certo insieme di operazioni. Le chiusure non si concentrano sul polimorfismo dei *dati* su cui operano, ma sul polimorfismo della funzione stessa (ad esempio, una funzione di ordine superiore che accetta una chiusura come argomento).
- **Differenza:** Il **meccanismo di polimorfismo** varia. L'ereditarietà opera per sottotipo. Interfacce/Trait operano definendo contratti comportamentali. Rust Traits possono portare a dispatch statico (monomorfizzazione) o dinamico, mentre Go Interfaces/Traits usano dispatch dinamico. Le chiusure abilitano il polimorfismo di funzioni di prima classe.

2. Incapsulamento:

- **Analogia:** Oggetti e Chiusure sono entrambi meccanismi di incapsulamento. Raggruppano dati e il codice che opera su quei dati.
- **Differenza:** Negli oggetti, i dati (campi) sono parte integrante della struttura dell'oggetto. Nelle chiusure, i dati (variabili libere) sono catturati dall'ambiente esterno alla funzione. I campi degli oggetti sono generalmente mutabili, mentre le variabili catturate dalle chiusure in contesti funzionali puri sono immutabili.

3. Stato vs. Contratto:

- **Differenza:** Oggetti contengono stato interno (campi). Interfacce e Trait (in sé) **non contengono stato**; definiscono un **contratto** per i tipi che invece contengono stato e implementano i metodi specificati. Le chiusure contengono lo stato catturato (valori delle variabili libere).

4. Natura dell'Entità:

- **Differenza:** Un oggetto è un'entità dati (un'istanza di una classe o una struttura con metodi). Un'interfaccia/Trait è una **specifica di un set di comportamenti**. Una chiusura è un'entità **comportamentale** (una funzione) che porta con sé il suo

ambiente dati. Come riassunto in un sorgente: le chiusure definiscono una funzione che può accedere a diversi dati, mentre i trait definiscono un dato per il quale disponiamo di diverse funzioni.

5. Implementazione e Vincoli:

- **Differenza:** L'implementazione di un'interfaccia può richiedere una dichiarazione esplicita (`implements` in Java), mentre in Go è implicita (`Duck Typing`). I trait in Rust possono essere usati come vincoli parametrici (`T: Trait`). L'implementazione dei trait in Rust ha regole specifiche sulla provenienza del tipo o del trait. L'implementazione di Type Classes in Haskell (concetto analogo a trait/interfacce) utilizza dizionari di funzioni passati implicitamente, con una differenza nel meccanismo di passaggio rispetto a Go/Rust. Le chiusure sono una caratteristica a livello di funzione, non richiedono implementazione esplicita come le interfacce/trait per i tipi.

Type Classes

Domanda

Type classes di Haskell: motivazione, definizione, implementazione, esempi.

Risposta

Haskell è presentato come un linguaggio funzionale fortemente tipato che sfrutta in modo fondamentale la *type reconstruction*, una forma avanzata di inferenza di tipi in cui il compilatore può dedurre interamente i tipi anche in assenza totale di annotazioni esplicite.

Motivazione

La motivazione principale per l'introduzione delle Classi di Tipo in Haskell nasce dalla necessità di gestire in modo più generico e scalabile il problema dell'**overloading di operatori e funzioni** su tipi diversi.

- Linguaggi con sistemi di tipi standard, come Standard ML o OCaml, incontrano difficoltà con operazioni comuni ma che si applicano a diversi tipi, come la moltiplicazione (`*`) o l'uguaglianza (`==`).
- In OCaml, ad esempio, per distinguere l'operazione tra interi e floating point è necessario usare operatori diversi (`*` per interi, `*.` per float), un approccio che non scala bene se si introducono nuovi tipi numerici.
- Analogamente, l'uguaglianza può avere significati diversi per tipi diversi (uguaglianza strutturale per record, uguaglianza referenziale per puntatori, ecc.), e sistemi come quello di Standard ML introducono notazioni ad hoc (apici singoli e doppi) per gestire queste proprietà, ma restano incomplete.
- Haskell cerca di risolvere questi problemi con meccanismi **più generici**, che non dipendano da soluzioni ad hoc.

Definizione

In Haskell, i tipi sono divisi in **classi**, che non sono necessariamente disgiunte. Una Classe di Tipo definisce un **set di operazioni (metodi)** che i tipi appartenenti a quella classe devono supportare. È un modo per raggruppare tipi basandosi sul loro **comportamento** comune.

La sintassi per definire una classe di tipo è la seguente:

```
class NomeClasse a where
  metodo1 :: firma_tipo1
  metodo2 :: firma_tipo2
  ...
```

Dove `a` è una variabile di tipo che viene vincolata ad appartenere a `NomeClasse`. Le firme dei metodi specificano i tipi degli argomenti e del risultato, spesso includendo la variabile di tipo `a` vincolata dalla classe.

Implementazione (Istanziamento e Compilazione)

- **Istanziamento:** Un tipo diventa un' **istanza** di una classe di tipo dichiarandolo esplicitamente tramite la parola chiave `instance`. Per ogni istanza, è necessario fornire le **realizzazioni concrete** (implementazioni) per tutti i metodi definiti nella classe.

```
instance NomeClasse TipoConcreto where
  metodo1 = implementazione_concreta1
  metodo2 = implementazione_concreta2
  ...
```

Le implementazioni concrete sono funzioni che operano sul `TipoConcreto`.

- **Implementazioni di Default:** È possibile fornire implementazioni di default per alcuni metodi all'interno della definizione della classe stessa. Se un'istanza non fornisce una implementazione specifica per un metodo con implementazione di default, verrà utilizzata quella predefinita.

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x == y = not (x /= y) -- Implementazione di default
  x /= y = not (x == y) -- Implementazione di default
```

Questo significa che, per rendere un tipo istanza di `Eq`, basta implementare uno solo tra `==` o `/=`, e l'altro verrà fornito automaticamente.

- **Istanze Generiche:** È possibile definire istanze per tipi polimorfi, come le liste. Per esempio, una lista di elementi di tipo `a` (`[a]`) può essere istanza della classe `Eq` se e solo se il tipo `a` è esso stesso istanza della classe `Eq`.

```
instance Eq a => Eq [a] where -- Vincolo: a deve essere Eq
  [] == [] = True
  (x :: xs) == (y :: ys) = x == y && xs == ys
  _ == _ = False
```

Questo definisce l'uguaglianza tra liste ricorsivamente, basandosi sull'uguaglianza dei loro elementi.

- **Sottoclassi:** È possibile definire sottoclassi, dove una classe estende un'altra. Ad esempio, la classe `Ord` (tipi ordinabili) è una sottoclasse di `Eq` (tipi confrontabili per uguaglianza).

```
class Eq a => Ord a where -- Ord è sottoclasse di Eq
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  ...
```

Questo richiede che un tipo sia prima istanza di `Eq` per poter essere istanza di `Ord`.

- **Implementazione del Compilatore:** Il compilatore Haskell gestisce le classi di tipo traducendole. Una classe di tipo `C` per un tipo `a` viene tradotta in un **record (un "dizionario")** contenente i puntatori alle implementazioni concrete dei metodi per quel tipo. Quando viene chiamata una funzione che ha un vincolo su una classe di tipo (es. `f :: Num a => a -> a -> a`), il compilatore aggiunge implicitamente un argomento a questa funzione: il dizionario delle operazioni per la classe `Num` relativa al tipo specifico che viene usato.

```
square :: Num a => a -> a
square x = x * x

result :: Int
result = square 3

-- Traduzione del compilatore (logica)

-- Dizionario
data Num a = Num { add :: a -> a -> a, mul :: a -> a -> a, neg :: a -> a }

-- Istanza concreta
numIntD :: Num Int = Num { add = addInt, mul = mulInt, neg = negInt }

-- Funzione con argomento implicito per il dizionario
square :: Num a -> a -> a

-- Usa il metodo 'mul' dal dizionario
square numD x = mul numD x x
```

```
result :: Int
result = square numIntD 3 -- Passa il dizionario per Int
```

Questa traduzione implica che il dizionario viene passato **una volta sola** come argomento alla funzione polimorfa. Questo differisce dall'implementazione di interfacce/trait in linguaggi come Go o Rust, dove il "dizionario" dei metodi (tabella dei metodi virtuale o parte del trait object) è tipicamente **associato al dato/oggetto stesso** e viene quindi passato implicitamente ad ogni chiamata di metodo su quell'oggetto.

Esempi di Classi di Tipo

Le sorgenti forniscono diversi esempi di classi di tipo comuni nella libreria standard di Haskell:

- **Num** : Per tipi numerici. Include operazioni come `(+)`, `(-)`, `(*)`, `negate`, `abs`. Permette di scrivere funzioni come `square :: Num a => a -> a` che funzionano per qualsiasi tipo numerico istanza di `Num`. Anche le costanti numeriche come `42` sono polimorfe e hanno tipo `Num a => a`.
- **Eq** : Per tipi per i quali è definita una nozione di uguaglianza. Include gli operatori `(==)` e `(/=)`. Permette funzioni come `member :: Eq a => a -> [a] -> Bool`.
- **Ord** : Per tipi ordinabili, è una sottoclasse di `Eq`. Include operatori di confronto come `(<)`, `(<=)`, `(>)`, `(>=)` e funzioni come `min`, `max`.
- **Show e Read** : Per tipi che possono essere rappresentati come stringhe (`Show`) o ricostruiti da stringhe (`Read`). La funzione `show :: Show a => a -> String` è analoga al `toString()` di Java. La funzione `read :: Read a => String -> a` è l'inversa, ma è necessariamente parziale (potrebbe fallire).
- **Foldable** : Per costruttori di tipo (come `List` o `Maybe`) che rappresentano contenitori i cui elementi possono essere "ridotti" a un singolo valore. Include operazioni come `length` e `sum`.
- **Functor** : Per costruttori di tipo che rappresentano contenitori su cui si può applicare una funzione a ogni elemento senza cambiare la struttura del contenitore. La funzione principale è `fmap :: (a -> b) -> t a -> t b`.
- **Monad** : Una classe di costruttori di tipo utilizzata per descrivere computazioni che possono avere effetti collaterali in un linguaggio puro come Haskell. Include le operazioni fondamentali `return` e `(>>=)` (`bind`). L'istanza `m in Monad m` è un **costruttore di tipo**, non un tipo semplice.

Le Classi di Tipo possono essere viste come un modo per ottenere **polimorfismo ad hoc (overloading)** o **polimorfismo parametrico vincolato** (bounded parametric polymorphism). Sono anche considerate concettualmente analoghe ai **Trait** presenti in altri linguaggi come Rust, sebbene l'implementazione sottostante (passaggio del dizionario dei metodi) differisca.

Domanda

Type classes su costruttori di tipo: esempi.

Risposta

Dopo l'introduzione delle classi di tipo classiche (definite su tipi come `Int`, `Float`, `[a]`), ci si è resi conto che poteva essere utile introdurre anche delle classi su **costruttori di tipo**.

Che cos'è un Costruttore di Tipo?

Un costruttore di tipo è una "funzione" a livello di tipi che applicata a uno o più tipi produce un nuovo tipo.

- Ad esempio, una lista da sola (`List` o `[]` in Haskell) non rappresenta un tipo completo, ma è un costruttore di tipo. Una lista associata a un tipo specifico, come una lista di interi (`List Int` o `[Int]`), è un tipo completo.
- Un ulteriore esempio di costruttore di tipo menzionato è `Maybe` (simile all'`Optional` in Java). `Maybe` da solo è un costruttore di tipo; `Maybe Int` è un tipo.

Le classi su costruttori di tipo permettono di definire comportamenti comuni per le "strutture" rappresentate da questi costruttori, indipendentemente dal tipo degli elementi contenuti.

Esempi di Classi di Tipo su Costruttori di Tipo:

1. Classe Foldable

- Definizione: `class Foldable t where ...` (dove `t` è una variabile su costruttori di tipo).
- Scopo: Rappresentare **strutture dati (contenitori)** che possono essere "ridotte" a un singolo valore. Ha senso applicare operazioni come il calcolo della lunghezza o la somma degli elementi su questi contenitori.
- Alcune operazioni (metodi) definite nella classe:
 - `length :: t a -> Int`
 - `sum :: Num a => t a -> a`
 - Sono presenti anche altre operazioni come `product` e `null`.
- L'istanza `t a` in `length :: t a -> Int` indica che il metodo opera su un tipo `t` applicato a un tipo `a` (es. `[Int]`, `Maybe Float`, ecc.).

2. Classe Functor

- Definizione: `class Functor t where ...` (dove `t` è una variabile su costruttori di tipo).
- Scopo: Rappresentare contenitori su cui puoi **applicare una funzione a ogni elemento senza cambiare la struttura del contenitore**.
- Operazione principale (metodo):
 - `fmap :: (a -> b) -> t a -> t b`. Questa funzione prende una funzione da `a` a `b` e la applica a ogni elemento di tipo `a` all'interno del contenitore `t a`, restituendo un contenitore `t b` con i risultati.
- Esempio di utilizzo: `fmap (+1)` restituisce ``.

3. Classe Monad

- Definizione: `class Monad m where ...` (dove `m` è una variabile su costruttori di tipo).
- Scopo: Descrivere matematicamente **computazioni che possono avere effetti collaterali** in un linguaggio puro come Haskell. L'istanza `m` non è un tipo semplice, ma un **costruttore di tipo**. Esempi di monadi menzionati sono `Maybe` (per gestire fallimenti/eccezioni), `Counter` (per gestione stato/contatori), e `Output` (per tracciare computazioni/gestione output).
- Le operazioni fondamentali (metodi) definite nella classe:
 - `return :: a -> m a`: Considerato un elemento neutro, rappresenta un'azione "pura" priva di effetti. Prende un valore puro e lo "impacchetta" nella monade.
 - `(>=>) :: m a -> (a -> m b) -> m b` (bind): Permette di concatenare computazioni monadiche. Prende una computazione nella monade `m a` e una funzione che accetta il risultato puro `a` e restituisce un'altra computazione nella monade `m b`, combinandole.
 - `(>>) :: m a -> m b -> m b` (and then): È una funzione utile quando si combinano due azioni monadiche e il risultato della prima non viene utilizzato dalla seconda; è importante solo la composizione degli effetti.

Domanda (Esercizio)

Cosa ci si può aspettare da una funzione `F` che ha tipo `T`?

Risposta

ciao

Domanda (Esercizio)

Qual è il tipo più generale che si può dare alla funzione `F`? (codice di `F` fornito o da realizzare)

Risposta

ciao

Domanda

Come sono implementate le type classes in Haskell? Analogie e differenze tra type classes, interfacce, trait.

Risposta

Implementazione delle Type Classes in Haskell

In Haskell, le classi di tipo non sono semplici astrazioni a livello di linguaggio che svaniscono dopo il type checking. Vengono tradotte dal compilatore in costrutti a runtime per gestire il polimorfismo.

Il compilatore Haskell traduce una dichiarazione di classe di tipo come `class Num a where add :: a -> a -> a ...` in un **record (o data structure, simile a una classe "nuda" in Java)**. Questo record, spesso chiamato **dizionario**, contiene come campi i puntatori alle implementazioni concrete delle funzioni (metodi) definite nella classe. Ad esempio, la classe `Num` genera un dizionario con campi per `add`, `mul`, `neg`, ecc..

Una dichiarazione di istanza, come `instance Num Int where add = addInt ...`, crea un'istanza specifica di questo dizionario. Questo dizionario (`numIntD` nell'esempio) contiene i puntatori alle funzioni primitive (`addInt`, `mulInt`, ecc.) che implementano le operazioni per il tipo `Int`. Analogamente, `instance Num Float ...` crea un dizionario `numFloatD` con le implementazioni per `Float`.

Una funzione vincolata da una classe di tipo, ad esempio `square :: Num a => a -> a` tradotta da `square x = x * x`, viene trasformata dal compilatore in una funzione che accetta un **argomento aggiuntivo esplicito**: il dizionario corrispondente al tipo `a`. La definizione di `square` diventa quindi concettualmente `square :: Num a -> a -> a`, dove il primo argomento `Num a` è il dizionario. All'interno del corpo della funzione tradotta, le chiamate ai metodi della classe (come `*` o `mul` nell'esempio) vengono risolte accedendo al campo appropriato del dizionario ricevuto.

Quando la funzione `square` viene invocata con un tipo concreto, ad esempio `square 3`, il compilatore, tramite l'inferenza di tipi, determina che il tipo concreto è `Int` e **passa automaticamente il dizionario corrispondente a `Int` (`numIntD`)** come primo argomento della funzione `square` tradotta.

Un punto chiave è che il dizionario viene **passato una sola volta** come argomento della funzione. Poiché il tipo `a` è fisso per tutta l'esecuzione della funzione, lo stesso dizionario viene riutilizzato per tutte le chiamate ai metodi della classe all'interno di quella specifica chiamata di funzione.

Questa tecnica di traduzione garantisce che il polimorfismo delle classi di tipo non abbia un costo a runtime significativo rispetto a una chiamata di funzione statica, una volta che il dizionario è stato determinato e passato.

Analogie tra Type Classes, Interfacce e Trait

Nonostante le differenze di implementazione e sintassi tra linguaggi diversi, Type Classes (Haskell), Interfacce (Java, Pascal) e Trait (Go, Rust) condividono un **obiettivo fondamentale**: definire un **contratto** o un insieme di **capacità** che i tipi o gli oggetti devono soddisfare. Permettono di raggruppare tipi diversi che supportano operazioni simili.

Tutti questi concetti consentono una forma di **polimorfismo basato sul comportamento** ("cosa un oggetto sa fare") piuttosto che sulla sola ereditarietà gerarchica ("cosa un oggetto è"). Questo aiuta a superare i problemi di **rigidità e fragilità** associati a gerarchie di ereditarietà strette, permettendo di scrivere codice più flessibile e riusabile.

In sintesi, fungono tutti da **specifiche** o **modelli** per descrivere un insieme di operazioni che possono essere applicate a diversi tipi, astrandosi dalla loro rappresentazione interna specifica.

Differenze tra Type Classes, Interfacce e Trait

Le differenze principali risiedono nella loro **implementazione**, nella **sintassi** e nel modo in cui si relazionano con i tipi che li "implementano".

1. Implementazione a Runtime:

- **Haskell Type Classes:** Vengono implementate tramite il **passaggio di un dizionario** (record di puntatori a funzioni) come argomento *aggiuntivo e implicito* (*gestito dal compilatore*) alle funzioni vincolate dalla classe. Il dizionario viene passato **una volta per chiamata di funzione**. Non c'è overhead per singola chiamata di metodo all'interno della funzione vincolata, ma l'overhead è nel passaggio iniziale del dizionario.
- **Go Trait/Interfacce:** Vengono rappresentati a runtime da una coppia (`dati`, `codice`). La parte `codice` è una tabella di metodi (simile a una virtual table) specifica per il tipo concreto dell'oggetto. Questa coppia viene passata **insieme al dato**. L'invocazione dei metodi avviene tramite **dynamic dispatch** (late binding) consultando la tabella a runtime.
- **Rust Traits:** Possono essere implementati in due modi:
 - **Monomorfizzazione:** Per il polimorfismo statico, il compilatore **duplica il codice** della funzione polimorfa per ogni combinazione di tipi concreti con cui viene usata. Non c'è alcun costo a runtime; è una **zero-cost abstraction**.
 - **Trait Objects (`dyn Trait`):** Per il polimorfismo dinamico, vengono gestiti in modo simile alle interfacce Go/Java, con una struttura che include il dato e un puntatore a una vtable/dizionario. Questo introduce dynamic dispatch a runtime.
- **Java Interfacce:** Implementate tramite **virtual tables** associate alla classe dell'oggetto. L'oggetto passato implicitamente contiene le informazioni necessarie per il dynamic dispatch.

2. Sintassi e Relazione con i Tipi:

- **Haskell Type Classes:** Si definisce una classe con `class ... where ...` e si dichiara che un tipo è un'istanza della classe con `instance ... where ...`. La relazione è **decoupled**: è possibile definire un'istanza per un tipo esistente e una classe esistente, potenzialmente in file diversi. Il programmatore non deve modificare la definizione originale del tipo.
- **Go Trait/Interfacce:** Si definisce un'interfaccia con `type InterfaceName interface { ... }`. L'implementazione è **implicita (Duck Typing)**: un tipo implementa un'interfaccia se definisce tutti i metodi richiesti, senza bisogno di una dichiarazione esplicita (`implements`). Questo è molto flessibile.
- **Rust Traits:** Si definisce un trait con `trait TraitName { ... }` e si implementa per un tipo con `impl TraitName for TypeName { ... }`. L'implementazione è

esplicita. È richiesto che almeno il trait o il tipo siano definiti nel file corrente per l'implementazione. Meno flessibile di Go.

- **Java Interfacce:** Si definisce un'interfaccia con `interface InterfaceName { ... }`. L'implementazione è **esplicita**: una classe deve dichiarare `implements InterfaceName`. La relazione è strettamente accoppiata tramite questa dichiarazione.

3. Implementazioni di Default:

- **Haskell Type Classes:** Supportano implementazioni di default per i metodi della classe.
- **Go Trait/Interfacce:** (Basato su conoscenze esterne, non esplicitamente nei testi) Le interfacce Go non supportano implementazioni di default.
- **Rust Traits:** Supportano implementazioni di default per i metodi del trait.
- **Java Interfacce:** (Basato su conoscenze esterne) Supportano implementazioni di default a partire da Java 8.

4. Ereditarietà/Composizione:

- **Haskell Type Classes:** Supportano l'ereditarietà tramite sottoclassi (`class Eq a => Ord a where ...`).
- **Go Trait/Interfacce:** Supportano la composizione di interfacce tramite embedding.
- **Rust Traits:** Supportano l'ereditarietà/composizione di trait (`trait FooBar : Foo { ... }`).
- **Java Interfacce:** Supportano l'estensione di interfacce.

In sintesi:

- **Haskell Type Classes** offrono un potente polimorfismo con forte supporto all'inferenza di tipi e un'implementazione basata su dizionari passati dal compilatore. Sono decoupled rispetto alle definizioni di tipi.
- **Go Trait/Interfacce** offrono Duck Typing implicito e un'implementazione basata su coppie (`data, codice`) per dynamic dispatch. Sono molto flessibili nella relazione tra interfacce e tipi.
- **Rust Traits** offrono sia polimorfismo statico (monomorfizzazione) che dinamico (trait objects) con implementazione esplicita e un sistema di tipi rigido (ownership, borrowing) che garantisce sicurezza a compile-time.
- **Java Interfacce** offrono un contratto esplicito per il polimorfismo basato su oggetti con implementazione esplicita e dynamic dispatch basato su vtables di classe.

Monadi

Domanda

Monadi in Haskell: motivazione, definizione, implementazione, esempi.

Risposta

Le Monadi in Haskell nascono come una soluzione elegante per gestire effetti collaterali (come Input/Output, gestione di errori, stato mutabile, non-determinismo) in un linguaggio che è fondamentalmente **puro**.

Motivazione:

Haskell è un linguaggio funzionale con caratteristiche chiave come la **trasparenza referenziale** e la **lazy evaluation**.

- La trasparenza referenziale implica che un'espressione può essere sostituita dal suo valore senza cambiare il comportamento del programma. Questo è possibile perché le funzioni pure restituiscono sempre lo stesso output per gli stessi input e non hanno effetti collaterali visibili esternamente.
- La lazy evaluation valuta le espressioni solo quando i loro risultati sono effettivamente necessari. Questo può migliorare l'efficienza e la modularità. Un esempio classico è l'implementazione di operatori logici o funzioni su liste infinite, dove solo le parti necessarie vengono calcolate.

Queste proprietà portano a vantaggi come la facilità di parallelizzazione. Tuttavia, rendono difficile o impossibile eseguire azioni con effetti collaterali, come stampare a schermo, leggere input dall'utente, interagire con file system o gestire eccezioni in modo sequenziale e prevedibile all'interno della logica pura del programma.

In un linguaggio puramente funzionale, un programma non "fa" esplicitamente qualcosa con effetti collaterali, ma piuttosto **crea una descrizione delle azioni** che dovrebbero essere eseguite. Questa descrizione viene poi delegata a un sistema esterno (come il runtime di Haskell o il sistema operativo) per l'esecuzione effettiva. Le Monadi forniscono un framework standard per costruire queste descrizioni di azioni e per combinarle in sequenza.

Come dimostrato con l'esempio del valutatore di espressioni, tentare di gestire manualmente effetti come la divisione per zero, il conteggio delle operazioni o il tracciamento dei passi in codice puramente funzionale porta a una **rapida complicazione e stravolgimento della struttura del programma**, rendendo la logica principale difficile da seguire. Le Monadi offrono un modo strutturato per astrarre e gestire questi effetti.

Definizione:

Concettualmente, una Monade è un costrutto che permette di **descrivere computazioni che possono avere effetti collaterali** e di **combinare sequenzialmente** queste computazioni.

In Haskell, una Monade è definita tramite la **classe di tipo Monad**. Questa classe si applica a **costruttori di tipo** m (che prendono un tipo a e producono un tipo $m\ a$, dove m incapsula l'effetto).

La classe `Monad` richiede l'implementazione di due operatori fondamentali:

- `return :: a -> m a` : Prende un valore puro `a` e lo "impacchetta" in un contesto monadico `m a`. Rappresenta una computazione nel contesto `m` che semplicemente produce il valore `a` senza effetti collaterali noti all'interno del contesto stesso (descrive un'azione "pura" all'interno della monade).
- `(>>=) :: m a -> (a -> m b) -> m b` : L'operatore di **bind**. Prende una computazione monadica `m a` e una funzione `f` che accetta il risultato `a` della prima computazione e produce una *nuova* computazione monadica `m b`. L'operatore `>>=` specifica come le due computazioni vengono eseguite in sequenza e come il risultato della prima (`a`) viene passato per influenzare la seconda (`f a`).

Affinché un costruttore di tipo `m` sia un'istanza valida della classe `Monad`, le sue implementazioni di `return` e `>>=` devono soddisfare tre **leggi** (identità sinistra, identità destra, associatività) che garantiscono un comportamento coerente per la sequenziazione delle azioni.

Implementazione:

L'implementazione delle Monadi in Haskell si basa sul meccanismo sottostante delle **Type Classes**. Come descritto per le Type Classes in generale, quando una funzione è vincolata da una classe di tipo (come `Monad m => ...`), il compilatore traduce questa funzione in una che accetta un **dizionario** come argomento aggiuntivo. Questo dizionario contiene i puntatori alle implementazioni concrete dei metodi (nel caso di `Monad`, `return` e `>>=`) per lo specifico costruttore di tipo `m` con cui la funzione viene utilizzata.

Quindi, quando si usa `>>=` (o `return`) con un tipo monadico concreto (es. `Maybe Int`), il compilatore seleziona l'implementazione corretta di `>>=` (o `return`) definita nell'istanza `instance Monad Maybe where ...`, accedendo al puntatore alla funzione appropriata tramite il dizionario della classe di tipo `Monad Maybe`.

Questo passaggio del dizionario avviene **una sola volta** per chiamata di funzione vincolata dalla classe di tipo, poiché il tipo `m` è fissato all'interno di quella funzione. All'interno del corpo della funzione, le chiamate ai metodi monadici vengono risolte tramite questo dizionario.

La notazione `do` è puro **zucchero sintattico** fornito dal compilatore Haskell. Un blocco `do` viene automaticamente tradotto dal compilatore in una sequenza di chiamate all'operatore `>>=` (e `return` per le espressioni finali o i binding senza `<-`). Questo rende il codice monadico più simile a quello imperativo, migliorando la leggibilità.

Esempi:

1. **La Monade Maybe** : Gestisce computazioni che possono fallire.

- **Definizione dell'istanza:** `return` è `Just`, `>>=` propaga `Nothing` in caso di fallimento della prima computazione e applica la funzione al valore incapsulato in `Just` altrimenti.

- **Uso:** L'esempio del valutatore di espressioni `evalM :: Expr -> Maybe Int` usa `Maybe` per gestire la divisione per zero, ritornando `Nothing` in caso di errore. Il codice diventa più pulito rispetto alla gestione manuale con `case Nothing of ...`.

2. **La Monade Counter** : Gestisce computazioni che mantengono e modificano un contatore o uno stato.

- **Definizione concettuale:** Una computazione che restituisce una coppia (valore, nuovo_stato) dato lo stato iniziale, rappresentata come `Int -> (a, Int)`. Nella versione monadica, è `Counter a = Int -> (a, Int)` o equivalentemente `Counter a = State Int a` in librerie standard (questo ultimo è conoscenza esterna).
- **Definizione dell'istanza:** `return` incapsula il valore senza modificare lo stato, `>=>` passa lo stato modificato dalla prima computazione alla seconda. Viene introdotta un'operazione specifica `tick` per incrementare il contatore.
- **Uso:** L'esempio del valutatore `evalM :: Expr -> Counter Int` usa `Counter` per contare le operazioni di divisione, incapsulando la logica del contatore separatamente dalla valutazione.

3. **La Monade Output** : Gestisce computazioni che producono un output log/trace.

- **Definizione concettuale:** Una computazione che restituisce una coppia (output_accumulato, valore), rappresentata come `(String, a)`.
- **Definizione dell'istanza:** `return` incapsula il valore con una stringa vuota, `>=>` concatena le stringhe di output delle due computazioni e passa il valore della prima alla seconda. Viene introdotta un'operazione specifica `output` per aggiungere testo all'output.
- **Uso:** L'esempio del valutatore `evalM :: Expr -> Output Int` usa `Output` per tracciare i passi di valutazione, accumulando la stringa di log.

Note aggiuntive:

- L'operatore `and-then (>>)` : Utile per sequenziare due azioni quando il risultato della prima non è necessario per la seconda. È definibile in termini di `>=>` e `const`.
- **Monad Transformers:** Come `MaybeT m a = m (Maybe a)`, permettono di combinare gli effetti di due monadi. Ad esempio, `MaybeT Counter` (nominata `MaybeCounter` nell'esempio) combina il fallimento con il contatore. Richiedono la classe di tipo `MonadT` e la funzione `lift` per "sollevare" le azioni della monade interna.
- **La Monade Id** : Una monade banale (`type Id a = a`) usata per rappresentare computazioni pure senza effetti. Può servire come base per costruire altre monadi o trasformazioni di monadi.

In conclusione, le Monadi in Haskell, implementate come Type Classes con operatori `return` e `>=>` che rispettano specifiche leggi, forniscono un potente strumento per gestire in modo strutturato e componibile gli effetti collaterali all'interno di un linguaggio puro, mantenendo chiara la logica principale del programma separata dalla gestione degli effetti. La notazione `do` semplifica notevolmente l'utilizzo quotidiano.

Domanda

Leggi di una monade: enunciato, dimostrazione per monadi specifiche.

Risposta

Le Monadi in Haskell sono state introdotte per gestire computazioni che possono avere effetti collaterali in un linguaggio altrimenti puro, come l'I/O, la gestione degli errori o il mantenimento di uno stato. Concettualmente, una Monade permette di **descrivere computazioni con effetti collaterali** e di **combinarle in sequenza**.

Il concetto di Monade si lega, ma si distingue, da quello di **monoide**, una struttura algebrica con un'operazione associativa e un elemento neutro. Mentre un monoide cattura l'idea di comporre azioni senza trasferire informazioni significative tra di esse, una Monade richiede la possibilità di trasferire il risultato di una computazione alla successiva.

Per garantire che gli operatori fondamentali di una Monade (`return` e `>>=`) si comportino in modo coerente per la sequenziazione delle azioni e la gestione degli effetti, essi devono soddisfare un insieme di tre **leggi**.

Enunciato delle Leggi di una Monade:

La classe di tipo `Monad` in Haskell definisce l'interfaccia per un costruttore di tipo `m` che rappresenta un contesto monadico. Questa classe richiede l'implementazione degli operatori `return` e `>>=`:

- `return :: a -> m a`: Impacchetta un valore puro `a` in un contesto monadico `m a`. Rappresenta una computazione "pura" all'interno del contesto monadico specificato.
- `(>>=) :: m a -> (a -> m b) -> m b`: L'operatore di **bind**. Combina due computazioni monadiche in sequenza, passando il risultato della prima (`a`) alla seconda, che produce una nuova computazione monadica.

Affinché un costruttore di tipo `m` sia un'istanza valida di `Monad`, le sue implementazioni di `return` e `>>=` devono obbedire alle seguenti tre leggi:

1. **Identità Sinistra (Left Identity)**: `return a >>= f ≡ f a` Questa legge afferma che impacchettare un valore puro `a` con `return` e poi passarlo a una funzione monadica `f` tramite `>>=` è equivalente a semplicemente applicare `f` al valore `a`. Cioè, `return` agisce come un'identità "a sinistra" rispetto a `>>=`.
2. **Identità Destra (Right Identity)**: `m >>= return ≡ m` Questa legge afferma che prendere una computazione monadica `m` e passare il suo risultato all'operatore `return` tramite `>>=` è equivalente alla computazione `m` stessa. Cioè, `return` agisce come un'identità "a destra" rispetto a `>>=`. Entrambe le prime due leggi indicano che `return` rappresenta un'azione pura, priva di effetti nel contesto, che non altera la computazione.
3. **Associatività (Associativity)**: `(m >>= \a -> n) >>= f ≡ m >>= (\a -> n >>= f)` Questa legge afferma che la composizione sequenziale di tre computazioni monadiche (`m`, seguita da `n`, seguita da una funzione `f` applicata al risultato di `n`) può essere

raggruppata in due modi equivalenti. L'ordine delle azioni viene preservato.

L'associatività è valida purché la variabile `a` non compaia libera nella funzione `f`.

Dimostrazione per Monadi Specifiche:

Le leggi di una Monade devono essere dimostrate per ogni specifica istanza del costruttore di tipo. Le fonti fornite presentano la **dimostrazione completa per la monade `Maybe`**.

Ricordiamo le definizioni di `return` e `>>=` per la monade `Maybe`:

- `return :: a -> Maybe a` `return = Just (return a = Just a)`
- `(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b` `(>>=) Nothing _ = Nothing`
`(>>=) (Just a) f = f a`

Dimostriamo ora le tre leggi per `Maybe` basandoci sui passaggi descritti nelle fonti:

1. Identità Sinistra: `return a >>= f ≡ f a`

- Lato sinistro: `return a >>= f`
- Usando la definizione di `return` per `Maybe` (`return a = Just a`): `Just a >>= f`
- Usando la seconda clausola della definizione di `(>>=)` per `Maybe` (`(>>=) (Just value) function = function value`): `f a`
- Quindi, `return a >>= f` si riduce a `f a`. La legge è dimostrata.

2. Identità Destra: `m >>= return ≡ m` Questa legge richiede la dimostrazione per i due possibili casi di `m`, poiché `Maybe a` può essere `Nothing` o `Just value`:

- **Caso 1: `m = Nothing`**
 - Lato sinistro: `m >>= return` che diventa `Nothing >>= return`
 - Usando la prima clausola della definizione di `(>>=)` per `Maybe` (`(>>=) Nothing _ = Nothing`): `Nothing`
 - Lato destro: `m` che è `Nothing`
 - I lati sinistro e destro sono uguali (`Nothing ≡ Nothing`), quindi la legge vale per questo caso.
- **Caso 2: `m = Just a`**
 - Lato sinistro: `m >>= return` che diventa `Just a >>= return`
 - Usando la seconda clausola della definizione di `(>>=)` per `Maybe` (`(>>=) (Just value) function = function value`): `return a`
 - Usando la definizione di `return` per `Maybe` (`return a = Just a`): `Just a`
 - Lato destro: `m` che è `Just a`
 - I lati sinistro e destro sono uguali (`Just a ≡ Just a`), quindi la legge vale anche per questo caso.

Poiché la legge vale per entrambi i casi possibili di `m`, è dimostrata per la monade `Maybe`.

3. Associatività: `(m >>= \a -> n) >>= f ≡ m >>= (\a -> n >>= f)` Questa legge, come la precedente, richiede la dimostrazione per i due casi possibili di `m`.

- **Caso 1: $m = \text{Nothing}$**

- **Lato Sinistro:** $(m \gg= \backslash a \rightarrow n) \gg= f$ che diventa $(\text{Nothing} \gg= \backslash a \rightarrow n) \gg= f$
- Usando la prima clausola di $(\gg=)$ $((\gg=) \text{Nothing } _ = \text{Nothing}): \text{Nothing} \gg= f$
- Usando di nuovo la prima clausola di $(\gg=)$: Nothing
- **Lato Destro:** $m \gg= (\backslash a \rightarrow n \gg= f)$ che diventa $\text{Nothing} \gg= (\backslash a \rightarrow n \gg= f)$
- Usando la prima clausola di $(\gg=)$: Nothing
- I lati sinistro e destro sono uguali $(\text{Nothing} \equiv \text{Nothing})$, quindi la legge vale per questo caso.

- **Caso 2: $m = \text{Just } a$**

- **Lato Sinistro:** $(m \gg= \backslash a \rightarrow n) \gg= f$ che diventa $(\text{Just } a \gg= \backslash a \rightarrow n) \gg= f$
- Usando la seconda clausola di $(\gg=)$ $((\gg=) (\text{Just value}) \text{function} = \text{function value}): (\backslash a \rightarrow n) a$
- Applicando la funzione lambda: n
- La parte sinistra diventa quindi $n \gg= f$.
- **Lato Destro:** $m \gg= (\backslash a \rightarrow n \gg= f)$ che diventa $\text{Just } a \gg= (\backslash a \rightarrow n \gg= f)$
- Usando la seconda clausola di $(\gg=)$: $(\backslash a \rightarrow n \gg= f) a$
- Applicando la funzione lambda: $n \gg= f$
- I lati sinistro e destro sono uguali $(n \gg= f \equiv n \gg= f)$, quindi la legge vale anche per questo caso.

Poiché la legge vale per entrambi i casi possibili di m , è dimostrata per la monade `Maybe`.

Avendo dimostrato che l'implementazione di `return` e `>>=` per `Maybe` soddisfa tutte e tre le leggi, `Maybe` è una valida monade.

Domanda

Scrittura di funzioni con monadi

Risposta

Le monadi vengono introdotte in linguaggi funzionali puri, come Haskell, per gestire effetti collaterali (Input/Output, gestione errori, stato, ecc.) che non sarebbero altrimenti possibili in un ambiente dove le funzioni producono sempre lo stesso output per gli stessi input (trasparenza referenziale). Invece di *fare* esplicitamente un'azione impura, il programma *crea* azioni che, se eseguite dal sistema (ad esempio, il runtime Haskell), hanno l'effetto desiderato.

Per scrivere funzioni che operano in un contesto monadico, si utilizzano principalmente due operatori fondamentali, definiti dalla classe di tipo `Monad` :

1. **`return`**: `return :: a -> m a` Questa funzione prende un valore puro `a` e lo "impacchetta" in un contesto monadico `m a`. Rappresenta una computazione nel contesto `m` che non ha effetti collaterali significativi e produce semplicemente il valore `a`. È concettualmente l'operazione per introdurre un valore puro nel contesto monadico.
2. **`>>=` (`bind`)**: `(>>=) :: m a -> (a -> m b) -> m b` Questo operatore, chiamato "bind", è il cuore della composizione monadica. Prende una computazione monadica `m a` e una funzione `a -> m b` (una funzione che dato il risultato puro della prima computazione `a` produce una nuova computazione monadica `m b`) e le combina in sequenza. Il risultato della prima computazione (`a`) viene "legato" al parametro della funzione successiva. Questo permette di far dipendere la computazione successiva dal risultato della precedente, trasferendo informazioni tra le azioni.

Affinché un costruttore di tipo `m` possa essere un'istanza valida della classe `Monad`, le sue implementazioni di `return` e `>>=` devono soddisfare tre leggi fondamentali (Identità Sinistra, Identità Destra, Associatività), che garantiscono un comportamento coerente per la sequenziazione e la gestione degli effetti. Le fonti dimostrano esplicitamente queste leggi per la monade `Maybe`.

Scrittura di funzioni tramite `return` e `>>=` (e operazioni specifiche della monade):

L'approccio generale per scrivere una funzione che opera all'interno di una monade `M` è il seguente:

- Se il risultato è un valore puro e non si desidera alcun effetto, si usa `return value`.
- Se si ha una computazione monadica `monadicAction1 :: M a` e si vuole usare il suo risultato `a` per eseguire un'altra computazione monadica `action2 :: a -> M b`, si usa l'operatore `>>=`: `monadicAction1 >>= \a -> action2 a`. La lambda `\a -> ...` cattura il risultato `a` della prima azione.
- Si utilizzano operazioni specifiche definite per quella particolare monade per produrre effetti collaterali specifici (ad esempio, `abort` per `Maybe`, `tick` per `Counter`, `output` per `Output`).

Le fonti illustrano questo con l'esempio della funzione `evalM` che valuta espressioni aritmetiche in diversi contesti monadici.

- **Monade `Maybe` (gestione errori/fallimento)**: `evalM :: Expr -> Maybe Int`
`evalM (Const n) = return n` -- Impacchetta il valore puro `n`
`evalM (Div t s) = evalM t >>= \m -> evalM s >>= \n -> if n == 0 then abort else return (m 'div' n)`
Qui, `evalM t` produce una computazione `Maybe Int`. `>>= \m -> ...` estrae il risultato `m` (se presente) e lo rende disponibile nella lambda. Analogamente per `evalM s` e il risultato `n`. La divisione avviene solo se `n` non è zero; altrimenti si usa `abort`.
- **Monade `Counter` (stato mutabile/contatore)**: `evalM :: Expr -> Counter Int`
`evalM (Const n) = return n` -- Impacchetta il valore puro `n`
`evalM (Div t s) =`

`evalM t >=> \m -> evalM s >=> \n -> tick >=> \() -> return (m 'div' n)` La computazione `evalM t` in questo caso è una funzione `Int -> (Int, Int)` (il contatore in ingresso e la coppia (risultato, contatore in uscita)). L'operatore `>=>` per `Counter` gestisce il passaggio automatico del contatore da una computazione alla successiva. `tick` è un'operazione specifica di `Counter` che incrementa il contatore.

- **Monade Output (tracciamento/log):** `evalM :: Expr -> Output Int` `evalM (Const n) = output (line (Const n) n) >=> \() -> return n` `evalM (Div t s) = evalM t >=> \m -> evalM s >=> \n -> output (line (Div t s) (m 'div' n)) >=> \() -> return (m 'div' n)` Qui, `evalM t` e `evalM s` producono risultati accoppiati con una stringa di output. L'operatore `>=>` per `Output` concatena automaticamente le stringhe di output. `output` è un'operazione specifica di `Output` che aggiunge una stringa al log.

Come si può notare, la struttura delle funzioni `evalM` nei tre casi è notevolmente simile, nonostante gestiscano effetti collaterali diversi. Questo è uno dei principali vantaggi delle monadi: forniscono un *pattern* strutturato per la composizione di computazioni con effetti.

Notazione do (Zucchero Sintattico):

Scrivere lunghe catene di `>=>` con funzioni lambda può diventare verboso. Per semplificare la scrittura di funzioni monadiche, Haskell e altri linguaggi offrono la **notazione do**.

La notazione `do` è puro zucchero sintattico per le chiamate a `>=>` e `>>` (quest'ultimo usato quando il risultato della prima azione non è necessario per la seconda). Un blocco `do` è una sequenza di azioni monadiche:

```
do
  azione1
  azione2
  ...
  azioneN
```

Se un'azione produce un risultato che deve essere utilizzato nella sequenza, si usa la sintassi `variabile <- azione`:

```
do
  risultato1 <- azione1
  risultato2 <- azione2 risultato1
  azione3 risultato1 risultato2
  return valore_finale
```

Questo si traduce automaticamente in chiamate a `>=>`. Ad esempio, il codice `evalM` per `Maybe` scritto con `>=>` può essere riscritto in notazione `do`:

```
evalM :: Expr -> Maybe Int
evalM (Const n) = return n
```



```
evalM (Div t s) = do
  m <- evalM t -- Esegue evalM t >=> \m -> ...
  n <- evalM s -- Esegue il risultato del >=> precedente >=> \n -> ...
  if n == 0 then abort else return (m 'div' n)
-- ... \n -> if n == 0 then abort else return (m 'div' n)
```

Questo rende il codice monadico più lineare e simile alla programmazione imperativa, facilitando la lettura.

In sintesi, scrivere funzioni con le monadi implica definire computazioni che restituiscono valori "impacchettati" in un contesto monadico (`m a`), utilizzare `return` per introdurre valori puri nel contesto e `>=>` (o la notazione `do`) per sequenziare azioni monadiche, passando i risultati da un passo all'altro e gestendo gli effetti collaterali specifici della monade in uso. Questo approccio permette di strutturare in modo coerente la gestione degli effetti in un linguaggio funzionale puro.

Per scrivere funzioni che gestiscono *effetti multipli* contemporaneamente (ad esempio, sia fallimento che contatore), le fonti introducono il concetto di **trasformatori di monadi** (Monad Transformers, es. `MaybeT`, `CounterT`) che "aggiungono" un effetto a una monade esistente. In questo caso, per usare le operazioni della monade "interna", è necessario utilizzare la funzione `lift`.

Domanda (Esercizio)

Implementazione di monadi (quelle discusse a lezione, presentate nelle slide o nell'articolo di Wadler, altre definite "al volo")

Risposta

ciao

Domanda

Il costrutto `do` : significato, uso, esempi, traduzione con `>=>` e `>>`.

Risposta

Il **costrutto** `do` in linguaggi funzionali come Haskell è principalmente **zucchero sintattico**. Non introduce nuove funzionalità, ma fornisce un modo più leggibile e lineare per scrivere sequenze di operazioni che operano all'interno di un **contesto monadico**.

Il suo significato e uso sono strettamente legati alle **monadi**, che sono concetti utilizzati per strutturare computazioni che possono avere **effetti collaterali** (come Input/Output, gestione errori, stato, tracciamento, ecc.) in un linguaggio altrimenti **puro**. In un linguaggio puro, le funzioni restituiscono sempre lo stesso output per gli stessi input (trasparenza referenziale) e non possono direttamente modificare lo stato o interagire con l'esterno. Le monadi consentono di descrivere queste interazioni o modifiche come "azioni" che vengono create dal programma puro e successivamente eseguite dal sistema.

Le operazioni fondamentali per lavorare con le monadi sono:

- **return** : prende un valore puro (`a`) e lo "impacchetta" nel contesto monadico (`M a`), rappresentando una computazione pura all'interno della monade.
- **>>= (bind)**: permette di sequenziare due computazioni monadiche (`M a` e `a -> M b`), passando il risultato puro (`a`) della prima come input alla funzione che produce la seconda computazione (`M b`). Questo è il meccanismo per concatenare azioni monadiche dipendenti dal risultato della precedente.

Scrivere lunghe sequenze di operazioni monadiche usando esplicitamente l'operatore `>>=` e funzioni lambda (`\a -> ...`) può diventare complesso e difficile da leggere. Qui interviene la **notazione do**.

Significato e Uso del Costrutto `do` :

Un blocco `do` rappresenta una sequenza di azioni monadiche che verranno eseguite una dopo l'altra nel contesto definito dalla monade. L'aspetto sequenziale ricorda la programmazione imperativa, ma in realtà sta solo costruendo una descrizione della computazione.

All'interno di un blocco `do` :

- Un'azione monadica il cui **risultato è necessario** per i passi successivi viene scritta usando la sintassi `variabile <- azione_monadica`. Il valore "scompattato" dal contesto monadico viene legato a `variabile` e può essere usato nelle righe successive del blocco `do`.
- Un'azione monadica il cui **risultato non è necessario** per i passi successivi viene semplicemente scritta come `azione_monadica`.
- L'ultima riga del blocco `do` è tipicamente l'azione monadica finale o un `return` del valore puro risultante.

Esempi:

Gli esempi forniti nei testi si concentrano sul valutatore di espressioni `evalM`. Consideriamo l'esempio per la monade `Maybe`, che gestisce i fallimenti dovuti alla divisione per zero.

La funzione `evalM` per `Maybe` scritta con `>>=` è:

```
evalM :: Expr -> Maybe Int
evalM (Const n) = return n
evalM (Div t s) =
  evalM t >>= \m ->
    -- Esegue evalM t, lega il risultato a m
    evalM s >>= \n ->
    -- Esegue evalM s, lega il risultato a n
```

```
if n == 0 then abort else return (m 'div' n)
-- Usa m e n, produce il risultato finale Maybe Int
```

Usando la **notazione** `do`, lo stesso codice diventa:

```
evalM :: Expr -> Maybe Int
evalM (Const n) = return n
evalM (Div t s) = do
  m <- evalM t
  -- Esegue evalM t, lega il risultato a m (corrisponde a >=> \m -> ...)
  n <- evalM s
  -- Esegue evalM s, lega il risultato a n (corrisponde a >=> \n -> ...)

  if n == 0 then abort else return (m 'div' n)
  -- L'ultima espressione è il risultato del blocco do
```

Questo esempio mostra chiaramente come la notazione `do` rende la sequenza di operazioni più lineare e intuitiva.

Un altro esempio dall'origine mostra l'uso implicito dell'operatore `>>` nella monade `Counter`: La versione con `>=>` era:

```
evalM :: Expr -> Counter Int
evalM (Const n) = return n
evalM (Div t s) =
  evalM t >=> \m ->
  evalM s >=> \n ->
  tick >=> \() ->
  -- Esegue tick, ma il risultato (Unit, ()) non è necessario

  return (m 'div' n)
```

Usando l'operatore `>>` (definito come `m >> n = m >=> const n`, dove `const n = _ -> n`), che ignora il risultato della prima azione, la riga `tick >=> \() -> ...` diventa `tick >> ...`. Questo si riflette nella notazione `do`. Se un'azione è scritta semplicemente come `azione_monadica` (senza variabile `<-`), significa che il suo risultato viene ignorato, corrispondendo all'uso dell'operatore `>>`.

Traduzione con `>=>` e `>>`:

La notazione `do` viene tradotta dal compilatore in una serie annidata di chiamate a `>=>` (e a `>>` per le azioni il cui risultato viene scartato).

In generale, un blocco `do`:

```
do
  azione1
  variabile2 <- azione2
  variabile3 <- azione3 variabile2
  azione4 variabile3
  return risultato
```

viene tradotto, passo dopo passo, in:

```
azione1 >>
  (azione2 >=> \variabile2 ->
    (azione3 variabile2 >=> \variabile3 ->
      (azione4 variabile3 >>
        return risultato))))
```

(Nota: le parentesi nella traduzione non sono sempre letterali in Haskell ma rappresentano la nidificazione implicita creata da `>=>`).

In sintesi, il costrutto `do` è un potente **zucchero sintattico** in Haskell e linguaggi simili che semplifica la scrittura e la lettura del codice che opera in contesti monadici. Permette di esprimere sequenze di azioni monadiche (che descrivono computazioni con effetti collaterali) in un formato che ricorda la programmazione imperativa, mentre la logica sottostante rimane puramente funzionale e gestita dagli operatori `return` , `>=>` , e `>>` specifici della monade. Questo aiuta a mantenere la separazione tra la descrizione della computazione (il programma puro) e la sua esecuzione (gestita dal runtime).

Domanda

Trasformazioni di monadi: motivazione, definizione, implementazione, esempi.

Risposta

Le **Trasformazioni di Monadi** (Monad Transformers) rappresentano un concetto avanzato in linguaggi funzionali come Haskell, utilizzato per affrontare la sfida di gestire **effetti collaterali** di natura diversa contemporaneamente.

Motivazione:

La motivazione principale dietro le trasformazioni di monadi nasce dal bisogno di combinare effetti collaterali multipli all'interno di un programma puro. I testi mostrano come, in un linguaggio puro come Haskell, l'introduzione anche di un singolo effetto (come la gestione degli errori con `Maybe` , il conteggio delle operazioni con `Counter` , o il tracciamento con `Output`) renda il codice che originariamente era semplice e diretto (come la funzione `eval` di un valutatore di espressioni) "stravolto" e la logica del programma si perda nella gestione di quell'effetto specifico.

Quando si tenta di gestire **effetti molteplici** con una singola monade "custom", si incontrano due problemi principali:

1. **Problema Concettuale (Ambiguità Semantica):** Non è sempre chiaro come gli effetti debbano interagire o l'ordine in cui debbano manifestarsi. Ad esempio, valutare un'espressione che può fallire (`Maybe`) e contemporaneamente contare le operazioni (`Counter`) può avere semantiche diverse a seconda che il contatore venga incrementato anche in caso di fallimento della computazione finale, oppure se il fallimento porti alla perdita anche del conteggio parziale (logica transazionale).
2. **Problema Ingegnistico (Duplicazione del Codice):** Creare una monade unica per ogni combinazione di effetti richiede di riscrivere il codice della monade da zero per ogni combinazione desiderata, portando a una notevole duplicazione.

L'idea delle trasformazioni di monadi risolve questi problemi: **invece di definire una singola monade complessa per una combinazione di effetti, si definisce una trasformazione che aggiunge un effetto a una monade esistente**. Questo permette di comporre gli effetti "impilando" le trasformazioni.

Definizione e Implementazione:

Una trasformazione di monade, concettualmente, è una funzione a livello di tipi che prende una monade e ne restituisce una nuova con capacità aggiuntive. L'implementazione concreta in linguaggi come Haskell avviene tipicamente definendo un nuovo **costruttore di tipo parametrico** `t m a`, dove `m` è la monade sottostante che viene trasformata, e `a` è il tipo del valore gestito.

Il testo fornisce l'esempio della trasformazione `MaybeT`:

- **Definizione del tipo:** `type MaybeT m a = m (Maybe a)`. Questo significa che una computazione nella monade `MaybeT m` che produce un valore di tipo `a` è rappresentata da una computazione nella monade sottostante `m` che produce un valore di tipo `Maybe a`. Il `Maybe` interno aggiunge la capacità di fallimento.
- **Implementazione dell'istanza `Monad`:** Se la monade sottostante `m` è essa stessa un'istanza della classe di tipo `Monad`, allora anche `MaybeT m` può essere resa un'istanza di `Monad`. Ciò richiede l'implementazione delle operazioni fondamentali `return` e `>>=` per `MaybeT m`.
 - `return :: a -> MaybeT m a`: La versione estesa data è `return a = return (Just a)`. Qui, il primo `return` è quello della monade `MaybeT m`, mentre il secondo `return` è quello della monade sottostante `m`. Un valore puro `a` viene prima impacchettato in `Just a` (l'effetto "fallimento non si è verificato"), e poi questo risultato viene impacchettato dalla monade sottostante `m`.
 - `(>>=) :: MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b`:
L'implementazione combina il `bind` della monade esterna (`m`) con la logica specifica del `Maybe`. Riceve una computazione `m (MaybeT m a)` e una funzione `f (a -> MaybeT m b)`. Esegue la computazione `m`. Se il suo risultato interno è

Nothing (fallimento), restituisce immediatamente Nothing impacchettato nella monade esterna (return Nothing , dove questo return è quello di MaybeT m). Se il risultato è Just a , scompatta a e applica la funzione f .

L'operazione lift :

Un aspetto cruciale delle trasformazioni di monadi è la possibilità di utilizzare le operazioni specifiche della monade sottostante all'interno del contesto della monade trasformata. Questo si ottiene tramite l'operazione lift .

- Viene definita una classe di tipo MonadT (o nomi simili) che richiede un'implementazione della funzione lift .
- lift :: Monad m => m a -> t m a : Questa funzione prende una computazione nella monade sottostante m a e la "solleva" (lifts) nel contesto della monade trasformata t m a .
- Per MaybeT , l'implementazione di lift è lift m = m >=> (return . Just) . Questo prende una computazione m dalla monade sottostante, usa il suo bind (>=>) per prendere il risultato puro a , lo impacchetta in Just a e poi usa il return della monade trasformata (MaybeT m) per posizionarlo nel contesto MaybeT m a .

Esempi:

L'esempio fornito nel testo mostra come utilizzare MaybeT per aggiungere la capacità di fallimento a una monade Counter .

- Si definisce un nuovo tipo combinato: type MaybeCounter = MaybeT Counter . Questa è una monade che rappresenta computazioni con effetti di conteggio (Counter) che possono anche fallire (MaybeT).
- Dentro un blocco di codice che opera nella monade MaybeCounter , le operazioni specifiche della monade interna Counter , come tick , non possono essere chiamate direttamente.
- Per utilizzare tick (che ha tipo Counter ()) nel contesto MaybeCounter , è necessario "sollevarla" usando lift: lift tick . Il tipo di lift tick diventa MaybeCounter () .

Questo permette di scrivere codice come:

```
evalM :: Expr -> MaybeCounter Int
evalM (Const n) = return n           -- Questo return è quello di
MaybeCounter
evalM (Div t s) = do
  m <- evalM t
  n <- evalM s
  lift tick                          -- Solleva l'azione tick da Counter a
MaybeCounter
  if n == 0 then abort               -- Questo abort è quello di MaybeT
  else return (m `div` n)           -- Questo return è quello di MaybeCounter
```

In questo modo, le operazioni di fallimento (`abort` , che è specifica di `MaybeT` come mostrato in) e le operazioni di conteggio (`tick` , usata tramite `lift`) possono essere combinate fluidamente.

Le trasformazioni di monadi possono essere impilate, creando una "pila di trasformazioni" (stack of transformers). Partendo da una monade base "banale" come `Id` (che rappresenta computazioni pure, prive di effetti), si possono aggiungere effetti applicando trasformazioni come `MaybeT` , `CounterT` , `StateT` , `ReaderT` , `ExceptT` , ecc., nell'ordine desiderato. Ad esempio, `MaybeT (CounterT Id)` rappresenta computazioni che contano e possono fallire, dove il fallimento potrebbe non annullare il conteggio parziale (a seconda della semantica della pila). `CounterT (MaybeT Id)` rappresenterebbe computazioni che possono fallire e contano, con una semantica potenzialmente diversa per la combinazione degli effetti.

Domanda

Definizione di monadi per mezzo di monad transformers. Differenze tra monadi ottenute applicando trasformazioni in ordini diversi.

Risposta

Le **Trasformazioni di Monadi (Monad Transformers)** sono uno strumento concettuale e implementativo fondamentale nei linguaggi funzionali, in particolare in Haskell, per gestire la combinazione di **effetti collaterali multipli** in modo strutturato.

Motivazione:

Nei linguaggi funzionali puri, l'introduzione di qualsiasi effetto (come la gestione di errori `Maybe` , il conteggio `Counter` , o il tracciamento `Output`) "stravolge" la struttura del codice puro originale, rendendolo più complesso e meno leggibile. Quando si cerca di combinare **effetti molteplici**, sorgono due problemi principali:

1. **Ambiguità Semantica:** Non è sempre univoco definire come gli effetti debbano interagire. Ad esempio, combinare fallimento e conteggio può avere semantiche diverse (il contatore viene incrementato anche in caso di fallimento o il conteggio si perde?). Il testo illustra questa ambiguità confrontando le semantiche `Expr -> Int -> (Maybe Int, Int)` e `Expr -> Int -> Maybe (Int, Int)` .
2. **Duplicazione del Codice (Ingegneristica):** Creare una singola monade "all-in-one" per ogni specifica combinazione di effetti richiederebbe di riscrivere il codice per `return` e `>>=` ogni volta, portando a un'enorme duplicazione.

Le trasformazioni di monadi risolvono questi problemi. Invece di definire una monade unica per una combinazione di effetti, si definisce una **trasformazione** che **aggiunge un effetto a una monade esistente**.

Definizione di Monadi tramite Trasformazioni (concetto):

L'idea non è tanto *definire* una monade base usando trasformazioni, ma piuttosto *ottenere*

monadi con effetti combinati applicando trasformazioni a una monade più semplice (spesso una monade base come `Id`, che rappresenta computazioni pure, prive di effetti).

Una trasformazione di monade `t` può essere vista come un **costruttore di tipo di ordine superiore** che prende un costruttore di tipo monadico `m` (una monade) e restituisce un nuovo costruttore di tipo `t m` che è a sua volta una monade, a patto che `m` sia una monade. Concettualmente, `t` arricchisce la monade `m` con un ulteriore effetto.

Implementazione:

L'implementazione di una trasformazione di monade in Haskell si concretizza nella definizione di:

1. Un **nuovo tipo di dato o alias** che incapsula la monade sottostante con l'effetto aggiunto. L'esempio nel testo è `type MaybeT m a = m (Maybe a)`. Questo significa che una computazione con capacità di fallimento (`MaybeT`) sopra una monade `m` è una computazione nella monade `m` che produce un risultato di tipo `Maybe a`.
2. Un'istanza della classe di tipo **`Monad`** per il costruttore di tipo trasformato `t m`. Se `m` è un'istanza di `Monad`, allora anche `t m` può esserlo, implementando le operazioni fondamentali `return` e `>>=`.
 - `return` per la monade trasformata `t m` deve utilizzare il `return` della monade sottostante `m` e impacchettare il valore puro `a` nella struttura dati specifica della trasformazione. Per `MaybeT`, `return a = return (Just a)`. Il `return` più esterno è quello di `MaybeT m`, quello interno è quello di `m`.
 - `>>=` per `t m` deve combinare il `>>=` della monade sottostante `m` con la logica che gestisce l'effetto specifico della trasformazione. L'implementazione per `MaybeT` mostra come il `bind` di `MaybeT m` chiami il `bind` di `m`, e poi esamini il risultato (`Nothing` o `Just a`) per decidere se propagare il fallimento (`Nothing`) o continuare l'esecuzione applicando la funzione `f`.
3. Un'istanza della classe di tipo **`MonadT`** (o simile), che definisce l'operazione `lift`. Questa operazione è cruciale perché permette di **sollevare (lift) una computazione dalla monade sottostante `m` a nel contesto della monade trasformata `t m`**. Senza `lift`, non si potrebbero utilizzare le operazioni specifiche della monade sottostante all'interno della monade trasformata.
 - `lift :: Monad m => m a -> t m a`. L'implementazione per `MaybeT` è `lift m = m >>= (return . Just)`. Prende una computazione `m a`, la esegue nel contesto `m`, prende il risultato `a`, lo impacchetta in `Just a`, e usa il `return` della monade trasformata (`MaybeT m`) per metterlo nel nuovo contesto.
 - L'esempio mostra `lift tick` per utilizzare l'azione `tick` (originaria della monade `Counter`) nel contesto della monade trasformata `MaybeCounter` (che è `MaybeT Counter`).

Differenze tra Monadi ottenute applicando Trasformazioni in Ordini Diversi:

Le trasformazioni di monadi possono essere **impilate (stacked)**. Partendo da una monade base (come `Id` per computazioni pure), è possibile applicare una sequenza di

trasformazioni, creando una "pila" di effetti. Ad esempio, si potrebbe avere una monade che gestisce sia il conteggio (`Counter`) che il fallimento (`Maybe`). Le trasformazioni pertinenti sarebbero `CounterT` e `MaybeT` . È possibile applicarle in ordini diversi:

1. **MaybeT Counter** : Questa monade rappresenta computazioni nella monade `Counter` che possono anche fallire. La struttura interna sarebbe simile a `Counter (Maybe a)` . La semantica degli effetti rifletterà questo ordine: una computazione che fallisce produrrà `Nothing` nel contesto `Maybe` . Tuttavia, dato che il `Maybe` è *interno* alla monade `Counter` , è possibile che le operazioni di `Counter` (come l'incremento del contatore) si verifichino *prima* che il fallimento venga gestito dall'esterno. Potrebbe corrispondere alla semantica `eval :: Expr -> Int -> (Maybe Int, Int)` dove il conteggio viene mantenuto anche in caso di fallimento della computazione finale.
2. **CounterT Maybe** : Questa monade rappresenta computazioni nella monade `Maybe` che tengono anche un contatore. La struttura interna sarebbe simile a `Maybe (Int, a)` . La semantica qui è diversa: l'effetto di fallimento (`Maybe`) è *esterno* alla monade `CounterT` . Se una computazione nella monade `Maybe` fallisce producendo `Nothing` , l'effetto `CounterT` che la incapsula potrebbe non avere accesso al conteggio parziale accumulato *all'interno* della computazione fallita nella monade `Maybe` . Questo potrebbe corrispondere alla semantica `eval :: Expr -> Int -> Maybe (Int, Int)` dove il fallimento porta alla perdita del conteggio parziale, implementando una logica "transazionale".

In sintesi, l'ordine in cui si applicano le trasformazioni definisce la monade risultante e, crucialmente, la **semantica di combinazione degli effetti**. Una trasformazione applicata "sopra" un'altra monade gestirà il proprio effetto in relazione agli effetti gestiti dalla monade sottostante in un modo specifico determinato dalla sua implementazione di `return` e `>>=` , e dall'uso di `lift` per interagire con la monade inferiore.

L'impilamento permette di costruire monadi complesse in modo modulare e flessibile, superando i limiti della creazione di monadi custom "piatte" per ogni combinazione di effetti desiderata.

Domanda

Lifting di operazioni.

Risposta

Le trasformazioni di monadi (ad esempio, `MaybeT` o `CounterT`) permettono di aggiungere effetti a una monade esistente, costruendo una "pila" di effetti. Quando si applica una trasformazione `t` a una monade `m` per ottenere la monade trasformata `t m` , sorge la necessità di poter utilizzare le operazioni specifiche che appartengono alla monade sottostante `m` anche all'interno del contesto della monade trasformata `t m` .

I sorgenti indicano chiaramente che **non possono essere mescolate direttamente azioni di monadi diverse**. Ad esempio, in un contesto `MaybeT Counter` (una monade `MaybeT`

applicata alla monade `Counter`), l'operazione `tick`, che è specifica della monade `Counter`, non può essere chiamata direttamente.

La soluzione a questo problema è il **Lifting**.

- **Definizione e Scopo:** Il lifting è una **trasformazione di computazioni**. Permette di **sollevare (lift) una computazione dalla monade sottostante m a nel contesto della monade trasformata $t\ m\ a$** . L'obiettivo è rendere utilizzabili le operazioni della monade interna (m) nel contesto della monade esterna trasformata ($t\ m$). Quando una computazione $m\ a$ viene sollevata in $t\ m\ a$ utilizzando `lift`, idealmente non dovrebbe aggiungere l'effetto della trasformazione t .
- **Implementazione:** La funzionalità di lifting è definita tramite una classe di tipo, tipicamente chiamata `MonadT` (o simile). Questa classe richiede che ogni trasformatore di monade t che ne è istanza definisca una funzione `lift`.
 - La funzione `lift` ha la seguente firma di tipo: `lift :: Monad m => m a -> t m a`. Questo indica che prende una computazione nella monade m (che produce un valore di tipo a) e restituisce una computazione nella monade trasformata $t\ m$ (che produce un valore di tipo a), a condizione che m sia una monade.
 - Ogni trasformazione di monade t deve fornire la propria implementazione specifica dell'istanza `MonadT t` e quindi della funzione `lift`.
- **Esempio con MaybeT:** Il sorgente fornisce l'implementazione di `lift` per la trasformazione `MaybeT`:

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

Questa implementazione per `MaybeT` prende una computazione m (di tipo $m\ a$), la esegue utilizzando il `>>=` della monade m . Il risultato di questa computazione interna a viene poi impacchettato in `Just a`, e infine `return` (il `return` della monade *trasformata* `MaybeT m`, che è `Just`) viene usato per posizionare questo valore nel contesto `MaybeT m`. Questo significa che una computazione sollevata dalla monade sottostante m usando `lift` non fallirà nel contesto `MaybeT`, a meno che non fallisca già nella monade m (cosa che `MaybeT` gestirebbe implicitamente tramite il `>>=`).

- **Utilizzo pratico:** L'esempio nel testo mostra come l'operazione `tick` (originaria della monade `Counter`) venga resa utilizzabile all'interno della monade `MaybeCounter` (che è `MaybeT Counter`) tramite il lifting:

```
evalM :: Expr -> MaybeCounter Int
evalM (Const n) = return n
evalM (Div t s) =
    evalM t >>= \m ->
    evalM s >>= \n ->
```

```
lift tick >=> \() ->
if n == 0 then abort else return (m 'div' n)
```

Senza `lift tick`, il compilatore segnalerebbe un errore perché `tick` appartiene alla monade `Counter`, non a `MaybeCounter`. `lift tick` trasforma l'azione `tick` (che è di tipo `Counter ()`) in un'azione di tipo `MaybeT Counter ()`, che può essere correttamente utilizzata nel contesto `MaybeCounter`.

In conclusione, il lifting è il meccanismo essenziale che permette di interagire con le funzionalità delle monadi sottostanti quando si utilizzano stack di monad transformers, garantendo la composizione modulare degli effetti.

Rust

Domanda

Caratteristiche e motivazioni per Rust

Risposta

Rust è un linguaggio di sistema, paragonabile a C o C++. La sua notorietà deriva in gran parte dalla sua **gestione della memoria**.

La motivazione principale dietro la progettazione di Rust sembra essere quella di offrire le capacità e le prestazioni di un linguaggio di sistema, con un **supporto a runtime minimo** (ad esempio, non include un garbage collector), pur prevenendo molti degli errori comuni legati alla gestione esplicita della memoria e alla concorrenza con memoria condivisa, tipici di linguaggi come C e C++.

Rust raggiunge questo obiettivo principalmente attraverso due meccanismi ortogonali che interagiscono tra loro:

1. **Un complesso sistema di ownership con borrowing in lettura e/o scrittura.** Questo meccanismo si basa sul sistema di tipi di Rust e **non ha costi a runtime**. Ogni cella di memoria sull'heap ha un *owner* (proprietario) responsabile della sua deallocazione. Quando un puntatore a una cella sull'heap viene assegnato a una variabile sullo stack, quest'ultima ne diventa l'owner. Il sistema di tipi, verificato a compile-time, garantisce che a runtime ci sia sempre al più una referenza mutabile *oppure* un numero arbitrario di referenze in sola lettura su un dato, prevenendo così le *data race*.
2. **Smart pointers**, utilizzati quando il meccanismo di ownership e borrowing diventa troppo complesso da gestire manualmente. Gli smart pointer sono strutture dati che si comportano come puntatori, ma implementano determinati *traits* (caratteristiche) per forzare specifiche politiche di gestione della memoria, come l'allocazione nell'heap (`Box<T>`), il reference counting (`Rc<T>`, `Arc<T>`), o la gestione di accessi mutabili controllati a runtime (`RefCell<T>`). Permettono di allocare dati sull'heap anche quando le loro dimensioni non sono uniformi o per definire tipi di dato algebrici ricorsivi. Smart

pointer specifici come `Weak<T>` sono usati per risolvere problemi come i riferimenti ciclici che impedirebbero la deallocazione con il solo reference counting.

Altre caratteristiche chiave di Rust includono:

- **Assenza di valori NULL**; vengono usati *option types* per gestire l'assenza di un valore, prevenendo un'altra fonte comune di errori.
- **Algebraic Data Types e Pattern Matching**, strumenti potenti per definire e manipolare strutture dati complesse in modo sicuro ed espressivo.
- **Polimorfismo parametrico *bounded*** (con vincoli) che utilizza un sistema di *templates/generics* con i *traits* come vincoli. Questo permette di scrivere codice generico che opera su diversi tipi, purché tali tipi implementino i tratti richiesti.
- **Traits**, concetti simili a interfacce in altri linguaggi, che definiscono un insieme di metodi che un tipo deve implementare. I trait possono avere implementazioni di default che possono essere sovrascritte. Vengono utilizzati anche come *Trait Objects* per il dynamic dispatch (risoluzione dinamica delle chiamate a metodi a runtime).
- **Moduli annidabili** per organizzare il codice.
- **Chiusure (Closures)**, che possono catturare variabili libere per trasferimento di ownership (`move`) o per borrowing (mutabile o immutabile).
- **Slices** (`&str` , `&[T]`), che sono smart pointer utilizzati per fare borrowing (mutabile o immutabile) di una parte di una struttura dati, molto utili per parallelizzare le computazioni.

L'analisi statica di Rust, basata sul suo rigoroso sistema di tipi, aiuta a **prevenire moltissimi errori comuni** nella gestione esplicita della memoria e nella concorrenza con memoria condivisa. Tuttavia, i messaggi di errore possono essere complessi da interpretare, e può essere necessario riformulare il codice per conformarsi alle regole di Rust.

L'implementazione del polimorfismo (*generics* e *traits*) in Rust avviene tramite **monomorfizzazione a tempo di compilazione**, il che significa che il compilatore crea una versione specializzata del codice generico per ogni combinazione di tipi utilizzata, evitando overhead a runtime a scapito di tempi di compilazione potenzialmente maggiori e una dimensione dell'eseguibile aumentata.

Domanda

Ownership e suo trasferimento

Risposta

Rust è un linguaggio di sistema, noto principalmente per il suo approccio alla **gestione della memoria**. L'obiettivo è fornire le prestazioni di un linguaggio di basso livello senza un garbage collector esplicito a runtime, prevenendo al contempo molti errori comuni.

Rust raggiunge questo obiettivo attraverso **due meccanismi ortogonali che interagiscono**:

1. **Un complesso sistema di ownership con borrowing in lettura e/o scrittura.**
2. **Smart pointers**, utilizzati quando il primo meccanismo diventa troppo complesso da gestire.

Concentrandoci sul primo meccanismo, l'**Ownership**, ecco le sue caratteristiche e come avviene il suo trasferimento:

- **Principio fondamentale:** Il meccanismo di ownership è basato sul **sistema di tipi di Rust** e viene **interamente imposto a compile-time**, senza avere costi a runtime.
- **Owner (Proprietario):** Ogni cella di memoria allocata sullo **heap** ha un **owner**, che è l'entità **responsabile della sua deallocazione**.
- **Assegnazione dell'Owner:**
 - Quando una cella sullo heap viene creata e un puntatore a essa viene assegnato a una variabile sullo **stack**, quest'ultima ne diventa l'owner.
 - Quando il puntatore viene assegnato a un'altra cella sullo **heap**, questa nuova cella ne diventa l'owner.
- **Regola chiave:** Una cella sullo heap ha sempre **uno e un solo owner**.
- **Deallocazione:** Quando un owner viene deallocato (ad esempio, una variabile sullo stack esce dallo scope e il suo blocco termina), le celle sullo heap che erano **ricorsivamente possedute** da quell'owner **vengono rilasciate**. Il codice necessario per questa deallocazione viene **inserito dal compilatore** alla fine del blocco corrispondente.

Trasferimento dell'Ownership (Move):

- Il **trasferimento (move)** dell'ownership avviene tramite le **assegnazioni** e il **passaggio come parametri** della variabile o cella che detiene l'ownership.
- Quando una variabile perde l'ownership attraverso un trasferimento, **non può più essere utilizzata!**. Questo comportamento è verificato a compile-time.
- Esempi pratici:
 - In un'assegnazione come `let t = s;`, dove `s` è l'owner di un dato sullo heap (es. una `String`), l'ownership del dato viene trasferita da `s` a `t`. Dopo questa operazione, la variabile `s` non è più valida e non può essere usata.
 - Quando una variabile che è owner di un dato viene passata come argomento a una funzione, l'ownership viene **trasferita alla funzione**. Similmente, se una funzione restituisce un dato di cui è owner, l'ownership viene **trasferita al chiamante**.

Borrowing (Prestito):

Il sistema di ownership è rigido (una volta trasferita, la variabile originale non è più usabile). Per consentire l'accesso ai dati senza trasferire l'ownership, Rust introduce il concetto di **References**, che sono una sorta di **sistema di prestiti** della ownership.

- Esistono due tipi principali di reference (prestiti):
 - `&T` : una reference **immutabile** (in sola lettura).
 - `&mut T` : una reference **mutabile** (che permette di modificare il dato).

- **Regole del Borrowing:** Il compilatore di Rust impone regole rigorose sui prestiti per **prevenire le data race**. Queste regole sono controllate a compile-time e garantiscono che, in un dato momento, per una specifica porzione di dato, tu possa avere **o una reference mutabile O un numero qualsiasi di reference immutabili, ma mai entrambe contemporaneamente**.
- **Implicazioni delle regole:** Se una variabile viene "borrowed" (data in prestito) mutably, nessun altro prestito è possibile, e l'owner originale è "frozen" (non può accedere alla variabile) finché il prestito non termina. Similmente, se l'owner ha ownership mutabile ma presta la variabile, l'owner è frozen.

Concetti Correlati:

- **Lifetimes:** Il sistema di lifetime è un concetto correlato che garantisce che le references non sopravvivano al dato a cui puntano, prevenendo i **dangling pointers**. I lifetimes vengono verificati a compile-time.
- **Slices:** Strutture dati come le slices (`&str` , `&[T]`) sono smart pointer che facilitano il borrowing (mutabile o immutabile) di una parte di una struttura dati, utili ad esempio per la parallelizzazione.
- **Smart Pointers:** Come accennato, gli smart pointer (`Box` , `Rc` , `Arc` , `RefCell` , `Weak` , `Mutex` , etc.) sono un altro meccanismo che permette di gestire la memoria e l'ownership in scenari più complessi rispetto alla semplice ownership singola. `Box<T>` ad esempio gestisce l'allocazione sullo heap con ownership singola, `Rc<T>` e `Arc<T>` permettono l'ownership condivisa con reference counting, `Weak<T>` è usato per risolvere i cicli di riferimento nel reference counting, e `RefCell<T>` o `Mutex<T>` gestiscono accessi mutabili controllati a runtime. Sebbene usino e interagiscano con il concetto di ownership, offrono modelli di gestione diversi (es. ownership condivisa invece che esclusiva).
- **Monomorfizzazione:** Il polimorfismo parametrico in Rust (usando generics con constraints via traits) viene implementato tramite monomorfizzazione a compile-time, creando versioni specializzate del codice per ogni tipo utilizzato. Questo si contrappone ad approcci come la rappresentazione uniforme dei dati usata da linguaggi come Erlang o OCaml.

In sintesi, il **trasferimento dell'ownership** in Rust è un'operazione di **move** implicita nelle assegnazioni e nel passaggio di parametri che rende la sorgente non più utilizzabile. Questo, combinato con le rigide regole del **borrowing** verificate a compile-time, è il fondamento su cui Rust costruisce la sua garanzia di **memory safety** e l'assenza di **data race**, senza ricorrere a un garbage collector. Sebbene il sistema sia potente, i messaggi di errore del compilatore possono essere complessi, richiedendo talvolta una riformulazione del codice per rispettare le regole.

Domanda

Borrowing mutabile e immutabile; references

Risposta

Rust è un linguaggio di sistema, paragonabile a C o C++, ma si distingue per la sua **gestione della memoria**, che è diventata la sua caratteristica più famosa. A differenza di molti altri linguaggi, Rust non utilizza un garbage collector a runtime, mantenendo un supporto a runtime minimo. L'obiettivo è garantire la sicurezza della memoria (memory safety) e prevenire le *data race*, tipiche della concorrenza con memoria condivisa, attraverso controlli statici eseguiti a compile-time.

Rust raggiunge questo obiettivo attraverso **due meccanismi ortogonali** che interagiscono tra loro:

1. Un complesso sistema di **ownership** con **borrowing** in lettura e/o scrittura.
2. **Smart pointers**, utilizzati in scenari più complessi.

Concentriamoci sul primo meccanismo, che include References e Borrowing.

Ownership (Proprietà)

Il sistema di ownership è il fondamento della gestione della memoria in Rust e si basa sul sistema di tipi del linguaggio. Le sue caratteristiche principali sono:

- **Unico Proprietario:** Ogni cella di memoria allocata sullo **heap** ha un **owner** (**proprietario**), che è l'unico responsabile della sua deallocazione.
- **Assegnazione:** Quando un dato sullo heap viene creato e il suo puntatore è assegnato a una variabile sullo **stack**, quest'ultima diventa l'owner. Se il puntatore viene assegnato a un'altra cella sullo heap, questa nuova cella ne diventa l'owner.
- **Deallocazione Automatica:** Quando l'owner esce dallo scope (ad esempio, una variabile sullo stack viene deallocata alla fine del suo blocco), le celle sullo heap che erano ricorsivamente possedute da quell'owner **vengono automaticamente rilasciate**. Il codice per questa deallocazione è **inserito dal compilatore**.

Trasferimento dell'Ownership (Move)

Quando l'ownership di un dato viene trasferita da una variabile a un'altra (tramite assegnamento o passaggio come parametro), la variabile originale **non è più valida** e non può essere utilizzata. Questo comportamento è verificato a compile-time. Questo meccanismo di "move" è rigido ma garantisce che la deallocazione avvenga una sola volta.

References e Borrowing (Prestito)

Per permettere l'accesso ai dati senza trasferire l'ownership (che renderebbe la variabile originale inutilizzabile), Rust introduce il concetto di **References**. Le references sono una sorta di **sistema di "prestiti" della ownership**.

Esistono due tipi principali di references:

- **&x** : una reference **immutabile** a **x**. Permette di leggere il dato. Se **x** ha tipo **T**, **&x** ha tipo **&T**.

- `&mut x` : una reference **mutabile** a `x` . Permette di modificare il dato. Se `x` ha tipo `T` , `&mut x` ha tipo `&mut T` .

Quando si crea una reference a una variabile, si sta "facendo borrowing" (prendendo in prestito) quella variabile. Il compilatore di Rust impone **regole rigorose sul borrowing a compile-time** per prevenire le *data race*, anche in scenari concorrenti con memoria condivisa. Queste regole sono:

1. Puoi avere un numero qualsiasi di **references immutabili** (`&T`) a un dato.
2. Puoi avere **al più una reference mutabile** (`&mut T`) a un dato.
3. **Non puoi avere references mutabili e immutabili contemporaneamente** allo stesso dato.

Quando una variabile viene data in prestito (borrowed), l'owner può essere temporaneamente "congelato" (frozen):

- Se una variabile è data in prestito in modo mutabile (`&mut`), nessun altro prestito è possibile (né mutabile né immutabile), e l'owner **non può accedere o modificare la variabile** finché il prestito non termina.
- Anche se l'owner ha ownership mutabile (`let mut x`), se la variabile viene data in prestito (mutabilmente o immutabilmente), l'owner **non può modificare la variabile** finché il prestito non termina.

Il prestito termina quando la reference (la variabile sullo stack che la detiene) esce dallo scope. Il compilatore verifica che queste regole siano rispettate analizzando il codice. Gli esempi nei sorgenti illustrano come il compilatore genera errori se si cerca di violare queste regole, ad esempio:

- Assegnare un nuovo valore a `x` mentre esiste un prestito immutabile di `x` causa un errore.
- Tentare di creare un prestito mutabile (`&mut x`) di una variabile `x` che è stata dichiarata come immutabile (`let x = 4`).
- Tentare di creare un prestito mutabile (`&mut x`) mentre esiste già un prestito immutabile (`&x`) sulla stessa variabile `x` .
- Tentare di creare più di un prestito mutabile (`&mut x`) contemporaneamente sulla stessa variabile `x` .

Lifetimes

Strettamente legato al concetto di references e borrowing è quello di **Lifetimes**. Ogni dato e ogni reference in Rust ha un *lifetime* che indica il periodo durante il quale è valido. Il sistema di lifetimes, anch'esso verificato a **compile-time**, garantisce che una reference **non possa mai sopravvivere al dato a cui punta**. Questo previene i **dangling pointers**, un problema comune in linguaggi con gestione manuale della memoria. I lifetimes vengono spesso inferiti dal compilatore, ma in alcuni casi (come nelle firme delle funzioni o nelle definizioni di struct

con references) possono dover essere annotati esplicitamente usando sintassi come `'a` o `'static`.

Interazioni e Concetti Correlati

- **Slices:** Le slices (`&str` , `&[T]`) sono viste come **smart pointer** che permettono di fare **borrowing** (mutabile o immutabile) di una porzione contigua di una struttura dati, utili ad esempio per parallelizzare le computazioni su array o vettori.
- **Chiusure (Closures):** Le chiusure in Rust possono **catturare** le variabili dal loro ambiente circostante in diversi modi: per trasferimento di ownership (`move`) o per **borrowing** (mutabile o immutabile). La necessità di gestire questi diversi modi di cattura ha portato a diversi tipi di chiusure e trait associati.
- **Smart Pointers e Borrowing Runtime:** Mentre il core borrowing con `&` e `&mut` è gestito a compile-time, alcuni smart pointer come `RefCell<T>` , `Mutex<T>` e `Arc<T>` permettono di implementare scenari di condivisione e mutabilità più complessi. In particolare, `RefCell<T>` permette di avere **più references immutabili o al più una reference mutabile** allo stesso dato **a runtime**, sollevando un `panic!` se le regole vengono violate. Questo è utile in situazioni monothread in cui il controllo a compile-time è troppo restrittivo. `Mutex<T>` e `Arc<T>` forniscono meccanismi simili per scenari **concurrenti** con memoria condivisa. L'uso di questi smart pointer comporta costi a runtime (come la gestione dei lock) e va valutato attentamente.

In sintesi, il **borrowing** in Rust, attraverso le **references** immutabili (`&T`) e mutabili (`&mut T`), è un meccanismo basato sul tipo e verificato a compile-time che permette di accedere ai dati senza trasferirne la proprietà. Le rigide regole imposte dal compilatore (una mutabile O molte immutabili, mai entrambe) e il sistema di **lifetimes** garantiscono che il codice sia sicuro, prevenendo dangling pointers e data race, pur mantenendo le prestazioni di un linguaggio di basso livello.

Domanda

Slices (mutabili e immutabili)

Risposta

Le Slices in Rust sono definite come degli **smart pointer**. Il loro scopo principale è quello di permettere il **borrowing** (prestito) di una **parte di una struttura dati**. Questo borrowing può essere **mutabile o meno**. Le Slices sono particolarmente utili per **parallelizzare le computazioni** su una struttura dati, delegando una slice per ogni thread.

I tipi più comuni di slices menzionati nelle fonti sono le `str` (string slices) e i `&[T]` (vector slices).

Concettualmente, una slice equivale a un **puntatore + numero di byte + capacità residua**. Come smart pointer, le slices **hanno dei lifetimes**. I lifetimes sono strettamente legati al

concetto di references e borrowing in Rust, e servono a garantire che una reference non sopravviva al dato a cui punta.

Le fonti includono un esempio di come una funzione possa prendere in prestito una slice immutabile:

```
// Questa funzione prende in prestito una slice
fn analyze_slice(slice: &[i32]) {
    println!("first element of the slice: {}", slice);
    println!("the slice has {} elements", slice.len());
}

fn main() {
    // Array di dimensione fissa
    let xs: [i32; 5] =;

    // Slice contenente gli ultimi 2 elementi
    analyze_slice(&xs[2 .. 4])
    // &xs[2 .. 4] permette di fare borrowing di una slice dell'array xs.
}
```

Questo esempio mostra l'utilizzo di una slice immutabile (`&[i32]`). L'espressione `&xs[2 .. 4]` crea un prestito (borrow) di una porzione dell'array `xs` come una slice.

Borrowing Mutabile e Immutabile con le Slices

Come accennato, le slices consentono sia il borrowing mutabile che quello immutabile.

- Una slice **immutabile** è rappresentata dal tipo `&[T]` (o `&str` per le stringhe). Permette di leggere gli elementi della porzione di dati a cui punta, ma non di modificarli.
- Una slice **mutabile** è rappresentata dal tipo `&mut [T]` (o `&mut str`). Permette di leggere e modificare gli elementi della porzione di dati a cui punta.

L'applicazione del borrowing mutabile e immutabile alle slices segue le **regole generali del borrowing in Rust**:

1. Puoi avere un numero qualsiasi di **references immutabili** (`&T`) a un dato.
2. Puoi avere **al più una reference mutabile** (`&mut T`) a un dato.
3. **Non puoi avere references mutabili e immutabili contemporaneamente** allo stesso dato.

Queste regole si applicano al dato *sottostante* su cui la slice è basata. Ad esempio, se hai un `Vec<i32>`, puoi creare molte slices immutabili `&[i32]` che puntano a diverse (o anche sovrapposte) parti del vettore. Tuttavia, puoi creare al massimo una slice mutabile `&mut [i32]` alla volta per l'intero vettore o per una sua porzione, e mentre quella slice mutabile esiste, non puoi creare altre slices (mutabili o immutabili) che puntano a quella stessa

porzione del vettore, né puoi accedere direttamente al vettore originale per leggerlo o modificarlo.

Domanda

Lifetimes, loro dichiarazione e loro uso come bound per il polimorfismo parametrico

Risposta

Nel contesto della gestione della memoria di Rust, basata sul sistema di **Ownership** e **Borrowing**, i **Lifetimes** giocano un ruolo cruciale. Ogni cella di memoria ha un *owner* che è responsabile della sua deallocazione. Ogni dato e ogni *reference* in Rust ha un **Lifetime**, che indica il periodo durante il quale è valido.

La funzione principale dei Lifetimes è quella di prevenire i **dangling pointers**, che sono reference che puntano a memoria che è già stata deallocata. Rust verifica a compile-time che una reference **non possa mai sopravvivere al dato a cui punta**.

È importante distinguere il Lifetime dallo *scope*.

Lo scope si riferisce al blocco di codice all'interno del quale una variabile è accessibile.

Il Lifetime di un dato è determinato da quando il suo *owner* esce dallo scope e il dato viene deallocato. Nel caso del trasferimento di ownership (*move*), il Lifetime di un dato non corrisponde semplicemente allo scope della variabile originale che lo deteneva.

Ogni reference (`&` o `&mut`) ha di fatto due Lifetimes associati:

1. Il Lifetime della reference stessa (ovvero, la durata della variabile che detiene la reference).
2. Il Lifetime del dato a cui la reference punta. Rust garantisce che il primo Lifetime sia sempre inferiore o uguale al secondo.

Dichiarazione dei Lifetimes

I Lifetimes sono spesso inferiti dal compilatore, un processo chiamato *elisione*.

Tuttavia, in alcuni casi, devono essere **esplicitati** (annotati) nella firma del codice. Le variabili di Lifetime vengono indicate con nomi che iniziano con un apice singolo, come `'a` , `'b` , `'c` , ecc.. Esiste anche un Lifetime costante chiamato `'static` , che indica che un dato vive per l'intera durata del programma (ad esempio, nel caso di variabili globali).

Le annotazioni esplicite dei Lifetimes sono necessarie principalmente in due contesti:

1. Nella firma di funzioni che restituiscono una reference, per specificare la relazione tra i Lifetimes degli input e quello dell'output. Ad esempio, una funzione che prende in input reference e ne restituisce una che "prende in prestito" dai suoi input deve indicare che il Lifetime della reference restituita non supererà quello degli input. L'esempio della funzione `dangle()` in mostra un errore tipico che i Lifetimes prevengono: la funzione tenta di restituire una reference (`&s`) a una variabile locale (`s`) il cui Lifetime termina

alla fine della funzione, mentre la firma indicava un Lifetime arbitrario ('a) per la reference di ritorno.

2. Nelle definizioni di `struct` o `enum` che contengono reference. Questo è necessario per garantire che la struttura dati non sopravviva ai dati a cui i suoi membri puntano.

L'esempio di `ImportantExcerpt` in mostra una `struct` che contiene una slice immutabile (`&'a str`), dove 'a indica che il Lifetime della slice (`part`) sarà lo stesso della stringa completa (`novel`) da cui deriva. In generale, un elemento che appartiene a una struttura dati deve avere lo stesso Lifetime (o uno più breve) della struttura stessa.

Nella dichiarazione dei Lifetimes, è possibile specificare dei **bounds** (vincoli) usando la sintassi `':'`. Ad esempio, `'a : 'b` significa che il Lifetime 'a deve terminare *dopo* il Lifetime 'b.

Uso dei Lifetimes come Bound per il Polimorfismo Parametrico

Rust supporta il **polimorfismo parametrico** tramite l'uso di *generics* (simili ai template in C++), i quali possono avere vincoli (`bounds`) definiti tramite **Traits**. I Lifetimes aggiungono un ulteriore tipo di vincolo per il polimorfismo parametrico, specificamente quando si lavora con le reference.

In questo contesto, i Lifetimes possono essere utilizzati come *bounds* in due modi principali:

1. **Bounds sui parametri di Lifetime:** Nella firma di funzioni generiche sui Lifetimes, si possono imporre relazioni tra i Lifetimes dei diversi argomenti o tra gli argomenti e il valore di ritorno. L'esempio `fn max<'a, 'b : 'a, 'c : 'a>(x: &'b i32, y: &'c i32) -> &'a i32` mostra questo uso. Qui:
 - La funzione `max` è generica sui Lifetimes 'a , 'b e 'c .
 - `x` è una reference con Lifetime 'b , `y` è una reference con Lifetime 'c .
 - La reference restituita ha Lifetime 'a .
 - I bounds `'b : 'a` e `'c : 'a` impongono che i Lifetimes di `x` ('b) e `y` ('c) debbano durare **almeno quanto** il Lifetime della reference restituita ('a). Questo è essenziale per garantire che la reference di ritorno non "pendoli" dopo che i dati a cui punta (provenienti da `x` o `y`) sono stati deallocati. L'esempio in `main` dimostra che la variabile `z` , che riceve la reference restituita da `max` , deve essere dichiarata in uno scope tale che il suo Lifetime termini *prima* che i Lifetimes di `x` e `y` terminino.
2. **Bounds sui parametri di Tipo (in relazione a un Lifetime):** Quando una funzione o una struttura è generica su un tipo `T` e prende in input una reference con un certo Lifetime 'a , può essere necessario imporre un vincolo sul tipo `T` che coinvolga questo Lifetime. La sintassi `T: 'a` (vista nell'esempio `fn print_ref<'a, T>(t: &'a T) where T: Debug + 'a`) significa che il tipo `T` (o, più precisamente, *tutte le reference contenute all'interno del tipo T*) deve vivere **almeno quanto** il Lifetime 'a associato alla reference `t` che punta a `T`. Questo garantisce che, se la funzione accede a dati interni a `T` tramite reference, quelle reference siano valide per tutta la durata del Lifetime 'a .

In sintesi, i **Lifetimes** sono un meccanismo di **controllo statico** (a compile-time) di Rust per garantire la sicurezza della memoria e prevenire i dangling pointers. Vengono esplicitati solo quando necessario, utilizzando la sintassi `'a`.

Nel contesto del **polimorfismo parametrico**, i Lifetimes possono essere usati come **bounds** (`:`) sia su altri parametri di Lifetime (`'b : 'a`) che su parametri di Tipo (`T: 'a`), per specificare le relazioni di durata necessarie a mantenere la sicurezza del codice che utilizza reference. Questo approccio, verificato rigorosamente dal compilatore, è uno dei pilastri che permette a Rust di offrire garanzie di sicurezza della memoria paragonabili a quelle di linguaggi con garbage collector, ma senza i relativi costi a runtime.

Domanda

Smart pointers: Box, Rc, RefCell, Arc, Mutex, ...

Risposta

Gli Smart Pointers rappresentano il **secondo meccanismo ortogonale** per la gestione della memoria in Rust, complementare al sistema di ownership e borrowing. Sono delle **strutture dati** (definite dall'utente o di sistema) che si comportano come puntatori (possono essere dereferenziati per accedere al valore puntato) ma implementano uno o più **traits** che aggiungono funzionalità e forzano specifiche politiche di gestione della memoria. Sono particolarmente utili quando il meccanismo di ownership e borrowing diventa troppo complesso da gestire. Rust ha una libreria standard che fornisce diversi tipi di smart pointers. Sono fondamentali per implementare il sistema di **fearless concurrency**.

Ecco i tipi di smart pointers specifici menzionati nelle fonti:

1. Box<T>

- **Scopo:** Permette di allocare dati sullo **heap**. Il dato viene allocato nello heap, mentre la variabile di tipo `Box<T>` (che è lo smart pointer) viene allocata sullo stack e punta al dato.
- **Costo:** È l'unico smart pointer che non ha un vero costo computazionale significativo (è una **zero-cost abstraction**).
- **Ownership:** Viene utilizzato quando il dato nello heap ha un **ed un solo puntatore entrante** (singolo owner).
- **Utilizzo:** È necessario per allocare dati di **grandi dimensioni** sullo heap, accedendovi tramite uno smart pointer di dimensione fissa e piccola. È **fondamentale per implementare tipi di dati algebrici (ADT) ricorsivi**, come le liste linkate, in cui i nodi potrebbero altrimenti avere dimensione arbitraria. Quando la variabile `Box<T>` esce dallo scope, il dato puntato sullo heap viene automaticamente deallocato.
- Esempi di dichiarazione ed uso sono mostrati nelle fonti, inclusa l'implementazione corretta di una lista linkata ricorsiva

```
enum List { Cons(i32, Box<List>), Nil, }
```

.

2. Rc<T> (Reference Counting)

- **Scopo:** Permette di avere **multipli owner** per gli stessi dati. Viene utilizzato quando i dati devono essere condivisi e avere più di un riferimento contemporaneamente.
- **Meccanismo:** Implementa un **contatore di riferimenti** (Reference Counter - RC) associato al dato sullo heap. `Rc::new` alloca il dato e il contatore sullo heap. `Rc::clone` incrementa il contatore. Il dato viene deallocato dallo heap solo quando il contatore raggiunge zero, cioè quando tutti i riferimenti `Rc` ad esso escono dallo scope.
- **Limitazione:** `Rc<T>` permette solamente **reference di sola lettura** al contenuto. È utilizzabile solo in scenari a **singolo thread**.
- Esempi di dichiarazione ed uso per condividere sotto-liste sono mostrati nelle fonti.

3. `RefCell<T>`

- **Scopo:** Permette la **mutabilità interna** (`interior mutability`) per dati che altrimenti sarebbero immutabili, in particolare quando usati in combinazione con `Rc<T>` . Risolve il problema di reference di sola lettura di `Rc<T>` .
- **Meccanismo:** **Verifica a runtime** le regole di borrowing (al più una reference mutabile o un numero arbitrario di immutabili) tramite un sistema di lock. In caso di violazione, viene sollevato un `panic` .
- **Utilizzo:** Permette di avere strutture dati con parti condivise immutabilmente (tramite `Rc`) ma con elementi mutabili (tramite `RefCell`) all'interno di un **singolo thread**.
- **Cautela:** Le fonti suggeriscono di usarlo con parsimonia, preferendo il controllo a compile-time di Rust quando possibile.
- Metodi: `borrow()` e `borrow_mut()` per ottenere riferimenti immutabili o mutabili al contenuto, con verifica a runtime.
- Un esempio di lista linkata con teste mutabili e code condivisibili è mostrato per illustrarne l'uso.

4. `Weak<T>`

- **Scopo:** Viene utilizzato in combinazione con `Rc<T>` per **rompere i cicli di riferimento** e prevenire memory leaks in strutture dati che altrimenti li causerebbero (es. grafi, liste doppiamente linkate con puntatori al predecessore).
- **Meccanismo:** Un `Weak<T>` è un puntatore che **non incrementa il contatore di riferimenti** dell' `Rc<T>` a cui punta. Il dato viene deallocato quando non ci sono più `strong pointers` (`Rc<T>`) che puntano ad esso, indipendentemente dalla presenza di `weak pointers` .
- **Utilizzo:** Tipicamente usato per riferimenti opzionali non-owning, come il puntatore al padre in una struttura ad albero o i puntatori dalle chiavi di una hash table ai dati per la memoization.
- Metodi: `upgrade()` (cerca di ottenere un `Rc<T>` dal `Weak<T>` , restituendo `None` se il dato è già stato deallocato) e `downgrade()` (crea un `Weak<T>` da un `Rc<T>`).

5. `Mutex<T>`

- **Scopo:** Permette **accesso sicuro a dati condivisi** in scenari **concorrenti** (multi-thread). Garantisce che solo un thread alla volta possa accedere ai dati protetti dal

mutex.

- **Meccanismo:** Utilizza un metodo **bloccante** `lock()`. Un thread che chiama `lock()` attende che nessun altro thread stia accedendo ai dati prima di ottenere una reference (mutabile) al contenuto.
- **Utilizzo:** Proteggere dati che vengono letti e/o scritti da più thread contemporaneamente.

6. `Arc<T>` (Atomic Reference Counting)

- **Scopo:** È la versione **thread-safe** di `Rc<T>`. Permette di avere multiple owner per gli stessi dati **attraverso i thread**.
- **Meccanismo:** Implementa un **contatore di riferimenti atomico**. Questo significa che le operazioni di incremento/decremento del contatore sono sicure da usare in un contesto multi-thread.
- **Utilizzo:** Condividere dati (tipicamente immutabili, o protetti da un altro smart pointer come `Mutex` al suo interno) tra più thread.

Smart Pointers Definiti dall'Utente

Le fonti menzionano anche la possibilità di definire smart pointers personalizzati implementando specifici **traits**. I tratti principali da implementare sono `Deref` (per definire il comportamento dell'operatore `*`) e `Drop` (per definire cosa accade quando lo smart pointer esce dallo scope). In alcuni casi, potrebbe essere necessario ricorrere a costrutti `unsafe` del linguaggio per manipolare la memoria direttamente, bypassando i normali controlli di Rust. Un esempio di `MyBox<T>` che implementa `Deref` e `Drop` è fornito nelle fonti.

Ogni tipo di smart pointer è ottimizzato per uno specifico scenario d'uso (singolo owner su heap, multipli owner single-thread, mutabilità interna single-thread, cicli di riferimento, condivisione multi-thread con mutabilità o immutabilità controllata).

Domanda

Implementazione di uno smart pointer

Risposta

E' possibile definire smart pointers personalizzati in Rust. Ciò si realizza definendo una `struct` che incapsula i dati o un puntatore ai dati, e quindi **implementando specifici traits**. I tratti fondamentali per creare un tipo che si comporta come un puntatore e gestisce la deallocazione sono:

- **Deref** : Permette di definire il comportamento dell'operatore di dereferenziazione (`*`) per il tipo. Implementando `Deref`, si consente al compilatore di applicare automaticamente la dereferenziazione per accedere ai metodi del tipo interno. L'associated type `Target` definisce il tipo a cui lo smart pointer "punta".
- **Drop** : Permette di definire codice personalizzato da eseguire quando un'istanza del tipo smart pointer esce dallo scope. Questo è il meccanismo principale per gestire la deallocazione delle risorse (come la memoria sullo heap).

Le fonti forniscono l'esempio di una `struct MyBox<T>` che incapsula un valore `T` e implementa i tratti `Deref` e `Drop` per dimostrare come si crea uno smart pointer definito dall'utente.

In alcuni casi, per implementare funzionalità avanzate o per interagire con basso livello, potrebbe essere necessario ricorrere a costrutti `unsafe` del linguaggio, che bypassano i controlli di sicurezza standard di Rust sulla memoria.

Domanda

Chiusure in Rust (e loro interazione con l'ownership e il borrowing)

Risposta

In generale, una chiusura è definita come l'insieme del codice di una funzione e dei valori delle variabili libere (quelle non definite all'interno della funzione stessa ma utilizzate nel suo corpo). L'idea è che una funzione possa essere invocata in un punto e in un momento molto distanti da dove è stata definita, e la chiusura "impacchetta" i dati necessari affinché l'esecuzione del corpo della funzione possa avvenire senza problemi.

Mentre in alcuni linguaggi ciò può implicare la copia dei valori delle variabili libere, che tendono a essere immutabili all'interno della chiusura, o l'uso di meccanismi come puntatori a catena statica, Rust integra questo concetto con il suo rigoroso sistema di gestione della memoria.

In Rust, le chiusure **catturano le variabili libere** utilizzate al loro interno. Il modo in cui questa cattura avviene è strettamente legato al sistema di **ownership e borrowing** di Rust. Le chiusure possono catturare le variabili in due modi principali:

1. **Per trasferimento di ownership (`move`)**: Quando si usa la parola chiave `move` prima dei parametri della chiusura (`move |params| { body }`), la chiusura **acquisisce la ownership** delle variabili libere utilizzate nel suo corpo. Questo significa che le variabili catturate in questo modo non possono più essere utilizzate nel codice chiamante dopo la definizione della chiusura, perché la loro ownership è stata trasferita alla chiusura stessa. Questa modalità è necessaria, ad esempio, quando la chiusura deve vivere più a lungo dello scope in cui è definita, come spesso accade con i thread o quando si restituisce una chiusura da una funzione.
2. **Per borrowing (mutabile o immutabile)**: Se la parola chiave `move` non è utilizzata, la chiusura **prende in prestito (borrows)** le variabili libere. Le regole standard del borrowing di Rust si applicano a questa cattura:
 - Se la chiusura ha bisogno di **modificare** una variabile catturata, prenderà un **prestito mutabile (`&mut`)** di quella variabile.
 - Se la chiusura ha solo bisogno di **leggere** una variabile catturata, prenderà un **prestito immutabile (`&`)**.

Il compilatore di Rust analizza l'uso delle variabili libere all'interno del corpo della chiusura e determina il tipo di borrowing necessario. Le rigide regole di borrowing di Rust vengono

quindi applicate per garantire la memory safety. Ad esempio, se una chiusura cattura mutabilmente una variabile per borrowing, non sarà possibile avere altri prestiti mutabili o immutabili di quella stessa variabile attivi contemporaneamente nello scope chiamante, finché la chiusura (il prestito) non termina.

Il sistema di tipi di Rust, essendo molto rigoroso, richiede che esistano **diversi tipi (e di conseguenza, diversi trait)** per le chiusure, a seconda del modo in cui catturano le variabili e di come possono essere chiamate. Sebbene i nomi specifici (`Fn` , `FnMut` , `FnOnce`) non siano dettagliati nei testi forniti, il concetto fondamentale è che una chiusura che acquisisce ownership (tramite `move`) e quindi "consuma" le variabili catturate, può essere chiamata solo una volta (`FnOnce`), mentre una che prende in prestito mutabilmente (`FnMut`) o immutabilmente (`Fn`) può essere chiamata più volte, con le restrizioni imposte dalle regole del borrowing.

L'interazione con l'ownership e il borrowing è fondamentale per la **"fearless concurrency"** di Rust. Ad esempio, quando si usano thread, spesso è necessario che la chiusura passata al thread acquisisca la ownership dei dati di cui ha bisogno (tramite `move`) per garantire che quei dati non vengano deallocati o modificati dallo scope originale prima che il thread finisca. Per condividere dati mutabili tra thread, si combinano spesso smart pointer come `Arc` (Reference Counting atomico per sharing thread-safe) e `Mutex` (per l'accesso esclusivo ai dati), e una chiusura `move` può catturare un `Arc<Mutex<T>>` per poter accedere e modificare i dati condivisi in modo sicuro nel nuovo thread. Questo è un esempio di come smart pointer e chiusure interagiscano all'interno del sistema di ownership di Rust.

GADT

Domanda

Sintassi dei Generalized Abstract Data Types e loro rappresentazione a run-time

Risposta

I Generalized Abstract Data Types estendono il concetto degli Algebraic Data Types (ADT). L'idea fondamentale dei GADT è che **i tipi dei dati possono dipendere da termini o valori**. Questo permette di definire strutture dati che **catturano invarianti** (proprietà che la struttura dati deve rispettare) a livello di tipo, permettendo al compilatore di verificare la correttezza del codice a compile-time.

Nei linguaggi che supportano i GADT, come OCaml negli esempi forniti, la sintassi di dichiarazione di un tipo GADT differisce dagli ADT standard perché il tipo risultante del costruttore di dati può essere specificato in modo più flessibile. Negli esempi forniti, una dichiarazione di un tipo `expr` con GADT in OCaml si presenta così:

```
type 'a expr =  
  Num : int -> int expr  
| Bool : bool -> bool expr
```

```
| Mult : int expr * int expr -> int expr
| And : bool expr * bool expr -> bool expr
| Eq : (* forall 'b *) 'b expr * 'b expr -> bool expr
```

In questa dichiarazione, 'a rappresenta una variabile di tipo che indica il *tipo di ritorno* o il *tipo di valore* che l'espressione incapsulata nel costruttore rappresenta. Ogni costruttore di dati (come Num, Bool, Mult, And, Eq) ha una firma di tipo che non solo specifica il tipo degli argomenti che accetta, ma anche il tipo esatto della *risultante istanza* del tipo expr. Ad esempio:

- Num : int -> int expr significa che il costruttore Num prende un int e produce un expr di tipo int expr.
- Mult : int expr * int expr -> int expr significa che il costruttore Mult prende una coppia di espressioni che sono entrambe di tipo int expr e produce un'espressione di tipo int expr (una moltiplicazione tra interi deve risultare in un intero).
- Eq : (* forall 'b *) 'b expr * 'b expr -> bool expr significa che il costruttore Eq prende una coppia di espressioni dello stesso tipo ('b expr , dove 'b può essere qualsiasi tipo purché le due espressioni siano dello stesso tipo) e produce un'espressione di tipo bool expr (un confronto deve risultare in un booleano).

Questa sintassi consente di **codificare nel sistema di tipi le regole semantiche** che altrimenti dovrebbero essere controllate a run-time o tramite logica applicativa più complessa. Il compilatore, durante il pattern matching su istanze di GADT, può dedurre equazioni tra tipi e risolverle, cambiando dinamicamente il tipo delle variabili locali e il tipo di ritorno della funzione in base al ramo del pattern matching che corrisponde. Ciò permette di scrivere funzioni come eval in modo più semplice e leggibile, molto simile alla versione non tipata vista in Python, ma con la garanzia di correttezza fornita dal compilatore a compile-time.

Rappresentazione a Run-time dei GADT (basato sugli esempi in OCaml)

Nei linguaggi con GADT come OCaml (e in generale nei linguaggi funzionali che non usano monomorfizzazione per tutto), la rappresentazione a run-time dei dati segue un approccio basato su **tag e payload**. Per i dati complessi (boxed), viene allocata memoria sull'heap, e il dato include una prima word contenente un tag che ne identifica il tipo, la dimensione e, in alcuni sistemi, un reference counter o altre informazioni.

Per quanto riguarda specificamente i GADT e il controllo di tipo che offrono, **"A runtime il discorso sui tipi viene cancellato"**. Questo significa che l'informazione di tipo "dipendente" che era presente a compile-time per permettere le verifiche e le deduzioni del compilatore **non ha un costo a run-time**. La struttura dati effettiva in memoria per un'istanza di un GADT sarà una sequenza di bit, organizzata secondo la convenzione di rappresentazione del linguaggio (tag + payload per i tipi boxed). Le verifiche sulla correttezza dei tipi sono già state effettuate dal compilatore durante la fase di tipizzazione basata sulle equazioni di tipo

dedotte dal pattern matching. Il "cast" implicito che il compilatore permette di fare, ad esempio nella funzione `equal` mostrata con l'uso delle dimostrazioni, non altera la memoria, ma cambia solo l'interpretazione associata ai bit.

In sintesi, la potenza dei GADT risiede nella loro capacità di arricchire il sistema di tipi statico per catturare invarianti complesse, permettendo al compilatore di fare verifiche approfondite, ma la loro rappresentazione a run-time è tipicamente la stessa degli ADT normali nello stesso linguaggio, con l'informazione di tipo dipendente che viene "cancellata" dopo la compilazione.

Domanda

Applicazioni dei GADT: imposizione di invarianti alle strutture dati e relative semplificazioni del codice; ad-hoc polymorphism (e.g. implementazione di una `printf`); tipo `eq`

Risposta

L'idea fondamentale dei Generalized Abstract Data Types (GADT) è che **i tipi possono dipendere da termini o valori**. Questo consente di rappresentare, con un tipo di dato, tutte le prove di un determinato enunciato. In un linguaggio con tipi dipendenti (una forma avanzata dei GADT), è possibile scrivere specifiche esatte, come ad esempio il tipo di una lista che è una permutazione di un'altra lista. Questo approccio trasforma la programmazione nell'atto di "abitare tipi" o "scrivere dimostrazioni", dove passare gli input alle funzioni significa anche passare le prove che gli input soddisfano le precondizioni.

I linguaggi che supportano i GADT scelgono un compromesso che privilegia fortemente la correttezza del codice.

1. Imposizione di Invarianti alle Strutture Dati e Relative Semplificazioni del Codice: I

GADT permettono di **codificare direttamente nel sistema di tipi le proprietà (invarianti)** che una struttura dati deve rispettare. Consideriamo l'esempio di una struttura dati che rappresenta alberi di sintassi astratta per espressioni. Utilizzando Abstract Data Types (ADT) standard, come mostrato negli esempi Python o OCaml pre-GADT, è possibile definire espressioni sintatticamente corrette ma semanticamente errate o mal tipate, come una moltiplicazione tra un intero e un booleano, o un confronto di uguaglianza tra una concatenazione di stringa/intero e un intero. Queste espressioni mal tipate non vengono rilevate dal compilatore e causano errori solo a run-time, spesso durante la valutazione dell'espressione.

Con i GADT, la definizione del tipo `expr` può essere arricchita specificando che il tipo di ritorno (indicato dalla variabile di tipo `'a`) dipende dal costruttore:

```
type 'a expr =  
  (* Num produce un int expr *)  
  Num : int -> int expr  
  (* Bool produce un bool expr *)  
  | Bool : bool -> bool expr
```

```
(* Mult tra due int expr produce un int expr *)
| Mult : int expr * int expr -> int expr
(* And tra due bool expr produce un bool expr *)
| And : bool expr * bool expr -> bool expr
(* Eq tra due 'b expr produce un bool expr *)
| Eq : (* forall 'b *) 'b expr * 'b expr -> bool expr
```

Questa sintassi `Costruttore : TipoArgomenti -> TipoRisultanteDelGADT` permette al compilatore di **garantire a compile-time che solo espressioni ben tipate possano essere costruite**. Ad esempio, il costruttore `Mult` richiede esplicitamente due argomenti di tipo `int expr` e produce un `int expr`. Un'espressione come `Mult(Bool(...), Num(...))` sarebbe rifiutata dal compilatore.

Questa imposizione di invarianti a livello di tipo porta a **significative semplificazioni del codice** che elabora queste strutture dati, come una funzione di valutazione (`eval`). Senza i GADT, una funzione `eval` in OCaml dovrebbe restituire un tipo di dato disgiunto (`res`) per gestire i diversi tipi possibili di risultati (`int`, `bool`, o un errore in caso di espressione mal tipata non rilevata prima). Il codice risulta complesso e verboso a causa della necessità di pattern matching sul tipo di risultato intermedio. Con i GADT, la funzione `eval` può avere un tipo di ritorno polimorfo `type a. a expr -> a`. Il compilatore, durante il pattern matching sull'input GADT, è in grado di dedurre equazioni tra tipi e risolverle. Se il pattern `Num n` corrisponde, il compilatore deduce che l'input era di tipo `int expr`, e poiché il tipo dell'input generico era `a expr`, conclude che `a` deve essere `int`. Di conseguenza, sa che il tipo di ritorno in questo ramo deve essere `int`. Questo permette di scrivere la funzione `eval` in modo molto più diretto e leggibile, simile alla versione non tipata in Python, ma con la **garanzia che il compilatore verificherà la correttezza dei tipi staticamente**.

2. **Polimorfismo Ad-Hoc (Implementazione di una `printf`):** I GADT sono molto utili in casi in cui il tipo di ritorno o il comportamento di una funzione dipendono dal *valore* (o dalla struttura) di uno degli argomenti, permettendo una forma di polimorfismo ad-hoc controllato staticamente. Un esempio classico è l'implementazione tipo-sicura di funzioni come la C `printf`. Si può definire un tipo GADT `spec` per rappresentare gli specificatori di formato (`%d`, `%f`, `%s`, ecc.), dove il tipo `'a` di `a spec` indica il tipo di dato che quello specificatore si aspetta:

```
type 'a spec =
(* %d si aspetta un int *)
| D : int spec
(* %f si aspetta un float *)
| F : float spec
(* %s si aspetta una stringa *)
| S : string spec
(* Combinazione di specificatori si aspetta una tupla *)
| Pair : 'a spec * 'b spec -> ('a * 'b) spec
```

```
(* Specificatore per lista si aspetta una lista *)  
| List: 'a spec -> ('a list) spec
```

Una funzione `to_string : type a. a spec -> a -> string` può quindi prendere uno specificatore (`a spec`) e il dato corrispondente (`a`) e produrre una stringa. Il tipo `a` è dedotto dal compilatore in base allo specificatore fornito. Ad esempio, se si usa `List (Pair (D,F))`, che ha tipo `((int * float) list) spec`, il compilatore per la chiamata a `to_string` dedurrà che `a` deve essere `(int * float) list`. La funzione si aspetterà quindi una lista di coppie `(int, float)`, e il compilatore garantirà che l'argomento fornito sia di quel tipo.

Questo è un esempio di polimorfismo ad-hoc dove il comportamento (quale specificatore processare) e il tipo dell'argomento successivo sono determinati dallo specificatore GADT, garantendo la correttezza a compile-time.

3. **Il Tipo `eq` (Dimostrazioni di Uguaglianza):** In linguaggi con sistemi di tipi espressivi che supportano i GADT, è possibile rappresentare le dimostrazioni logiche come dati. Un esempio è il tipo `('a, 'b) eq`, usato per rappresentare una dimostrazione che i tipi `'a` e `'b` sono uguali. L'unico costruttore di questo tipo è `Refl : ('c, 'c) eq`, che rappresenta la proprietà riflessiva dell'uguaglianza (qualsiasi tipo `c` è uguale a se stesso). Avere un dato di tipo `(A, B) eq` significa avere una dimostrazione che il tipo `A` è uguale al tipo `B`. Poiché l'unico modo per costruire un dato di tipo `('a, 'b) eq` è usare `Refl` (che ha tipo `('c, 'c) eq`), il compilatore deduce che `a` deve essere uguale a `b` affinché la dimostrazione sia valida. Questo tipo può essere usato per implementare un **cast tipo-sicuro**: `let cast : type a b. (a, b) eq -> a -> b = function Refl -> (fun x -> x)`. Questa funzione prende una dimostrazione che `a` è uguale a `b` e un valore di tipo `a`, e restituisce lo stesso valore, ma con il tipo `b`. Questo cast è sicuro perché la sua validità è garantita dalla dimostrazione passata come argomento.

È cruciale notare che questo **cast non altera in alcun modo la memoria**; cambia solo l'interpretazione o il tipo che il compilatore associa alla sequenza di bit sottostante.

Il tipo `eq` e la funzione `cast` possono essere usati per rendere tipo-sicure operazioni che altrimenti sarebbero problematiche. Ad esempio, nella funzione `equal` che confronta due `expr`, può essere utile verificare se due espressioni hanno lo stesso tipo sottostante (e.g., entrambe `int expr`) anche se la firma generica le tratta come potenzialmente diverse (`a expr, b expr`). Una funzione `same_type : a expr -> b expr -> (a expr, b expr) eq option` può controllare se i costruttori indicano tipi compatibili e, in tal caso, restituire `Some Refl` (una dimostrazione che `a expr` e `b expr` sono, di fatto, uguali come tipi). La funzione `equal` può quindi utilizzare questa dimostrazione per "convincere" il compilatore che i due valori possono essere confrontati in modo tipo-sicuro, eventualmente dopo aver "castato" uno di essi al tipo dell'altro usando la funzione `cast` basata sulla dimostrazione.