**Emerging Programming Paradigms Core Concepts**

**1. Programmazione ad Attori + Erlang**

**Paradigma ad attori**

Attori NON condividono stato/behaviour/memoria comunicano attraverso messaggi.

Hanno PID + mailbox + behaviour (mappa: msg -> [azioni] + new behaviour). 1 attore = 1 thread

**Paradigma ad agenti**

Agenti NON condividono stato/behaviour/memoria comunicano solo modificando l’ambiente condiviso.

**Linguaggi funzionali**

Caratteristiche: funzioni higher-order (funzioni come oggetti di prima classe), purezza (assenza di side effects e immutabilità dei dati).

**Erlang**

Sviluppato nel ‘87 in Ericsson per disponibilità e concorrenza massiva. Paradigma ad attori (no memoria condivisa, si scambio messaggi). Politica di fault tolerance “let it fail”.

BEAM unico kernel process multi-threaded (1 kernel thread per core), ogni attore un language thread.

Pattern matching: =, funzioni, receive, case of, try-catch.

Erlang (come altri linguaggi recenti), simula lo stack all’interno dello heap come array estensibile, quando si riempie viene copiato in un’area di memoria di dimensione doppia.

**Rappresentazione a run-time dei valori in memoria in Erlang**

Tutti i dati occupano una parola, quelli unboxed sono contenuti per intero nella parola, per i boxed la parola contiene un puntatore al dato nello heap.

I linguaggi non tipati a runtime non sanno che dati stanno manipolando: Erlang usa i bit meno significativi come tag variabile (max 6 bit), 0 puntatore (solo celle pari), 01 interi, ecc.

Altri linguaggi usano due parole (payload e puntatore allo heap con altri dati e/o tag).

OCaml non è tipato a runtime usa solo un bit: 0 per i puntatori (per GC), 1 per il resto.

**Garbage Collection (GC)**

Gestione della memoria: manuale (C/C++), smart pointers (C++/Rust), garbage collection (OCaml, Erlang, ...).

Reference counting: ogni dato è una coppia RC-payload, dato puntato RC+=1, se RC=0 deallocato.

Contro: memory leaks in strutture circolari; soluzione: weak pointers (non incrementano il RC).

Costo: O(n), si liberano ricorsivamente tutti gli elementi della lista.

Mark and sweep: heap diviso in 2 aree (from space / to space) e rootset (registri ed elementi nello stack che puntano ad elementi nello heap). Quando l’occupazione del from space raggiunge una certa soglia si fa mark: si marca come “vivo” ogni elemento nel from space puntato da un elemento nel rootset; e sweep: si spostano questi elementi dal from space al to space sostituendole con un “forward pointer”, poi da scan start a scan stop del to space si ripete il procedimento ricorsivamente. Al termine from space diventa to space (i puntatori vengono invertiti). Con mark and sweep non si ha più aritmetica dei puntatori. Costo: O(#celle vive).

Ipotesi generazionale: la maggior parte dei dati o muore subito o dopo tanto. Ottimizzazione: si divide lo heap in 3 aree (young heap (from/to space) e old heap) e si ha un indicatore watermark. Ad ogni passata si spostano i dati sotto al watermark dal from space all’old heap, e quelli sopra dal from space al to space (minor collection). Ogni tanto si fa un mark and weep anche dell’old heap (major collection). Costo: O(#celle vive).

Se c’è un puntatore dall’old heap al from space la cella non verrebbe considerata viva. Questo scenario è assente nei linguaggi con immutabilità, nei linguaggi con mutabilità i puntatori dall’old allo young sono spostati nel rootset.

**Tail recursion**

Una chiamata di coda è una chiamata a funzione dopo la quale non avviene nessuna operazione. Un’ottimizzazione per chiamate ricorsive di coda è quella di mantenere fisso l’indirizzo di ritorno della prima chiamata, e sovrascrivere i parametri della funzione ad ogni chiamata ricorsiva.

Alcune trasformazioni meccaniche per riscrivere una chiamata ricorsiva in TR sono le seguenti:

- Simulazione dello stack: ad ogni chiamata TR viene passato uno stack a cui è aggiunta una funzione A -> B; alla condizione di terminazione viene invocata un’altra chiamata TR che invoca ogni funzione sull’output della precedente, passandosi lo stack a cui vengono man mano sottratte le funzioni invocate.

- Continuation Passing Style: ad ogni chiamata TR invece che passare uno stack con tutte le chiamate da fare si passa una sola funzione; la chiamata ricorsiva passa una nuova funzione che contiene la funzione precedente (ricevuta in input) invocata con un nuovo input (ovvero la computazione da eseguire). Al termine si avranno varie funzioni annidate racchiuse in un’unica funzione che verranno eseguite.

- Trampolines: per la chiamata TR viene creato un record con un atomo call e una funzione che contiene la chiamata ricorsiva. Al termine con un semplice ciclo se l’atomo è call viene invocata effettivamente la funzione.

**Gestione eccezioni**

Quando si entra in un blocco try-catch viene inserito sullo stack uno speciale catch frame, se viene sollevata un’eccezione viene rimosso ogni frame fino a quando non se ne trova uno speciale in grado di gestirla. Dopodichè viene rimosso anche questo frame speciale ed è eseguito il codice del catch. Se è presente un blocco of viene eseguito solo dopo aver rimosso lo speciale frame, uguale per il blocco after che tuttavia aggiunge di per se un frame allo stack, è quindi impossibile ottimizzare la tail ricorsione. È ottimizzabile invece se viene fatta nell’of.

**Accesso ai bit a basso livello**

<< X:4 || X <= 16#FF23A3:24 >> >>. list comprehension in cui si chiedono tutte le “porzioni” di 4 bit nel colore.

**Concorrenza**

In Erlang la receive è bloccante mentre la send è asincrona. La receive ottiene i messaggi in base all’ordine di ricezione, non di invio (i due coincidono solo se il mittente e il nodo sono gli stessi).

Hot code swapping: la BEAM mantiene due versioni, specificando il nome del modulo prima, si invoca la nuova.

Linking: il collegamento di due attori è bidirezionale, quando uno termina invia un segnale a tutti i collegati, exit(normal) è solo un informazione, exit(kill) obbliga tutti a terminare, exit(PID, msg) obbliga un attore a terminare.

Monitoring: simile al linking ma è unidirezionale e può essere fatto più volte, la morte di un attore invia un segnale a tutti ma non li obbliga a terminare. Per evitare race conditions si può usare spwn\_link e spawn\_monitor.

**Parallelizzazione**

Creare un attore per ogni porzione di codice da parallelizzare sarebbe molto dispendioso, perché creare un numero di attori molto superiore al numero di core nel PC creerebbe una quantità di content switch elevata. Un’alternativa è avere un pool fissato di attori già creati da cui attingere per eseguire le operazioni in parallelo, oppure creare attori e distruggerli in base al bisogno (con un limite superiore).

Si possono creare cookie per riconoscere un processo tramite make\_ref().

**Porte**

In caso si vogliano utilizzare librerie esterne che non usano un approccio funzionale in Erlang è possibile wrapparle in un attore ed accedere alle loro funzioni tramite porte (del tutto simili a PID). Le funzioni usate sono open\_port(), read(P) per leggere su una porta, invio alla porta di messaggi con atomi command e close.

**Distribuzione**

I nodi Erlang si scambiano un token che risiede su ogni macchina prima di poter comunicare. La migrazione di codice può avvenire forwardando le richieste da una macchina ad un’altra, oppure spostando tutti i dati.

**Rappresentazione degli atomi**

Gli atomi sulla BEAM sono rappresentati da un valore progressivo, la tabella atomo-numero viene scambiata quando si trasferiscono gli atomi tra due nodi (le volte successive sono trasferiti solo i nuovi atomi).

OCaml invece utilizza l’hash dell’atomo, possono esserci collisioni, anche se rare, che vanno risolte manualmente.

**Runtime Architecture**

Erlang adotta un’architettura ibrida in cui ogni processo ha il proprio heap (e lo stack di conseguenze) ma tutti condividono un’area (in sola lettura) per lo scambio dei messaggi (scrittura con lock).

**Multitasking**

Il multitasking con content switch può essere: collaborativo o pre-emptive. Il primo lascia al programmatore la decisione di quanto eseguire, mentre il secondo viene deciso dal sistema tramite un interrupt hardware.

Erlang usa una via di mezzo, in cui l’utente può rilasciare la computazione (collaborativo) ma dopo un po’ viene automaticamente liberata, questo è possibile grazie alla presenza della BEAM che usa una yield.

**2. Blockchain**

Un ledger è una struttura dati astratta simile ad uno stack, le cui uniche operazioni sono empty, push (di una transazione) ed iter (in sola lettura). Una blockchain è un’implementazione peer-to-peer di un ledger. Uno smart contract è un attore il cui stato quiescente è memorizzato sulla blockchain.

Il problema della consistenza delle copie di un ledger è indecidibile, la soluzione consiste nell’indebolire l’ipotesi di consistenza e di immutabilità. Proof of work: fornire computazione per assicurare un “investimento” (di energia) nell’utilizzo della blockchain. Proof of stake: garanzia del corretto operato di un utente sulla blockchain dato dal fatto di aver immobilizzato una certa quantità di capitale.

**3. Futures**

Un future è un tipo di dato astratto che descrive una computazione. Nei linguaggi logici le variabili sono future. Il valore è calcolato al massimo una volta (memoization) e memorizzato in una tabella hash (così come i suoi side effects). La computazione può avvenire o alla creazione del future, o quando è letto la prima volta (yield) [thunk]. Permettono computazione lazy in linguaggi eager.

Una promise è un future a cui non viene specificata la funzione quando è creato, ma in un secondo momento.

In JS il then di un future crea un nuovo future che fa una get del precedente, mentre per le promises non avviene alcun get e il codice rimane asincrono. Il future pipelining permette di eseguire tutta la computazione sul nodo e restituire solo il risultato, invece che passare avanti e indietro i risultati.

**4. Chiusure**

Una chiusura è una funzione di prima classe definita all’interno di uno o più blocchi che accede a variabili locali ai blocchi circostanti (se accedesse solo a variabili locali o globali non sarebbe una chiusura). Il C non permette chiusure perché non ha modo di salvare le variabili usate da una funzione e dichiarate fuori da essa. Il Pascal non ha chiusure ma ha funzioni annidate, queste possono essere usate solo dentro al blocco in cui sono dichiarate.

In C è possibile implementare una chiusura memorizzando le variabili sullo heap e restituendo una coppia (puntatore al codice della funzione, record di puntatori alle variabili utilizzate).

NOTA: nei linguaggi a oggetti (es. js) una chiusura è un oggetto di una classe con un solo metodo, “this” in una chiusura si riferisce a oggetto chiusura, non all’oggetto contenente la chiusura.

Un linguaggio è di ordine superiore quando le funzioni (e quindi anche le chiusure) sono oggetti di prima classe. Le chiusure sono utilizzate per: data-hiding (funzione che crea una variabile e ritorna una chiusura getter e una setter), strutture di controllo (passare in input una chiusura che viene eseguita a certe condizioni), callback (al termine di una computazione si esegue la chiusura ricevuta in input), decoratori (wrappare una funzione in un altra per poterne estendere il comportamento senza doverla modificare in modo permanente), limitare l’accessibilità (una chiusura che limiti l’accesso solo alle variabili di cui ha bisogno).

**5. OCaml**

Linguaggio funzionale ma non puro di tipo eager con thunks, oggetti usati solo per late binding, basato su type inference, sequenziale (non supporta multi-core), fortemente tipato.

Con la type inference Hindley-Milner [polimorfismo UNIFORME/GENERICO] non c’è polimorfismo ad-hoc (funzioni reimplementate con tipi in input differenti, ovvero overloading) esempio per gli interi si usa “+” e per i float “+.”, il tipo delle variabili viene inferito dal loro uso, in assenza di vincoli vengono generati schemi di tipi (quantificatori universali), i tipi possono essere ristretti esplicitamente, questo sistema di tipi è decidibile, a runtime non si ha rappresentazione dei tipi.

Quando si usa un ADT il compilatore controlla che i pattern matching siano esaustivi e privi di duplicati.

Vige la closed world assumption: ovvero tutte le possibili forme dei dati sono note, i cambiamenti sono rari e richiedono un ripensamento generale, gli invarianti quindi rimangono corretti.

**6. ADT**

Un ADT è un’unione disgiunta, taggata da atomi, di tuple/record di tipi.

In C un ADT è implementabile con una struttura che contiene un tag intero e un union, in memoria in un dato momento ci sarà sempre e solo un insieme di byte, il tag serve per “decifrare” questi byte. In Erlang gli ADT sono letteralmente record taggati da atomi (qui gli ADT sono visti come enumerated types).

In OO se sono disponibili le chiusure si crea un’interfaccia che implementi i metodi utilizzabili dagli ADT (di base basta match) e che prendono in input una serie di chiusure (una per ogni tipo dell’ADT), dopodichè si istanzia concretamente questa interfaccia per ogni tipo dell’ADT e si implementa il match invocando la chiusura opportuna. Se non sono disponibili le chiusure è possibile utilizzare il pattern visitor, quindi si passa a match l’interfaccia Visitor con un generics, questa implementerà una funzione per ogni tipo dell’ADT, che sarà poi chiamata nel match. Passare un’interfaccia appesantisce l’esecuzione in quanto viene fatto late binding a runtime.

Con le classi però si lavora in ipotesi di mondo aperto per chiuderlo è necessario fare sealing, ovvero impedire che l’interfaccia possa essere estesa e di conseguenza anche le sue istanziazioni. In Java è possibile farlo creando una classe astratta con un costruttore privato (non è estendibile) e le implementazioni concrete come inner class.

In Kotlin è sufficiente dichiarare una classe sealed che implementi come data class (classe che contiene principalmente dati) i tipi dell’ADT, dopodichè con uno switch si analizza il tipo e si esegue l’azione (per lo switch avviene static dispatch, ogni ramo “sa” già il tipo del dato).

**Null values**

I valori option sono una forma di ADT e servono a gestire l’eventualità di valori null. Per la natura degli ADT in OCaml un option può avere a sua volta un tipo option. Come contro a runtime occupa più spazio (per il tag) e può essere boxed (es Ocaml e Haskell), quindi l’accesso è più lento. Le eccezioni sono più veloci che gli option, occupano solo uno stack frame in più mentre un ADT va sempre allocato sullo heap.

Kotlin non usa ADT per gli option (dovrebbe spacchettare ogni volta che interagisce con Java) ma nullable values. Quesi non possono essere annidati. T? e T hanno la stessa rappresentazione in memoria (spreca meno memoria che gli ADT). Se viene controllato che T? != null il compilatore fa il cast a T. Casting: T y = x as T (eccezione se x non è T x è null), T? y = x as T? (eccezione de x non è T), T? y = x as? T (ritorna un T? che è null se x non è T o x è null).

**7. Expression problem e varianti polimorfe**

L’expression problem ha come obiettivo il definire un tipo di dato per casi, in cui è possibile aggiungere nuovi casi al tipo di dato o aggiungere nuove funzioni che lavorino con questo, senza ricompilare il codice esistente. Una variante prevede che sia anche possibile dati due tipi di dato crearne un terzo che è l’unione disgiunta dei due.

Static polym.: stesso nome di metodo overloaded con tipi num param o ritorno diversi nella della stessa classe.

Dynamic polym.: stesso nome di metodo overridden con uguale signature in classi differenti.

I linguaggi OO possono risolvere il problema usando 4 delle 5 features che li caratterizzano: inheritance, subtype polymorphism, dynamic dispatch, open recursion (la quinta è encapsulation).

In Java si risolve creando un’interfaccia con le funzioni che si vuole siano eseguite sul tipo di dato la quale è concretizzata da classi che ne implementano i costruttori di dato. Per aggiungere nuovi costruttori si aggiungono nuove implementazioni dell’interfaccia, per aggiungere nuove funzioni si eredita l’interfaccia ma poi vanno reimplementati i costruttori. La soluzione lavora solo in mondo aperto ed ha molti boilerplate (codice copiato in giro senza modifiche).

In OCaml bisogna aprire in profondità e in larghezza. L’apertura in profondità avviene passando all’ADT un tipo generico e ricorrendo su quello invece che sul tipo di dato stesso, alle funzioni che lavoreranno sul dato viene passata anche una funzione per trattare questi tipi parametrizzati, il mondo si può sempre chiudere passando “type e1 = e1 e” e le funzioni dovranno prendere “let rec eval1 x = eval eval1 x”. L’apertura in larghezza avviene definendo il tipo di dato con un ulteriore costruttore “More of ‘b”, che prende un altro tipo parametrizzato ed è estensibile a piacere (soluzione usata in Haskell).

In OCaml sono presenti le varianti polimorfe. La differenza con le varianti normali (ADT) è che non è necessario definire un costruttore dentro ad un variante prima di poterlo usare, ma bensì il compilatore inferirà un tipo di dato generico che comprenda ALMENO il costruttore indicato. Questo si chiama polimorfismo di riga, ovvero un’espressione è tipata come ∀α.[‘Costr of int | α] si indica qualsiasi riga che contiene almeno quella indicata. Si può tipare anche con polimorfismo bounded ovvero ∀α.α ma con i vincoli che α sia una riga (unione di tipi taggata) e contenga almeno ‘Costr of int. Una lettura non corretta è tramite polimorfismo subtype quindi tipata con ∀α.α dove α è un sottotipo del tipo [‘Costr of int], ma il tipaggio implicito violerebbe quello principale.

Il confronto è sempre strutturale non nominale (tag diversi e stessa struttura sono la stessa cosa). Con le varianti polimorfe è possibile risolvere l’expression problem e la sua estensione: type ‘a ef = [‘a e | ‘a f] le funzioni dovranno poi trattare i casi in cui il tipo ef abbia la forma di e o di f. Anche qui il mondo è chiudibile: type exp = exp ef.

**8. GADT**

**Tipi dipendenti**

L’idea è che un tipo possa dipendere da un termine (es. list 3), in questo modo è possibile scrivere tipi che rappresentino proprietà arbitrarie del codice. Es. tipo funzione: l1:list -> l2:list \* is\_permutation l1 l2. Nell’esempio is\_permutation è un elemento abitabile ovvero è possibile costruire un elemento di questo tipo di dato solo se l1 è una permutazione di l2 (abitabile: ha un valore). In questo modo vengono passati coi dati anche delle prove (il cui codice va comunque scritto dal programmatore), il compilatore diventa un verificatore di problemi.

**GADT**

Simili agli ADT ma in cui è possibile tipare esplicitamente anche i costruttori di dato. Invece che avere un per ogni all’esterno della dichiarazione di tipo lo si ha all’interno, quindi diventa un esiste. Es. (∀α . α exp \* α exp) -> int exp diventa ∃α . ((α exp \* α exp) -> int exp). Un GADT a runtime ha la stessa forma di un ADT (una tupla taggata). I GADT sono spesso usati per imporre invarianti.

La reflection è una caratteristica dei linguaggi che portano informazioni sul tipaggio di un dato (es. per le classi ottenere i metodi) a runtime (appesantendolo). Con i GADT è possibile farne a meno rappresentando l’informazione sul tipo a runtime solo quando necessaria, es. let rec to\_string: **type** t . t typ -> t -> string = fun y x -> …

Un tipo di dato astratto (o esistenziale) può essere rappresentato con i GADT, es. type stringable = W : ‘a \* (‘a -> string) -> stringable, ovvero si definisce il tipo stringable che contiene la descrizione della funzione W (che dovrà poi essere implementata) la quale è essa stessa tipata come stringable (in Java sarebbe un’interfaccia, molto pesante a runtime).

Una dimostrazione può essere rappresentata tramite GADT ad es. type (\_,\_) eq = Eq : (‘a,‘a) eq, quindi il cast avviene tramite let cast: type a b . (a, b) eq -> a -> b = fun Eq x -> x così viene fornita anche la prova che a == b (qui la proprietà dei dati “uguaglianza” è usata come oggetto di prima classe).

Un esempio di invarianti è dato il tipo type ‘a int\_or\_bool = I : int -> int int\_or\_bool | B : bool -> bool int\_or\_bool la funzione let get: type a . a int\_or\_bool -> a = fun … obbliga i tipi ad essere solo I o B.

**9. Extensible Records a Oggetti Funzionali**

Mentre una variante polimorfa è un’unione non disgiunta (coprodotto) di valori taggati, tipata con polimorfismo di riga, il suo duale, il record estensibile è un record (prodotto cartesiano non disgiunto) tipato con polimorfismo di riga, qui ogni campo è il “tag” e sono tutti a disposizione contemporaneamente, a differenza delle varianti.

Un record estensibile supporta tre operazioni: accesso a un campo, aggiornamento di un campo, e aggiunta di un campo. Con i record estensibili si utilizza static dispatch, si parla infatti di ipotesi di mondo chiuso, gli oggetti invece utilizzano dynamic dispatch. Un pre-metodo è una funzione che prende in input l’oggetto (self o this) da cui estrarrà le funzioni su cui operare, il polimorfismo è banale (dynamic dispatch).

OCaml implementa oggetti, hanno campi e metodi, i campi sono nascosti all’esterno (data-hiding), e utilizzano dynamic dispatch. I campi sono mutabili, i metodi no (metodi uguali tra oggetti diversi puntano ad un’unica tabella comune), ma i pre-metodi usati come campi si. Anche qui viene usato polimorfismo di riga per il tipaggio. Gli oggetti non sono estensibili (a differenza dei RE). Si parla di oggetti funzionali quando nessun campo è mutabile.

Le liste sono covarianti (se s1 <: s2 allora [s1] <: [s2]) ovvero monotone come costruttori di tipo, in Ocaml +’a list; le funzioni sono covarianti in output e controvarianti in input ((a1 -> b1) <: (a2 -> b2) sse b1 <: b2 e a2 <: a1) in OCaml +’a -> -’b.

Problema in Java: dati t2 <: t1 e t3 <: t1 java dice che [s] <: [t] se s <: t ma essendo le strutture mutabili è possibile quindi fare: t1[0] = t2 poi si modifica una cella con t1[0] = t3 il compilatore lo permette ma a runtime crasha. Per avere polimorfismo di riga in Java si usa ? extends T (almeno gli elementi di T) e ? super T (al più gli elementi di T).

Object-based (js, ocaml, …) ≠ class-based (java, python, …), anche se OCaml permette di creare classi ma solo per il riuso di codice (ereditarietà), NON hanno a che fare con il tipaggio (in Java invece le due cose coincidono).

Inheritance is not subtyping: esempio classe point e colored\_point, g : colored\_point -> colored\_point non è sottotipo di f : color -> color anche se colored\_point <: point perché non è controvariante in input.

**10. Traits**

Importante distinguere le classi (cosa un oggetto È) dalle interfacce (cosa un oggetto FA). Il fragile base class problem mostra come i cambiamenti ad una classe (base) possano ripercuotersi sulle classi derivate (ad es. ridefinire un metodo nella base class fa saltare tutte le assunzioni fatte dalle classi derivate che dovranno essere reimplementate). Una soluzione a questo problema può essere la composition, ovvero una classe B non eredita dalla classe A ma un oggetto B prende in input un oggetto A da cui B estrarrà i metodi di cui ha bisogno (e che A vorrà mettere a disposizione); togliendo le classi si possono creare object-less methods (metodi liberi).

Pre-metodo: o.m(o, a1, …, an) ≠ metodo libero: m(o, a1, …, an).

**Go**

Linguaggio a microservizi, ha garbage collection, un sistema debole di tipi, usa thread del linguaggio (non di sistema), ha metodi liberi (a runtime del tutto uguali alle funzioni) che possono quindi essere invocati (dispatch) staticamente (non deve essere estratto da un oggetto). Le funzioni per lavorare con i campi di un oggetto, possono prenderlo in input o dichiararlo come receiver. Nel secondo caso la funzione verrà automaticamente aggiunta ai metodi che l’oggetto mette a disposizione.

Per avere polimorfismo di riga Go introduce le interfacce, un insieme di method signatures (righe) per tipare gli argomenti delle funzioni polimorfe, una funzione che prende in input un tipo interfaccia usa dynamic dispatch. Un elemento di tipo interfaccia a runtime può essere nil oppure (v, p), dove v è il valore passato ((numerico) int, float, (puntatori) struct, array, …) e p è un puntatore ad una tabella con tutti i metodi implementati dal tipo di dato passato v (nei OO viene passato l’oggetto che altro non è che un record con i campi e un puntatore alla tabella dei metodi, qui sono passati i dati e la tabella separatamente). Casting: con x := o.(T) si assegna ad x il valore di o sse o ha tipo interfaccia T; con x, err := o.(T) se non va a buon fine x sarà nil ed err sarà true.

Go NON ha polimorfismo uniforme (generics/templates) quindi per alberi e stack ad esempio si hanno tipi con interfaccia vuota, non sono presenti nemmeno gli ADT. Go NON ha inheritance ma ha composition (viene incluso un oggetto da cui estrapolare i metodi).

**Rust**

Anche in Rust, come in Go, è possibile definire una struct e poi implementarne le funzioni separatamente. Mentre in Go veniva definita una funzione con receive la struttura, in Rust si una impl NomeStruct e un blocco di codice contenente le funzioni. Un trait, come le interfacce di Go, è una lista di dichiarazioni di metodi (righe). I trait sono usati come bound del polimorfismo generico (e polimorfismo di riga), es. impl<T: TrtA + TrtB> StrName<T>{ }. I trait possono avere implementazioni di default, ma non hanno campi, hanno ereditarietà (problema del diamante risolto implementando i metodi di tutti i trait ereditati). Le chiamate di metodo sono statiche (monomorfizzazione), anche in presenza di trait, ma dinamiche in presenza di trait objects.

Un trait object a runtime è un puntatore all’oggetto e un puntatore alla vtable del trait per un tipo concreto (come avviene in Go). Il dynamic dispatch inibisce l’inlining (il codice di una funzione viene iniettato nella funzione chiamante anziché fare una chiamata distinta). Un trait object è un trait che può essere conosciuto solo a runtime [es. statico: f(x); dinamico: f(&x as &Trait)], un trait object può essere ottenuto da un puntatore ad un tipo concreto tramite casting o coercing [fn f(x: &Trait) ; f(&x as &Trait)].

Gli associated types di Rust sono un meccanismo di raggruppamento, ha senso usarli quando è necessario raggruppare tipi insieme (es. in un grafo, una volta definito non si vuole che nodi e archi differiscano più di tanto).

I traits definiscono solo funzioni e implementazioni di default, NON campi (solo behaviour); i mix-ins sono simili ai traits ma permettono ANCHE di definire campi (stato + behaviour), in OCaml il diamond problem è risolto usando regole di precedenza. Le interfacce sono simili ai trait ma NON possono avere implementazioni di default (tranne rari casi tipo in Java >= 8); mix-ins e abstract classes sono equivalenti (di solito i primi hanno ereditarietà multipla mentre i secondi singola).

**Haskell**

Le type classes predicano su un tipo l’esistenza di alcune funzioni, inoltre forniscono un modo strutturato per avere polimorfismo ad-hoc (≡ overloading: posso comparare due tipi qualsiasi purché implementino la type-class Eq).

La definizione avviene tramite class Eq a where … l’implementazione avviene tramite instance (Eq m) => Eq (Maybe m) where … la freccia indica che è richiesto che il tipo sia ciò che è definito a sinistra per essere ciò che è definito a destra (simile a T: TraitA + TraitB in Rust); la freccia nella definizione della type-class fornisce una sorta di inheritance. Type classes e polimorfismo sui costruttori di tipo sono essenziali per l’implementazione delle monadi.

**Polimorfismi**

- ad hoc (≡ overloading): funzioni polimorfe che possono essere applicate ad argomenti di tipi differenti, ma che si comportano in modo diverso in base al tipo dell’argomento a cui sono applicate (stesso nome, sign. diversa).

- parametrico (≡ uniforme/generico): funzioni scritte in modo generico, così da poter gestire valori senza dipendere dal loro tipo (templates/generics).

- subtype (≡ inclusion): funzioni che possono prendere in input un certo tipo T, ma funzionano correttamente anche se viene passato un tipo S che sia sottotipo di T (perché S ha tutte le caratteristiche di T con in più qualcosa).

- di riga (≡ duck typing): simile al subtype ma applicato a tipi record, permette di scrivere programmi che operino solo su una porzione di record e che quindi potranno usare tutti i record che abbiano almeno quei campi.

- bounded: questo polimorfismo è un’intersezione tra il polimorfismo parametrico e il subtype, permette di scrivere funzioni che prendano tipi generici, ma che abbiamo almeno certe caratteristiche.

expre

**11. Monadi**

La referential transparency è una proprietà matematica per la quale f(i) può essere sostituita con o dove o è il risultato della chiamata f(i). Questo non è possibile in presenza di side effects, perché questi andrebbero risollevati ogni volta che f(i) viene eseguita. Le leggi monadiche indicano i requisiti minimi che la monade deve soddisfare:

(1) return x >>= f = f x; (2) m >>= return = m; (3) (m >>= f) >>= g = m >>= (f >>= g); (4) return x >> m = m; (5) m >> return x = return x; (6) (m >> n) >> o = m >> (n >> o);

Un monoide è una tripla contenente: elemento neutro, chiusura (dati due elementi dell’insieme ne restituisce un terzo sempre dell’insieme), associatività.

La writer monad è una coppia di elementi uno il valore delle computazioni, l’altro il side effect.

La reader monad è una funzione da stato (non utilizzato) in un valore r.

Le monadi possono essere viste come descrizione di una computazione (es. reader), arricchimento di una computazione (es. writer), contenitore di dati (es. list).

In Haskell è facile scrivere monadi perché ha type classes e polimorfismo sui costruttori di tipo (Go e Rust hanno i traits ma non hanno rispettivamente polimorfismo generico (Go) e sui costruttori di tipo (Rust)).

**12. Rust**

Rust ha un runtime minimale, usa thread di sistema e NON ha garbage collection. Con la zero-cost abstraction è possibile avere più costrutti ad alto livello possibile purché non introducano costi addizionali a runtime. Rust ha polimorfismo parametrico tramite monomorfizzazione. La guaranteed memory safety garantisce, tramite il sistema di tipi, l’assenza di memory leaks, doppie allocazioni, dangling pointers e data races. Sempre il sistema di tipi garantisce fearless concurrency.

La gestione della memoria avviene tramite due meccanismi: un meccanismo basato su un sistema di ownership dei dati e di borrowing e un meccanismo a runtime basato su smart pointers. Uno smart pointer è un puntatore ed una struttura dati generica che astrae tutte le operazioni effettuabili con un puntatore. Smart pointers comuni sono: Box<T> (puntatore sullo heap), Rc<T> (puntatore con reference counting), RefCell<T> (controllo a runtime che ci sia o un solo puntatore che possa anche scrivere, o N puntatori in sola lettura), Weak<T> (puntatori deboli), Mutex<T> (accede a T acquisendo il lock e rilasciando al termine), Arc<T> (reference counter atomico). È possibile creare uno smart pointer definendo un dato e facendogli implementari certi trait degli smart pointers.

Ogni cella sullo heap ha uno e un solo owner responsabile per la sua deallocazione (ricorsivamente verranno deallocate anche le celle puntate), con l’assegnamento si trasferisce l’ownership della cella. Si può cedere il possesso della cella tramite il meccanismo di borrowing, solitamente passando la reference (&var), si fa borrowing mutabile indicando la mutabilità della reference (&mut x), una variabile prestata blocca l’utilizzo da parte dell’owner.

I lifetime indicano quando una cella di memoria verrà deallocata dall’owner (ogni cella ha il lifetime della reference vera e propria e quello di ciò che viene puntato). In Rust NON si hanno dangling pointers perché viene verificato che il lifetime della reference sia inferiore di quello di ciò che viene puntato (a’ : ‘b lifetime a’ minore di b’). ‘static indica il lifetime del programma.

Uno slice è uno smart pointer per fare borrowing (mutabile o meno) di una parte di struttura dati (le slices hanno lifetimes), a runtime una slice è una tripla: (puntatore alla struttura, numero di bytes, capacità residua della struttura). Le chiusure usano barre al posto di parentesi per gli argomenti e con move se ne trasferisce l’ownership.