**Emerging Programming Paradigms**

[ lezioni 20 - 22 aprile ]

**OCaml**

OCaml è un linguaggio funzionale (di ordine superiore ma non puro) di tipo **eager** con però *lazy thunk* per poter avere computazione lazy. Gli oggetti sono presenti ma utilizzati in modo molto limitato, solo per *late binding* (il metodo che viene chiamato su un oggetto o la funzione chiamata con argomenti viene cercato per nome in fase di esecuzione), per il resto il core è funzionale. OCaml è basato su **type inference**.

La O in Ocaml sta per objects. Il linguaggio nasce in Francia ed è un’implementazione di ML. ML (che sta per metalinguaggio) nasce nel 1973 da Robin Milner ad Edimburgo. Viene sviluppato perchè in quegli anni Milner stava lavorando a LCF, uno dei primi dimostratori interattivi di teoremi. L’idea di Milner era quella di fornire un linguaggio di macro che ponesse enfasi sul calcolo simbolico e sui tipi di dato astratto. Un tipo di dato astratto è un tipo di dato di cui viene nascosta l’implementazione ma si può solamente costruire o usare i valori utilizzando delle funzioni che sono dichiarate in interfaccia e implementate internamente al un modulo.

La prova di un teorema in ML è un tipo di dato astratto, e quindi è impossibile manipolare direttamente una prova per creare una prova sbagliata e gli unici metodi che esponeva una prova erano regole di introduzione e eliminazione della logica. Dalla composizione di prove corrette ottengo prove corrette.

ML ha un enorme successo, nel 1990 viene creato un comitato per standardizzarlo perchè c’erano tantissime implementazioni.

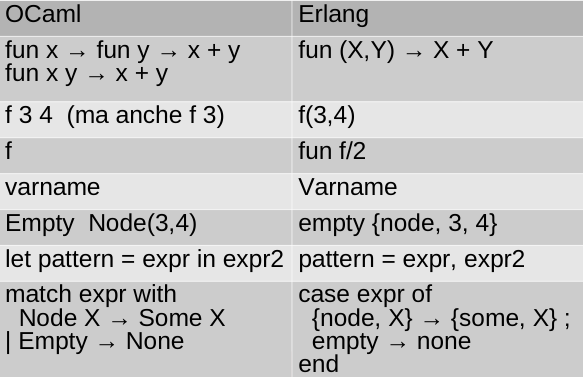
Caml è un’implementazione di ML. Nasce nel INRIA, l’istituto nazionale francese di ricerca in informatica e automazione. Basandosi su una teoria matematica molto solida era diventato molto lento (ribattezzato infatti Caml Heavy). I francesi vennero esclusi da una conferenza e continuarono a lavorare creando prima Caml Light (1995) poi OCaml (1996). Ora Ocaml è l’implementazione più popolare.

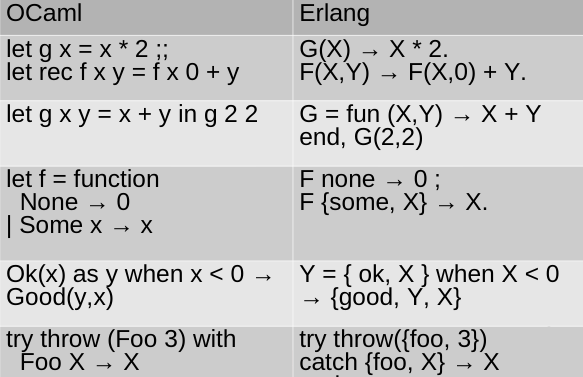
Molti compilatori sono scritti in Ocaml e spesso si mettono feature stesse di Ocaml nei linguaggi che si stanno implementando. Per esempio Rust all’inizio aveva il compilatore in OCaml e ha tutt’ora feature prese da OCaml.

Pro: linguaggio conciso, efficiente, il codice è tendenzialmente molto corretto. Si ha un’ampia scelta di costrutti non standard (varianti polimorfe, oggetti funzionali, funtori, moduli di prima classe, …).

Contro: **non supporta il multi-core** e quindi non è ottimale se si necessita di concorrenza. Ha sofferto per lo sviluppo in ambiente accademico (modello a cattedrale, bazar tardivo e poco fruttuoso) ed è stato superato, sempre in ambiente accademico, da Haskell.

Differenze di sintassi con Erlang





Nota: con fun dichiari funzioni con più argomenti ma senza poter fare pattern matching, con function dichiari funzioni con pattern matching ma con un solo parametro.

Non offrendo concorrenza OCaml è il punto di riferimento per la programmazione funzionale sequenziale.

Gli argomenti di una funzione durante il tipaggio della funzione sono sempre considerati *monomorfi* (tutte funzioni unarie, x -> y -> z non x y -> z).

Il vantaggio delle currying è che facilmente puoi fare applicazione parziale.

Se non si specifica la keyword ‘rec’ nelle funzioni ricorsive la funzione interna al corpo fa riferimento ad un’altra definita esternamente.

OCaml è un linguaggio **fortemente tipato**, quindi cattura staticamente una qualche proprietà indecidibile di un linguaggio approssimandola. La proprietà è che se un linguaggio è ben tipato si possono evitare crash a runtime dati da problemi di tipi. In OCaml essendo fortemente tipato non è presente casting. Mantenere informazioni sul tipo a runtime è costoso, quindi non è stato implementato casting.

**Type inference Hindley-Milner**

- **Non c’è polimorfismo ad-hoc** (ovvero overloading), ad esempio 1 + 2 funziona ma 1.5 + 2.5 non funziona, per i float si usa l’operatore di somma apposito “+.”

- Il **tipo** delle variabili viene **inferito dal** loro **uso**, ad esempio

let f g x = g (g x) + 1 è tipato come val f : (int -> int) -> int -> int

- Il compilatore ritorna un errore di tipo quando le variabili sono usate in maniera incoerente, ad esempio: let f g = g 3, g “ciao”.

- In assenza di vincoli vengono generati schemi di tipi, ovvero i tipi iniziano con **quantificazioni universali** su variabili di tipo indicate con “lettere greche”, ad esempio:

let f g (x, y) = g x, (y, y)

val f : (‘a -> ‘b) -> ‘a \* ‘c -> ‘b \* (‘c \* ‘c)

- I **tipi possono essere ristretti** esplicitamente (l’utente specifica il tipaggio o inline tipo (x : int) o con val f : float -> int).

- Questo **sistema di tipi** è **decidibile** (esiste un algoritmo che determina se un programma è ben tipato), ammette most general unifier (il programma calcola un tipo di cui tutti gli altri sono istanza) ed è iper-conciso.

- **A runtime non si ha** alcuna **rappresentazione dei tipi** (come in C) quindi non c’è nessun overhead ed ha la stessa rappresentazione dei dati di Erlang

Il polimorfismo alla Hindley-Milner è chiamato anche **polimorfismo uniforme** o **generico**.

**Abbreviazione di tipi**

L’idea è dare un nome e un’implementazione ad un tipo, ad esempio:

type currency = int

type ‘a set = ‘a list (il parametro in questo caso va posto a sinistra del nome)

type (‘a, ‘b) mappa = (‘a \* ‘b) list (tipo con due parametri)

L’uguaglianza in OCaml è **strutturale**. Nei linguaggi con uguaglianza nominale due tipi sono uguali solo se hanno solo stesso nome. Nei linguaggi con uguaglianza strutturale i nomi sono semplicemente sinonimi (nell’esempio di prima int e currency sono lo stesso tipo).

Oltre all’abbreviazione di tipo è possibile definire **tipi di dato algebrici**. Un tipo di dato algebrico è un tipo di dato ricorsivo. Questo tipo cattura la stragrande maggioranza delle strutture dati. L’unica struttura dati che non sono nativamente un tipo di dato algebrico sono i grafi, i quali vanno eventualmente codificati in tipo di dato algebrico. Ad esempio:

type (‘a,‘b) tree = Leaf of ‘b | Node of (‘a\*‘b) tree \* ‘a \* (‘a,‘b) tree

tree è un tipo di dato algebrico parametrico su ‘a e ‘b mentre Leaf e Node sono costruttori rispettivamente unario e ternario, in modo simile in Erlang: { leaf, X } e { node, T1, K, T2 }.

I costruttori **non** possono essere applicati parzialmente, ad esempio chiamare solo Node;; darebbe errore in quanto si aspetterebbe 3 argomenti, ma ne vengono forniti 0.

Le tonde in OCaml quasi sempre servono per la precedenza ma quando si dichiara un costruttore indicano un tipo di dato ‘coppia’.

I vantaggi dei tipi di dato algebrici sono che il compilatore controlla che tutti i pattern match siano **esaustivi** e **privi di duplicati**.

Esaustivo: quando si opera su un ADT il compilatore verifica che l’operazione sia svolta per ogni possibile caso del ADT.

Privo di duplicati: quando si opera su un dato ADT il compilatore verifica che l’operazione su un caso del ADT non sia stata definita più volte.

Nel caso in cui i pattern abbiano delle guardie allora l’esaustività significa verificare che tutti i casi vengano coperti ed almeno uno dei casi venga eseguito. Il problema però è indecidibile quindi sono possibili falsi positivi, come ad esempio (che è corretto ma il compilatore lancia un warning):

let f = function x when x < 0 → 0

| x when x >= 0 → 1

Il controllo di esaustività è molto utile in quanto:

- non si ha nessun caso dimenticato per sbaglio (se si evitano pattern catch-all tipo \_ → ...)

- in caso di aggiunta di un costruttore al tipo il compilatore enumera tutti i punti dove occorre aggiornare il codice, e l’editor ci porta automaticamente al punto dove fare la modifica.

Se si pensa a funzioni come metodi e costruttori come classi allora è possibile vedere in cosa differiscano i due approcci (con o senza dati algebrici). In presenza di dati algebrici l’aggiunta di un nuovo metodo/funzione o di una nuova classe/costruttore è immediata.

I tipi di dati algebrici partono da un'assunzione molto forte sulla struttura dati, e quando questa vale il tipo di dato algebrico è la soluzione più corretta al problema. Se l'assunzione è falsa la soluzione corretta è l’approccio a oggetti.

Questa assunzione è la **closed world assumption**:

- tutte le possibili forme sono note (es. protocolli, strutture dati, …)

- i cambiamenti nelle possibili forme sono rari/inaspettati e richiedono ripensamento generale (ad esempio se si cambia un protocollo di rete).

- un’altro vantaggio dei dati algebrici è mantenere corretti gli invarianti (ad esempio in un albero binario di ricerca tutte le chiavi a sinistra del nodo devono essere <= del valore nel nodo)

- mantenere compatte le definizioni di funzione ne agevola la comprensione.

Se l’assunzione non è valida si parla di **open world assumption** e la soluzione migliore è l’approccio ad oggetti. Le assunzioni qui negano quelle viste in precedenza.

CURIOSITÀ: in logica mondo chiuso significa che se non si conosce la verità di un’affermazione allora è considerata falsa (perché tutto ciò che è vero è definito in partenza), in mondo aperto invece non si esclude la veridicità di un’affermazione che non si conosce (perché nuove affermazioni che ne indicano la veridicità potrebbero subentrare in futuro).

**Algebraic Data Types (ADT)**

Introdotti in ML a fine anni 70 e solitamente implementati in tutti i linguaggi funzionali.

Un **ADT** è un’**unione disgiunta** (coprodotto), **taggata** da atomi/costruttori, di **tuple/record** (prodotti) **di tipi** e può essere mutuamente ricorsivo.

In matematica prodotti/coprodotti sono nozioni fondamentali in algebra, teoria delle categorie, ... con ricca teoria matematica. Coprodotti di prodotti sono una tipica forma canonica completa (i prodotti distribuiscono sui coprodotti).

Un’**unione disgiunta** è un’unione insiemistica che considera distinti elementi appartenenti ad insiemi distinti.

Esempio (in Ocaml):

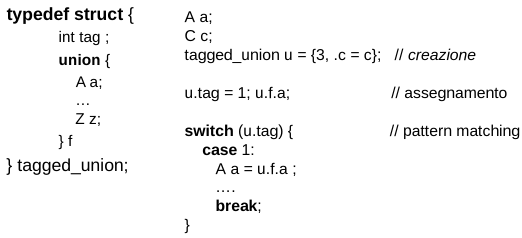
type tagged\_union = Ka of A | ... | Kz of Z

match Ka a with Ka a => “a” | ... | Kz z => “z”

Partiamo dalle teoria degli insiemi. Unione e intersezione. Attenzione che si parla di unione **disgiunta**. Il bello dell’unione disgiunta è che mi porto dietro l’informazione di quale insieme apparteneva un certo elemento. L’informazione è nel primo elemento della coppia.

Tutto questo assomiglia a come salva i dati Erlang. Erlang infatti si ispira alle unioni disgiunte.

**Implementazione di ADT in C**



In C c’è sia il costrutto struct (che alla fine è una tupla) sia l’unione. Di solito l’unione non si usa da sola ma si mette dentro una struct che ha un tag che dice quale dei tipi possibili della union in quel momento la union è. Scrivere .c = c è una versione shorthand per dire che il membro c della union diventa quello attivo.

C’è un errore nell’ assegnamento perchè dovrebbe essere u.f.a = a

Sarebbe meglio chiamare le variabili locali dichiarate per assegnare diverso dalle lettere minuscole della union.

E simile al approccio OCaml sintatticamente ma è error-prone. Bisogna per esempio ricordarsi di tenere allineati il tag e i valori all’interno. Inoltre il costrutto union viene sempre usato insieme a una struct, quindi è un costrutto a basso livello che viene sempre usato per implementare disjoint unions, quindi a posteriori è un errore linguistico se da solo è poco usato.

Se voi implementate Erlang in C, diventa esattamente questa rappresentazione e il tag diventa il numero che avete associato al vostro atomo.

**Implementazione ADT in Erlang**

case { ka, a } of

{ ka, A } => “a” ;

...

{ kz, Z} => “z”

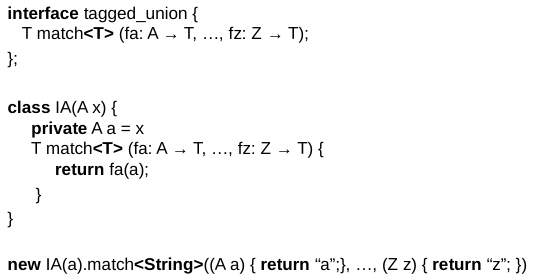
end

Per implementare un ADT in Erlang basta semplicemente creare una coppia composta da atomo-tipo e poi fare pattern matching su di essa per creare un tipo a partire da una certa forma tra tutte quelle possibili elencate.

In memoria un ADT nella forma ad esempio type pippo = Nil | Cons of int \* int list avrà un atomo che è Nil per il caso Nil, e un record {Cons, 4, Nil} ad esempio, per il caso Cons. Gli atomi sono unboxed quindi gli si assegna un intero e e lo si specifica anche nella tabella dei simboli del processo. Invece i record sono boxed.

Non c’è spreco di interi per rappresentare gli atomi perchè l’unicità degli atomi non è globale ma solo locale al ADT, cioè gli interi per rappresentare gli atomi vengono riciclati.

**- Encoding tramite oggetti e chiusure**



L’unico metodo che la classe algebrica deve implementare è il pattern matching (generico su un tipo T). In teoria IA dovrebbe implementare tagged\_union ma il prof si è dimenticato e l’ha ammesso la lezione dopo.

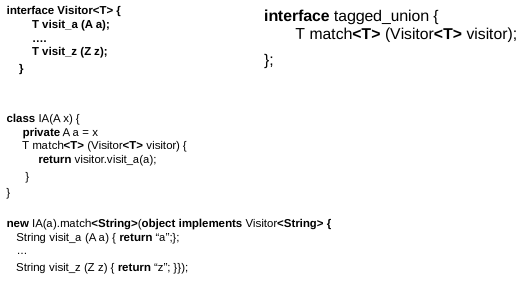
L’idea è definire un’interfaccia che rappresenta qualunque dato. **Quello che caratterizza un ADT è il pattern matching, ovvero analizzare la propria forma e fare qualcosa**. Si può fare pattern matching sullo stesso dato per ottenere qualcosa di diverso (es. una stringa o un booleano). La funzione match della classe dovrà invocare il ramo opportuno, e il ramo opportuno lo sà perchè gli viene passato in input, il codice da eseguire per ciascuno dei casi.

Bisogna fare una classe IA, IB, IC … IZ

Il problema di questa codifica è una restrizione al numero di classi che possono implementare l'interfaccia tagged union, quindi il criterio di assenza di duplicati.

Questo codice è di un ordine di grandezza più lento del codice C che semplicemente fa uno switch ed è più predicibile dal branch prediction della CPU.

**- Encoding tramite oggetti e pattern visitor**

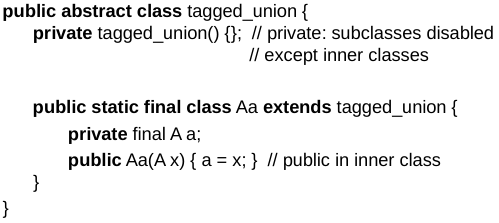


In assenza di chiusure nel linguaggio che stiamo usando è possibile utilizzare il pattern visitor per implementare gli ADT. La differenza è che il metodo match non prende in input le mie chiusure ma prende in input un visitor. Il visitor è un'interfaccia con ognuna delle chiusure come metodo. Questa implementazione è più pesante perché fa riferimento a **runtime** a **late binding**. Viene fatto un controllo su quale implementazione eseguire.

**Sealing**

Il sealing prevede di passare da ‘open world’ a ‘closed world’ in OO. Questo è possibile impedendo la possibilità di aggiungere nuove implementazioni ad un interfaccia.

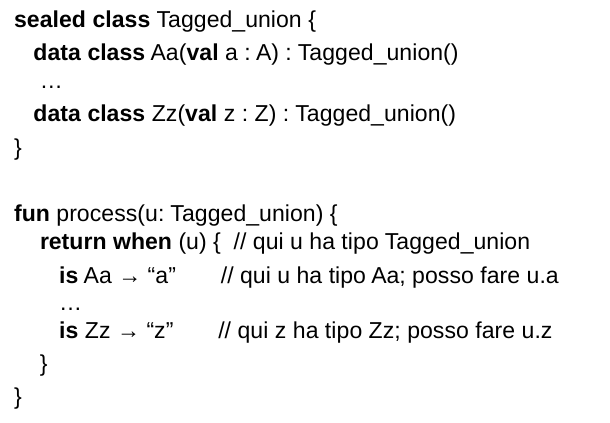
In Java ad esempio:



*taggen\_union* è una classe astratta quindi non è possibile implementarla, mettendo un costruttore privato non si possono mettere sottoclassi che rendano la classe da astratta a concreta. Tuttavia è possibile utilizzare una inner-class interna che estende la classe astratta, può accedere al costruttore in quanto interna e può concretizzare la classe. Trasformando di fatto la classe astratta in un’interfaccia non modificabile dall’esterno ma accessibile tramite la inner-class.

Le classi interne sono ognuna un diverso caso del tipo ADT.

**ADT in Kotlin**

****

L'implementazione ADT di Kotlin consiste nel:

1. Dichiarare una classe sealed (non modificabile) detta tagged\_union, che è composta da n classi, dove n è il numero di forme del ADT preso in considerazione. Ognuna di queste classi è un sottotipo nested della classe tagged\_union ed è una data class che è sostanzialmente una classe che ha solamente dei campi istanziati (come le case class di scala);
2. Un’istanza del tipo tagged union avrà anche una corrispondente istanza di una delle data class nested;
3. Si definisce una funzione che effettua l'analisi per casi con cast impliciti. In pratica questa funzione prende in input una variabile u di tipo tagged union e restituisce un certo valore in base al tipo inferito su u.

Il vantaggio di questa implementazione è che, quando si fa analisi per casi con cast implicito, il compilatore è in grado di conoscere il tipo della variabile nei vari rami del case. Questo porta a due vantaggi:

* Posso utilizzare i metodi dei diversi oggetti dentro i vari rami;
* Evito overhead dovuto a dynamic dispatch.

**Riassumendo**

- gli ADT sono ormai un costrutto mainstream nella maggior parte dei linguaggi, anche se tipicamente molti dei linguaggi basati su classi li hanno codificati sui costrutti di classi il che ha creato overhead e rimane un costrutto meno espressivo.

- prima visione: ADT come **tipi enumerati** che trasportano valori, sintassi concisa, semantica ovvia, implementazione semplice/efficiente.

- seconda visione: ADT come **classi con particolari proprietà**, sintassi e semantiche barocche, implementazioni inefficienti.

**Null values**

In C/Java/… ogni tipo ha un valore di default (0 per int, NULL per pointer, …). Le variabili non inizializzate possono prendere tale valore (o ricevere un valore casuale: dipende dal linguaggio). Accedere al valore non inizializzato è quasi sempre errato, NULL pointer dereferencing è uno dei bug più tipici dei programmi C (ma anche Java, etc.).

Billion dollar mistake: avere aggiunto il tipo di dato null ai linguaggi di programmazione perché la maggior parte dei bug sono generati dal mancato controllo del null.

Nei tipi di dati algebrici è possibile evitare il null (nessuno ha questo tipo di dato) grazie alla presenza di particolari valori option.

**Optional values**

Gli **optional values** sono una forma di ADT.

Gli *aspetti positivi* di questi tipi sono:

- il compilatore forza l’utente a considerare i casi negativi

- cambiare un tipo da/in uno opzionale causerà un messaggio d’errore in cui è necessario effettuare cambiamenti.

- (‘a option) option != ‘a option perchè il left value ha più informazione, più livelli. Per esempio l’option esterno può dire se sono riuscito o no a calcolare un valore, e l’option interno dice se questo valore è nella forma che mi aspettavo o in un altra non significativa in cui non sono interessato .

- gli invarianti diventano espliciti e documentati (per esempio il fatto che un valore NON può essere null)

I *contro* sono:

- a run-time un valore opzionale occupa più spazio (per il tag) e può essere boxed (p.e. in Ocaml/Haskell/...).

- l’accesso al valore opzionale è più lento.

- il codice è più verboso (ma può essere mitigato).

‘a option è un container i valori di tipo ‘a. Le tipiche operazioni standard sui container sono:

val map : (‘a → ‘b) → ‘a option → ‘b option

*Map* applica una funzione ad ogni elemento di un container. Un esempio tipico è la propagazione di un’eccezione.

val iter : (‘a → unit) → ‘a option → unit

*Iter* può restituire o meno un dato.

val bind : (‘a → ‘b option) → ‘a option → ‘b option

La *bind* insieme alla *some* fornisce ad option una struttura monade, una generalizzazione dei container su cui ci aspettiamo di avere strutture dati in modo da implementare funzioni.

val default : ‘a option → ‘a → ‘a

iter si utilizza invece che map quando siamo interessati solo al side effect che la funzione genera.

bind è simile al map, voglio applicare un'operazione solo se un valore c’è, questa operazione però a sua volta può fallire, può darmi un output oppure no. Ritorna sempre option, ma esegue l’operazione sugli elementi in cui dentro l’option ci sia un valore.

**Valori opzionali vs Eccezioni**

Similitudini:

* mettere a null sarebbe lanciare un eccezione
* map/bind possono servire a far *propagare* un eccezione
* default/match a fermarne la propagazione

Ma a differenza delle eccezioni i valori opzionali obbligano la gestione

Però sono più verbosi (non sempre), un po’ di più di memoria ma soprattutto più lenti a runtime, abbiamo visto che le exception sono solo uno frame stack in più e inoltre non hanno slowdown se il problema non esiste ([EAFP](https://docs.python.org/3.6/glossary.html#term-eafp) style ("it's easier to ask for forgiveness than permission") ), invece un ADT va sempre allocato nello heap.

**Nullable References in Kotlin**

Per implementare i valori Nullable però **Kotlin non utilizza gli ADT** perchè per mantenere la compatibilità con java dovrebbe “aprire” ogni volta l’optional e ritornare o null o il valore.

Un tipo nullable in Kotlin è indicato con un punto di domanda alla fine.

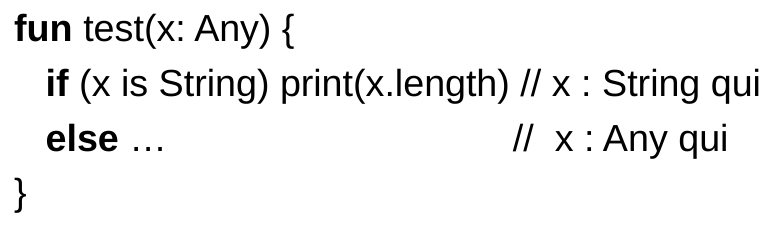
T e T? hanno la stessa rappresentazione a runtime.

Kotlin non permette di avere optional annidati, (per esempio T??) perchè non riusciresti più a distinguere il null esterno da quell interno mentre (‘a option) option e option hanno proprio rappresentazioni a runtime differenti.

Il compilatore dà errore se si utilizza un valore non nullable e non lo si assegna.

Oppure se si legge un valore da un tipo nullable ma non si fa un controllo che non sia null.

Kotlin cambia automaticamente il tipo delle variabili da T? a T a seguito di un controllo di diversità da null.



Kotlin estende questo cast automatico a tutti i tipi, una funzione potrebbe prendere un tipo Any per poi promuoverlo a String dopo un check.

- Se T è un reference type, T? è un nullable reference type

- T e T? hanno la stessa rappresentazione a run-time (no overhead, T?? non ammesso)

- Il compilatore inibisce sia l’uso di T? come valore che l’assegnamento di null a T

- Kotlin promuove un T? in un T se nel codice viene fatto un controllo esplicito che T non sia nullo

- se o : T?, o ?. f restituisce o.f se o != null e null altrimenti (map f o)

- se o : T?, o ?: d restituisce o se o != null e d altrimenti (default o d)

- se o : T?, o!! solleva un’eccezione se o è null, altrimenti ritorna o promosso a T

Casting:

- T y = o as T solleva un’eccezione se o non è un T oppure o è null

- T? y = o as T? solleva un’eccezione se o non è un T

- T? y = o as? T ritorna sempre un T? che è null se o non è un T o è null

- Come nel caso della nullità, il compilatore di Kotlin tiene conto dei test di tipo per promuovere automaticamente gli oggetti, ad esempio dato un x: Any se si controlla x is String x viene convertito a String dal compilatore.

**Confronto tipo option/maybe e nullable in Kotlin**

Se un linguaggio ha gli ADT li utilizza per rappresentare l’assenza o no di un dato. Ocaml/Rust/Haskell fanno così. I Nullable in Kotlin non usano ADT e hanno la stessa rappresentazione in memoria del valore non nullable. Kotlin quindi spreca meno memoria ma non permette di avere nullable annidati. Gli Option/Maybe sono leggermente più lenti ma più espressivi.

**Riassumendo**

- gli ADT option permettono di rappresentare altro oltre alla nullità dei puntatori, ma sono pesanti quando usati a tale scopo.

- le null references di Kotlin sono linguisticamente molto vicine all’ADT option, ma con un uso specializzato e non introducono alcun overhead a run-time.

- Kotlin traccia le condizioni per prevenire casts e null-pointer checks in una maniera decisamente interessante.