Tipi Polimorfi

Tipi Monomorfici e Polimorfici

In generale, se un sistema di tipi non ci permette di esprimere che alcuni tipi accettano altri tipi come parametri, chiamiamo il sistema di tipi **monomorfico**.

Ad esempio, in un sistema di tipi monomorfico non possiamo definire una funzione “generica" per trovare il massimo tra due elementi, indipendentemente dal loro tipo, e dovremmo definire funzioni specifiche per ogni istanziazione di un tipo di dati comparabile, ad es,

`max_int: int->int->int`, `max_float: float->float->float`, etc.

anche se nell'implementazione potremmo probabilmente astrarre dal sapere che stiamo gestendo interi e float, purché possiamo confrontare i loro abitanti.

Polimorfismo di Tipo

Riassumendo, un sistema di tipi polimorfi consente agli sviluppatori di specificare un insieme di operazioni su un dato tipo parametrico che **astrae da alcuni dettagli dell'istanziazione concreta del tipo**.

Sebbene il concetto di polimorfismo nei tipi abbia una interpretazione generalmente condivisa, il *modo* in cui i linguaggi di programmazione lo interpretano e lo implementano varia.

In generale, esistono tre tipi di polimorfismo:

- **ad-hoc** (**sovraccarico/overloading**), in cui si sovraccarica la definizione di una data operazione su diversi tipi specifici;
- **di sottotipo** (**subtyping**), in cui si stabiliscono relazioni da-astratto-a-specifico tra i tipi e si ottiene un polimorfismo con operazioni su tipi astratti;
- **parametrico** (**universale**), in cui abbiamo simboli astratti che rappresentano parametri di tipo.

Polimorfismo Ad-hoc (Sovraccarico/Overloading)

Come suggeriscono i termini **ad-hoc** e **sovraccarico**/**overloading**, questo tipo di polimorfismo sfrutta la capacità del compilatore/runtime del linguaggio di distinguere dal contesto di chiamata definizioni alternative di operazioni con lo stesso nome.

L'esempio più comune di polimorfismo ad-hoc sono gli operatori aritmetici. Ad esempio, `+`, nella maggior parte dei linguaggi è sovraccaricato (su numeri, come `int` e `float`, ma anche su caratteri e stringhe).

A seconda del linguaggio e del contesto del sito di chiamata, il compilatore o il runtime devono disporre di informazioni sufficienti sui tipi per sapere quale implementazione di una determinata operazione sovraccaricata il programma deve utilizzare in un determinato punto di invocazione.

Polimorfismo Ad-hoc (Sovraccarico/Overloading)

L'overloading è una sorta di abbreviazione sintattica che scompare non appena risolviamo l'invocazione, rendendo l'overloading un meccanismo di **indirizzamento** (**dispatch**).

Quando il dispatch avviene **staticamente** (per esempio, in fase di compilazione), sostituiamo ogni simbolo sovraccaricato con un nome non ambiguo che denota in modo univoco la specifica implementazione.

Quando l'indirizzamento avviene in fase di esecuzione, abbiamo un indirizzamento **dinamico**, che di solito avviene tramite tabelle di ricerca.

In particolare, l'overloading e la coercizione dei tipi vanno spesso di pari passo, ad esempio (a seconda del linguaggio) possiamo scrivere `1 + 2.0` che, ad esempio, "coercizza" `1` in `1.0` e utilizza la definizione di `+` per i `float` (che restituisce un `float`, ad esempio `3.0`).

Polimorfismo Ad-hoc (Sovraccarico/Overloading)

Dal nostro esempio sulla funzione generica "max", la soluzione polimorfica ad-hoc potrebbe essere (scritta in Java — C e Rust non supportano l'overloading)

```
int max( int i, int j ){
 return i > j ? i : j;
}

float max( float i, float j ){
 return i > j ? i : j;
}
```

Polimorfismo di Sottotipo (Subtyping o di inclusione)

Il polimorfismo dei sottotipi si basa su una relazione binaria $<:$ sui tipi in cui $S <: T$ si legge "S è un sottotipo di T". Il significato della relazione è solitamente quello di **specificazione**, cioè se $S <: T$ allora S è un tipo più specifico di T e possiamo usare tranquillamente S in qualsiasi posto che necessiti un T (come postulato dal *principio di sostituzione di Liskov* [1]).

Un tipico esempio di polimorfismo dei sottotipi è il tipo `Animale` e i suoi sottotipi `Gatto` e `Cane`. Possiamo usare i valori dei tipi `Gatto` e `Cane` finché abbiamo bisogno di un `Animale` (ad esempio, entrambi possono respirare e dormire).

[1] Liskov, Barbara H., and Jeannette M. Wing. "**A behavioral notion of subtyping**." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.6 (1994): 1811-1841.

Polimorfismo di Sottotipo (Subtyping o di inclusione)

Di solito, $<:$ è un **preordine** ($T <: T$ e $S <: T \land R <: S \implies R <: T$) e, come tale, non possiamo usare `Animale` dove assumiamo di avere un `Gatto` o un `Cane`, ad esempio, non possiamo supporre che tutti i valori di tipo `Animale` abbiano l'operazione `abbaiare`, come farebbe un `Cane`.

Concludendo, $<:$ è di solito anche antisimmetrico, il che lo rende un **preordine parziale**, cioè vale anche che $T <: S \land S <: T \implies T = S$.

Il polimorfismo dei sottotipi si trova tipicamente nei linguaggi orientati agli oggetti, data la sua vicinanza al concetto di ereditarietà (lo vedremo quando parleremo di OO).

[1] Liskov, Barbara H., and Jeannette M. Wing. "**A behavioral notion of subtyping**." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.6 (1994): 1811-1841.

Polimorfismo di Sottotipo, sottotipaggio dei record

Intuitivamente, possiamo considerare $S <: T$ come una relazione tra un tipo $Target$ che il contesto assume e un tipo $Sostituto$ che vogliamo usare al posto di T. Consideriamo i tipi record

```
type Animal:{name:string}

type Dog:{name:string,bark:string}
```

dove, per <:, `Dog <: Animal`.

```
type AnimalHouse:{tenant: Animal}

type DogHouse:{tenant:Dog}
```

`Animal` e `Dog` sono un esempio di **sottotipaggio in larghezza**, in cui i record dei sottotipi aggiungono più campi dei loro supertipi.

`AnimalHouse` e `DogHouse` esemplificano il **sottotipaggio in profondità**, in cui sostituiamo i campi con i loro sottotipi.

Polimorfismo di Sottotipo, sottotipaggio dei record

Intuitivamente, possiamo considerare $S <: T$ come una relazione tra un tipo $Target$ che il contesto assume e un tipo $Sostituto$ che vogliamo usare al posto di T. Consideriamo i tipi record

```
type Animal:{name:string}

type Dog:{name:string,bark:string}
```

dove, per <:, `Dog <: Animal`.

```
type AnimalHouse:{tenant: Animal}

type DogHouse:{tenant:Dog}
```

Ad esempio, se abbiamo una `AnimalHouse` ed estraiamo `tenant`, `DogHouse` e `AnimalHouse` sono **covarianti** rispetto a `Animal` e `Dog`, perché mantengono la stessa direzione di sottotipaggio dei tipi di riferimenti, cioè

Dog <: Animal

DogHouse <: AnimalHouse

Polimorfismo di Sottotipo, sottotipaggio dei record

Intuitivamente, possiamo considerare $S <: T$ come una relazione tra un tipo $Target$ che il contesto assume e un tipo $Sostituto$ che vogliamo usare al posto di T. Consideriamo i tipi record

```
type Animal:{name:string}

type Dog:{name:string,bark:string}
```

dove, per <:, `Dog <: Animal`.

```
type AnimalHouse:{tenant: Animal}

type DogHouse:{tenant:Dog}
```

⚠ dato che sono sottotipi in profondità possiamo **usare** DogHouse e AnimalHouse **in maniera covariante solo per le letture**, e.g., possiamo usare `DogHouse` al posto di `AnimalHouse` se dal valore dato ne estraiamo (leggiamo) il `tenant`—che ci aspettiamo essere un Animal, per cui avere un Dog non pone problemi.

Polimorfismo di Sottotipo, sottotipaggio dei record

Intuitivamente, possiamo considerare $S <: T$ come una relazione tra un tipo $Target$ che il contesto assume e un tipo $Sostituto$ che vogliamo usare al posto di T. Consideriamo i tipi record

```
type Animal:{name:string}

type Dog:{name:string,bark:string}
```

dove, per <:, `Dog <: Animal`.

```
type AnimalHouse:{tenant: Animal}

type DogHouse:{tenant:Dog}
```

⚠️ Al contrario, dobbiamo **usare** `DogHouse` e `AnimalHouse` **in maniera controvariante** (cioè in opposizione al verso del sottotipaggio dei tipi di riferimento) **per le scritture**, e.g., possiamo dare `AnimalHouse` al posto di `DogHouse` se nel valore dato vogliamo metterci (scriverci) un `Dog` come `tenant`—`AnimalHouse` vuole un `Animal`, per cui non è un problema dargli un sottotipo.

Polimorfismo di Sottotipo, covarianza and controvarianza

Intuitivamente, possiamo considerare $S <: T$ come una relazione tra un tipo $Target$ che il contesto assume e un tipo $Sostituto$ che vogliamo usare al posto di T. Consideriamo i tipi record

```
type Animal:{name:string}

type Dog:{name:string,bark:string}
```

dove, per <:, `Dog <: Animal`.

```
type A2B: Animal -> Bool

type D2B: Dog -> Bool
```

Un fenomeno simile a quello dei sottotipi di profondità, in questo caso **legato a consumo** (input) vs **produzione** (output) si verifica con i **tipi di funzione:**

Consumo -> Controvariante

Produzione -> Covariante

Polimorfismo di Sottotipo, covarianza and controvarianza

Intuitivamente, possiamo considerare $S <: T$ come una relazione tra un tipo $Target$ che il contesto assume e un tipo $Sostituto$ che vogliamo usare al posto di T. Consideriamo i tipi record

```
type Animal:{name:string}

type Dog:{name:string,bark:string}
```

dove, per <:, `Dog <: Animal`.

```
type A2B: Animal -> Bool

type D2B: Dog -> Bool
```

I **tipi funzione in consumo** (cioè quando il **contesto** di aspetta di avere una funzione a cui **dare un input**) **sono controvarianti** rispetto ai tipi di riferimento.

Ad esempio, `A2B` <: `D2B` (stesso output, `Bool`) sono in relazione **controvariante** rispetto a `Dog` <: `Animal` perché possiamo sostituire `D2B`, a cui il contesto si aspetta di fornire `Dog`, con `A2B`, che utilizza meno informazioni.

Polimorfismo di Sottotipo, covarianza and controvarianza

Intuitivamente, possiamo considerare $S <: T$ come una relazione tra un tipo $Target$ che il contesto assume e un tipo $Sostituto$ che vogliamo usare al posto di T. Consideriamo i tipi record

```
type Animal:{name:string}

type Dog:{name:string,bark:string}
```

dove, per <:, `Dog <: Animal`.

```
type U2A: Unit -> Animal

type U2D: Unit -> Dog
```

I **tipi funzione in produzione** (cioè quando il **contesto** si aspetta una funzione da cui **avere un** valore in **output**) **sono covarianti** rispetto ai tipi di riferimento.

Ad esempio, `U2D <: U2A` (con lo stesso input `Unit`) hanno una relazione **covariante** rispetto a `Dog <: Animal` perché possiamo sostituire U2D, che produrrà un Dog, a U2A, da cui il contesto si aspetta di ricevere un `Animal`.

Polimorfismo di Sottotipo, covarianza and controvarianza

Intuitivamente, possiamo considerare $S <: T$ come una relazione tra un tipo $Target$ che il contesto assume e un tipo $Sostituto$ che vogliamo usare al posto di T.

Un esempio un po' più elaborato sui tipo record

```
type EliteDog: { name: string, bark: string, pedigree: string }
```

dove, per <:, `EliteDog` <: `Dog` <: `Animal`.

Si possono quindi definire due funzioni D2D e A2E

```
type D2D: Dog -> Dog

type A2E: Animal -> EliteDog
```

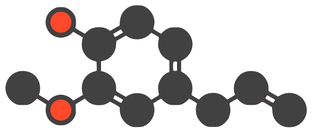
`A2E` <: `D2D`, questo esemplifica ulteriormente la regola dei tipi funzionali vista in precedenza: **i tipi degli argomenti sono controvarianti** (A2• <: D2• ma Dog <: Animal) e **i tipi di ritorno sono covarianti** (•2E <: •2D e EliteDog <: Dog).

Polimorfismo di Sottotipo, covarianza and controvarianza

Controvarianza

Il super-tipo sostituisce il sottotipo

Carburante :> Legno :> Bamboo



E.g., se il contesto produce Legno, possiamo usare il consumatore di Carburante

Consumatori

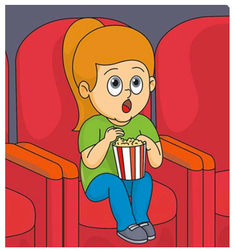


Consumatore di Carburante <: Consumatore di Legno <: Consumatore di Bamboo

Covarianza

Il sottotipo sostituisce il super-tipo

Intrattenimento :> Musica :> Metal



E.g., se il contesto consuma Musica, possiamo usare il produttore di Metal

Produttori



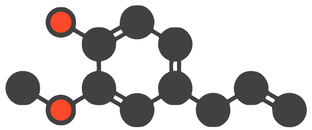
Produttore di Intrattenimento :> Produttore di Musica :> Produttore di Metal

Polimorfismo di Sottotipo, covarianza and controvarianza

Controvarianza

Il super-tipo sostituisce il sottotipo

Carburante :> Legno :> Bamboo



E.g., se il contesto produce Legno, possiamo usare il consumatore di Carburante

Consumatori

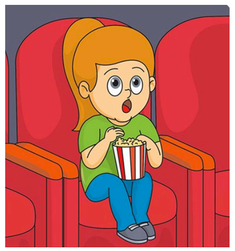


Carbur. -> Unit Legno -> Unit Bamboo -> Unit

Covarianza

Il sottotipo sostituisce il super-tipo

Intrattenimento :> Musica :> Metal



E.g., se il contesto consuma Musica, possiamo usare il produttore di Metal

Produttori



Unit -> Intrat. Unit -> Musica Unit -> Metal

Polimorfismo di Sottotipo, sussunzione (subsumption)

L'atto di decidere se $S <: T$ si chiama **sussunzione**.

Esistono principalmente due strategie per definire la sussunzione (che dipendono dai due modi in cui possiamo definire l'appartenenza ad un tipo): **estensionale** o **intensionale**.

Per estensione, si ha che $S <: T$ e $\forall s \in S, [\![s]\!] \in T$.

Intensionalmente, se $S <: T$, allora il predicato che definisce l'appartenenza a T 1) deve far parte del predicato di S e 2) deve essere applicato sullo stesso dominio di quello per S.

In molti casi, i sistemi di tipi definiscono relazioni di sottotipaggio specifiche per i tipi di base, ad esempio `int` <: `float` o `char` <: `string`.

Polimorfismo di Sottotipo

Dal nostro esempio sulla funzione generica "max", la soluzione con polimorfismo di sottotipo potrebbe essere scritta

```
Integer <: Float

max( Float i, Float j ) -> Float { … }
```

Utilizzando questa soluzione, perdiamo informazione sui valori dei sottotipi in ingresso e possiamo restituire con sicurezza solo un valore del **super-tipo** (`Float`) e non quello specifico (che è noto solo al momento della chiamata di max).

Questo può causare problemi, ad esempio, a causa della necessità di forzare un casting del valore restituito da `max` per essere utilizzabile in un determinato contesto (e.g., troviamo il massimo tra due numeri interi e inseriamo il primo in una funzione che accetta numeri interi).

Polimorfismo di Sottotipo

Similarmente

```
Integer <: Comparable

Float <: Comparable

max( Comparable i, Comparable j ) -> Comparable { … }
```

Utilizzando questa soluzione, si perdono le informazioni sui valori dei "comparabili" che abbiamo come input e possiamo restituire con sicurezza solo un valore del super-tipo (`Comparable`) e non quello specifico (che è noto solo al momento della chiamata di max), con gli stessi possibili problemi della soluzione precedente.

Tipi Parametrici

Alcune strutture di dati hanno delle invarianti che ci permettono di fornire type safety anche se non conosciamo completamente la loro forma.

Un esempio che abbiamo già visto di queste strutture dati è il tipo `Set`, per il quale abbiamo detto che di solito vengono definite alcune operazioni tipiche come l'unione, l'intersezione, la sottrazione e il test di inclusione.

Un'osservazione che possiamo fare è che, per un implementatore della struttura dati `Set`, le operazioni sugli insiemi rimangono le stesse sia che una data istanza di `Set` gestisca interi, caratteri o array. Più precisamente, **le operazioni** degli insiemi **sono parametriche** agli elementi dell'insieme, ad esempio (semplificando) se dobbiamo verificare l'inclusione, applicheremo la definizione di equivalenza degli interi se abbiamo un insieme di interi e lo stesso vale per i caratteri, gli array, ecc.

In questo caso, diciamo che `Set` è un **tipo parametrico** e, quando viene istanziato con, ad esempio, gli interi, quest'ultimo è il suo **parametro** (attuale) **di tipo**.

Polimorfismo Parametrico (Universale)

L'estensione di un sistema di tipi con tipi parametrici introduce la possibilità del **polimorfismo parametrico**.

In generale, quando si usa il polimorfismo parametrico, non si possono fare ipotesi sulla forma dei parametri del tipo, il che essenzialmente costringe noi (e il sistema di tipi) a considerare tutti i tipi possibili nelle implementazioni.

Questo è il motivo per cui il polimorfismo parametrico è chiamato anche polimorfismo **universale**, perché leggiamo la dichiarazione dei parametri di tipo Type(T) come $\forall$ T . Type(T).

Theorems for free, un esempio

A titolo di esempio, si consideri

```
r( T ): List( T ) -> List( T ),
f: A -> B,
map( T, S ): List( T ) -> (T -> S) -> List( S )
```

quindi, sia `l: List( A )`

```
map( r( l ), f ) = r( map( l, f ) )
```

Intuitivamente, poiché `r` ignora la forma di `T` (che può essere qualsiasi tipo, i.e., ∀`T`), `r` può lavorare solo sul tipo non parametrico `List`. Quindi, tutto ciò che `r` può fare è riorganizzare la lista (per esempio, rimuovere elementi in base al loro indice, duplicarli, ecc.) indipendentemente dai loro valori attuali. Questo ci permettere di scambiare tranquillamente (per quanto i tipi possono "vedere") l'applicazione di `f` a ciascun elemento della lista riordinata (a sinistra di =) con l'applicazione di `f` a ciascuno dei suoi elementi, seguita dall'ordinamento (a destra di =).

[1] Wadler, Philip. "**Theorems for free!**." Proceedings of the fourth international conference on Functional programming languages and computer architecture. 1989.

Polimorfismo Parametrico Ibrido/di Sottotipo

Mentre il **polimorfismo parametrico non presuppone alcuna conoscenza dei parametri del tipo** (e potrebbe fornire utili proprietà su questa astrazione [1]), alcuni linguaggi con polimorfismo parametrico forniscono un **operatore di introspezione** del tipo (ad esempio, `instanceof` in Java) che consente all'utente di verificare se un valore appartiene a un determinato tipo, a **scapito delle proprietà di astrazione.**

Il polimorfismo parametrico cattura l'universalità delle espressioni di tipo, ma a volte è utile **esprimere dei vincoli sul quantificatore universale**, ad esempio per limitare la quantificazione solo ai tipi con determinate proprietà.

Un modo comune di esprimere questi vincoli è quello di mescolare polimorfismo parametrico e di sottotipo, come avviene in Java e Rust.

Per esempio, una versione polimorfica della nostra funzione `max` può avere il tipo

```
∀ T, T <: Comparable, max: T -> T -> T
```

[1] Wadler, Philip. "**Theorems for free!**." Proceedings of the fourth international conference on Functional programming languages and computer architecture. 1989.

Polimorfismo Parametrico (Universale)

Per poter esprimere i parametri nei tipi, dobbiamo introdurre una nuova notazione che renda esplicito quando i tipi accettano parametri.

Ad esempio, Java e Rust supportano i tipi parametrici tramite **generici** e **traits**/**tratti**. Possiamo scrivere una versione parametrica di `Set` come `Set<T>`, dove `Set` è un tipo polimorfico che accetta un **parametro di tipo** (formale), qui catturato con la **variabile di tipo** `T`. Analogamente, possiamo definire `max` come (Java e Rust)

```
<T extends Comparable> T max (T x, T y){…}
```

```
fn max<T: Comparable>(x:T, y:T)->T{…}
```

dove 1) i due argomenti sono dello stesso tipo (`T`), 2) il loro tipo è sottotipo di `Comparable` e 3) il valore di ritorno è dello stesso tipo di quelli in input.

Polimorfismo Parametrico (Universale)

Il polimorfismo parametrico introduce la nozione di tipi universali—come abbreviazione di tipi **quantificati universalmente**—dove definizioni come

```
<T extends Comparable> T max (T x, T y){…}
```

e

```
fn max<T: Comparable>(x:T, y:T)->T{…}
```

vedono `max` con tipo $\forall T . T <:$ Comparable $. T \times T -> T$ dove l'universale $\forall$ indica che la definizione è valida per (e parametrica rispetto a) qualsiasi tipo T (fintanto che è un sottotipo di Comparable).

Polimorfismo Parametrico e di Sottotipo

La combinazione di polimorfismo parametrico e sottotipaggio introduce la nozione di sottotipaggio dei tipi parametrici, ad es,

se Dog <: Animal allora Set<Dog> <: Set<Animal>?

La questione è simile a quella del sottotipaggio di profondità per i record, dove è sicuro considerare Set<Dog> <: Set<Animal>, purché non si eseguano scritture/ inserimenti nell'insieme.

Questo è vero anche con il sottotipaggio in larghezza dei tipi parametrici, ad es,

```
type Set<T>: {add:T->(),remove:T->(),includes:T->Bool}

type List<T>:{add:T->(),remove:T->(),includes:T->Bool,get:int->T}
```

dove `List<T> <: Set<S>` dipendentemente dall'uso del tipo parametrico e la relazione dei parametri di tipo T ed S.

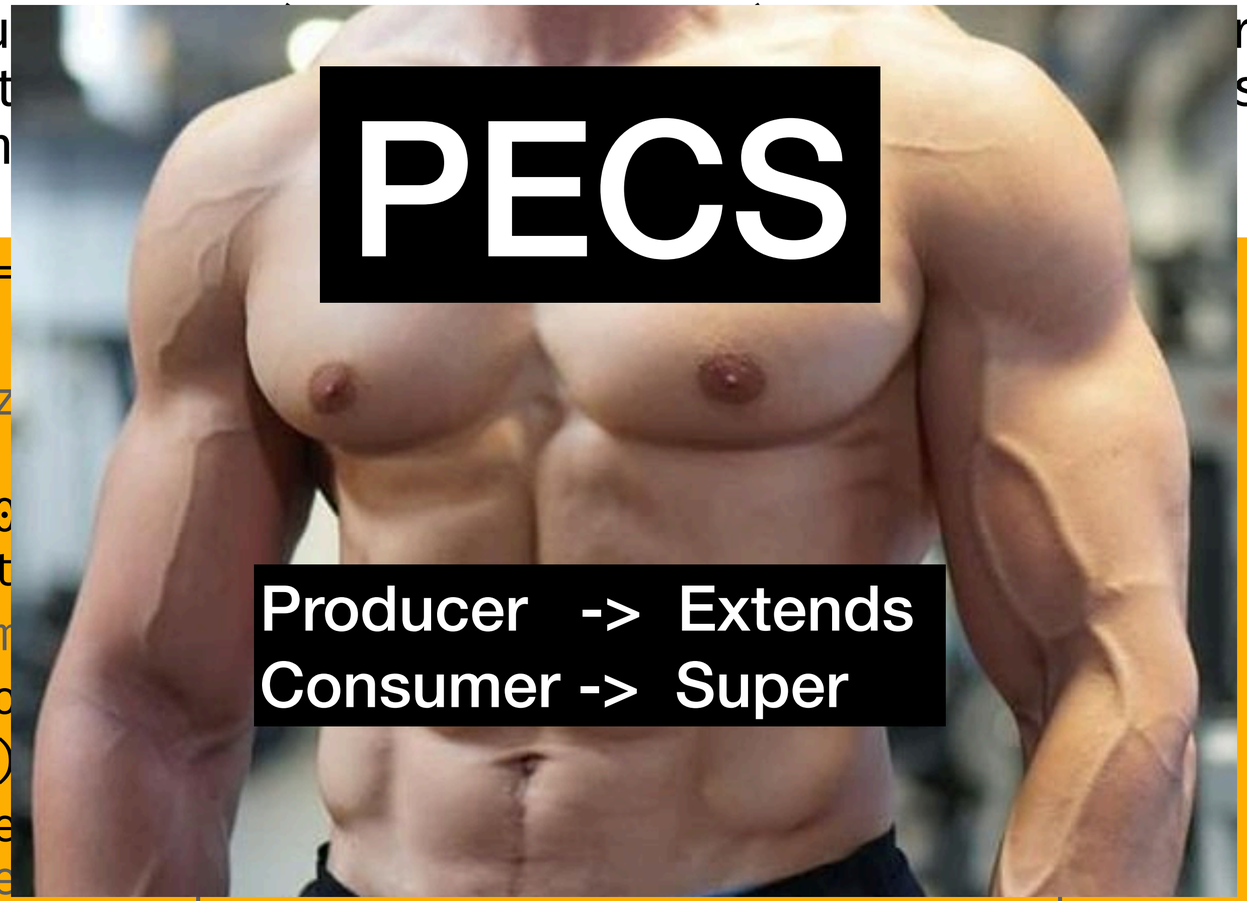
Polimorfismo Parametrico e di Sottotipo

Per questo, i linguaggi con (qualche forma di) polimorfismo parametrico di sottotipo permettono di annotare esplicitamente la "direzione" che l'utente si aspetta per l'uso dei tipi parametrici, ad esempio

```
Vector< Dog > dv = new Vector< Dog >();
dv.add( new Dog() );
// usato in produzione, i.e., int -> Dog, covariante
Vector< ? extends Dog > cov = dv;
Dog d = cov.get( 0 ); // Dog <: Dog
Animal a = cov.get( 0 ); // Dog <: Animal
// usato in consumo, i.e., Dog -> Unit, controvariante
Vector< ? super Dog > con = dv;
con.add( new Dog() ); // Dog <: Dog
con.add( new EliteDog() ); // EliteDog <: Dog
// sotto, dato che non specifichiamo il suo uso nel tipo
// il type-check assume "invarianza" e riporta un errore di tipo
Vector< Animal > inv = dv;  ❌
```

Polimorfismo Parametrico e di Sottotipo

Per questo, i lingu rametrico di sottotipo permettono di annot spetta per l'uso dei tipi parametrici, ad esem




```
Vector< Dog > dv =
dv.add( new Dog()
// usato in produz
Vector< ? extends
Dog d = cov.get( 0
Animal a = cov.get
// usato in consum
Vector< ? super Do
con.add( new Dog()
con.add( new Elite
// sotto, dato che
// il type-check assume "invarianza" e riporta un errore di tipo
Vector< Animal > inv = dv;  ❌
```

Lo strano caso degli array in C++/Java

In Java e C++ il tipo **array** è un tipo parametrico polimorfico... sotto mentite spoglie. Entrambi i linguaggi non avevano supporto per i tipi polimorfi, e gran parte del codice preesistente utilizzava array non polimorfi. Per rendere utili gli array, senza polimorfismo, gli implementatori hanno deciso di rendere covarianti sia scritture che letture. Pertanto, il sistema dei tipi non rifiuta

```
Animal[] aa = new Dog[ 1 ];

aa[ 0 ] = new Dog(); // in consumo, usato come Array< ? super Dog >

Animal a = aa[ 0 ]; // in produzione, usato come Array< ? extends Dog >

aa[ 0 ] = new Animal(); // usato come Array< Animal > ❌
```

Costosi

uguale se avessimo usato `new Cat()` con Cat <: Animal

... ma rompe i controlli di runtime (per la controvarianza dell'aggiornamento dell'array — ad esempio, accetta correttamente EliteDog, ma non Animal) e produce una ArrayStoreException.

Tipi monadici: Tipi Opzione e Risultato

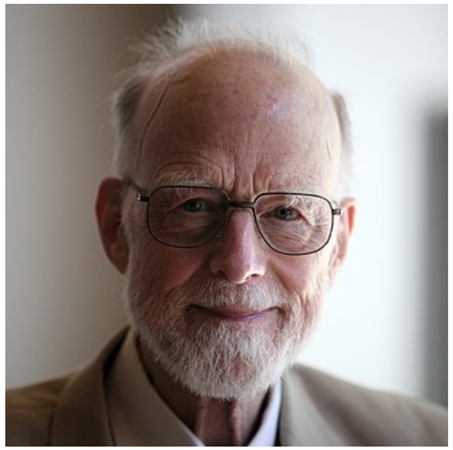
Le monadi, proposte da Eugenio Moggi per rappresentare le computazioni nei linguaggi di programmazione, sono un'astrazione utilizzata principalmente nei linguaggi funzionali per semplificare la composizione e la risoluzione (gestione del flusso di controllo, effetti collaterali, ecc.) di catene di funzioni.

Qui, consideriamo le monadi in modo semplicistico e le vediamo come "contenitori" che incapsulano alcune funzionalità e ci concentriamo sui **tipi monadici** che sono diventati sempre più comuni nei linguaggi di programmazione tradizionali: **Opzione** e **Risultato**.

Tipi Opzione e Maybe

Il tipo Opzione (chiamato Option o Maybe) è utile per gestire in maniera strutturata (e dispensare un linguaggio dalla presenza di) puntatori nulli.

Ricordando i tipi di puntatore, abbiamo notato l'esistenza, in alcuni sistemi di tipi, di un abitante speciale del tipo, chiamato `null`, che indica che il puntatore non riferisce una posizione di memoria valida (inizializzata/utilizzabile). L'inventore di `null`, Tony Hoare, vincitore del premio Turing, lo ha definito il suo "errore da un miliardo di dollari".



È noto tra l'altro per **quicksort**, la **logica di Hoare**, il **problema dei filosofi** e del **linguaggio dei processi sequenziali comunicanti** (CSP).

> [...] in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to **innumerable errors**, **vulnerabilities**, and **system crashes**, which have probably caused a billion dollars of pain and damage in the last forty years.

Tipi Opzione e Maybe

I tipi Option/Maybe mitigano (se non eliminano) il problema dei puntatori nulli presentando, a livello di tipo, la dualità tra puntatori validi e non validi,. Per fare ciò, mescolano tipi parametrici e tipi somma.

Esistono diverse interpretazioni equivalenti dei tipi Option/Maybe, ad es.

```
type Maybe<T> : Some<T> + None
```

dove un valore di tipo Maybe<T> è una capsula attorno a un valore di tipo T (Some<T>) o il valore singleton di tipo None (equivalente a Unit).

Java ha introdotto il tipo Optional nella versione 8. Optional fornisce un'interfaccia dedicata, di ispirazione funzionale, ad es,

```
Optional< Integer > opt = Option.of( 42 );
var d = 2 * opt.orElseGet( () -> 0 );
```

Rust utilizza gli enum per codificare i tipi di Opzione e si affida al pattern matching per gestire i casi

```
let opt: Option< i32 > = Some( 42 );
let d = match opt {
 Some( x ) => 2*x,
 None => 0        }
```

Tipi Risultato

Possiamo pensare ai tipi Risultato (**Result**) come ad un raffinamento dei tipi Maybe/Option, dove usiamo tipi polimorfi e somma per **distinguere tra il risultato di un calcolo e un'esecuzione errata**, segnalata/definita da un errore.

Una possibile implementazione di questa idea è

```
type Result< T, E >  : Ok< T > + Err< E >
```

In pratica, i tipi di risultato rappresentano un'**alternativa alla gestione delle eccezioni** (e al ragionamento sul comportamento dinamico ad esse associato) che permette una gestione più lineare degli stati del programma, dal momento che il programmatore deve (finché il sistema di tipi lo costringe) o scartare completamente il risultato (ad esempio, perché non gli interessa la sua esecuzione) o considerare tutti i suoi possibili stati.

```
let res: [ Result< i32, &str >; 3 ] =
 [Ok(40), Err("Error"), Ok(2)];
let mut acc: i32 = 0;
for r in &res {
 acc += match r {
  Ok( _r ) => _r,
  Err( _ ) => 0
 }
}
```