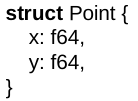
**Emerging Programming Paradigms**

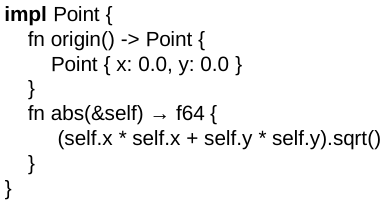
[ lezione 11 maggio ]

**Rust**

Esempio di codice Rust:

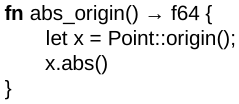


Si definisce una struttura Point con due campi floating point.



Con la keyword **impl** si indica un blocco di codice che implementa i metodi della struttura Point. Non è una classe, si possono dichiarare metodi su qualunque tipo di dato e i dati sono indicati separatamente (in struct ad esempio).

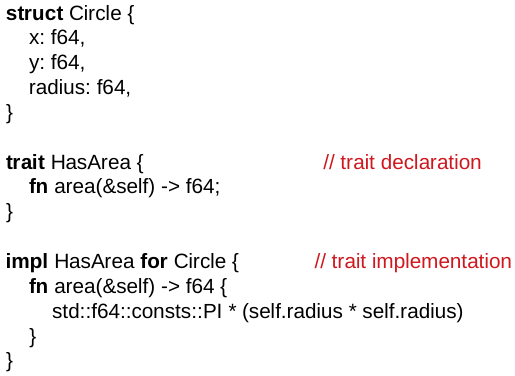
La funzione abs ad esempio prende in input una reference, chiamata self; la sintassi è simile a Go, si indica prima il dato e poi il metodo da invocarvi sopra.



Con la keyword **fn** si indica la dichiarazione di una funzione, con let si dichiara un nuovo nome non mutabile, in questo caso x è il risultato della chiamata origin dentro all’implementazione Point.

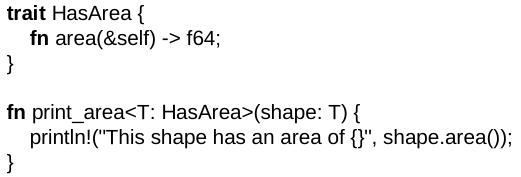
**Traits**

Un *trait*, come nel caso delle interfacce di Go è una lista di dichiarazioni di metodi.



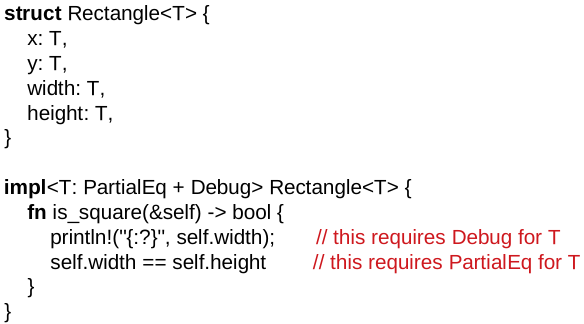
NOTA: self usato come tipo indica proprio il tipo per cui si sta implementando il trait (in questo caso Circle).

I trait sono usati come bound del polimorfismo generico.



In questo caso si astrae su tipo, ovvero si indica che la print\_area funziona su un qualunque tipo T che implementa tutti i metodi del trait HasArea. Questo è esattamente il **polimorfismo di riga**, ovvero “per ogni tipo T che abbia almeno i campi indicati in HasArea”.

Go invece non ha strutture polimorfe quindi non sarebbe possibile usare il “per ogni”.



In questo esempio si ha una struttura Rectangle con tipo parametrico T, nell’implementazione si implementa un metodo is\_square che fa uso nell’uguaglianza tra due rettangoli e della print. Quindi questo tipo Rectangle non può essere usato per ogni tipo T ma solo sui tipi T che abbiano implementato almeno PartialEq e Debug.

Una limitazione è che per implementare il trait T per un tipo U è necessario che almeno uno dei due sia stato definito nel file corrente (utile per i tool di autocompletion).

I traits in Rust possono definire implementazioni di default, le quali possono essere ridefinite quando si implementa il trait.

NOTA: i trait non hanno campi, possono avere solo dichiarazione di codice e implementazioni di default.

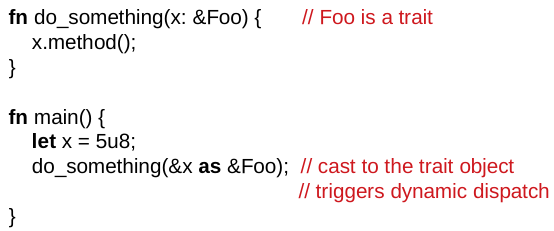
Le implementazioni di default fanno si che sia presente il *problema del diamante*, in quanto nel momento in cui si indica di implementare da due traits diversi e questi hanno entrambi del codice di default per un metodo ma diverso tra loro, il compilatore dovrà decidere quale dei due andare ad utilizzare.

Il Rust il problema del diamante si risolve come descritto alla fine di [questa pagina](https://www.bookstack.cn/read/cpp-to-rust-book/fixing_problems_in_c_cpp-multiple_inheritance.md). Cioè se una struttura implementa i metodi dei trait B e C, e questi due trait ereditano dal trait A, allora la struttura sarà obbligata ad implementare anche A.

I traits di Rust, così come le interfacce di Go, hanno un meccanismo di inheritance.

Tutte le chiamate di metodo sono statiche, a meno che non si dichiarino variabili di tipo trait. Il polimorfismo di Rust, con in C++, utilizza monomorfizzazione. Quindi quando si scrive un metodo polimorfo su T, questo viene monomorfizzato generando più copie dello stesso codice specializzate sul tipo concreto.

Tuttavia se si utilizza un’interfaccia e il compilatore non è in grado di capire a tempo di compilazione qual è il tipo che verrà passato, avverrà dispatch dinamico a runtime. Ad esempio:



All’interno di do\_something si prende x che è un tipo concreto, lo si casta all’interfaccia (quindi si indica al compilatore di ignorare che sia un intero e considerarlo come un elemento di tipo Foo) quindi il compilatore non riesce più a fare monomorfizzazione ma genera codice do\_something, che **prende in input l’intero e la sua tabella dei metodi** (come Go).

Un trait object è sempre un puntatore a un trait, lo si alloca sullo heap. I puntatori hanno tutti la stessa dimensione e quindi possono essere passati allo stesso codice polimorfo (come nel caso di OCaml). Come per Go, un trait object in verità è sempre un puntatore all’oggetto più un puntatore alla *vtable* del trait per un particolare tipo concreto. In questo caso il dynamic dispatch inibisce l’inlining di codice (inline si riferisce all'ottimizzazione in fase di compilazione in cui una funzione di codice viene iniettata nella funzione chiamante anziché richiedere una chiamata separata).

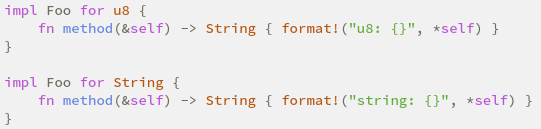
Non tutti i trait possono essere usati per dichiarare trait objects, quindi non per tutti i trait si possono utilizzare questi meccanismi di dynamic dispatch. Perché, intuitivamente, un trait può essere usato per trait object se tutti i suoi metodi sono polimorfizzabili, ovvero se prendono in input dati di dimensione nota a compile time (es. puntatori) o a run-time (se il loro tipo implementa il particolare trait Sized).

**Trait objects** come spiegato in [Trait Objects](https://doc.rust-lang.org/1.24.0/book/first-edition/trait-objects.html)

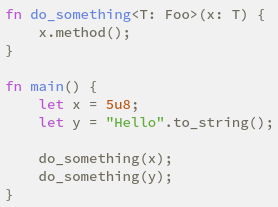
Quando il codice comprende polimorfismo, è necessario un meccanismo per determinare quale versione specifica sta attualmente eseguendo, questo è chiamato “dispatch”.

Definiamo il seguente trait e lo facciamo implementare a u8 e a String:

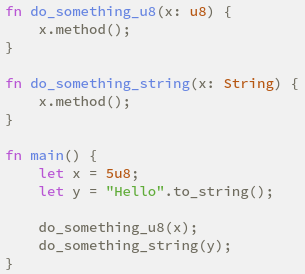




Questo trait può essere usato per eseguire **static dispatch** con trait bounds:

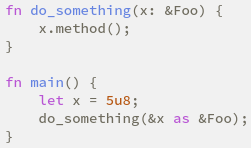


Rust usa **monomorfizzazione** per fare static dispatch quindi crea una speciale versione di do\_something() sia per u8 che per String:

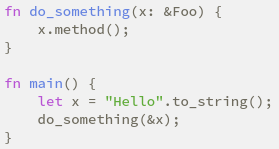


Rust fornisce anche **dynamic dispatch** tramite i **Trait Objects**. I trait objects come ad esempio &Foo o Box<Foo> sono valori normali che memorizzano un valore di *ogni tipo* che implementi quel dato trait (Foo in questo caso), e in cui il tipo preciso di trait può essere conosciuto solo a runtime. Un trait object può essere ottenuto da un puntatore ad un tipo concreto (al suo indirizzo di memoria) che implementa il trait, tramite *casting* (&x as &Foo) o *coercing* (usando &x come argomento per una funzione che prende &Foo). Tornando all’esempio di prima:

Dynamic dispatch usando casting:



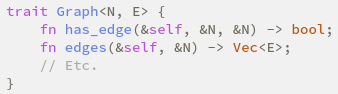
Dynamic dispatch usando coercing:



**Associated types**

Quando ad esempio si lavora con i grafi si implementano tre strutture dati: i nodi, gli archi e il grafo. Se si vogliono scrivere funzioni che lavorino per qualunque implementazione di un grafo, nel trait Graph si inseriscono tutte le funzioni che lavorano su grafo generico (es. has\_edge e edges).

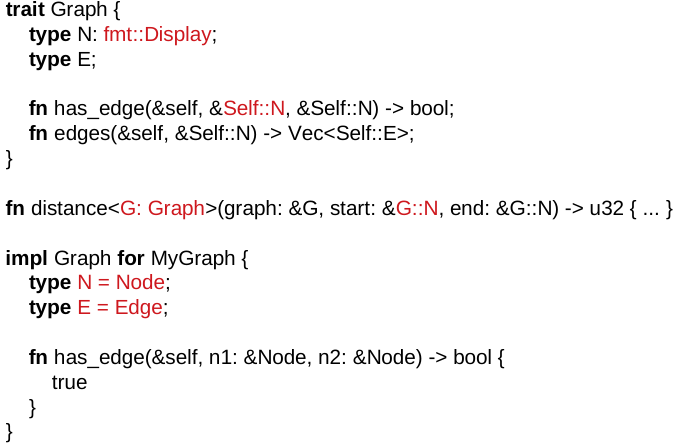
Ad esempio has\_edge prende un Graph e due nodi, e edges ritorna un vettori di archi.



Il problema è che il trait indica quali sono i metodi implementati da un tipo; in questo caso però si ha bisogno anche del tipo di dato dei nodi e del tipo di dato degli archi. Non è consigliabile passare tipi polimorfi a nodi e archi perché un grafo tipicamente ha una struttura per i nodi e per gli archi fissa, non è applicabile a qualsiasi tipo di dato, e nell’esempio sopra Graph potrebbe essere chiamato con qualsiasi tipo al posto di N ed E.

Il problema è: come si fa a dichiarare un trait su un tipo (il tipo del grafo), ma in cui il tipo del nodo e dell’edge debbano anche loro essere determinati dal tipo del grafo.

L’idea di Rust per risolvere questo problema è di mettere all’interno della dichiarazione del trait delle dichiarazioni di tipi (in questo caso i tipi N ed E, in cui N deve implementare almeno i metodi di fmt::Display). Dopodichè della dichiarazione dei metodi del trait si possono usare questi tipi.



Nel caso di distance, start ed end sono tipati con G::N, G non solo è una riga, che deve avere almeno i metodi di Graph, ma deve avere anche due tipi N ed E. Nell’implementazione di Graph poi si associa ad N ed E la vera implementazione dei tipi.

**Traits, Mix-ins, Interfaces, …**

Tipicamente i **traits** non definiscono campi, solo metodi in funzione di altri metodi. Quindi un trait ha a che fare esclusivamente con il behaviour. Si parla invece di **mix-ins** quando sono presenti anche campi, si mischia quindi lo stato e il behaviour.

Questa aggiunta di campi può portare al diamond problem, ovvero come gestire più trait che implementano lo stesso campo, quale scegliere? I traits possono chiedere al programmatore di risolvere il conflitto facendo override del metodo in conflitto all’interno della classe che eredita entrambi i traits conflittuali (è quello che succede in Go e in Java).

I mix-ins risolvono il conflitto favorendo il primo o l’ultimo incluso quando si effettua ereditarietà multipla (es. OCaml).

*Traits* e *interfacce* possono essere considerati la stessa cosa. Un’interfaccia dichiara metodi (a volte campi) senza implementarli, a differenza dei traits. Tuttavia in alcuni linguaggi (ad esempio Java >= 8) i metodi delle interfacce possono avere un’implementazione di default rendendoli di fatto traits.

*Mix-in* e *abstract classes* sono di fatto concetti equivalenti, spesso l’inheritance è ristretta a single inheritance per le abstract classes, e sempre multipla per i mix-ins e i traits.

**Haskell**

Haskell è il più puro dei linguaggi di programmazione, non solo è completamente privo di side-effect, ma le librerie sono organizzate su concetti matematici superiori, come le categorie.

**Type-classes**

L’idea è che esiste una corrispondenza che dovrebbe essere molto rigida tra i tipi di un linguaggio di programmazione e le formule di una logica. Questo induce il fatto che le dimostrazioni delle formule corrispondono ai programmi che abitano i tipi. Questa corrispondenza, che va sotto il nome di Curry-Howard, permette di prendere concetti della logica, che si sa essere corretti, e guardarli sottoforma di tipi e costrutti di un linguaggio di programmazione, e viceversa.

Esempi di questo isomorfismo sono l’and: dati A, B allora A and B è vero quando è vero A ed è vero B, uguale parlando di prove, ovvero si ha una prova di “A and B” se si ha una prova di A e si ha una prova di B.

Se le prove sono i termini (i valori), allora si ha un valore di tipo A and B se si ha un valore di tipo A e un valore di tipo B, ma questo vuol dire che l’and altro non è che un record, una tupla.

L’or invece corrisponde alla somma disgiunta, che abbiamo visto emergere in molti linguaggi come tipi di dato algebrici. In logica se abbiamo un elemento A allora abbiamo anche un elemento A or B, in programmazione in modo simile se si ha un elemento di tipo A si dovrà avere anche un elemento di tipo A “qualcosa” B. L’unione disgiunta permette di prendere un elemento di tipo A ed etichettarlo per metterlo in un tipo più ampio che può contenere sia A che B.

Ancora in logica se ho A or B e devo dimostrare C si va per casi, si analizza A or B, nel caso A si dimostra C e nel caso B si dimostra sempre C. Questo corrisponde al pattern matching (o switch) visto per i tipi di dato algebrici in cui si prende in input qualcosa di tipo “unione disgiunta A e B” e poi lo si analizza (pattern matching) se è di tipo A si fa qualcosa, se è di tipo B qualcos’altro.

Il per ogni corrisponde a polimorfismo uniforme, e l’esiste corrisponde a una forma di data hiding.

Per ora si è parlato di logica proposizionale, logica in cui sono presenti solo booleani, non valori. Ad esempio non si può trattare l’espressione “∀ lista l t.c. len(l) ≥ 0”, qui si sta quantificando su l che non è una proposizione ma una lista e si hanno operatori, come il maggiore uguale che sono dei predicati che prendono in input liste, numeri, ecc e hanno valori di verità.

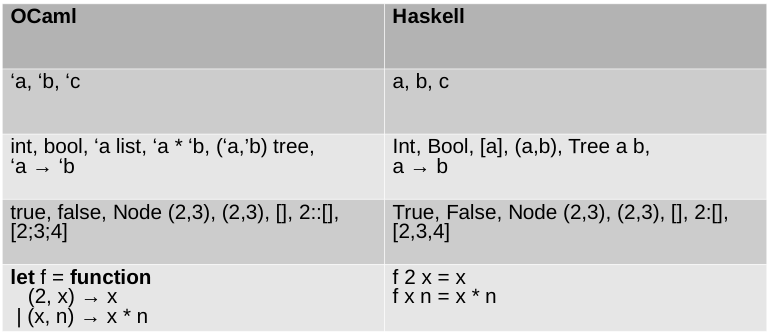
Un predicato in logica, indica per un qualche valore se una certa proprietà vale o meno. Ad esempio il predicato “essere pari” indica di un numero naturale se è pari o no (simile con il predicato “essere maggiore o uguale” e così via).

Nel mondo dei linguaggi di programmazione sui tipi un predicato su un tipo deve dire un qualcosa su un tipo, per esempio un predicato Num su un tipo “a” che indica se “a” è una rappresentazione di un numero. Come in logica un predicato va definito, ovvero un predicato vale se e solo se succede qualcosa. Nel caso di Num di “a” si può dire che “a” è un tipo numerico se e solamente se sono definite su di esso alcune operazioni (ad esempio somma, prodotto, ecc). Oppure analogamente un tipo “a” è un tipo Eq se e solamente se ha su di esso un metodo per confrontare elementi di tipo “a”.

L’idea è che una **type-class** altro non sia che il **predicare su un tipo l’esistenza di alcune funzioni**. Nel momento in cui si vede la type-class come un predicato, questo non deve essere necessariamente unario, in logica si utilizzano predicati binari (es. ≤ prende due numeri). In modo analogo la type-class è un predicato che data una lista di tipi predica l’esistenza di alcune funzioni. Nell’esempio precedente sui Grafi, un predicato utilizza Grafo, Nodi e Archi e predica il fatto che esistano certe funzioni che “parlano” di tutti e tre i tipi contemporaneamente (predicato ternario). *Mentre le interfacce di Go e traits di Rust, quando definiti su un tipo solo senza tipi interni, altro non sono che un predicato unario*.

Le type-classes sono un modo pulito per avere overloading.

Sintassi di Haskell



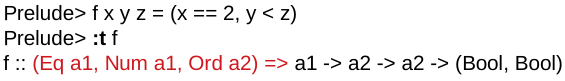
Haskell viene confrontato con Ocaml per quanto riguarda i costrutti tipati, OCaml è stato confrontato con Erlang per quanto riguarda i costrutti non tipati, e di conseguenza si possono dedurre anche i costrutti non tipati di Haskell.

Nel top-level di Haskel, **:t** chiede al compilatore di tipare quanto segue. Ad esempio:

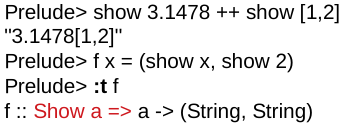


Qui sono evidenti le type-classes in azione. Prima di tutto compaiono b e c, i quali sono quantificati universalmente. La tripla non ha un tipo solo, ma per ogni b e c ha un certo tipo, precisamente una tripla in cui il primo elemento è una lista di caratteri (ovvero una stringa), il secondo è il tipo b e il terzo il tipo c. Per ogni b e c se vale (Num b, Fractional c) allora vale ([Char], b, c), questa è a tutti gli effetti un’implicazione logica. Nello specifico se b soddisfa il predicato Num (ha su di esso implementati i metodi per i numeri) e c soddisfa il predicato Fractional allora quella è una tripla ([Char], b, c).

L’idea è pensare al polimorfismo generico per ogni tipo b e c costretto da certi predicati (cosa si può sapere su b e c). Perfino il dato qui è stato tipato in maniera polimorfa, infatti b è Num ma non è specificato sia qualcosa di ancora più specifico.



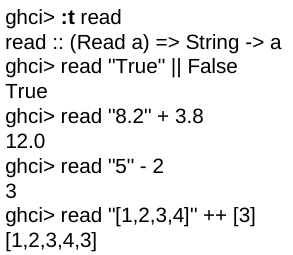
In quest’altro esempio viene tipata indicando che f per ogni tipo a1 e per ogni tipo a2 è una funzione che prende una x di tipo a1, una y di tipo a2 e una z di tipo a2 e restituisce una coppia di booleani. Tuttavia questo se e solo se a1 implementa un predicato di uguaglianza, a2 implementa i predicati numerici e a2 è un tipo ordinato che implementa maggiore, minore, ecc.



In OCaml non abbiamo polimorfismo ad-hoc (overloading), il + funzionava solo sugli interi ad esempio, il +. solo sui floating point, ecc, si rinuncia a questa funzionalità per avere type inference. In Haskell si ha sia type inference come OCaml ma non si rinuncia all’overloading, semplicemente tutti gli operatori overloaded sono overloaded utilizzando le type-classes.

Nell’esempio sopra f viene tipata in modo che prenda in input un elemento di tipo Showable e restituisce una coppia di stringhe.

Esempio funzione read

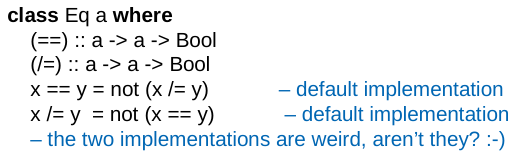


Se a è readable allora si ottiene una funzione che prende una stringa e restituisce a. La read prende sempre in input una stringa, è l’output che cambia, e ci saranno quindi varie implementazioni (ad esempio read “True” || False ritorna un booleano). Eseguendo solo read “4” senza nient’altro non ritorna nulla perché il compilatore non sa staticamente in cosa dover convertire la stringa “4”, non essendo associata ad altri tipi di dato.

L’idea delle type-classes è quella di avere polimorfismo ad-hoc in un linguaggio con type inference.

**Definire e implementare type-classes**

Ad esempio:



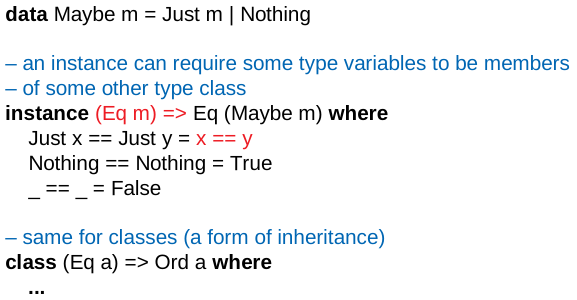
In questo caso si sta dicendo: il tipo a soddisfa il predicato Eq se sono implementate due funzioni, == che prende due a e restituisce Bool, e /= del tutto simile. In Haskell è possibile dare implementazioni di default (come nei traits).

Questa implementazione è stata data in modo così “incrociato” ovvero not (x /= y) in modo che se non vengono implementate ma lasciate così andrebbero in loop infinito (== chiama /= che chiama == e così via). La cosa interessante è che sarebbe sufficiente implementare solo una delle due, l’altra sarà la sua negazione.

Successivamente si definisce un tipo di dato algebrico TrafficLight che può essere di tipo Red, Yellow o Green. Poi bisogna dimostrare che TrafficLight soddisfa il predicato Eq, implementando le funzioni == e/o /= tramite instance … where.

La definizione di Eq è il predicato, mentre la instance è la dimostrazione che TrafficLight soddisfa il predicato Eq.

Altro esempio:

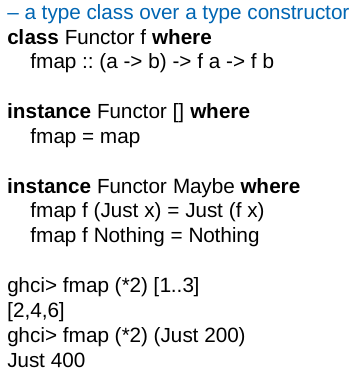


Il dato Maybe è l’option di OCaml, che permette di codificare l’assenza o la presenza di un dato. Successivamente si vuol far vedere che il tipo Maybe m soddisfa il predicato Eq, fornendo un’implementazione dell’uguaglianza. Chiamando l’operatore == su x e y è necessario che sia x che y, che sono di tipo m, abbiano implementato questo operatore. Quindi si riesce a dimostrare che Maybe m è un Eq se si suppone che m sia un Eq (tra parentesi in rosso). Letto in un altro modo: per ogni tipo m, se m è Eq allora anche Maybe m è Eq. Questo può essere paragonato all’esempio di Rectangle in Rust, la cui implementazione prendeva un tipo T: PartialEq + Debug, ovvero prendeva un tipo T che però implementasse sia le funzioni di PartialEq che di Debug, Haskell è visto in forma di predicato.

Con queste “se … allora …” il codice può essere visto come un programma logico. Si scrive un generatore di codice per un linguaggio funzionale utilizzando un (meta) linguaggio logico.

Anche per le classi è la stessa cosa, anche il predicato Ord a “essere ordinato” ha una premessa Eq a, quindi per ogni tipo a se a è già Eq ovvero ha già l’implementazione dell’uguale e diverso, allora a è ordinato se ha *anche* maggiore e minore ad esempio (=> è una vera e propria implicazione logica).

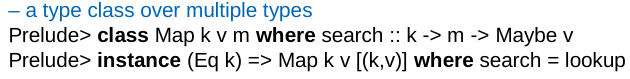
Altro esempio:



Una type-class può essere costruita non solo su un tipo ma anche su un costruttore di tipo (ad esempio lista, una lista di booleani è un tipo, ma lista e basta non è un tipo, coppia, ...).

Nell’esempio sopra, il costruttore di tipo f soddisfa il predicato Functor se c’è una funzione fmap che data una funzione da a in b restituisce una funzione da f di a ad f di b.

Tutti i contenitori implementano Functor, tutte le strutture dati che contengono qualcosa implementano una nozione di mappa. Ad esempio le liste, perché data una funzione da a in b si può trasformare una lista di tipo a in una di tipo b (si itera sulla lista e si applica la funzione a tutti gli elementi). Anche Maybe è un funtore, f di Nothing è Nothing, mentre f di Just di x è Just di f di x.



In questo esempio a Map viene indicato il tipo delle chiavi, il tipo dei valori e la mappa stessa. Il predicato Map è quindi un predicato ternario, un tipo di chiave, un tipo di valore, un tipo di mappa soddisfano il predicato Map “formare una mappa” se c’è un metodo search che prende un elemento di tipo chiave, prende un elemento di tipo mappa e restituisce un elemento di tipo valore opzionale. Un esempio di implementazione è dimostrare (tramite instance) che soddisfano il predicato Map la tripla: tipo per le chiavi, tipo per i valori e una lista di coppie chiave e valore, a patto che (la premessa), la chiave k soddisfi il predicato Eq (ovvero poter confrontare le chiavi). Per far vedere che questa implementazione è corretta si dimostra che search è la funzione di libreria lookup.

**Implementazione di type classes**

appunti presi da [qui](http://okmij.org/ftp/Computation/typeclass.html) da studiare perchè ci ha messo una domanda a proposito.

type classes = bounded polymorphism = parametric overloading.