Introduzione alla programmazione funzionale

Giulio Canti

April 26, 2018

Contents

1 Che cos'è la programmazione funzionale

> Though programming was born in mathematics, it has since largely been divorced from it. The idea is that there's some higher level than the code in which you need to be able to think precisely, and that mathematics actually allows you to think precisely about it
> - Leslie Lamport

La programmazione funzionale usa *modelli* formali per descrivere e meglio governare le *implementazioni*. E più l'implementazione si avvicina al suo corrispettivo ideale (il modello matematico) più diventa facile ragionare sul sistema.

Ecco un parziale elenco di concetti sfruttati dalla programmazione funzionale

- Higher-order functions (`map`, `reduce`, `filter`, ...)
- Funzioni pure
- Strutture dati immutabili
- Algebraic Data Types
- Trasparenza referenziale[1]
- Algebre (Semigruppi, Monoidi, ...)
- Teoria delle Categorie (Funtori, Funtori applicativi, Monadi, ...)
- Ottica funzionale

Nella programmazione funzionale sono innanzitutto le proprietà del codice ad essere portate in primo piano.

Esempio 1.1 *Perchè `map` è "più funzionale" di un ciclo `for`?*

```
const xs = [1, 2, 3]
const double = n => n * 2
```

[1] An expression is said to be *referentially transparent* if it can be replaced with its corresponding value without changing the program's behavior

```
const ys = []
for (var i = 0; i < xs.length; i++) {
  ys.push(double(xs[i]))
}

const zs = xs.map(double)
```

Un ciclo `for` è più flessibile: posso modificare l'indice di partenza, la condizione di fine e il passo. Ma questo vuol dire anche che ci sono più possibilità di introdurre bachi e non ho alcuna garanzia sul risultato. Una `map` invece mi da delle garanzie: gli elementi dell'input verrano processati tutti dal primo all'ultimo e qualunque sia l'operazione che viene fatta nella callback, il risultato sarà sempre un array con lo stesso numero di elementi dell'array di input.

1.1 Quali sono i suoi obiettivi?

- Design pattern derivati dai modelli formali
- Programmazione modulare[2]
- Gestire gli effetti in modo che valga la trasparenza referenziale
- Rendere gli stati illegali non rappresentabili

Vedremo come lo studio delle algebre e delle monadi permettano di raggiungere questi obiettivi in modo generale.

Il pattern fondamentale della programmazione funzionale è la *componibilità*, ovvero la costruzione di piccole unità che fanno qualcosa di specifico in grado di essere combinate al fine di ottenere entità più grandi e complesse.

DEMO `combinator.ts`

[2]By modular programming I mean the process of building large programs by gluing

Una gran parte delle tecniche utilizzate nella programmazione funzionale sono mutuate dalla matematica.

Facciamo due esempi che ci saranno utili anche in seguito

- come catturare il concetto di computazione parallelizzabile?
- che cos'è una funzione pura?

1.2 Un esempio di matematica applicata: la proprietà associativa

Il concetto di computazione parallelizzabile (e distribuibile) può essere catturato dalla nozione di operazione associativa.

Definizione 1.1 *Sia A un insieme, una operazione $* : A \times A \to A$ si dice* associativa *se per ogni $a, b, c \in A$ vale*

$$(a * b) * c = a * (b * c)$$

In altre parole la proprietà associativa garantisce che non importa l'ordine in cui vengono fatte le operazioni, il risultato sarà sempre lo stesso.

Possiamo quindi eliminare le parentesi, senza pericolo di ambiguità

$$a * b * c$$

Esempio 1.2 *La somma di interi gode della proprietà associativa.*

Se sappiamo che una data operazione gode della proprietà associativa possiamo suddividere una computazione in due sotto computazioni, ognuna delle quali può essere ulteriormente suddivisa

$$a * b * c * d * e * f * g * h = \big((a * b) * (c * d)\big) * \big((e * f) * (g * h)\big)$$

Le sotto computazioni possono essere distribuite ed eseguite parallelamente.

1.3 Funzioni pure

> Una funzione pura è una procedura che dato lo stesso input restituisce sempre lo stesso output e non ha alcun side effect osservabile.

Un tale enunciato informale può lasciare spazio a qualche dubbio (che cos'è un side effect? cosa vuol dire osservabile?). Vediamo una definizione formale del concetto di funzione

Definizione 1.2 *Una* funzione $f : X \to Y$ *è un sottoinsieme* f *di* $X \times Y$ *(il* prodotto cartesiano *di* X *e* Y*) tale che per ogni* $x \in X$ *esiste esattamente un* $y \in Y$ *tale che* $(x, y) \in f$[3]*.*

L'insieme X si dice il *dominio* di f, Y il suo *codominio*.

Esempio 1.3 *La funzione* `double` : $Nat \to Nat$ *è il sottoinsieme del prodotto cartesiano* $Nat \times Nat$ *dato da* $\{(1, 2), (2, 4), (3, 6), \ldots\}$*.*

In TypeScript

```
const f: { [key: number]: number } = {
  1: 2,
  2: 4,
  3: 6
  ...
}
```

Si noti che l'insieme f deve essere descritto *staticamente* in fase di definizione della funzione (ovvero gli elementi di quell'insieme non possono variare nel tempo e per nessuna condizione interna o esterna). Ecco allora che viene esclusa ogni forma di side effect e il risultato è sempre quello atteso.

Quella dell'esempio viene detta definizione *estensionale* di una funzione, ovvero si enumerano uno per uno gli elementi dell'insieme. Naturalmente quando l'insieme è infinito come in questo caso, la definizione può risultare un po' scomoda.

Si può ovviare a questo problema introducendo quella che viene detta definizione *intensionale*, ovvero si esprime una condizione che deve valere per tutte le coppie $(x, y) \in f$ ovvero $y = x * 2$. Questa è la familiare forma con cui scriviamo la funzione *double* e come la definiamo in JavaScript

together smaller programs - Simon Peyton Jones

[3] Questa definizione risale ad un secolo fa https://en.wikipedia.org/wiki/History_of_the_function_concept

```
const double = x => x * 2
```

o in TypeScript, ove risultano evidenti dominio e codominio sottoforma di type annotation

```
const double = (x: number): number => x * 2
```

La definizione di funzione come sottoinsieme di un prodotto cartesiano mostra come in matematica tutte le funzioni siano pure: non c'è azione, modifica di stato o modifica degli elementi (che sono considerati immutabili) degli insiemi coinvolti. Nella programmazione funzionale l'implementazione delle funzioni deve avvicinarsi il più possibile a questo modello ideale.

Che una funzione sia pura non implica necessariamente che sia bandita la mutabilità, localmente è ammissibile se non esce dai confini della implementazione.

```
              Mutable
                 ^
    (Good)       |      (Bad)
                 |
Non-Shared ----------> Shared
  State          |      State
                 |
    (Good)       |      (Good)
             Immutable
```

Lo scopo ultimo è garantirne le proprietà fondamentali: purezza e trasparenza referenziale.

Infine le funzioni compongono

Definizione 1.3 *Siano* $f : Y \to Z$ *e* $g : X \to Y$ *due funzioni, allora la funzione* $h : X \to Z$ *definita da*

$$h(x) = f(g(x))$$

si dice composizione *di* f *e* g *e si scrive* $h = f \circ g$

Si noti che affinchè due funzioni f e g possano comporre, il codominio di g deve coincidere col dominio di f.

1.4 Funzioni parziali

Definizione 1.4 *Una funzione* parziale *è una funzione che non è definita per tutti i valori del dominio.*

Viceversa una funzione definita per tutti i valori del dominio è detta *totale*.

Esempio 1.4

$$f(x) = \frac{1}{x}$$

La funzione $f : number \rightarrow number$ non è definita per $x = 0$.

Una funzione parziale $f : X \rightarrow Y$ può essere sempre ricondotta ad una funzione totale aggiungendo un valore speciale, chiamiamolo $None$, al codominio e associandolo ad ogni valore di X per cui f non è definita

$$f' : X \rightarrow Y \cup None$$

Chiamiamo $Option(Y) = Y \cup None$.

$$f' : X \rightarrow Option(Y)$$

In ambito funzionale si tende a definire solo funzioni totali.

2 Error handling funzionale

Consideriamo la funzione

```
const inverse = (x: number): number => 1 / x
```

Tale funzione è parziale perchè non è definita per $x = 0$. Come possiamo gestire questa situazione?

Una soluzione potrebbe essere lanciare un'eccezione

```
const inverse = (x: number): number => {
  if (x !== 0) return 1 / x
  throw new Error('cannot divide by zero')
}
```

ma così la funzione non sarebbe più da considerarsi pura [4].

Un'altra possibile soluzione è restituire `null`

```
const inverse = (x: number): number | null => {
  if (x !== 0) return 1 / x
  return null
}
```

Sorge però un nuovo problema quando si cerca di comporre la funzione `inverse` così modificata con un'altra funzione

```
// calcola l'inverso e poi moltiplica per 2
const doubleInverse = (x: number): number => double(inverse(x))
```

L'implementazione di `doubleInverse` non è corretta, cosa succede se `inverse(x)` restituisce `null`? Occorre tenerne conto

```
const doubleInverse = (x: number): number | null => {
  const y = inverse(x)
  if (y === null) return null
  return double(y)
}
```

[4] Le eccezioni sono considerate un side effect inaccettabile perchè modificano la normale

Appare evidente come l'obbligo di gestione del valore speciale `null` si propaghi in modo contagioso a tutti gli utilizzatori di `inverse`.

Questo approccio ha diversi svantaggi

- molto boilerplate
- prono ad errori (è facile dimenticarsi di gestire il caso di fallimento)
- le funzioni non compongono facilmente

2.1 Il tipo `Option`

La soluzione funzionale ai problemi illustrati precedentemente è l'utilizzo del tipo `Option`, eccone la definizione

```
type Option<A> = None | Some<A>

class None {
  readonly _tag = 'None'
}

class Some<A> {
  readonly _tag = 'Some'
  constructor(readonly value: A) {}
}

// in TypeScript never è un bottom type
const none: Option<never> = new None()

const some = <A>(a: A): Option<A> => new Some(a)
```

Ridefiniamo `inverse` sfruttando `Option`

```
const inverse = (x: number): Option<number> =>
  x === 0 ? none : some(1 / x)
```

Possiamo interpretare questa modifica in termini di successo e fallimento: se viene restituita una istanza di `Some` la computazione di `inverse` ha avuto successo, se viene restituita una istanza di `None` essa è fallita.

Il tipo `Option` codifica l'*effetto* di una computazione che può fallire

Aggiungiamo un metodo `map`

```
type Option<A> = None<A> | Some<A>

class None<A> {
  readonly _tag = 'None'
  map<B>(f: (a: A) => B): Option<B> {
    return none
  }
}

class Some<A> {
  readonly _tag = 'Some'
  constructor(readonly value: A) {}
  map<B>(f: (a: A) => B): Option<B> {
    return some(f(this.value))
  }
}
```

In un linguaggio che non supporta classi e metodi, `map` può essere definita come funzione

```
const map = <A, B>(f: (a: A) => B, fa: Option<A>): Option<B> => {
  switch (fa._tag) {
    case 'None' :
      return none
    case 'Some' :
      return some(f(fa.value))
  }
}
```

Ora è possibile definire `doubleInverse` senza boilerplate

esecuzione del codice e violano la trasparenza referenziale

```
const doubleInverse = (x: number): Option<number> =>
  inverse(x).map(double)

doubleInverse(2) // Some(1)
doubleInverse(0) // None
```

Inoltre è facile concatenare altre operazioni

```
const inc = (x: number): number => x + 1

inverse(0)
  .map(double)
  .map(inc) // None
inverse(4)
  .map(double)
  .map(inc) // Some(1.5)
```

> `Option` mi permette di concentrarmi solo sul *path di successo* in una serie di computazioni che possono fallire

Per questioni di interoperabilità con codice che non usa `Option` possiamo definire due utili funzioni

```
const fromNullable = <A>(
  a: A | null | undefined
): Option<A> => (a == null ? none : some(a))

const toNullable = <A>(fa: Option<A>): A | null => {
  switch (fa._tag) {
    case 'None':
      return null
    case 'Some':
      return fa.value
  }
}
```

2.2 Branching tramite la funzione `fold`

Prima o poi dovrò affrontare il problema di stabilire cosa fare sia nel caso di successo che di fallimento. La funzione `fold` permette di gestire i due casi

```
class Some<A> {
  ...
  fold<R>(f: () => R, g: (a: A) => R): R {
    return g(this.value)
  }
}

class None<A> {
  ...
  fold<R>(f: () => R, g: (a: A) => R): R {
    return f()
  }
}

const f = (): string => 'cannot divide by zero'
const g = (x: number): string => `the result is ${x}`

inverse(2).fold(f, g) // 'the result is 0.5'
inverse(0).fold(f, g) // 'cannot divide by zero'
```

Si noti come il branching è racchiuso nella definizione di `Option` e non necessita di alcun `if` e che l'utilizzo necessita solo di funzioni.

Inoltre le funzioni `f` e `g` sono generiche e riutilizzabili.

2.3 Il tipo `Either`

Il tipo `Option` è utile quando c'è un solo modo evidente per il quale una computazione può fallire, oppure ce ne sono diversi ma non interessa distinguerli.

Se invece esistono molteplici ragioni di fallimento ed interessa comunicare al chiamante quale si sia verificata, oppure se si vuole definire un errore personalizzato, è possibile impiegare il tipo `Either`. Eccone la definizione

```
type Either<L, A> = Left<L, A> | Right<L, A>
```

```
class Left<L, A> {
  readonly _tag = 'Left'
  constructor(readonly value: L) {}
  map<B>(f: (a: A) => B): Either<L, B> {
    return left(this.value)
  }
}

class Right<L, A> {
  readonly _tag = 'Right'
  constructor(readonly value: A) {}
  map<B>(f: (a: A) => B): Either<L, B> {
    return right(f(this.value))
  }
}

const left = <L, A>(l: L): Either<L, A> =>
  new Left(l)

const right = <L, A>(a: A): Either<L, A> =>
  new Right(a)
```

Tipicamente `Left` rappresenta il caso di fallimento mentre `Right` quello di successo.

Ridefiniamo la funzione `inverse` in funzione del tipo `Either`

```
const inverse = (x: number): Either<string, number> =>
  x === 0 ? left('cannot divide by zero') : right(1 / x)
```

Ancora una volta è possibile definire `doubleInverse` senza boilerplate

```
const doubleInverse = (x: number): Either<string, number> =>
  inverse(x).map(double)

doubleInverse(2) // Right(1)
doubleInverse(0) // Left('cannot divide by zero')
```

ed è facile comporre insieme altre operazioni

```
inverse(0)
  .map(double)
  .map(inc) // Left('cannot divide by zero')
inverse(4)
  .map(double)
  .map(inc) // Right(1.5)
```

Anche per il tipo `Either` è possibile definire una funzione `fold`

```
class Left<L, A> {
  ...
  fold<R>(f: (l: L) => R, g: (a: A) => R): R {
    return f(this.value)
  }
}

class Right<L, A> {
  ...
  fold<R>(f: (l: L) => R, g: (a: A) => R): R {
    return g(this.value)
  }
}
```

I vantaggi offerti dalla funzione `map` non sono esclusivi dei tipi `Option` e `Either`. Essi sono condivisi da tutti i membri di una vasta famiglia che prende il nome di *funtori*. Per definire in modo formale cosa sia un funtore, occorre prima introdurre il concetto di *categoria*.

3 Teoria delle categorie

3.1 Perchè è importante?

> And how do we solve problems? We decompose bigger problems into smaller problems. If the smaller problems are still too big, we decompose them further, and so on. Finally, we write code that solves all the small problems. And then comes the essence of programming: we compose those pieces of code to create solutions to larger problems. Decomposition wouldn't make sense if we weren't able to put the pieces back together. - Bartosz Milewski

Ma cosa vuol dire esattamente *componibile*? Quando possiamo davvero dire che due cose *compongono*? E se compongono quando possiamo dire che lo fanno in un *modo buono*?

> Entities are composable if we can easily and generally combine their behaviors in some way without having to modify the entities being combined. I think of composability as being the key ingredient necessary for acheiving reuse, and for achieving a combinatorial expansion of what is succinctly expressible in a programming model. - Paul Chiusano

Occorre poter fare riferimento ad una teoria **rigorosa** che possa fornire risposte a domande così fondamentali.

Opportunamente da più di 60 anni un vasto gruppo di studiosi appartenenti al più longevo e mastodontico progetto open source nella storia dell'umanità si occupa di sviluppare una teoria specificatamente dedicata a questo argomento: la *componibilità*.

Il progetto open source si chiama *matematica* e la teoria sulla componibilità ha preso il nome di *Teoria delle categorie*.

Studiare teoria delle categorie non è perciò un passatempo astratto, ma va dritto al cuore di ciò che facciamo tutti i giorni quando vogliamo sviluppare (buon) software.

3.2 Definizione

Definizione 3.1 *Una* categoria $\mathcal{C}$ *è una coppia* (O, M) *ove*

- O *è una collezione di* oggetti*, non meglio specificati. Considerate un oggetto come un corpo imperscrutabile, senza struttura né proprietà distintive, a meno della sua identità (ovvero considerati due oggetti sappiamo solo se sono uguali oppure diversi ma non il perchè).*

- M *è una collezione di* frecce *(o* morfismi*) che collegano gli oggetti. Tipicamente un morfismo* f *è denotato con* $f : A \to B$ *per rendere chiaro che è una freccia che parte da* A *detta* sorgente *e arriva a* B *detta* destinazione*.*

Mentre gli oggetti non hanno ulteriori proprietà da soddisfare, per i morfismi devono valere alcune condizioni note come leggi

Morfismi identità. *Per ogni oggetto X di $\mathcal{C}$ deve esistere un morfismo $1_X : X \to X$ (chiamato* morfismo identità per *X)*

Composizione di morfismi. *Deve esistere una operazione, indichiamola con il simbolo $\circ$, detta* composizione*, tale che per ogni coppia di morfismi $g : A \to B$ e $f : B \to C$ associa un terzo morfismo $f \circ g : A \to C$. Inoltre l'operazione $\circ$ di composizione deve soddisfare le seguenti proprietà:*

- *associatività: se $g : A \to B$, $f : B \to C$, $h : C \to D$, allora $h \circ (f \circ g) = (h \circ f) \circ g$*
- *identità: per ogni morfismo $f : A \to B$ vale $f \circ 1_A = f = 1_B \circ f$ (ove 1_A e 1_B sono rispettivamente i morfismi identità di A e B)*

Esempio 3.1

$$\begin{tikzcd} A \arrow[r, "g"] \arrow[loop above, "1_A"] \arrow[dr, "f \circ g"'] & B \arrow[d, "f"] \arrow[loop above, "1_B"] \\ & C \arrow[loop below, "1_C"] \end{tikzcd}$$

Le categorie possono essere interpretate come linguaggi di programmazione: gli oggetti rappresentano i tipi mentre i morfismi rappresentano le funzioni.

Esempio 3.2

$$\begin{tikzcd} \mathtt{string} \arrow[r, "length"] \arrow[dr, "length{>}2"'] & \mathtt{number} \arrow[d, ">2"] \\ & \mathtt{boolean} \end{tikzcd}$$

4 Funtori

Di fronte ad un nuovo oggetto di studio come le categorie, il matematico ha davanti due percorsi di indagine: il primo, che chiamerò, *ricerca in profondità*, mira a studiare le proprietà di una singola categoria. Il secondo (ed è quello che interessa a noi), che chiamerò *ricerca in ampiezza*, mira a studiare quando due categorie possono essere dette *simili*.

Per iniziare questo secondo tipo di indagine dobbiamo introdurre un nuovo strumento: le mappe tra categorie (pensate a mappa come ad un sinonimo di funzione).

Mappe tra categorie. Se $\mathcal{C}$ e $\mathcal{D}$ sono due categorie, cosa vuol dire costruire una mappa F tra $\mathcal{C}$ e $\mathcal{D}$?

Essenzialmente vuol dire costruire una associazione tra le parti costituenti di $\mathcal{C}$ e le parti costituenti di $\mathcal{D}$.

Siccome una categoria è composta da due cose, i suoi oggetti e i suoi morfismi, per avere una buona mappa non devo mischiarle, devo cioè fare in modo che agli oggetti di $\mathcal{C}$ vengano associati degli oggetti di $\mathcal{D}$ e che ai morfismi di $\mathcal{C}$ vengano associati dei morfismi di $\mathcal{D}$.

La costruzione di una buona mappa implica che oggetti e morfismi viaggiano su strade separate e non si mischiano tra loro.

Ma mi interessano proprio tutte le mappe che posso costruire così? No davvero, molte di quelle che posso costruire non sarebbero affatto interessanti: quello che voglio è perlomeno preservare la *struttura di categoria*, ovvero che le leggi rimangano valide anche dopo aver applicato la mappa.

Specifichiamo in modo formale che cosa vuol dire per una mappa preservare la struttura categoriale.

4.1 Definizione

Definizione 4.1 *Siano $\mathcal{C}$ e $\mathcal{D}$ due categorie, allora una mappa F si dice* funtore *se valgono le seguenti proprietà:*

- *ad ogni oggetto X in $\mathcal{C}$, F associa un oggetto $F(X)$ in $\mathcal{D}$*
- *ad ogni morfismo $f : A \to B$ in $\mathcal{C}$, F associa un morfismo $F(f) : F(A) \to F(B)$ in $\mathcal{D}$*
- *$F(1_X) = 1_{F(X)}$ per ogni oggetto X in $\mathcal{C}$*

- $F(f \circ_{\mathcal{C}} g) = F(f) \circ_{\mathcal{D}} F(g)$ *per tutti i morfismi* $g : A \to B$ *e* $f : B \to C$ *in* $\mathcal{C}$

Le prime due proprietà formalizzano il requisito che oggetti e morfismi viaggiano su strade separate.

Le ultime due formalizzano il requisito che la struttura categoriale sia preservata.

$$
\begin{tikzcd}
F(A) \arrow[r, "F(g)"] \arrow[rr, bend left, "F(f)\circ_{\mathcal{D}}F(g)"] & F(B) \arrow[r, "F(f)"] & F(C) \arrow[loop above, "F(1_C)"] & \mathcal{D} \\
A \arrow[u, "F"] \arrow[r, "g"'] \arrow[rr, bend right, "f\circ_{\mathcal{C}}g"'] & B \arrow[u, "F"] \arrow[r, "f"'] & C \arrow[u, "F"'] \arrow[loop below, "1_C"'] & \mathcal{C}
\end{tikzcd}
$$

L'associazione tra f e $F(f)$ si chiama *lifting* del morfismo f.

Quando $\mathcal{C}$ e $\mathcal{D}$ coincidono, si parla di *endofuntori* [5].

Esercizio 4.1 *Mostrare che* $F(\mathcal{C})$ *è una sotto-categoria di* $\mathcal{D}$.

4.2 La categoria $\mathcal{TS}$

Vediamo la categoria che rappresenta il linguaggio `TypeScript`.

Come ogni categoria, la categoria $\mathcal{TS}$ è composta da oggetti e morfismi:

- gli oggetti sono i tipi (per esempio `number`, `string`, `boolean`, `Array<number>`, `Array<Array<number>>`, etc ...)
- i morfismi sono funzioni tra tipi (per esempio `number` $\to$ `number`, `string` $\to$ `number`, `Array<number>` $\to$ `Array<number>`, etc ...)

Inoltre l'operazione di composizione $\circ$ è l'usuale composizione di funzioni.

[5] endo proviene dal greco e significa dentro

4.3 Endofuntori in $\mathcal{TS}$

Definire un (endo)funtore F nella categoria $\mathcal{TS}$ significa due cose:

- per ogni tipo A stabilire a quale tipo corrisponde $F(A)$
- per ogni funzione $f : A \to B$ stabilire a quale funzione corrisponde $F(f)$

Perciò un funtore è una coppia $F = ($`F<X>`$,$ `lift`$)$ ove

- `F<X>` è un *type constructor*[6], ovvero una procedura che, dato un qualunque tipo `X` produce un tipo `F<X>`
- `lift` è una funzione con la seguente firma[7]

```
lift: <A, B>(f: (a: A) => B) => ( (fa: F<A>) => F<B> )
```

La funzione `lift` è meglio conosciuta nella sua forma equivalente `map`.

```
map: <A, B>(f: (a: A) => B, fa: F<A>) => F<B>
```

Vediamo l'implementazione per `Option` e `Either`

```
// Option
const funtorOption = {
  map: <A, B>(f: (a: A) => B, fa: Option<A>): Option<B> =>
    fa.fold(() => none, a => some(f(a)))
}

// Either
const functorEither = {
```

---

[6]

- `number` è un 0-type constructor (kind $*$)
- `Option<A>` è un 1-type constructor (kind $* \to *$)
- `Either<L, A>` è un 2-type constructor (kind $* \to * \to *$)

[7] Si noti che la sintassi usata non è valida in TypeScript dato che non supporta gli Higher Kinded Types

```
  map: <L, A, B>(
    f: (a: A) => B,
    fa: Either<L, A>
  ): Either<L, B> => fa.fold(left, a => right(f(a)))
}
```

Per comodità di utilizzo (*chainable APIs*) abbiamo già visto che è utile implementare `map` in modo che l'argomento `fa` sia implicito (ovvero come metodo)

```
// Option
class None<A> {
  ...
  map<B>(f: (a: A) => B): Option<B> {
    return none
  }
}

class Some<A> {
  ...
  map<B>(f: (a: A) => B): Option<B> {
    return some(f(this.value))
  }
}

// Either
class Left<L, A> {
  ...
  map<B>(f: (a: A) => B): Either<L, B> {
    return left(this.value)
  }
}

class Right<L, A> {
  ...
  map<B>(f: (a: A) => B): Either<L, B> {
    return right(f(this.value))
  }
}
```

4.4 Esempi

Vediamo una raccolta dei funtori più comuni

Identity. Manda un tipo `A` ancora in `A`

```
class Identity<A> {
  constructor(readonly value: A) {}
  map<B>(f: (a: A) => B): Identity<B> {
    return new Identity(f(this.value))
  }
}
```

Array. Manda un tipo `A` nel tipo della lista di elementi di tipo `A`

```
const functorArray = {
  map: <A, B>(f: (a: A) => B, fa: Array<A>): Array<B> =>
    fa.map(f)
}
```

IO. Manda un tipo `A` nel tipo `() => A`[8]

```
class IO<A> {
  constructor(readonly run: () => A) {}
  map<B>(f: (a: A) => B): IO<B> {
    return new IO(() => f(this.run()))
  }
}
```

Task. Le `Promise` sono *eager*, ovvero eseguono il side effect immediatamente, Task è una variante *lazy* di una computazione asincrona: manda un tipo `A` nel tipo `() => Promise<A>`

```
class Task<A> {
  constructor(readonly run: () => Promise<A>) {}
  map<B>(f: (a: A) => B): Task<B> {
    return new Task(() => this.run().then(f))
```

[8] Una funzione senza argomenti viene detta *thunk*

```
  }
}
```

Esercizio 4.2 *Sia*

```
type Tuple<L, A> = [L, A]
```

definire una istanza di funtore.

Esercizio 4.3 *Sia*

```
type Decoder<A> = (s: string) => A
```

definire una istanza di funtore.

4.5 Composizione di funtori

I funtori compongono e la `map` della composizione è la composizione delle `map`.

Formalmente, siano $F : \mathcal{D} \to \mathcal{E}$ e $G : \mathcal{C} \to \mathcal{D}$ due funtori, allora possiamo costruire il funtore composizione $F(G) : \mathcal{C} \to \mathcal{E}$

$$
\begin{tikzcd}
F(G(A)) \arrow[r, "F(G(g))"] & F(G(B)) \arrow[r, "F(G(f))"] & F(G(C)) & & \mathcal{E} \\
G(A) \arrow[u, "F"] \arrow[r, "G(g)"] & G(B) \arrow[u, "F"] \arrow[r, "G(f)"] & G(C) \arrow[u, "F"] & & \mathcal{D} \\
A \arrow[u, "G"] \arrow[r, "g"'] & B \arrow[u, "G"] \arrow[r, "f"'] & C \arrow[u, "G"] & & \mathcal{C}
\end{tikzcd}
$$

Esempio 4.1 *Implementiamo una istanza di funtore per gli array di `Option`*

```
class ArrayOption<A> {
  constructor(readonly value: Array<Option<A>>) {}
  map<B>(f: (a: A) => B): ArrayOption<B> {
    return new ArrayOption(this.value.map(o => o.map(f)))
  }
}
```

5 Funtori controvarianti

```
// Funtori covarianti
map: <A, B>      (f: (a: A) => B, fa: F<A>) => F<B>

// Funtori controvarianti
contramap: <A, B>(f: (b: B) => A, fa: F<A>) => F<B>
```

Come esempi di tipi che ammettono istanze di funtore controvariante consideriamo

- i predicati[9]
- le relazioni di equivalenza[10]
- le funzioni di ordinamento[11]
- i serializzatori[12]

Vediamo un esempio con le funzioni di ordinamento

```
type Ordering = -1 | 0 | 1

type Comparison<A> = (x: A, y: A) => Ordering

const contramap = <A, B>(
  f: (b: B) => A,
  fa: Comparison<A>
): Comparison<B> => (x, y) => fa(f(x), f(y))

const comparisonString: Comparison<string> = (x, y) =>
  x < y ? -1 : x > y ? 1 : 0

const comparisonNumber: Comparison<number> = (x, y) =>
  x < y ? -1 : x > y ? 1 : 0
```

[9] `type Predicate<A> = (a: A) => boolean`
[10] `type Equivalence<A> = (x: A, y: A) => boolean`
[11] `type Comparison<A> = (x: A, y: A) => -1 | 0 | 1`
[12] ad esempio `type Encoder<A> = (a: A) => string`

```
interface Person {
  name: string
  age: number
}

const byName: Comparison<Person> = contramap(
  p => p.name,
  comparisonString
)

const byAge: Comparison<Person> = contramap(
  p => p.age,
  comparisonNumber
)

const sort = <A>(
  as: Array<A>,
  c: Comparison<A>
): Array<A> => as.slice().sort(c)

const persons: Array<Person> = [
  { name: 'A', age: 43 },
  { name: 'C', age: 26 },
  { name: 'B', age: 40 }
]

console.log(sort(persons, byName))
/*
[ { name: 'A', age: 43 },
  { name: 'B', age: 40 },
  { name: 'C', age: 26 } ]
*/
console.log(sort(persons, byAge))
/*
[ { name: 'C', age: 26 },
  { name: 'B', age: 40 },
  { name: 'A', age: 43 } ]
```

```
*/
```

Vediamo un altro esempio notevole di funtore controvariante: le componenti di `React`.

Component. Manda un tipo `A` nel tipo `(a: A) => ReactElement`

```
import * as React from 'react'
import * as ReactDOM from 'react-dom'

type Component<A> = (a: A) => React.ReactElement<any>

const contramap = <A, B>(
  f: (b: B) => A,
  fa: Component<A>
): Component<B> => b => fa(f(b))

const DisplayFullName = (a: { fullName: string }) => (
  <div>Hello {a.fullName}</div>
)

ReactDOM.render(
  <DisplayFullName fullName="Giulio Canti" />,
  document.getElementById('app')
)

type Person = {
  name: string
  surname: string
}

const DisplayPerson: Component<Person> = contramap(
  b => ({ fullName: `${b.name} ${b.surname}` }),
  DisplayFullName
)

ReactDOM.render(
  <DisplayPerson name="Giulio" surname="Canti" />,
```

```
  document.getElementById('app')
)
```

6 Bifuntori

Un *bifuntore* è un concetto associato ad un type constructor con kind `* -> * -> *` per il quale esiste una istanza di funtore covariante per ambedue i type parameter.

La sua operazione è chiamata `bimap`

```
interface Bifunctor<F> {
  bimap: <L, A, M, B>(
    fla: F<L, A>,
    f: (l: L) => M,
    g: (a: A) => B
  ) => F<M, B>
}
```

Tale che `bimap(x, identity, identity) = x`

Vediamo un esempio, una istanza per `Either`

```
const bimap = <L, V, A, B>(
  fla: Either<L, A>,
  f: (l: L) => V,
  g: (a: A) => B
): Either<V, B> => {
  return fla.fold(l => left(f(l)), a => right(g(a)))
}
```

7 Profuntori

Vediamo ora un particolare type constructor che ammette sia una istanza di funtore covariante sia una istanza di funtore controvariante: le funzioni.

```
// I = input, O = output
type Function1<I, O> = (i: I) => O
```

Si noti che `Function1` ha kind `* -> * -> *` dato che ha due type parameter [13].

Nel cercare una istanza di funtore abbiamo perciò due scelte

- fissare il type parameter `I`
- fissare il type parameter `O`

Prima fissiamo il tipo in input `I` e otteniamo una istanza di funtore **covariante**

```
const functorFunction1 = {
  map: <I, A, B>(
    f: (a: A) => B,
    fa: Function1<I, A>
  ): Function1<I, B> => {
    return i => f(fa(i))
  }
}
```

Notate che l'implementazione di `map` non è altro che la composizione di funzioni

```
// prima g poi f
const compose = <I, A, B>(
  f: (a: A) => B,
  g: (i: I) => A
): ((i: I) => B) => {
  return i => f(g(i))
}
```

[13] In Haskell `Function1` è indicato con `(->)`

```
const functorFunction1 = {
  map: compose
}
```

Ora fissiamo il tipo in output `O` e otteniamo una istanza di funtore **controvariante**

```
const contravariantFunction1 = {
  contramap: <B, A, O>(
    f: (b: B) => A,
    fa: Function1<A, O>
  ): Function1<B, O> => {
    return b => fa(f(b))
  }
}
```

Notate che l'implementazione di `contramap` non è altro che la funzione `pipe`

```
// prima f poi g
const pipe = <B, A, O>(
  f: (b: B) => A,
  g: (a: A) => O
): ((b: B) => O) => {
  return b => g(f(b))
}

const contravariantFunction1 = {
  contramap: pipe
}
```

Giungiamo alla definizione di `Profunctor`, concetto associato ad un type constructor con kind `* -> * -> *` che è controvariante nel primo type parameter e covariante nel secondo type parameter.

La sua operazione è chiamata `promap` (sinonimo `dimap`)

```
interface Profunctor<P> {
  promap: <A, B, C, D>(
```

```
    pbc: P<B, C>,
    f: (a: A) => B,
    g: (c: C) => D
  ) => P<A, D>
}
```

Tale che `promap(x, identity, identity) = x`

Alternativamente è possibile definire la coppia `lmap` e `rmap`.

```
lmap: <A, B, C>(f: (a: A) => B, pbc: P<B, C>) => P<A, C>

rmap: <A, B, C>(f: (b: B) => C, pab: P<A, B>) => P<A, C>
```

Tali che `lmap(identity, x) = x` e `rmap(identity, x) = x`.

Nota. Deve valere la seguente relazione

```
promap(f, g, x) = lmap(f, rmap(g, x))
```

8 Un po' di algebra: Semigruppi

8.1 Cos'è una algebra?

Potremmo aggiungere al termine programmazione funzionale quello di programmazione algebrica, infatti

Le algebre sono i design pattern della programmazione funzionale

Per algebra si intende generalmente una qualunque combinazione di

- insiemi
- operazioni
- leggi

Le algebre sono il modo in cui i matematici tendono a catturare un concetto nel modo più puro, ovvero eliminando tutto ciò che è superfluo.

Le algebre possono essere considerate una versione più potente delle interfacce: quando si manipola una struttura algebrica sono permesse solo le operazioni definite dall'algebra in oggetto e in conformità alle sue leggi.

I matematici lavorano con tali interfacce da secoli e funziona in modo egregio.

Facciamo un esempio di algebra, il magma.

Definizione 8.1 *Sia M un insieme e $*$ un'operazione* chiusa su *(o* interna a*) M ovvero $* : M \times M \to M$, allora la coppia $(M, *)$ si chiama* magma*.*

> Because the binary operation of a magma takes two values of a given type and returns a new value of the same type (*closure property*), this operation can be chained indefinitely

Il fatto che l'operazione sia chiusa è una proprietà non banale, per esempio sui numeri naturali la somma è una operazione chiusa mentre la sottrazione non lo è.

Un magma non possiede alcuna legge a parte il vincolo basilare di chiusura, vediamo un'algebra che ne definisce una: i semigruppi.

8.2 Definizione

Definizione 8.2 *Sia* $(S, *)$ *un magma, se* $*$ *è associativa* [14] *allora è un* semigruppo.

L'insieme S si dice insieme *sostegno* del semigruppo.

La proprietà associativa ci assicura che non ci dobbiamo preoccupare di utilizzare le parentesi in una espressione[15].

Come abbiamo visto nell'introduzione, i semigruppi catturano l'essenza di una operazione parallelizzabile

Ma possono anche essere usati per rappresentare l'idea astratta di

- concatenazione
- fusione (merging)
- riduzione

Vediamo qualche esempio

- `(number, +)` ove `+` è l'usuale addizione di numeri
- `(number, *)` ove `*` è l'usuale moltiplicazione di numeri
- `(string, +)` ove `+` è l'usuale concatenazioni di stringhe
- `(boolean, &&)` ove `&&` è l'usuale congiunzione
- `(boolean, ||)` ove `||` è l'usuale disgiunzione

8.3 Implementazione

Come facciamo a tradurre questa astrazione sottoforma di codice?

Possiamo implementare il concetto di semigruppo come una `interface`

[14] ovvero per ogni $a, b, c \in S$ vale

$$(a * b) * c = a * (b * c)$$

[15] ovvero possiamo scrivere semplicemente $a * b * c$

```
interface Semigroup<A> {
  concat: (x: A, y: A) => A
}
```

L'insieme sostegno è rappresentato dal type parameter `A` mentre l'operazione $*$ è chiamata `concat`.

L'associatività non può essere espressa nel type system di TypeScript

```
// deve valere per ogni a, b, c
concat(concat(a, b), c) = concat(a, concat(b, c))
```

Ecco come possiamo implementare il semigruppo $(number, +)$

```
const sum: Semigroup<number> = {
  concat: (x, y) => x + y
}
```

Notate che si possono definire differenti istanze di semigruppo per lo stesso insieme sostegno

```
const product: Semigroup<number> = {
  concat: (x, y) => x * y
}
```

Esercizio 8.1 *Implementare i seguenti semigruppi*

- `(string, +)`
- `(boolean, &&)`
- `(boolean, ||)`
- `(object, ...)`
  (spread operator)

L'insieme sostegno può essere costituito anche da funzioni

```
type Predicate<A> = (a: A) => boolean
```

```
const getPredicateSemigroup = <A>(
  S: Semigroup<boolean>
): Semigroup<Predicate<A>> => ({
  concat: (x, y) => a => S.concat(x(a), y(a))
})
```

Ora se definiamo una generica funzione `fold` che accetta un semigruppo come parametro [16]

```
const fold = <A>(S: Semigroup<A>) => (
  a: A,
  as: Array<A>
): A => as.reduce((a, b) => S.concat(a, b), a)
```

possiamo reimplementare alcune popolari funzioni

```
const monoidAll: Semigroup<boolean> = {
  concat: (x, y) => x && y
}

const every = <A>(p: Predicate<A>, as: Array<A>): boolean =>
  fold(monoidAll)(true, as.map(p))

const monoidAny: Semigroup<boolean> = {
  concat: (x, y) => x || y
}

const some = <A>(p: Predicate<A>, as: Array<A>): boolean =>
  fold(monoidAny)(false, as.map(p))

const monoidObject: Semigroup<Object> = {
  concat: (x, y) => ({ ...x, ...y })
}

const assign = (as: Array<Object>): Object =>
  fold(monoidObject)({}, as)
```

[16] Notate che ho bisogno anche di un valore di tipo A perchè l'array potrebbe essere vuoto

E' possibile definire una versione alternativa di `fold` nel caso non esista un valore sensato da restituire quando l'array è vuoto

```
const tryFold = <A>(S: Semigroup<A>) => (
  as: Array<A>
): Option<A> =>
  as.length === 0 ? none : some(fold(S)(as[0], as.slice(1)))

tryFold(sum)([1, 2, 3]) // some(6)
tryFold(sum)([]) // none
```

8.4 Il semigruppo duale

```
const getDualSemigroup = <A>(
  S: Semigroup<A>
): Semigroup<A> => ({
  concat: (x, y) => S.concat(y, x)
})
```

8.5 Non riesco a trovare un'istanza!

Cosa succede se, dato un tipo A, non si riesce a trovare un'operazione interna associativa?

Potete creare un'istanza di semigruppo per *ogni* tipo usando una delle seguenti costruzioni:

restituire sempre il primo argomento

```
const getFirstSemigroup = <A>(): Semigroup<A> => ({
  concat: (x, y) => x
})
```

restituire sempre l'ultimo argomento

```
const getLastSemigroup = <A>(): Semigroup<A> => ({
  concat: (x, y) => y
})
```

restituire sempre uno stesso elemento $a \in A$

```
const getConstSemigroup = <A>(a: A): Semigroup<A> => ({
  concat: () => a
})
```

Queste istanze possono sembrare banali[17] ma c'è un'altra tecnica che mantiene il contenuto informativo ed è quella di definire una istanza di semigruppo per `Array<A>`, chiamata il *semigruppo libero* (Free semigroup) di A

```
const getFreeSemigroup = <A>(): Semigroup<Array<A>> => ({
  concat: (x, y) => x.concat(y)
})
```

e poi mappare gli elementi di `A` sui singoletti di `Array<A>`

```
const of = <A>(a: A): Array<A> => [a]
```

Il semigruppo libero di A è il semigruppo i cui elementi sono tutte le possibili sequenze finite di elementi di A.

> Il semigruppo libero di A può essere visto come un modo *lazy* di concatenare elementi di A.

Anche se ho a disposizione una istanza di semigruppo per A, potrei decidere di usare ugualmente il semigruppo libero perchè

- evita di eseguire computazioni possibilmente inutili
- evita di passare in giro l'istanza di semigruppo
- permette al consumer delle mie API di stabilire la strategia di concatenazione

Esempio 8.1 *Reporters di `io-ts` (https://github.com/gcanti/io-ts)*

8.6 Semigruppi per i type constructor

In alcuni casi è possibile derivare una istanza di semigruppo per un type constructor, per esempio `Option<A>` o `Task<A>`, sfruttando una istanza per `A`. Vediamo qualche esempio

[17] Eppure a volte possono essere molto utili, in particolare `getLastSemigroup`

Option

```
const getOptionSemigroup = <A>(
  S: Semigroup<A>
): Semigroup<Option<A>> => ({
  concat: (x, y) =>
    x.fold(
      () => y,
      ax => y.fold(() => x, ay => some(S.concat(ax, ay)))
    )
})

fold(getOptionSemigroup(sum))(none, [
  some(2),
  none,
  some(3)
]) // Some(5)
```

Promise

```
const getTaskSemigroup = <A>(
  S: Semigroup<A>
): Semigroup<Task<A>> => ({
  concat: (x, y) =>
    new Task(() =>
      x
        .run()
        .then(rx => y.run().then(ry => S.concat(rx, ry)))
    )
})

const of = <A>(a: A): Task<A> =>
  new Task(() => Promise.resolve(a))

fold(getTaskSemigroup(semigroupSum))(of(0))([
  of(2),
  of(0),
  of(3)
```

```
])
  .run()
  .then(x => console.log(x)) // 5
```

8.7 Il semigruppo prodotto

Dati due semigruppi $(A, *)$ e $(B, +)$ è possibile definire il semigruppo prodotto $(A \times B, *+)$ ove

$$*+\Big((a_1, b_1), (a_2, b_2)\Big) = (a_1 * a_2, b_1 + b_2)$$

```
const getProductSemigroup = <A, B>(
  A: Semigroup<A>,
  B: Semigroup<B>
): Semigroup<[A, B]> => ({
  concat: ([ax, bx], [ay, by]) => [
    A.concat(ax, ay),
    B.concat(bx, by)
  ]
})

const str: Semigroup<string> = {
  concat: (x, y) => x + y
}

getProductSemigroup(sum, str).concat([2, 'a'], [3, 'b']) // [ 5, 'ab' ]
```

Il teorema[18] può essere generalizzato al prodotto di n semigruppi (n-tuple) e ai record.

9 Uguaglianza e ordinamento

Se `A` è totalmente ordinabile allora è possibile definire un'istanza di semigruppo su `A` usando `min` (o `max`) come operazioni

```
const meet: Semigroup<number> = {
  concat: (x, y) => Math.min(x, y)
}

const join: Semigroup<number> = {
  concat: (x, y) => Math.max(x, y)
}
```

Nota. Definire una istanza di `Setoid` per `A` equivale a definire una *partizione* di `A` in cui due elementi $x, y \in A$ appartengono alla stessa partizione se e solo se `equals(x, y) = true`.

E' possibile catturare la nozione di *totalmente ordinabile*? Per farlo prima dobbiamo catturare la nozione di *uguaglianza*.

18

Teorema 8.1 *Siano $(A, *)$ e $(B, +)$ due semigruppi, allora $P = (A \times B, * \times +)$ è un semigruppo*

Dimostrazione. P è un magma dato che $a_1 * a_2$ appartiene a A e $b_1 + b_2$ appartiene a B (per definizione dei semigruppi $(A, *)$ e $(B, +)$). Inoltre vale la proprietà associativa:

$$
\begin{aligned}
(a_1, b_1) * +((a_2, b_2) * +(a_3, b_3)) = & \quad (1)\\
(a_1, b_1) * +(a_2 * a_3, b_2 + b_3) = & \quad (2)\\
(a_1 * (a_2 * a_3), b_1 + (b_2 + b_3)) = & \quad (3)\\
((a_1 * a_2) * a_3, (b_1 + b_2) + b_3) = & \quad (4)\\
(a_1 * a_2, b_1 + b_2) * +(a_3, b_3) = & \quad (5)\\
((a_1, b_1) * +(a_2, b_2)) * +(a_3, b_3) & \quad (6)
\end{aligned}
$$

QED

9.1 Relazioni di equivalenza

I matematici parlano di *relazione di equivalenza*. Definiamo una nuova interfaccia

```
interface Setoid<A> {
  equals: (x: A, y: A) => boolean
}
```

Devono valere le seguenti leggi

riflessiva	`equals(x, x) = true` per ogni $x \in A$
simmetrica	`equals(x, y) = equals(y, x)` per ogni $x, y \in A$
transitiva	se `equals(x, y) = true` e `equals(y, z) = true` allora `equals(x, z) = true`

Esempio 9.1 *Una relazione di equivalenza per `number`*

```
const setoidNumber: Setoid<number> = {
  equals: (x, y) => x === y
}
```

9.2 Relazioni d'ordine

Possiamo catturare la nozione di ordinabile introduciamo l'interfaccia `Ord`

```
type Ordering = -1 | 0 | 1;

interface Ord<A> extends Setoid<A> {
  compare: (x: A, y: A) => Ordering
}
```

Esempio 9.2 *Una relazione d'ordine totale per `number`*

```
const ordNumber: Ord<number> = {
  ...setoidNumber,
  compare: (x, y) => (x < y ? -1 : x > y ? 1 : 0)
}
```

Ora possiamo definire l'equivalente di $\leq$

```
const leq = <A>(O: Ord<A>) => (x: A, y: A): boolean =>
  O.compare(x, y) <= 0
```

scriviamo che $x \leq y$ se e solo se `leq(ord)(x, y) = true`.
Devono valere le seguenti leggi

riflessiva	$x \leq x$ per ogni $x \in A$
antisimmetrica	se $x \leq y$ e $y \leq x$ allora $x = y$ (se non vale questa proprietà, si chiama *preordine*)
transitiva	se $x \leq y$ e $y \leq z$ allora $x \leq z$

Per essere compatibile con `Setoid` deve valere una proprietà aggiuntiva

compatibilità	`compare(x, y) = 0` se e solo se `equals(x, y) = true`

Ora possiamo definire `min` e `max` in modo generale

```
const min = <A>(O: Ord<A>) => (x: A, y: A): A =>
  O.compare(x, y) === -1 ? x : y

const max = <A>(O: Ord<A>) => (x: A, y: A): A =>
  O.compare(x, y) === -1 ? y : x
```

con i quali infine possiamo definire due nuovi combinator

```
const getMeetSemigroup = <A>(O: Ord<A>): Semigroup<A> => ({
  concat: min(O)
})

const getJoinSemigroup = <A>(O: Ord<A>): Semigroup<A> => ({
  concat: max(O)
})
```

Ora possiamo calcolare il massimo di n numeri

```
tryFold(getJoinSemigroup(ordNumber))([1, 2, 3]) // some(3)
```

Oppure selezionare la persona con l'età minore in un gruppo

```
type Person = {
  name: string
  age: number
}

const persons: Array<Person> = [
  { name: 'Giulio', age: 44 },
  { name: 'Guido', age: 47 }
]

const contramap = <A, B>(
  O: Ord<A>,
  f: (b: B) => A
): Ord<B> => ({
  equals: (x, y) => O.equals(f(x), f(y)),
  compare: (x, y) => O.compare(f(x), f(y))
})

tryFold(
  getMeetSemigroup(
    contramap(ordNumber, (p: Person) => p.age)
  )
)(persons) // some({ name: 'Giulio', age: 44 })
```

Qui abbiamo dovuto usare `tryFold` perchè non c'è un valore iniziale sensato per `fold`.

Tuttavia esistono semigruppi per i quali questo valore esiste: i *monoidi*.

10 Monoidi

Se aggiungiamo una condizione in più alla definizione di semigruppo, ovvero che esista un elemento $u \in M$ tale che per ogni elemento $m \in M$ vale

$$u * m = m * u = m$$

allora la terna $(M, *, u)$ viene detta *monoide* e l'elemento u viene detto *unità* (sinonimi: *elemento neutro*, *elemento identità*).

Teorema 10.1 *L'unità di un monoide è unica.*

Dimostrazione. Siano u_1 e u_2 due unità allora

$$u1 * u2 = u1 \tag{7}$$
$$u1 * u2 = u2 \tag{8}$$

Molti dei semigruppi che abbiamo visto possono essere estesi a monoidi

- `(number, +, 0)`
- `(number, *, 1)`
- `(string, +, "")`
- `(boolean, &&, true)`
- `(boolean, ||, false)`
- `(Object, ..., {})`

Esercizio 10.1 *Generalizzare il monoide* `(Object, ..., {})` *in modo che possa lavorare con un generico dizionario*

```
type Dictionary<A> = { [key: string]: A }
```

I monoidi sono ovunque

- Money amounts define a Monoid under summation with the null amount as neutral element
- Relative paths in a file system form a Monoid under appending
- Access rights to files form a Monoid under intersection or union of rights

10.1 Implementazione

```
interface Monoid<A> extends Semigroup<A> {
  empty: A
```

```
}
```

Come esempi non banali possiamo implementare i seguenti fatti.

Dato un tipo `A`, gli endomorfismi[19] su `A` ammettono una istanza di monoide

```
type Endomorphism<A> = (a: A) => A

const identity = <A>(a: A): A => a

const getEndomorphismMonoid = <A>(): Monoid<
  Endomorphism<A>
> => ({
  concat: (x, y) => a => x(y(a)),
  empty: identity
})
```

Se il tipo `M` ammette una istanza di monoide allora il tipo `(a: A) => M` ammette una istanza di monoide per ogni tipo `A`

```
const getFunctionMonoid = <M>(M: Monoid<M>) => <
  A
>(): Monoid<(a: A) => M> => ({
  concat: (f, g) => a => M.concat(f(a), g(a)),
  empty: () => M.empty
})
```

Come corollario otteniamo che i reducer ammettono una istanza di monoide [20]

[19] Un endomorfismo non è altro che una funzione il cui dominio e codominio coincidono

[20] Alternativamente, se i reducer non sono curried, l'istanza può essere definita manualmente

```
type Reducer<S, A> = (s: S, a: A) => S

const getReducerMonoid = <S, A>(): Monoid<
  Reducer<S, A>
> => ({
  concat: (x, y) => (s, a) => y(x(s, a), a),
  empty: (s, _) => s
})
```

```
type Reducer<S, A> = (a: A) => (s: S) => S

const getReducerMonoid = <S, A>(): Monoid<Reducer<S, A>> =>
  getFunctionMonoid(getEndomorphismMonoid<S>())<A>()
```

10.2 Monoide prodotto

```
const getProductMonoid = <A, B>(
  MA: Monoid<A>,
  MB: Monoid<B>
): Monoid<[A, B]> => ({
  ...getProductSemigroup(MA, MB),
  empty: [MA.empty, MB.empty]
})
```

Questo combinatore può essere generalizzato al prodotto di n monoidi (n-tuple) e ai record.

10.3 Non tutti i semigruppi sono monoidi

Esiste una istanza di un semigruppo che non è possibile estendere a monoide?

```
class NonEmptyArray<A> {
  constructor(readonly head: A, readonly tail: Array<A>) {}
}

const getNonEmptyArraySemigroup = <A>(): Semigroup<
  NonEmptyArray<A>
> => ({
  concat: (x, y) =>
    new NonEmptyArray(
      x.head,
      x.tail.concat([y.head]).concat(y.tail)
    )
})

getNonEmptyArraySemigroup().concat(
```

```
  new NonEmptyArray(1, [2]),
  new NonEmptyArray(3, [4, 5])
) // { head: 1, tail: [ 2, 3, 4, 5 ] }
```

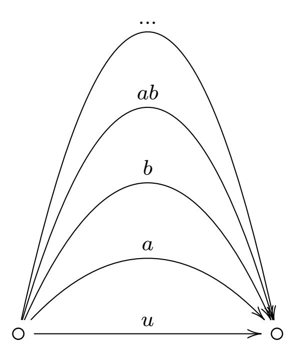
Ma non esiste nessun elemento `u: NonEmptyArray` che concatenato ad un altro `x: NonEmptyArray` dia ancora `x`.

10.4 Monoidi come categorie

Un monoide $(M, *, u)$ *assomiglia* ad una categoria

- c'è un'operazione che *compone* gli elementi
- l'operazione è associativa
- c'è il concetto di *identità*

La somiglianza non è casuale. Ad un monoide $(M, *, u)$ può essere associata una categoria con un solo oggetto, i cui morfismi sono gli elementi di M e la cui operazione di composizione è $*$.



La funzione `fold` che abbiamo definito per i semigruppi può essere ridefinita per i monoidi

```
const fold = <A>(M: Monoid<A>) => (
  as: Array<A>
): A => as.reduce((a, b) => M.concat(a, b), M.empty)
```

Notate che non c'è più bisogno di un elemento iniziale `a: A` perchè ora possiamo sfruttare `empty()`

```
const product: Monoid<number> = {
  concat: (x, y) => x * y,
  empty: 1
}

const monoidString: Monoid<string> = {
  concat: (x, y) => x + y,
  empty: ''
}

fold(monoidString)(['a', 'b', 'c']) // 'abc'
fold(product)([2, 3, 4]) // 24
fold(product)([]) // 1
```

10.5 Monoidi liberi

Cosa succede se ho un insieme X al quale non posso associare facilmente un'istanza di monoide? Esiste un'operazione su X che produce in modo *automatico* un monoide? E se si, il monoide generato che caratteristiche ha?

Consideriamo come X l'insieme costituito dalle due stringhe `'a'` e `'b'` e come operazione $*$ la giustapposizione:

$$*(a, b) = ab$$

Ovviamente quello che abbiamo non è un monoide: non c'è traccia di un elemento unità e appena applichiamo $*$ *cadiamo fuori* dall'insieme X. Possiamo però costruire il seguente monoide $M(X)$ che viene chiamato *monoide generato da* X o *monoide libero di* X:

$$M(X) = (Y, *, u)$$

ove

- $*$ è l'operazione di giustapposizione
- u è un elemento speciale che fa da unità

- $Y = u, a, b, ab, ba, aa, bb, aab, aba, ...$

Attenzione, perchè valga la proprietà associativa dobbiamo anche identificare alcuni elementi generati, ad esempio $(aa)b = a(ab)$

Gli elementi di X vengono detti *elementi generatori* di $M(X)$.

E' possibile dimostrare che:

- $M(X)$ è il *più piccolo* monoide che contiene X[21]
- $M(X)$ è *isomorfo* a $(Array < X >, concat, [])$

11 Diagramma delle algebre

$$
\begin{tikzcd}
 & Set & \\
Setoid \arrow[ur, "\texttt{equivalence}"] & & Magma \arrow[ul, "\texttt{closed operation}"'] \\
Ord \arrow[u, "\texttt{total order}"'] & & Semigroup \arrow[u, "\texttt{associativity}"'] \\
 & & Monoid \arrow[u, "\texttt{unit}"']
\end{tikzcd}
$$

[21] il termine libero si usa quando sussiste questa proprietà

DEMO

`shapes.ts`

12 Funtori applicativi

Nel capitolo sui funtori abbiamo visto come trasformare, tramite l'operazione `lift`, una computazione che non fallisce mai $f : A \to B$ in una computazione che può fallire `lift`$(f) : Option(A) \to Option(B)$, per esempio se fallisce il recupero dell'input.

Ma `lift` opera solo su funzioni unarie.

Cosa succede se abbiamo una funzione con due o più argomenti? Possiamo ancora effettuare una operazione che sia simile al lifting che già conosciamo?

Consideriamo una funzione con due argomenti

$$f : A \times B \to C$$

ove $A \times B$ indica il prodotto cartesiano degli insiemi A e B. La funzione f può essere riscritta in modo che sia una composizione di due funzioni, ognuna con un solo argomento

$$f : A \to B \to C$$

Questo processo di riscrittura prende il nome di *currying*[22].

Se F è un funtore, quello che si vorrebbe ottenere è una funzione $F(f)$ tale che

$$F(f) : F(A) \to F(B) \to F(C)$$

Proviamo a costruire $F(f)$ con i soli mezzi che abbiamo a disposizione. Siccome sappiamo che la composizione di funzioni è associativa possiamo evidenziare il secondo elemento della composizione di f vedendola come una funzione che accetta un solo parametro di tipo A e restituisce un valore di tipo $B \to C$.

$$f : A \to (B \to C)$$

ora che ci siamo ricondotti ad avere una funzione con un solo parametro, possiamo operare un lifting tramite il funtore F

$$F(f) : F(A) \to F(B \to C)$$

[22]Fu introdotta da Gottlob Frege (filosofo, logico e matematico tedesco), sviluppata da Moses Schönfinkel (logico e matematico russo), e sviluppata ulteriormente da Haskell Curry (logico e matematico americano)

Ma a questo punto siamo bloccati. Perchè non c'è nessuna operazione lecita che ci permette di passare dal tipo $F(B \to C)$ al tipo $F(B) \to F(C)$.

Il fatto che F ammetta una istanza di funtore non basta, deve avere una proprietà in più, quella cioè di ammettere una operazione che permette di spacchettare il tipo delle funzioni da B a C mandandolo nel tipo delle funzioni da $F(B)$ a $F(C)$. Indichiamo questa operazione con il nome `ap` e la corrispondente asptrazione con `Apply`.

Inoltre sarebbe oppurtuno avere un'altra operazione che, dato un valore di tipo A associa un valore di tipo $F(A)$. In questo modo, una volta ottenuta la funzione $F(f) = F(A) \to F(B) \to F(C)$ e avendo a disposizione un valore di tipo $F(A)$ (magari ottenuto da un'altra computazione) e un valore di tipo B, sono in grado di eseguire la funzione $F(f)$.

Chiamiamo questa operazione `of`.

12.1 Definizione

Sia F un funtore, allora si dice *funtore applicativo* (`Applicative`) se esistono due operazioni

```
interface Apply<F> extends Functor<F> {
  ap: <A, B>(f: F<(a: A) => B>, fa: F<A>) => F<B>
}

interface Applicative<F> extends Apply<F> {
  of: <A>(a: A) => F<A>
}
```

tali che valgono le seguenti leggi

Associative composition	`ap(ap(map(compose, f), g), h) = ap(f, ap(g, h))`
Identity	`ap(of(identity), x) = x`
Composition	`ap(ap(ap(of(compose), f), g), h) = ap(f, ap(g, h))`
Homomorphism	`ap(of(f), of(x)) = of(f(x))`
Interchange	`ap(f, of(g)) = ap(of(x => g(x)), f)`

Vediamo un esempio: il tipo `Option`

```
const of = some

class None<A> {
  readonly _tag = 'None'
  map<B>(f: (a: A) => B): Option<B> {
    return none
  }
  ap<B>(fab: Option<(a: A) => B>): Option<B> {
    return none
  }
}

class Some<A> {
  readonly _tag = 'Some'
  constructor(readonly value: A) {}
  map<B>(f: (a: A) => B): Option<B> {
    return some(f(this.value))
  }
  ap<B>(fab: Option<(a: A) => B>): Option<B> {
    return fab.map(f => f(this.value))
  }
}
```

Oppure nella sua versione statica

```
const ap = <A, B>(
  fab: Option<(a: A) => B>,
  fa: Option<A>
): Option<B> =>
  fab.fold(
    () => none,
    f => fa.fold(() => none, a => some(f(a)))
  )
```

Nota. Si noti che per applicare una funzione inserita nel contesto funtoriale ad un valore che non è nel contesto funtoriale non occorre un funtore applicativo, un funtore è sufficiente

```
const flap = <A, B>(
  fab: Option<(a: A) => B>,
  a: A
): Option<B> => fab.map(f => f(a))
```

12.2 Lifting manuale

Consideriamo la seguente funzione `sum`

```
const sum = (a: number) => (b: number): number => a + b
```

E' possibile sfruttare `of` e `ap` per ottenere il suo lifting

```
const sumOptions = (fa: Option<number>) => (
  fb: Option<number>
): Option<number> => fb.ap(fa.ap(of(sum)))
```

oppure usando `map` e `ap`

```
const sumOptions = (fa: Option<number>) => (
  fb: Option<number>
): Option<number> => fb.ap(fa.map(sum))
```

12.3 La funzione `liftA2`

L'operazione di lifting può essere facilmente generalizzata per ogni funzione[23]

```
type Function2<A, B, C> = (a: A) => (b: B) => C

const liftA2 = <A, B, C>(
  f: Function2<A, B, C>
): Function2<Option<A>, Option<B>, Option<C>> => fa => fb =>
  fb.ap(fa.map(f))

const sumOptions = liftA2(sum)
```

Analogamente è possibile definire `liftA3` per funzioni con 3 argomenti,

[23] e per ogni funtore applicativo

`liftA4`, etc ...

E' importante sottolineare che mentre abbiamo avuto bisogno di una nuova astrazione per poter operare un lifting di una funzione binaria, per operare un lifting di una funzione n-aria un funtore applicativo è sufficiente.

Esercizio 12.1 *Mostrare che se `F` ammette una istanza di `Apply` allora è possibile definire le seguenti funzioni*

```
const map2 = <A, B, C>(
  fa: F<A>,
  fb: F<B>,
  f: (a: A, b: B) => C
): F<C> => ???

const map3 = <A, B, C, D>(
  fa: F<A>,
  fb: F<B>,
  fc: F<C>,
  f: (a: A, b: B, c: C) => D
): F<D> => ???

// etc...
```

12.4 Esempi

Identity

```
const of = <A>(a: A) => new Identity(a)

class Identity<A> {
  ...
  ap<B>(fab: Identity<(a: A) => B>): Identity<B> {
    return new Identity(fab.value(this.value))
  }
}
```

Either

```
const of = right

class Left<L, A> {
  ...
  ap<B>(fab: Either<L, (a: A) => B>): Either<L, B> {
    return fab.fold<Either<L, B>>(
      l => new Left(l),
      () => new Left(this.value)
    )
  }
}

class Right<L, A> {
  ...
  ap<B>(fab: Either<L, (a: A) => B>): Either<L, B> {
    return fab.fold<Either<L, B>>(
      l => new Left(l),
      f => new Right(f(this.value))
    )
  }
}
```

Array

```
export const applicativeArray = {
  ...functorArray,
  of: <A>(a: A): Array<A> => [a],
  ap: <A, B>(
    fab: Array<(a: A) => B>,
    fa: Array<A>
  ): Array<B> =>
    fab.reduce(
      (acc, f) => acc.concat(fa.map(f)),
      [] as Array<B>
    )
}
```

IO

```
const of = <A>(a: A): IO<A> => new IO(() => a)

class IO<A> {
  ...
  ap<B>(fab: IO<(a: A) => B>): IO<B> {
    return new IO(() => fab.run()(this.run()))
  }
}
```

Task

```
const of = <A>(a: A): Task<A> =>
  new Task(() => Promise.resolve(a))

class Task<A> {
  ...
  ap<B>(fab: Task<(a: A) => B>): Task<B> {
    return new Task(() =>
      Promise.all([fab.run(), this.run()]).then(([f, a]) =>
        f(a)
      )
    )
  }
}
```

Esercizio 12.2 *Sia*

```
type Tuple<L, A> = [L, A]
```

definire una istanza di funtore applicativo per ***`Tuple`****.*

DEMO `applicative.ts`

12.5 Monoidal lax functors

La definizione di funtore applicativo che abbiamo visto è equivalente alla seguente

```
interface Monoidal<F> extends Functor<F> {
  unit: F<void>
  mult: <A, B>(fa: F<A>, fb: F<B>) => F<[A, B]>
}
```

Esercizio 12.3 *Dimostrare che sono equivalenti.*

12.6 Composizione di funtori applicativi

I funtori applicativi compongono, ovvero dati due funtori applicativi `F` e `G`, allora la composizione `F(G)` è ancora un funtore applicativo.

```
import { Option, option } from 'fp-ts/lib/Option'
import { array } from 'fp-ts/lib/Array'

export const of = <A>(a: A) =>
  new ArrayOption(array.of(option.of(a)))

export class ArrayOption<A> {
  ...
  ap<B>(fab: Array<Option<(a: A) => B>>): Array<Option<B>> {
    return array.ap(
      array.map(fab, h => (ga: Option<A>) => ga.ap(h)),
      this.value
    )
  }
}
```

13 Monadi

13.1 Come si gestiscono i side effect?

A parte alcune tipologie di programmi come i compilatori o come `prettier`, che possono essere pensati come funzioni pure (`string => string`), quasi tutti i programmi che scriviamo comportano dei side effect.

> Come è possibile modellare un programma che produce side effect con una funzione pura?

La risposta è modellando i side effect tramite effetti, ovvero valori che rappresentano il side effect. Ci sono due modi per farlo

1. definire un DSL per gli effetti
2. usare i *thunk*

DSL. Il programma

```
const sum = (a: number, b: number): number => {
  console.log(a, b) // side effect
  return a + b
}
```

viene modellato con una funzione pura modificando il codominio e restituendo una descrizione dell'effetto

```
const sum = (a: number, b: number): [number, string] => [
  a + b,
  `please log: ${a}, ${b}`
]
```

fondamentalmente creando un DSL per gli effetti.

Thunk. La computazione viene racchiusa bel body di una funzione senza argomenti

```
class IO<A> {
  constructor(readonly run: () => A) {}
  ...
}

const sum = (a: number, b: number): IO<number> =>
  new IO(() => {
    console.log(a, b)
    return a + b
  })
```

Il programma `sum`, quando viene eseguito, non provoca immediatamente il side effect ma restituisce un valore che rappresenta la computazione (detta anche *azione*).

Vediamo un altro esempio, leggere e scrivere sul `localStorage`

```
const read = (name: string): IO<string | null> =>
  new IO(() => localStorage.getItem(name))

const write = (name: string, value: string): IO<void> =>
  new IO(() => localStorage.setItem(name, value))
```

Ritorneremo più avanti a occuparci di `IO` dato che è possibile associare una istanza di *monade*.

Nella programmazione funzionale si tende a spingere i side effect ai confini del sistema (la funzione `main`) ove vengono eseguiti da un interprete ottenendo il seguente schema

system = pure core + imperative shell

Nei linguaggi *puramente funzionali* (come Haskell, PureScript o Elm) questa divisione è netta ed è imposta dal linguaggio stesso.

Un intero programma che produce un valore di tipo `A` è rappresentato da una funzione il cui codominio è `IO<A>` (o `Task<A>`).

Come faccio a scrivere la funzione `main`? Davvero si pretende di scrivere tutta l'applicazione in una unica funzione?

E' possibile applicare la tecnica *divide et impera* ovvero decomporre il problema in sotto problemi più piccoli, per poi ricomporre le soluzioni trovate per i sotto problemi.

Cosa c'è di nuovo però? Il fatto che nella programmazione funzionale come decomporre e poi ricomporre il problema non è lasciato all'istinto del programmatore, la metodologia suggerita è quella di descrivere il programma tramite strutture algebriche (monoidi, categorie, funtori, ...) che godono di buone proprietà di composizione.

C'è un ostacolo però: **il fatto che due funzioni compongano è un evento raro!**

Perchè due funzioni g e f compongano, il codominio di g deve coincidere col dominio di f

$$g : A \to B, f : B \to C$$

Ma in generale non è così.

E in particolare non sappiamo ancora come comporre gli effetti, guardate cosa può accadere con `Option`

```
const head = <A>(as: Array<A>): Option<A> =>
  as.length ? some(as[0]) : none

const inverse = (x: number): Option<number> =>
  x === 0 ? none : some(1 / x)

// program: Option<Option<number>>
const program = head([2, 3]).map(inverse)
```

Qui il risultato è incapsulato due volte in una `Option`, circostanza affatto desiderabile. Vediamo se è possibile definire una funzione che *appiattisce* il risultato, chiamiamola `flatten`

```
const flatten = <A>(
  mma: Option<Option<A>>
): Option<A> => mma.fold(() => none, identity)

// program: Option<number>
const program = flatten(head([2, 3]).map(inverse))
```

Outer	Inner	Result
`None`	`None`	`None`
`Some`	`None`	`None`
`Some`	`Some`	`Some`

Vediamo un altro esempio: scrivere la funzione `echo` in stile funzionale.

```
const getLine: IO<string> = new IO(() => process.argv[2])

const putStrLn = (s: string): IO<void> =>
  new IO(() => console.log(s))

// program: IO<IO<void>>
const program = getLine.map(putStrLn)
```

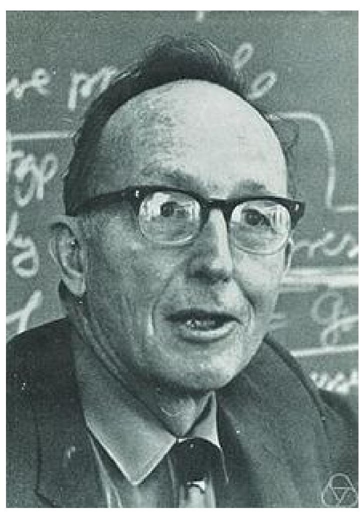
Anche in questo caso possiamo definire una funzione `flatten`

```
const flatten = <A>(mma: IO<IO<A>>): IO<A> =>
  new IO(() => mma.run().run())
```

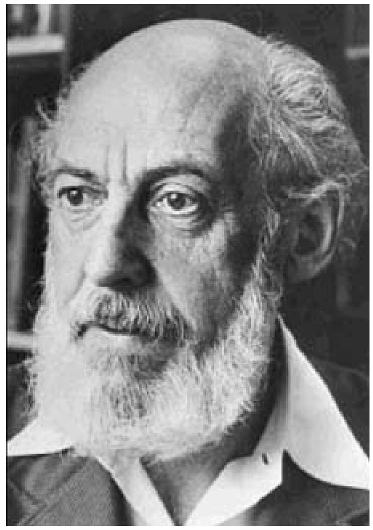
Cosa dire di `Either`, `Array` e degli altri funtori? E' possibile individuare un nuovo pattern funzionale?

Si, le monadi.

13.2 Un po' di storia

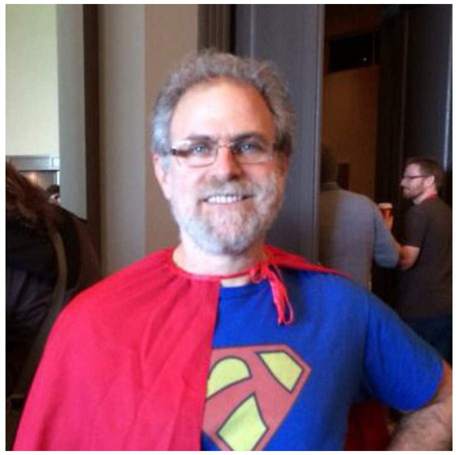


Saunders Mac Lane



Samuel Eilenberg

Eugenio Moggi is a professor of computer science at the University of Genoa, Italy. He first described the general use of monads to structure programs.



Philip Lee Wadler is an American computer scientist known for his contributions to programming language design and type theory.

13.3 Definizione

Quella seguente è una possibile definizione che si può trovare in rete:

Una monade M nella categoria $\mathcal{C}$ è un endofuntore di $\mathcal{C}$ con due operazioni aggiuntive

```
interface Monad<F> extends Functor<F> {
  of: <A>(a: A) => M<A>
  chain: <A, B>(f: (a: A) => M<B>, ma: M<A>) => M<B>
}
```

Inoltre devono valere le seguenti leggi

Left identity	`chain(f, of(x)) = f(x)`
Right identity	`chain(of, x) = x`
Associativity	`chain(g, chain(f, ma)) = chain(x => chain(g, f(x)), mx)`

Possibili sinonimi di `of`[24] sono `return`, `pure` e `point`, sinonimi di `chain` sono `bind` e `flatMap`.

[24]Nome contenuto nella specifica https://github.com/fantasyland/fantasy-land

- perchè ci sono esattamente quelle due funzioni?
- perchè hanno quelle firme?
- perchè devono valere quelle leggi?

Per rispondere a queste domande introduciamo un concetto equivalente a quello di monade: le *categorie di Kleisli*

13.4 Categorie di Kleisli

Heinrich Kleisli (Swiss mathematician)

Sia M un endofuntore nella categoria $\mathcal{C}$ e si considerino i due morfismi $g : A \to M(B)$, $f : B \to M(C)$

$$
\begin{tikzcd}
 & M(B) & M(C) \\
A \arrow[ur, "g"] & B \arrow[ur, "f"] & C
\end{tikzcd}
$$

Chiamiamo *Kleisli arrow* i morfismi come g ed f, ovvero i morifismi il cui target è l'immagine di M.

Una Kleisli arrow $f : A \to M(B)$ può essere interpretata come un programma che accetta un input di tipo A e che produce un output di tipo B insieme ad un effetto di tipo M

Le Kleisli arrows $g : A \to M(B)$, $f : B \to M(C)$ non compongono rispetto a $\circ$, l'operazione di composizione della categoria $\mathcal{C}$, poichè $M(B)$ è diverso da B.

Si consideri allora la seguente costruzione $K_{\mathcal{C}}$

$$A \xrightarrow{g'} B \xrightarrow{f'} C$$

ove

- $A, B, C, \ldots$ sono gli oggetti di $\mathcal{C}$
- esiste una morfismo $m' : A \to B$ in $K_{\mathcal{C}}$ se e solo se esiste un morfismo $m : A \to M(B)$ in $\mathcal{C}$

Definire una buona operazione di composizione per le Kleisli arrow in $\mathcal{C}$, indichiamola con $\bullet$, vuol dire imporre che $K_{\mathcal{C}}$ sia una categoria.

Per dimostrare che $K_{\mathcal{C}}$ è una categoria dobbiamo definire una operazione di composizione, indichiamola con $\diamond$, e dimostrare che valgono le leggi categoriali (identità sinistra, identità destra e associatività).

Composizione Se $K_{\mathcal{C}}$ è una categoria allora deve esistere un morfismo $f' \diamond g' : A \to C$. Ma allora il corrispondente morfismo $f \bullet g$ in $\mathcal{C}$ deve avere come sorgente A e come target $M(C)$, ovvero $h : A \to M(C)$. Proviamo a costruirlo

$$
\begin{tikzcd}
 & & M(M(C)) \arrow[d, "flatten"] \\
 & M(B) \arrow[ur, "M(f)"] & M(C) \\
A \arrow[ur, "g"] \arrow[urr, dashed, "f \bullet g"] & B \arrow[ur, "f"] & C
\end{tikzcd}
$$

$$
\begin{tikzcd}
A \arrow[r, "g'"] \arrow[rr, bend right, "f' \diamond g'"'] & B \arrow[r, "f'"] & C
\end{tikzcd}
$$

$$f \bullet g = flatten \circ M(f) \circ g$$

Morfismi identità Se $K_{\mathcal{C}}$ è una categoria allora per ogni A deve esistere un morfismo $1'_A : A \to A$, perciò deve esistere un morfismo $\eta_A : A \to M(A)$ in $\mathcal{C}$.

$$\begin{tikzcd} M(A) \\ A \arrow[u, "\eta_A"'] \end{tikzcd}$$

$$\begin{tikzcd} A \arrow[loop above, "1'_A"] \end{tikzcd}$$

Identità sinistra e destra

$$\begin{tikzcd}
& M(M(C)) \arrow[d, bend left, "flatten"] \\
M(B) \arrow[ur, "M(f)"] & M(C) \arrow[u, bend left, "M(\eta_C)"] \\
B \arrow[u, "\eta_B"] \arrow[ur, "f"] & C \arrow[u, "\eta_C"']
\end{tikzcd}$$

$$\begin{tikzcd}
B \arrow[loop above, "1'_B"] \arrow[r, "f'"] & C \arrow[loop above, "1'_C"]
\end{tikzcd}$$

- $1'_B \diamond f' = f'$ implica $\eta_B \bullet f = f$ ovvero `chain(f, of(b)) = f(b)`
- $f' \diamond 1'_C = f'$ implica $f \bullet \eta_C = f$ ovvero `chain(of, c) = c`

Associatività

$$
\begin{tikzcd}
 & M(B) & M(C) & M(D) \\
A \arrow[ur, "g"] \arrow[urr, dashed, "f\bullet g"] & B \arrow[ur, "f"] \arrow[urr, dashed, "h\bullet f"] & C \arrow[ur, "h"] & D
\end{tikzcd}
$$

$$
\begin{tikzcd}
A \arrow[r, "g'"] \arrow[rr, bend right, "f'\diamond g'"'] & B \arrow[r, "f'"] \arrow[rr, bend right, "f'\diamond g'"'] & C \arrow[r, "h'"] & D
\end{tikzcd}
$$

$$h \diamond (f \diamond g) = (h \diamond f) \diamond g$$

ovvero

```
chain(h, chain(f, mb)) = chain(b => chain(h, f(b)), mb)
```

13.5 Ricapitolando

Perchè le categorie sono importanti? Perchè sono alla base del concetto di composizione e di monade.

Cos'è una monade? M è una monade quanto le funzioni $A \to M(B)$, che rappresentano un programma che ha per input A, per output B e che produce un effetto M, sono i morfismi di una categoria.

Perchè le monadi sono importanti? Perchè se M è una monade posso comporre i programmi $A \to M(B)$ tra loro.

Una monade per ogni occasione.

- eseguire un'azione sincrona? monade `IO`
- eseguire computazioni asincrone? monade `Task`
- leggere una configurazione? monade `Reader`
- scrivere su un log? monade `Writer`
- gestire lo stato? monade `State`
- gestire gli errori? monade `Option` o `Either`
- gestire risultati non deterministici? monade `Array`

13.6 Esempi

Identity

```
const of = <A>(a: A) => new Identity(a)

class Identity<A> {
  ...
  chain<B>(f: (a: A) => Identity<B>): Identity<B> {
    return f(this.value)
  }
}
```

Option

```
const of = some

class None<A> {
  ...
  chain<B>(f: (a: A) => Option<B>): Option<B> {
    return none
  }
}

class Some<A> {
```

```
  ...
  chain<B>(f: (a: A) => Option<B>): Option<B> {
    return f(this.value)
  }
}
```

Either

```
const of = right

class Left<L, A> {
  ...
  chain<B>(f: (a: A) => Either<L, B>): Either<L, B> {
    return left(this.value)
  }
}

class Right<L, A> {
  ...
  chain<B>(f: (a: A) => Either<L, B>): Either<L, B> {
    return f(this.value)
  }
}
```

Array

```
const monadArray = {
  ...
  chain: <A, B>(
    f: (a: A) => Array<B>,
    fa: Array<A>
  ): Array<B> =>
    fa.reduce((acc, a) => acc.concat(f(a)), [] as Array<B>)
}
```

IO

```
const of = <A>(a: A): IO<A> => new IO(() => a)
```

```
class IO<A> {
  ...
  chain<B>(f: (a: A) => IO<B>): IO<B> {
    return new IO(() => f(this.run()).run())
  }
}
```

Task

```
const of = <A>(a: A): Task<A> =>
  new Task(() => Promise.resolve(a))

class Task<A> {
  ...
  chain<B>(f: (a: A) => Task<B>): Task<B> {
    return new Task(() => this.run().then(a => f(a).run()))
  }
}
```

Esercizio 13.1 *Sia*

```
type Tuple<L, A> = [L, A]
```

, definire una istanza di monade per ***`Tuple`***.

13.7 `Task vs Promise`

`Task` è una astrazione simile a `Promise`, la differenza chiave è che `Task` rappresenta una computazione asincrona mentre `Promise` rappresenta solo un risultato (ottenuto in maniera asincrona).

Se abbiamo un `Task`

- possiamo far partire la computazione che rappresenta (per esempio una richiesta network)
- possiamo scegliere di non far partire la computazione
- possiamo farlo partire più di una volta (e potenzialmente ottenere risultati diversi)

- mentre la computazione si sta svolgendo, possiamo notificagli che non siamo più interessati al risultato e la computazione può scegliere di terminarsi da sola
- quando la computazione finisce otteniamo il risultato

Se abbiamo una `Promise`

- la computazione si sta già svolgendo (o è addirittura già finita) e non abbiamo controllo su questo
- quando è disponibile otteniamo il risultato
- due consumatori della stessa `Promise` ottengono lo stesso risultato

13.8 Derivazione di `map`

L'operazione `map` può essere derivata da `chain` e `of`

```
const map = <A, B>(f: (a: A) => B) => (
  fa: Option<A>
): Option<B> => fa.chain(a => of(f(a)))
```

13.9 Derivazione di `ap`

L'operazione `ap` può essere derivata da `chain` e `map`

```
const ap = <A, B>(fab: Option<(a: A) => B>) => (
  fa: Option<A>
): Option<B> => fab.chain(f => fa.map(f))
```

13.10 Esecuzione parallela e sequenziale

```
// par-seq.ts

const liftA2 = <A, B, C>(
  f: (a: A) => (b: B) => C
): ((
  fa: Task<A>
) => (fb: Task<B>) => Task<C>) => fa => fb =>
```

```
  fb.ap(fa.map(f))

const sumTasks = liftA2(
  (a: number) => (b: number): number => a + b
)

const delay = (n: number) => <A>(a: A): Task<A> =>
  new Task(
    () =>
      new Promise(resolve => {
        setTimeout(() => resolve(a), n)
      })
  )

const oneSec = delay(1000)

sumTasks(oneSec(1))(oneSec(3))
  .run()
  .then(x => console.log(x))
```

Eseguendo il codice mostrando il tempo di esecuzione otteniamo [25]

```
$ time ts-node par-seq.ts

3

real    0m1.383s
user    0m0.327s
sys     0m0.058s
```

Il che mostra che le computazioni asincrone vengono eseguite in modo concorrente.

Se però come implementazione di `ap` per `Task` scegliamo quella derivata da `chain` otteniamo

```
$ time ts-node par-seq.ts
```

[25] `ts-node` è un wrapper di `node` in grado di eseguire codice TypeScript

```
3

real    0m2.402s
user    0m0.342s
sys     0m0.063s
```

Che cosa è successo? La spiegazione è che l'implementazione di `ap` derivata da `chain` è sempre **sequenziale**.

13.11 Trasparenza referenziale

```
const readFile = (filename: string): IO<string> =>
  new IO(() => fs.readFileSync(filename, 'utf-8'))

const writeFile = (
  filename: string,
  data: string
): IO<void> =>
  new IO(() =>
    fs.writeFileSync(filename, data, { encoding: 'utf-8' })
  )

const log = (message: string): IO<void> =>
  new IO(() => console.log(message))

const program1 = readFile('file.txt')
  .chain(log)
  .chain(() => writeFile('file.txt', 'hello'))
  .chain(() => readFile('file.txt'))
  .chain(log)
```

L'azione `readFile('file.txt')` è ripetuta due volte ma dato che vale la trasparenza referenziale possiamo catturare l'azione in una costante

```
const read = readFile('file.txt').chain(log)

const program2 = read
  .chain(() => writeFile('file.txt', 'foo'))
  .chain(() => read)
```

Possiamo anche definire un combinatore e sfruttarlo per rendere più compatto il codice

```
const aba = <A, B>(a: IO<A>, b: IO<B>): IO<A> =>
  a.chain(() => b).chain(() => a)

const program3 = aba(read, writeFile('file.txt', 'foo'))
```

DEMO
`game.ts`

13.12 Le monadi non compongono

In generale le monadi non compongono, ovvero date due istanze di monade, una per `M<A>` e una per `N<A>`, allora a `M<N<A>>` non è detto che possa ancora essere associata una istanza di monade.

Che non compongano in generale però non vuol dire che non esistano dei casi particolari ove questo succede.

Vediamo qualche esempio, se `M` ha una istanza di monade allora ammettono una istanza di monade i seguenti tipi

- `OptionT<M, A> = M<Option<A>>`
- `EitherT<M, L, A> = M<Either<L, A>>`

Notate come questi tipi collassino in quelli già conosciuti quando il type constructor `M` è `Identity`

- `Option<A> = OptionT<Identity, A>`
- `Either<L, A> = EitherT<Identity, L, A>`

`OptionT` e `EitherT` sono esempi di *monad transformer*.

14 Algebraic Data Types

Un *Algebraic Data Type* (o ADT) è un tipo composto da product e/o sum types, anche innestati.

14.1 Product types

Definizione 14.1 *Un product type è una collezione di tipi A_i indicizzati da un insieme I.*

Un product type è isomorfo[26] al prodotto cartesiano $\prod_i A_i$.

Esponenti notevoli di questa famiglia sono le n-tuple, ove I è un intervallo non vuoto dei numeri naturali [27]

```
type Tuple1 = [string]
type Tuple2 = [string, number]
type Tuple3 = [string, number, boolean]
```

e i record, ove I è una collezione di label [28]

```
type Person = {
  name: string,
  age: number
}
```

`Tuple2` e `Person` sono isomorfi tra loro e al prodotto cartesiano $string \times number$.

$$f : \texttt{Tuple2} \to \texttt{Person}$$

$$f(\texttt{[name, age]}) = \{\ \texttt{name, age}\ \}$$

$$f^{-1} : \texttt{Person} \to \texttt{Tuple2}$$

[26] Due insiemi A e B sono isomorfi se esiste una funzione $f : A \to B$ iniettiva e suriettiva, ovvero se esiste una funzione $f^{-1} : B \to A$, detta *funzione inversa* di f, tale che $f \circ f^{-1} = identity$

[27] $\{0\}$ per `Tuple1`, $\{0, 1\}$ per `Tuple2`, $\{0, 1, 2\}$ per `Tuple2`

[28] $\{"name", "age"\}$ per `Person`

$$f^{-1}(\{\ \texttt{name, age}\ \}) = \texttt{[name, age]}$$

L'isomorfismo è evidente se si implementa `Person` con una classe

```
class Person {
  name: string,
  age: number
  constructor(name: string, age: number) {
    this.name = name
    this.age = age
  }
}
```

in cui `constructor` realizza la funzione f.

Perchè si chiamano product types? Se indichiamo con $\|A\|$, detta *cardinalità* o *ordine* di A, il numero di elementi dell'insieme A è facile convincersi che vale la seguente formula

$$\|A \times B\| = \|A\| * \|B\|$$

ovvero la cardinalità del prodotto cartesiano è il prodotto delle cadinalità.

```
type Hour = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12
type Period = 'AM' | 'PM'
type Clock = [Hour, Period]
```

Il tipo `Clock` ha $12 * 2 = 24$ abitanti.

14.2 Sum types

Così come i product types sono analoghi ai prodotti cartesiani di insiemi, i sum types sono analoghi alle unioni di insiemi disgiunti

```
type Action =
  | { type: 'INCREMENT' }
  | { type: 'DECREMENT' }
```

Product types e sum types possono essere mischiati

```
type Action =
  | { type: 'ADD_TODO', text: string }
  | { type: 'UPDATE_TODO', id: number, text: string, completed: boolean }
  | { type: 'DELETE_TODO', id: number }
```

Il tipo `Array<A>` può essere interpretato come sum type

```
type Array<A> = [] | [A] | [A, A] | [A, A, A] | ...
```

Esempio 14.1 *Linked lists*[29]

```
type List<A> =
  | { type: 'Nil' }
  | { type: 'Cons', head: A, tail: List<A> }
```

Esempio 14.2 *Binary trees*[30]

[29] E' possibile definire una istanza di funtore per `List<A>`

```
const map = <A, B>(f: (a: A) => B) => (
  fa: List<A>
): List<B> => {
  switch (fa.type) {
    case 'Nil':
      return { type: 'Nil' }
    case 'Cons':
      return {
        type: 'Cons',
        head: f(fa.head),
        tail: map(f)(fa.tail)
      }
  }
}
```

[30] E' possibile definire una istanza di funtore per `Tree<A>`

```
const map = <A, B>(f: (a: A) => B) => (
  fa: Tree<A>
): Tree<B> => {
  switch (fa.type) {
    case 'Empty':
      return { type: 'Empty' }
    case 'Node':
```

```
type Tree<A> =
  | { type: 'Empty' }
  | {
      type: 'Node'
      left: Tree<A>
      value: A
      right: Tree<A>
    }
```

Perchè si chiamano sum types? E' facile convincersi che la cadinalità di un sum type è la somma delle cardinalità dei suoi membri

$$\|A|B\| = \|A\| + \|B\|$$

Il tipo `Option<boolean>` ha $1 + 2 = 3$ abitanti.

```
      return {
        type: 'Node',
        left: map(f)(fa.left),
        value: f(fa.value),
        right: map(f)(fa.right)
      }
  }
}
```

15 Make impossible states irrepresentable

Vediamo un'altra tecnica per ottenere type safety, questa volta addirittura per costruzione.

Sappiamo che la funzione `head` è parziale

```
const head = <A>(xs: Array<A>): A => xs[0]
```

e che per renderla totale occorre modificare il codominio

```
const head = <A>(xs: Array<A>): Option<A> =>
  xs.length > 0 ? some(xs[0]) : none
```

Tuttavia questo ci obbliga ad usare `Option`.

Un'altra opzione è quella di cambiare il dominio invece che estendere il codominio

15.1 Il tipo `NonEmptyArray`

```
class NonEmptyArray<A> {
  constructor(readonly head: A, readonly tail: Array<A>) {}
}

const head = <A>(fa: NonEmptyArray<A>): A => fa.head
```

15.2 Il tipo `Zipper`

Supponiamo di dover modellare la seguente struttura dati

> una lista non vuota di elementi di cui uno è considerato la selezione corrente

Un modello semplice potrebbe essere questo

```
type Selection<A> = {
  items: Array<A>
  current: number
}
```

Tuttavia questo modello ha diversi difetti

- la lista può essere vuota
- l'indice può essere out of range

Uno `Zipper` invece è un modello perfetto e type safe per il problema

```
type Zipper<A> = {
  prev: Array<A>
  current: A
  next: Array<A>
}
```

15.3 Smart constructors

Consideriamo la funzione `inverse`

```
const inverse = (x: number): Option<number> =>
  x === 0 ? none : some(1 / x)
```

Un altro modo per ottenere lo stesso grado di type safety senza avere una funzione parziale è l'utilizzo degli *smart constructors*.

In pratica si fa in modo che `Option` non compaia a valle, nel codominio di `inverse`, ma a monte, in fase di creazione dell'input `x`.

In generale, se voglio rappresentare un raffinamento di un tipo `A` (come per esempio il fatto che sia un numero diverso da zero), faccio in modo che il suo costruttore non sia invocabile al di fuori del suo modulo e fornisco un costruttore alternativo che però restituisce una `Option<A>` dato che a runtime verrà effettuato il controllo che il raffinamento sussista davvero.

```
class NonZero {
  // private
  private constructor(readonly value: number) {}
  // smart constructor
  static create(value: number): Option<NonZero> {
    return value === 0 ? none : some(new NonZero(value))
  }
}

const inverse = (x: NonZero): number => 1 / x.value
```

In questo modo spesso si spingono i controlli a runtime là dove dovrebbe essere il loro posto naturale: ai confini del sistema, dove vengono fatte tutte le validazioni dell'input.

16 Foldable

`Foldable` rappresenta una struttura che può essere ridotta tramite l' operazione `reduce` (sinonimi: `reduceLeft`, `foldLeft`, `foldl`)

16.1 Definizione

```
interface Foldable<F> {
  reduce: <A, B>(fa: F<A>, b: B, f: (b: B, a: A) => B) => B
}
```

Definizioni equivalenti[31] di `Foldable` coinvolgono l'operazione

```
reduceRight: <A, B>(fa: F<A>, b: B, f: (a: A, b: B) => B) => B
```

(sinonimi: `foldRight`, `foldr`) oppure l'operazione

```
foldMap: <M>(M: Monoid<M>) =>
  <A>(fa: F<A>, f: (a: A) => M) => M
```

Un modo per afferrare il concetto di `Foldable` è che una struttura che ammette una sua istanza è in grado di essere rappresentata sotto forma di array.

Infatti è sempre possibile definire la seguente funzione

```
const toArray = <F>(F: Foldable<F>) => <A>(fa: F<A>): Array<A>
```

Esercizio 16.1 *Perchè? Suggerimento: usare `foldMap`.*

E' importante notare che affinchè sia possibile scrivere una istanza di `Foldable` è necessario che la produzione di valori di tipo `A` abbia un ordine deterministico.

Esercizio 16.2 *`Option<A>` ammette una istanza di `Foldable`, qual'è la sua rappresentazione come array?*

[31] ovvero ogni operazione può essere derivata da una qualsiasi delle altre

16.2 Differenze tra `reduceLeft` e `reduceRight`

`reduceLeft` è *associativa a sinistra*

$$reduce(b, f, [x_1, x_2, \ldots, x_n]) == f(\ldots f(f(b, x_1), x_2), \ldots x_n)$$

mentre `reduceRight` è *associativa a destra*

$$reduceRight(b, f, [x_1, x_2, \ldots, x_n]) == f(f(\ldots f(b, x_n) \ldots, x_2), x_1)$$

Esempio 16.1 *Ridurre* ***`Array<string>`***

```
reduce(["a", "b", "c"], "", (b, a) => b + a) // "abc"

reduceRight(["a", "b", "c"], "", (a, b) => b + a) // "cba"
```

16.3 `Foldable` e `Functor`

`Foldable` e `Functor` sono indipendenti, ovvero esistono strutture che ammettono una istanza di `Foldable` ma non una di `Functor` e viceversa.

`Task<A>` ammette una istanza di `Functor` ma non di `Foldable`.

La seguente struttura dati

```
class Weird<A> {
  constructor(
    readonly value: A,
    readonly endo: (a: A) => A
  ) {}
}
```

ammette una istanza di `Foldable`

```
const reduce = <A, B>(
  fa: Weird<A>,
  b: B,
  f: (b: B, a: A) => B
): B => f(b, fa.endo(fa.value))
```

ma non di `Functor` perchè `A` compare in posizione controvariante.

16.4 Esempi

Identity

```
const reduce = <A, B>(
  fa: Identity<A>,
  b: B,
  f: (b: B, a: A) => B
): B => f(b, fa.value)
```

Option

```
const reduce = <A, B>(
  fa: Option<A>,
  b: B,
  f: (b: B, a: A) => B
): B => (isSome(fa) ? f(b, fa.value) : b)
```

Either

```
const reduce = <L, A, B>(
  fa: Either<L, A>,
  b: B,
  f: (b: B, a: A) => B
): B => (isRight(fa) ? f(b, fa.value) : b)
```

Array

```
const reduce = <A, B>(
  fa: Array<A>,
  b: B,
  f: (b: B, a: A) => B
): B => fa.reduce(f, b)
```

Esercizio 16.3 *Mostrare che `reduce`, `reduceRight` e `foldMap` sono definizioni equivalenti per `Foldable`.*

Esercizio 16.4 *Definire una istanza di `Foldable` per `Tree<A>`*

```
type Tree<A> =
  | { type: 'Empty' }
  | {
      type: 'Node'
      left: Tree<A>
      value: A
      right: Tree<A>
    }
```

Esercizio 16.5 *E' possibile definire una istanza di `Foldable` per `Set<A>`?*

Esercizio 16.6 *E' possibile definire una istanza di `Foldable` per `Task<A>`?*

16.5 I `Foldable` compongono

Dati due type constructor `F<A>` e `G<A>` tali che ammettono una istanza di `Foldable`, allora la composizione `F<G<A>>` ammette una istanza di `Foldable`.

Esempio 16.2 *`ArrayOption<A>`*

```
const reduce = <A, B>(
  fa: ArrayOption<A>,
  b: B,
  f: (b: B, a: A) => B
): B => fa.value.reduce((b, o) => option.reduce(o, b, f), b)

reduce(
  new ArrayOption([some("a"), none, some("b")]),
  "",
  (b, a) => b + a
) // "ab"
```

DEMO
`wizard.ts`

17 Traversable

`Traversable` rappresenta una struttura che può essere attraversata da sinistra a destra, eseguendo una azione per ogni elemento.

17.1 Definizione

```
interface Traversable<T> extends Functor<T>, Foldable<T> {
  traverse: <F>(F: Applicative<F>) =>
    <A, B>(ta: T<A>, f: (a: A) => F<B>) => F<T<B>>
}
```

Alternativamente `Traversable` può essere definita tramite la funzione

```
sequence: <F>(F: Applicative<F>) => <A>(tfa: T<F<A>>) => F<T<A>>
```

Una istanza di `Traversable` deve soddisfare le seguenti leggi

naturality	`at(traverse(F1)(f, ta)) = traverse(F2)(compose(at, f), ta)` per ogni *applicative transformation* `at`
identity	`traverse(F)(a => new Identity(a), ta) = new Identity(ta)`
composition	`traverse(F)(a => compose(fa => A.map(g, fa), f), ta) = F.map(tfa => traverse(A)(g, tfa), traverse(F)(f, ta))`

Una applicative transformation è una funzione con la seguente firma

```
at: (F: Applicative<F>, g: Applicative<G>) =>
  <A>(fa: F<A>) => G<A>
```

17.2 Esempi

Identity.

```
function traverse<F>(
```

```
  F: Applicative<F>
): <A, B>(
  ta: Identity<A>,
  f: (a: A) => F<FB>
) => F<Identity<B>> {
  return (ta, f) => F.map(identity.of, f(ta.value))
}
```

Array.

```
const snoc = <A>(as: Array<A>) => (a: A): Array<A> =>
  as.concat([a])

function traverse<F>(
  F: Applicative<F>
): <A, B>(
  ta: Array<A>,
  f: (a: A) => F<B>
) => F<Array<B>> {
  const snocLifted = liftA2(F)(snoc)
  return (ta, f) =>
    reduce(ta, F.of([]), (fab, a) => snocLifted(fab)(f(a)))
}
```

Esercizio 17.1 *Definire una istanza di `Traversable` per `Option<A>`*

Esercizio 17.2 *Definire una istanza di `Traversable` per `Either<L, A>`*

Esercizio 17.3 *E' possibile definire una istanza di `Traversable` per `Task<A>`?*

17.3 La funzione `sequence`

Da `traverse` è possibile derivare la seguente funzione [32]

```
sequence: <F, T>(F: Applicative<F>, T: Traversable<T>) =>
  (tfa: T<F<A>>) => F<T<A>>
```

[32] e viceversa

From	To
`Array<Option<A>>`	`Option<Array<A>>`
`Either<L, IO<A>>`	`IO<Either<L, A>>`
`Array<Either<L, A>>`	`Either<L, Array<A>>`
`Either<L, Task<A>>`	`Task<Either<L, A>>`

17.4 I `Traversable` compongono

Ovvero se sia `F<A>` che `G<A>` ammettono una istanza di `Traversable`, allora `F<G<A>>` ammette una istanza di `Traversable`

DEMO
`traversable.ts`

18 Alternative

18.1 Alt

```
interface Alt<F> extends Functor<F> {
  alt: <A>(fx: F<A>, fy: F<A>) => F<FA>
}
```

associativity	`A.alt(A.alt(a, b), c) = A.alt(a, A.alt(b, c))`
distributivity	`A.map(f, A.alt(a, b)) = A.alt(A.map(f, a), A.map(f, b))`

18.2 Plus

```
interface Plus<F> extends Alt<F> {
  zero: <A>() => F<A>
}
```

right identity	`P.alt(a, P.zero()) = a`
left identity	`P.alt(P.zero(), a) = a`
annihilation	`P.map(f, P.zero()) = P.zero()`

18.3 Alternative

```
interface Alternative<F> extends Applicative<F>, Plus<F> {}
```

distributivity	`A.ap(A.alt(a, b), c) = A.alt(A.ap(a, c), A.ap(b, c))`
annihilation	`A.ap(A.zero(), a) = A.zero()`

18.4 Esempi

Option.

```
const alt = <A>(fx: Option<A>, fy: Option<A>): Option<A> => {
  return isSome(fx) ? fx : fy
```

```
}

const zero = <A>(): Option<A> => {
  return none
}
```

Either. `Either` possiede una istanza di `Alt`

```
const alt = <L, A>(
  fx: Either<L, A>,
  fy: Either<L, A>
): Either<L, A> => {
  return isRight(fx) ? fx : fy
}
```

Array.

```
const zero = <A>(): Array<A> => []

const alt = concat
```

DEMO
`router.ts`

19 Diagramma delle type class

$$
\begin{tikzcd}
Semigroup & Monoid \arrow[l] & & \\
Setoid & Ord \arrow[l] & & \\
& Alt \arrow[ddl] & Plus \arrow[l] & Alternative \arrow[l] \arrow[dl] \\
& Apply \arrow[dl] & Applicative \arrow[l] & Monad \arrow[l] \arrow[dl] \\
Functor & & Chain \arrow[ul] & \\
& Extend \arrow[ul] & Comonad \arrow[l] & \\
Foldable & Traversable \arrow[l] \arrow[uul] & &
\end{tikzcd}
$$

20 Trasformazioni naturali

In Teoria delle Categorie, una *trasformazione naturale* offre una modo per trasformare un funtore in un altro rispettando la struttura interna, ovvero la composizione di morfismi, delle categorie coinvolte.

Una trasformazione naturale può essere considerata come un morfismo tra funtori. In effetti questa intuizione può essere formalizzata definendo la *categoria dei funtori*: date due categorie $\mathcal{C}$ e $\mathcal{D}$, i funtori tra $\mathcal{C}$ e $\mathcal{D}$ costituiscono gli oggetti mentre i morfismi sono le trasformazioni naturali tra i funtori.

Le trasformazioni naturali, dopo le categorie e i funtori, sono uno dei più importanti concetti in Teoria delle Categorie e perciò appaiono nella maggior parte delle sue applicazioni.

20.1 Definizione

Siano F e G due funtori tra le categorie $\mathcal{C}$ e $\mathcal{D}$, allora una *trasformazione naturale* η da F a G è una famiglia di morfismi tale che

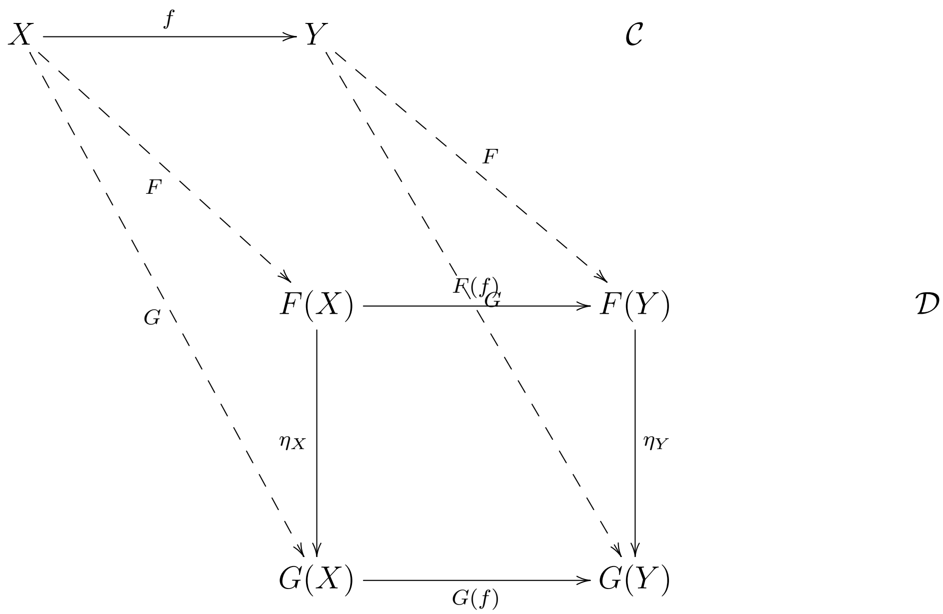
- ad ogni oggetto X in $\mathcal{C}$ è associato un morfismo $\eta_X : F(X) \to G(X)$ tra oggetti di $\mathcal{D}$. Il morfismo η_X è chiamato la *componente* di η in X.

- le componenti devono essere tali che per ogni morfismo $f : X \to Y$ in $\mathcal{C}$ valga
$$\eta_y \circ F(f) = G(f) \circ \eta_X$$

L'ultima equazione può essere rappresentata dal seguente *diagramma commutativo*

$$\begin{tikzcd}
F(X) \arrow[r, "F(f)"] \arrow[d, "\eta_X"'] & F(Y) \arrow[d, "\eta_Y"] \\
G(X) \arrow[r, "G(f)"'] & G(Y)
\end{tikzcd}$$

Ecco lo stesso diagramma ma con l'aggiunta degli oggetti X, Y e il morfismo f appartenenti alla categoria $\mathcal{C}$



Dal punto di vista implementativo una trasformazione naturale ha il seguente tipo

```
type NaturalTransformation<F, G> = <A>(fa: F<A>) => G<A>
```

20.2 Esempi

20.2.1 Da Option<A> a Array<A>

```
const fromOption = <A>(fa: Option<A>): Array<A> =>
  fa.fold([], a => [a])
```

20.2.2 Da Array<A> a Option<A>

```
const head = <A>(fa: Array<A>): Option<A> =>
  fromNullable(fa[0])
```

20.2.3 Da Either<L, A> a Option<A>

```
const fromEither = <L, A>(fa: Either<L, A>): Option<A> =>
  fa.fold(() => none, some)
```

20.2.4 Da IO<A> a Task<A>

```
const fromIO = <A>(fa: IO<A>): Task<A> =>
  new Task(() => Promise.resolve(fa.run()))
```

21 Gestire lo stato in modo funzionale: la monade `State`

Se sono bandite le mutazioni, come è possibile gestire lo stato? Cambiare lo stato in modo compatibile alla programmazione funzionale vuol dire restituirne una nuova copia modificata.

Il modello che descrive nel modo più generale un cambiamento di stato è quello definito dalla seguente firma

```
(s: S) => [A, S]
```

ove `S` è il tipo dello stato e `A` è il tipo del valore restituito dalla computazione.

Definiamo l'istanza di monade

```
class State<S, A> {
  constructor(readonly run: (s: S) => [A, S]) {}
  // monad instance
  map<B>(f: (a: A) => B): State<S, B> {
    return this.chain(a => of(f(a))) // <= derived
  }
  ap<B>(fab: State<S, (a: A) => B>): State<S, B> {
    return fab.chain(f => this.map(f)) // <= derived
  }
  chain<B>(f: (a: A) => State<S, B>): State<S, B> {
    return new State(s => {
      const [a, s1] = this.run(s)
      return f(a).run(s1)
    })
  }
  // utils
  eval(s: S): A {
    return this.run(s)[0]
  }
  exec(s: S): S {
    return this.run(s)[1]
  }
}
```

```
const of = <S, A>(a: A): State<S, A> =>
  new State(s => [a, s])
```

21.1 Combinatori

get

```
/** Get the current state */
const get = <S>(): State<S, S> =>
  new State(s => [s, s])
```

put

```
/** Set the state */
const put = <S>(s: S): State<S, undefined> =>
  new State(() => [undefined, s])
```

modify

```
/** Modify the state by applying a function to the current state */
const modify = <S>(
  f: (s: S) => S
): State<S, undefined> => new State(s => [undefined, f(s)])
```

gets

```
/** Get a value which depends on the current state */
const gets = <S, A>(f: (s: S) => A): State<S, A> =>
  new State(s => [f(s), s])
```

Vediamo qualche semplice esempio

```
/** `of` set the result value but leave the state unchanged */
of('foo').run(1) // [ 'foo', 1 ]

/** `get` set the result value to the state and leave the state unchanged */
get().run(1) // [ 1, 1 ]
```

```
/** `put` set the result value to `undefined` and set the state value */
put(5).run(1) // [ undefined, 5 ]

const inc = (n: number): number => n + 1

modify(inc).run(1) // [ undefined, 2 ]

gets(inc).run(1) // [ 2, 1 ]
```

Vediamo ora un semplice programma che gestisce un contatore

```
type S = number

const increment = modify<S>(n => n + 1)

const decrement = modify<S>(n => n - 1)

const program = increment
  .chain(() => increment)
  .chain(() => increment)
  .chain(() => decrement)

console.log(program.run(0)) // [undefined, 2]
```

Si noti che `increment` è un *valore* che rappresenta un programma, che se eseguito modificherà lo stato incrementando il contatore. Essendo un valore è inerte fino a quando non viene eseguito (chiamando il metodo `run`). `program` è un programma ottenuto dalla combinazione di due sottoprogrammi (`increment` e `decrement`). E qui emerge il fatto che vale la trasparenza referenziale: si noti che vengono fatti tre incrementi ma `increment` è definito una volta sola.

alla fine decido di eseguire il programma fornendo lo stato iniziale

```
console.log(program.run(0)) // [undefined, 2]
```

naturalmente rappresentando `program` l'intero programma posso eseguirlo tutte le volte che voglio, anche cambiando lo stato iniziale

```
console.log(program.run(2)) // [undefined, 4]
```

DEMO

`state.ts`

22 Dependency injection funzionale: la monade `Reader`

> Represents a computation which can read values from a shared environment, pass values from function to function and execute sub-computations in a modified environment

Se leggere da uno stato mutabile può invalidare la trasparenza referenziale, come è possibile leggere da una configurazione globale? Ancora una volta la risposta è nelle funzioni, il modello che descrive nel modo più generale la lettura da una configurazione globale è quello definito dalla seguente firma

```
(e: E) => A
```

ove `E` è il tipo della configurazione e `A` è il tipo del valore restituito dalla computazione.

Definiamo una istanza di monade

```
class Reader<E, A> {
  constructor(readonly run: (e: E) => A) {}
  map<B>(f: (a: A) => B): Reader<E, B> {
    return this.chain(a => of(f(a))) // <= derived
  }
  ap<B>(fab: Reader<E, (a: A) => B>): Reader<E, B> {
    return fab.chain(f => this.map(f)) // <= derived
  }
  chain<B>(f: (a: A) => Reader<E, B>): Reader<E, B> {
    return new Reader(e => f(this.run(e)).run(e))
  }
}

const of = <E, A>(a: A): Reader<E, A> =>
  new Reader(() => a)
```

22.1 Combinatori

ask

```
/** Reads the current context */
const ask = <E>(): Reader<E, E> => new Reader(e => e)
```

asks

```
/** Projects a value from the global context in a Reader */
const asks = <E, A>(f: (e: E) => A): Reader<E, A> =>
  new Reader(f)
```

local

```
/**
 * Changes the value of the local context
 * during the execution of the action `fa`
 */
const local = <E>(f: (e: E) => E) => <A>(
  fa: Reader<E, A>
): Reader<E, A> => new Reader((e: E) => fa.run(f(e)))
```

DEMO
`reader.ts`

23 Logging funzionale: la monade `Writer`

Il modello che descrive nel modo più generale una operazione con logging è quello definito dalla seguente firma

```
() => [A, W]
```

ove `A` è il tipo del valore restituito dalla computazione e `W` è il tipo del valore usato come log.

Definiamo una istanza di funtore

```
class Writer<W, A> {
  constructor(readonly run: () => [A, W]) {}
  map<B>(f: (a: A) => B): Writer<W, B> {
    const [a, w] = this.run()
    return new Writer(() => [f(a), w])
  }
  eval(): A {
    return this.run()[0]
  }
  exec(): W {
    return this.run()[1]
  }
}
```

Proviamo a definire una istanza di monade, incominciamo da `of`

```
const of = <W, A>(a: A): Writer<W, A> =>
  new Writer(() => [a, w]) // w ???
```

C'è un problema: non so cosa usare come `w`.

Vediamo adesso `chain`

```
class Writer<W, A> {
  ...
  chain<B>(f: (a: A) => Writer<W, B>): Writer<W, B> {
    return new Writer(() => {
      const [a, w1] = this.run()
      const [b, w2] = f(a).run()
```

```
      return [b, w] // w ???
    })
  }
}
```

Anche qui c'è un problema: non so cosa usare come `w`, inoltre intuitivamente dovrebbe contenere l'informazione sia di `w1` sia di `w2`.

Ricapitolando

- per `of` mi servirebbe un valore di tipo `W` che sia *neutro*
- per `chain` mi servirebbe un modo per *combinare* due valori di tipo `W`

23.1 `Monoid` to the rescue

```
const of = <W>(M: Monoid<W>) => <A>(a: A): Writer<W, A> => {
  return new Writer(() => [a, M.empty])
}

const chain = <W>(S: Semigroup<W>) => <A, B>(
  fa: Writer<W, A>,
  f: (a: A) => Writer<W, B>
): Writer<W, B> => {
  return new Writer(() => {
    const [a, w1] = fa.run()
    const [b, w2] = f(a).run()
    return [b, S.concat(w1, w2)]
  })
}
```

Infine defininiamo `ap` (anche se si potrebbe derivare)

```
const ap = <W>(S: Semigroup<W>) => <A, B>(
  fab: Writer<W, (a: A) => B>,
  fa: Writer<W, A>
): Writer<W, B> => {
  return new Writer(() => {
    const [f, w1] = fab.run()
    const [a, w2] = fa.run()
```

```
    return [f(a), S.concat(w1, w2)]
  })
}
```

23.2 Combinatori

tell

```
/**
 * Appends a value to the accumulator
 */
export const tell = <W>(w: W): Writer<W, void> => {
  return new Writer(() => [undefined, w])
}
```

listen

```
/**
 * Modifies the result to include the
 * changes to the accumulator
 */
export const listen = <W, A>(
  fa: Writer<W, A>
): Writer<W, [A, W]> => {
  return new Writer(() => {
    const [a, w] = fa.run()
    return [tuple(a, w), w]
  })
}
```

pass

```
/**
 * Applies the returned function to the accumulator
 */
export const pass = <W, A>(
  fa: Writer<W, [A, (w: W) => W]>
): Writer<W, A> => {
  return new Writer(() => {
```

```
    const [[a, f], w] = fa.run()
    return [a, f(w)]
  })
}
```

listens

```
/**
 * Projects a value from modifications made
 * to the accumulator during an action
 */
export const listens = <W, A, B>(
  fa: Writer<W, A>,
  f: (w: W) => B
): Writer<W, [A, B]> => {
  return new Writer(() => {
    const [a, w] = fa.run()
    return [tuple(a, f(w)), w]
  })
}
```

censor

```
/**
 * Modify the final accumulator value by applying a function
 */
export const censor = <W, A>(
  fa: Writer<W, A>,
  f: (w: W) => W
): Writer<W, A> => {
  return new Writer(() => {
    const [a, w] = fa.run()
    return [a, f(w)]
  })
}
```

DEMO

`writer.ts`

24 Monad transformer

Supponiamo di avere una computazione con le seguenti proprietà

- asincrona
- può fallire

Come possiamo modellarla?

Sappiamo che questi due effetti possono essere rispettivamente codificati dai seguenti tipi

- `Task<A>`
- `Either<L, A>`

e che ambedue hanno una istanza di monade.

Potremmo anche dire che l'istanza di monade di `Task<A>` rappresenta la feature di asincronicità, mentre l'istanza di monade di `Either<L, A>` rappresenta la feature di fallimento.

Come posso combinare queste due feature?

In due modi

- `Task<Either<L, A>>` rappresenta una computazione asincrona che può fallire
- `Either<L, Task<A>>` rappresenta una computazione che può fallire oppure che restituisce una computazione asincrona

Diciamo che sono interessato al primo dei due modi

```
// type alias
type TaskEither<L, A> = Task<Either<L, A>>
```

è possibile definire una istanza di monade per `TaskEither`?

In generale le monadi non compongono, ovvero date due istanze di monade, una per `M<A>` e una per `N<A>`, allora a `M<N<A>>` non è detto che possa ancora essere associata una istanza di monade.

Esercizio 24.1 *In generale le monadi non compongono, perchè?*

Che non compongano in generale però non vuol dire che non esistano dei casi particolari ove questo succede.

Obiettivo: poter definire la seguente funzione

$$flatten : M(N(M(Na))) \to M(Na)$$

Vediamo tre condizioni sufficienti

$$prod : N(M(Na)) \to M(Na)$$

$$dorp : M(N(Ma)) \to M(Na)$$

$$swap : N(Ma) \to M(Na)$$

Esercizio 24.2 *Mostrare che `swap` è sufficiente per le altre due*

Esercizio 24.3 *Mostrare che ognuna delle tre condizioni è sufficiente*

Ora vediamo qualche esempio, se il type constructor `M` ha una istanza di monade allora ammettono una istanza di monade i seguenti type constructor

- `OptionT<M, A> = M<Option<A>>`
- `EitherT<M, L, A> = M<Either<L, A>>`
- `StateT<M, S, A> = (s:  S) => M<[A, S]>`
- `ReaderT<M, E, A> = Reader<E, M<A>>`

Notate come questi tipi collassino in quelli già conosciuti quando il type constructor `M` è `Identity`

- `Option<A> = OptionT<Identity, A>`
- `Either<L, A> = EitherT<Identity, L, A>`
- `State<S, A> = StateT<Identity, S, A>`
- `Reader<E, A> = ReaderT<Identity, E, A>`

Vediamo la funzione `swap` per i quattro casi in esame

OptionT

```
import { HKT } from 'fp-ts/lib/HKT'
import { Monad } from 'fp-ts/lib/Monad'

import { Option, none, some } from 'fp-ts/lib/Option'

const swapOption = <M>(M: Monad<M>) => <A>(
  nma: Option<HKT<M, A>>
): HKT<M, Option<A>> => {
  return nma.fold(M.of(none), ma => M.map(ma, some))
}
```

EitherT

```
import { Either, left, right } from 'fp-ts/lib/Either'

const swapEither = <M>(M: Monad<M>) => <L, A>(
  nma: Either<L, HKT<M, A>>
): HKT<M, Either<L, A>> => {
  return nma.fold(
    l => M.of(left(l)),
    ma => M.map(ma, a => right(a))
  )
}
```

ReaderT

```
import { Reader } from 'fp-ts/lib/Reader'

const swapReader = <M>(M: Monad<M>) => <E, A>(
  nma: HKT<M, Reader<E, A>>
): Reader<E, HKT<M, A>> => {
  return new Reader(e =>
    M.map(nma, reader => reader.run(e))
  )
}
```

StateT

```
const swapState = <M>(M: Monad<M>) => <S, A>(
  nma: HKT<M, (s: S) => [A, S]>
): ((s: S) => HKT<M, [A, S]>) => {
  return s => M.chain(nma, state => M.of(state(s)))
}
```

Ora vediamo una implementazione completa `TaskEither`

```
import { Task, task } from 'fp-ts/lib/Task'
import { Either, either, left } from 'fp-ts/lib/Either'

const of = <L, A>(a: A): TaskEither<L, A> =>
  new TaskEither(task.of(either.of(a)))

class TaskEither<L, A> {
  constructor(readonly value: Task<Either<L, A>>) {}
  run(): Promise<Either<L, A>> {
    return this.value.run()
  }
  chain<B>(
    f: (a: A) => TaskEither<L, B>
  ): TaskEither<L, B> {
    return new TaskEither(
      this.value.chain(e =>
        e.fold(l => task.of(left(l)), a => f(a).value)
      )
    )
  }
}
```

e due utili combinatori

```
import { TaskEither, fromLeft, taskEither } from 'fp-ts/lib/TaskEither'
import { Either } from 'fp-ts/lib/Either'

/** Return `Right` if the given action succeeds, `Left` if it throws */
export const attempt = <L, A>(fa: TaskEither<L, A>): TaskEither<L, Either<L, A>> =
```

```
  return new TaskEither(fa.value.map(e => e.map(() => e)))
}

/**
 * Make sure that a resource is cleaned up in the event of an exception. The
 * release action is called regardless of whether the body action throws or
 * returns.
 */
export const bracket = <L, A, R>(
  acquire: TaskEither<L, R>,
  program: (r: R) => TaskEither<L, A>,
  release: (r: R) => TaskEither<L, void>
): TaskEither<L, A> => {
  return acquire.chain(r =>
    attempt(program(r)).chain(result => release(r).chain(() => result.fold<TaskEit
  )
}
```

DEMO
eitherT.ts

25 Ottica funzionale

25.1 A cosa serve?

Si consideri il problema di modificare delle strutture dati immutabili. Per capire la ragione per cui potremmo voler utilizzare l'ottica funzionale vediamo un semplice esempio, definiamo due record

```
type Street = {
  num: number
  name: string
}
type Address = {
  city: string
  street: Street
}
```

Data una istanza di `Address`, ricavare il nome della strada è immediato

```
const a1: Address = {
  city: 'london',
  street: { num: 23, name: 'high street' }
}
const name = a1.street.name
```

Tuttavia sostituirne il valore è laborioso

```
const a2: Address = {
  ...a1,
  street: {
    ...a1.street,
    name: 'main street'
  }
}
```

L'ottica funzionale serve a manipolare (leggere, scrivere, modificare) le strutture dati immutabili in modo semplice e componibile.

25.2 Iso

Il tipo `Iso<S, A>` rappresenta un isomorfismo tra `S` e `A`

$$
\begin{tikzcd}
S \arrow[r, bend right, "\texttt{get}"'] & A \arrow[l, bend right, "\texttt{reverseGet}"']
\end{tikzcd}
$$

```
class Iso<S, A> {
  constructor(
    readonly get: (s: S) => A,
    readonly reverseGet: (a: A) => S
  ) {}
}
```

Devono valere le seguenti leggi

- `get` $\circ$ `reverseGet = identity`
- `reversetGet` $\circ$ `get = identity`

Esempio 25.1 *Convertire metri in chilometri e chilometri in miglia*

```
const mTokm = new Iso<number, number>(
  m => m / 1000,
  km => km * 1000
)
const kmToMile = new Iso<number, number>(
  km => km * 0.621371,
  mile => mile / 0.621371
)
```

E' possibile effettuare il lifting di un endomorfismo[33] di `A` ad un endomorfismo di `S` tramite la funzione `modify`.

Inoltre gli `Iso` compongono

$$
\begin{tikzcd}
S \arrow[r, bend right, "\texttt{get}"'] & A \arrow[l, bend right, "\texttt{reverseGet}"'] \arrow[r, bend right, "\texttt{get}"'] & B \arrow[l, bend right, "\texttt{reverseGet}"']
\end{tikzcd}
$$

[33] Un *endomorfismo* di un insieme A è una funzione $f : A \to A$

```
class Iso<S, A> {
  constructor(
    readonly get: (s: S) => A,
    readonly reverseGet: (a: A) => S
  ) {}
  modify(f: (a: A) => A): (s: S) => S {
    return s => this.reverseGet(f(this.get(s)))
  }
  compose<B>(ab: Iso<A, B>): Iso<S, B> {
    return new Iso(
      s => ab.get(this.get(s)),
      b => this.reverseGet(ab.reverseGet(b))
    )
  }
}
```

Usando la composizione si ottiene facilmente un isomorfismo tra metri e miglia

```
const mToMile = mTokm.compose(kmToMile)
```

25.3 Lens

Il tipo `Lens` è la reificazione dell'operazione di focalizzazione su una parte di un product type.

Data una lente ci sono essenzialmente tre cose che si possono fare

- vedere la parte
- modificare l'intero cambiando la parte
- combinare due lenti per guardare ancora più in profondità

Una lente non è altro che una coppia di funzioni, un getter e un setter. Il tipo `S` rappresenta l'intero `A` la parte

$$
\begin{tikzcd}
S \to S & \\
S \arrow[r, "\mathtt{get}"'] & A \arrow[ul, "\mathtt{set}"']
\end{tikzcd}
$$

```
class Lens<S, A> {
  constructor(
    readonly get: (s: S) => A,
    readonly set: (a: A) => (s: S) => S
  ) {}
}
```

Definiamo una lente per il tipo `Address` con focus sul campo `street`

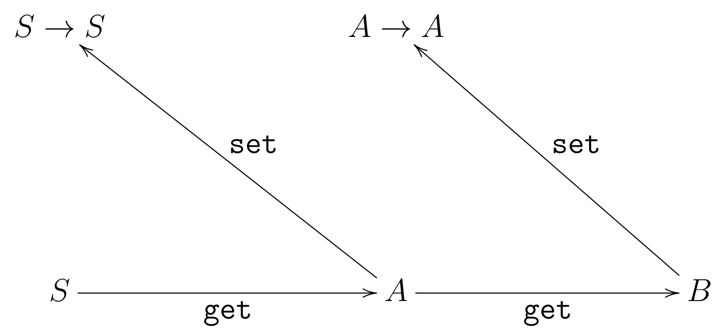
```
const address = new Lens<Address, Street>(
  s => s.street,
  a => s => ({ ...s, street: a })
)

address.get(a1)
// { num: 23, name: "high street" }

address.set({ num: 23, name: 'main street' })(a1)
// { city: "london", street: { num: 23, name: "main street" } }
```

Ora definiamo una lente per il tipo `Street` con focus sul campo `name`

```
const street = new Lens<Street, string>(
  s => s.name,
  a => s => ({ ...s, name: a })
)
```

C'è un modo per ottenere una lente per il tipo `Address` con focus sul campo innestato `name`?

Le lenti, così come gli `Iso`, compongono




```
class Lens<S, A> {
  constructor(
    readonly get: (s: S) => A,
    readonly set: (a: A) => (s: S) => S
  ) {}
  compose<B>(ab: Lens<A, B>): Lens<S, B> {
    return new Lens(
      s => ab.get(this.get(s)),
      b => s => this.set(ab.set(b)(this.get(s)))(s)
    )
  }
}
```

Ora gestire il campo `name` risulta banale

```
const name = address.compose(street)

name.get(a1)
// "high street"

name.set('main street')(a1)
// { city: "london", street: { num: 23, name: "main street" } }
```

Come per gli `Iso` è possibile definire una funzione `modify` per le lenti.

Per esempio supponiamo di volere il nome della via tutto in maiuscolo

```
class Lens<S, A> {
  constructor(
    readonly get: (s: S) => A,
    readonly set: (a: A) => (s: S) => S
  ) {}
  compose<B>(ab: Lens<A, B>): Lens<S, B> {
    return new Lens(
      s => ab.get(this.get(s)),
      b => s => this.set(ab.set(b)(this.get(s)))(s)
    )
  }
  modify(f: (a: A) => A): (s: S) => S {
    return s => this.set(f(this.get(s)))(s)
  }
}

const toUpperCase = (s: string): string => s.toUpperCase()

name.modify(toUpperCase)(a1)
// { city: 'london', street: { num: 23, name: 'HIGH STREET' } }
```

25.4 Prism

Il tipo `Prism` è in qualche modo il duale di `Lens`, ovvero è la reificazione dell'operazione di focalizzazione su una parte di un sum type.

```
class Prism<S, A> {
  constructor(
    readonly getOption: (s: S) => Option<A>,
    readonly reverseGet: (a: A) => S
  ) {}
}
```

Il tipo `S` rappresenta l'intera unione mentre `A` un suo membro.

$$\begin{tikzcd} & A \arrow[dl, "\texttt{reverseGet}"'] \\ S \arrow[r, "\texttt{getOption}"'] & Option(A) \end{tikzcd}$$

Esempio 25.2 *Convertire un valore di tipo $A|null$ in un valore di tipo $Option(A)$*

```
const fromNullable = new Prism<
  string | null,
  string
>(s => (s === null ? none : some(s)), a => a)
```

Un altro esempio tipico di prisma è una coppia di parser / formatter

```
const number = new Prism<string, number>(
  s => {
    const n = parseFloat(s)
    return isNaN(n) ? none : some(n)
  },
  a => String(a)
)
```

Un altro prisma, questa volta tra numeri e interi

```
const integer = new Prism<number, number>(
  s => (s % 1 === 0 ? some(s) : none),
  a => a
)
```

Anche i prismi compongono

$$
\begin{tikzcd}
 & & B \arrow[dl, "\texttt{reverseGet}"'] \\
 & A \arrow[r, "\texttt{getOption}"'] \arrow[dl, "\texttt{reverseGet}"'] & Option(B) \\
S \arrow[r, "\texttt{getOption}"'] & Option(A) &
\end{tikzcd}
$$

```
class Prism<S, A> {
  constructor(
    readonly getOption: (s: S) => Option<A>,
    readonly reverseGet: (a: A) => S
  ) {}
  compose<B>(ab: Prism<A, B>): Prism<S, B> {
    return new Prism(
      s => this.getOption(s).chain(a => ab.getOption(a)),
      b => this.reverseGet(ab.reverseGet(b))
    )
  }
}
```

Posso perciò facilmente ottenere un prisma tra una stringa e un intero

```
const integerFromString = number.compose(integer)
```

Anche per i prismi è possibile definire una funzione `modify`

```
class Prism<S, A> {
```

```
  constructor(
    readonly getOption: (s: S) => Option<A>,
    readonly reverseGet: (a: A) => S
  ) {}
  compose<B>(ab: Prism<A, B>): Prism<S, B> {
    return new Prism(
      s => this.getOption(s).chain(a => ab.getOption(a)),
      b => this.reverseGet(ab.reverseGet(b))
    )
  }
  modify(f: (a: A) => A): (s: S) => S {
    return s =>
      this.getOption(s)
        .map(a => this.reverseGet(f(a)))
        .fold(() => s, s => s)
  }
}
```

25.5 Optional

TODO

25.6 Diagramma delle ottiche

Esempio 25.3

$$
\begin{tikzcd}
 & Iso & \\
Lens \arrow[ur] & & Prism \arrow[ul] \\
 & Optional \arrow[ul] \arrow[ur] &
\end{tikzcd}
$$