Monadi

Una delle caratteristiche principali di Haskell è l'essere un linguaggio **lazy**.

Questa caratteristica porta ad alcune conseguenze positive, come:

- Purezza: in un linguaggio completamente puro è presente la **trasparenza referenziale**.

  La trasparenza referenziale si può definire come:

  Se `a = b` posso usare `a` al posto di `b` e viceversa.

  Un possibile esempio di questo concetto può essere:

  ```
  f x = a + a
      where
          a = g x
          b = h 3
  ```

  il quale viene tradotto dal compilatore Haskell in:

  ```
  f x = g x + g x
  ```

  E' quindi possibile osservare come `a` venga sostituita con la sua definizione sottostante (ottimizzazione, **inlining**), andando a rimuovere completamente la definizione di `b` (non utilizzata).

  Questo è possibile solo se il linguaggio è puro, dove non è possibile alterare lo stato esterno.

  Se il linguaggio non fosse stato puro, `g` ed `h` avrebbero potuto causare effetti collaterali con la loro esecuzione. In questo caso, la traduzione effettuata dal compilatore non sarebbe stata corretta.

- Modularità dei programmi: è possibile dividere il programma in moduli differenti, richiamati solo se strettamente necessario.
- Normalizzazione (ad esempio, nel caso della funzione `const 1 undefined`, dove `undefined` non verrà mai valutato perché non utilizzato).
- Parallelismo: nei linguaggi pigri, e di conseguenza puri, è semplice parallelizzare l'esecuzione.

ma anche conseguenze negative, come:

- Purezza: in un linguaggio completamente puro non è possibile stampare a schermo, stabilire connessioni di rete, ecc. Questo rende il linguaggio abbastanza inutile.
- Imprevedibilità: ad esempio, nel caso della funzione `const (print 1) (print 2)`, non si riesce a capire guardando solamente il codice quale sarà l'esito dell'esecuzione.

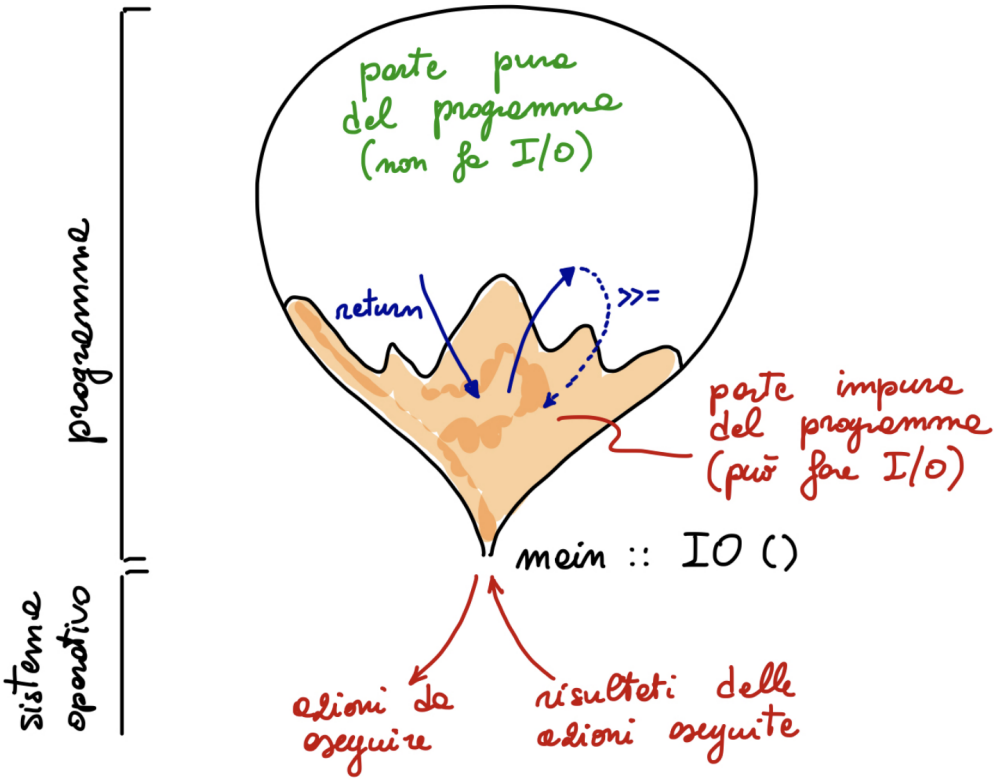
Un possibile esempio di purezza come conseguenza negativa in Haskell è la gestione di Input/Output (dialoghi). Prima delle monadi, questo procedimento richiedeva una logica contorta.

Prima di introdurre il concetto di **monade**, bisogna fare una distinzione tra il *fare qualcosa* rispetto al *pensare di fare qualcosa*.

Nella programmazione tradizionale viene scritto in maniera esplicita quello che il programma deve fare, passo dopo passo.

Nella programmazione con monadi, il progrmama **crea delle azioni che se eseguite hanno un effetto collaterale** (come una stampa a schermo).

Una possibile rappresentazione grafica di questo concetto può essere:



E' quindi presente una suddivisione tra la parte *pura* e *impura* del programma. Il nostro programma rimane puro, delegando al sistema operativo l'esecuzione di funzionalità impure.

Inizialmente l'idea delle monadi era stata pensata per gestire il problema dell'input/output. Successivamente ci si è resi conto che poteva essere utilizzata anche per altri scopi (alcuni possono essere eccezioni, parallelismo, transazioni).

Una monade può essere anche definita come: concetto nell'area della semantica dei linguaggi per descrivere matematicamente **computazioni** che possono avere **effetti collaterali**.

E' possibile trovare monadi anche in linguaggi moderni, come Scala e Javascript, fornendo un modo strutturato di organizzare computazioni con effetti collaterali.

Di seguito vengono mostrati esempi pratici di applicazione di monadi in Haskell.

Valutatore di espressioni

Si vuole realizzare un valutatore per semplici espressioni aritmetiche.

```
data Expr = Const Int | Div Expr Expr

eval :: Expr -> Int
eval (Const n) = n
eval (Div t s) = eval t `div` eval s
```

`Expr` indica una struttura dati custom, la quale può avere un valore `Int` oppure rappresentare una espressione aritmetica.

Vengono successivamente definiti due casi possibili: restituzione del valore inviato e calcolo dell'espressione.

Si vuole ora raffinare il valutatore in modo che supporti:

- Gestione delle **divisioni per zero**
- **Conteggio** delle operazioni "difficili"
- **Tracciamento** dei passi di valutazione

Se usassimo un linguaggio impuro sarebbe tutto facile.

Partiamo dalla divisione per zero.
Per implementare la gestione della divisione per zero viene modificato il codice nel seguente modo:

```
eval :: Expr -> Maybe Int
eval (Const n) = Just n
eval (Div t s) =
    case eval t of
        Nothing -> Nothing
        Just m -> case eval s of
            Nothing -> Nothing
            Just 0 -> Nothing
            Just n -> Just (m `div` n)
```

Ciò che era facile fare in un linguaggio impuro ha stravolto la struttura del programma in un linguaggio puro. La logica del programma si perde nella gestione dell'eccezione.

Si passa ora al conteggio delle operazioni "difficili".
Per implementare il conteggio delle operazioni viene modificato il codice nel seguente modo:

```
type Counter a = Int -> (a, Int)

eval :: Expr -> Counter Int
eval (Const n) x = (n, x)
eval (Div t s) x =
    let (m, y) = eval t x in
        let (n, z) = eval s y in
            (m `div` n, z + 1)
```

Ciò che era facile fare in un linguaggio impuro ha stravolto la struttura del programma in un linguaggio puro. La logica del programma si perde nella gestione del contatore.

Per concludere, si passa al tracciamento dei passi di valutazione.
Per implementare il tracciamento dei passi di valutazione viene modificato il codice nel seguente modo:

```
type Output a = (String, a)

eval :: Expr -> Output Int
eval (Const n) = (line (Const n) n, n)
eval (Div t s) =
    let (x, m) = eval t in
        let (y, n) = eval s in
            (x ++ y ++ line (Div t s) (m `div` n), m `div` n)

line :: Expr -> Int -> String
line t n = "eval (" ++ show t ++ ") = " ++ show n ++ "\n"
```

Ciò che era facile fare in un linguaggio impuro ha stravolto la struttura del programma in un linguaggio puro. La logica del programma si perde nella gestione della traccia.

In tutti e tre i casi, l'utilizzo di un linguaggio puro porta ad un codice più complesso.

Si vuole ora fare una sorta di esercizio di astrazione.

Prendendo una versione **pura** di `eval`, si ha `eval :: Expr -> Int`

Prendo una versione **impura** di `eval`, si ha `eval :: Expr -> M Int`
dove `M` può essere:

- **Maybe**, nel caso delle eccezioni
- **Counter**, nel caso del contatore
- **Output**, nel caso del tracciamento

Su queste `M` occorrono diverse operazioni specifiche che dipendono dalla `M` stessa. Sono però sempre presenti due operazioni:

- **return**: l'idea dietro la funzione `return` è che la sua computazione debba essere pura, senza effetti collaterali. Viene definita come

  `return :: a -> M a`

- **bind (>>=)**: l'idea dietro la funzione `bind` è il combinare in sequenza due computazioni della monade `M`, dove però la seconda computazione può aver bisogno di conoscere il valore prodotto dalla computazione precedente. Viene definita come

  `(>>=) :: M a -> (a -> M b) -> M b`

Questi due sono gli operatori fondamentali di ogni monade, che ci permettono di dire come viene fatta una computazione pura e come vengono combinate in sequenza più operazioni.

Oltre a queste due, ogni monade avrà delle operazioni ad hoc per gestire i casi specifici inerenti.

Tornando all'esempio precedente, osserviamo ora come implementare le monadi in quel contesto, andando a semplificare il codice.

Partiamo dalla divisione per zero. Il codice in versione monadica è il seguente:

```
return :: a -> Maybe a
return = Just

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) Nothing _ = Nothing
(>>=) (Just a) f = f a

abort :: Maybe a
abort = Nothing

evalM :: Expr -> Maybe Int
evalM (Const n) = return n
evalM (Div t s) =
    evalM t >>= \m ->
    evalM s >>= \n ->
    if n == 0 then abort
    else return (m `div` n)
```

In questo caso, la funzione `return` viene scritta in maniera abbreviata. La sua forma completa è `return a = Just a`. Questa funzione ci permette di indicare la computazione pura andando ad affermare che dato un argomento `a`, la funzione `return` restituirà semplicemente un valore `Just a`.

Per quanto riguarda la funzione `bind` sono presenti due casi.

Nel primo caso viene gestito il fallimento, con `Nothing` come parametro di sinistra che restituisce un ulteriore valore `Nothing` per indicare il fallimento della composizione di operazioni.

Nel secondo caso, invece, la prima computazione è andata a buon fine e sarà quindi possibile applicare `f` ad `a` (tramite `f a`).

E' inoltre presente una operazione specifica chiamata `abort`, la quale permette di restituire un `Nothing` abortendo la computazione.

Grazie a queste nuove funzioni, si può riscrivere il codice in maniera più ordinata e leggibile.

Si passa ora al conteggio delle operazioni "difficili". Il codice in versione monadica è il seguente:

```
return :: a -> Counter a
return a = \x -> (a, x)

(>>=) :: Counter a -> (a -> Counter b) -> Counter b
(>>=) m f = \x -> let (a, y) = m x in f a y

tick :: Counter ()
tick = \x -> ((), x + 1)

evalM :: Expr -> Counter Int
evalM (Const n) = return n
evalM (Div t s) =
    evalM t >>= \m ->
    evalM s >>= \n ->
    tick >>= \() ->
    return (m `div` n)
```

In questo caso, la funzione `return` può esprimere una computazione pura solo nel caso in cui non viene modificato il contatore. Viene quindi passato l'argomento `x` senza essere alterato.

Per quanto riguarda la funzione `bind`, il valore del contatore `x` viene passato ad `m` per essere alterato. Il valore modificato `y` rappresenterà il nuovo valore del contatore. Il contatore viene quindi passato dalla prima computazione alla successiva, per essere alterato.

E' inoltre presente una operazione specifica chiamata `tick`, la quale permette di incrementare il valore del contatore di uno. Le parentesi tonde aperte chiuse indicano una `Unit`, un valore vuoto.

Grazie a queste nuove funzioni, si può riscrivere il codice in maniera più ordinata e leggibile.

Si può anche notare come i due codici appena presentati risultano essere simili nella loro forma, anche se gli effetti collaterali causati dalle monadi sono differenti.

Per concludere, si passa al tracciamento dei passi di valutazione. Il codice in versione monadica è il seguente:

```
return :: a -> Output a
return a = ("", a)

(>>=) :: Output a -> (a -> Output b) -> Output b
(>>=) m f =
  let (x, a) = m in
    let (y, b) = f a in
      (x ++ y, b)

output :: String -> Output ()
output x = (x, ())

evalM :: Expr -> Output Int
evalM (Const n) =
  output (line (Const n) n) >>= \() ->
  return n
evalM (Div t s) =
  evalM t >>= \m ->
  evalM s >>= \n ->
  output (line (Div t s) (m `div` n)) >>= \() -> return (m `div` n)
```

Anche in questo caso , come per i precedenti, le funzioni `return` e `bind` permettono di esprimere la computazione pura e la composizione di operazioni.

Per quanto riguarda le operazioni speciali, è presente la funzione `output`, la quale permette di ottenere una traccia a schermo.

Volendo ricapitolare i casi appena visti, si può effettuare la seguente suddivisione di monadi:

- **Maybe a**: per le computazioni che possono fallire (con `abort`). Se hanno successo producono un valore di tipo `a`.
- **Counter a**: per le computazioni che possono incrementare un contatore (con `tick`). Producono un valore di tipo `a`.
- **Output a**: per le computazioni che possono produrre una traccia (con `output`). Producono un valore di tipo `a`.

Si vuole ora dare una definizione di **monade**.

Il termine monade deriva da *monoide*, ovvero una struttura algebrica dotata di un'operazione **assiociativa**:

```
(a * b) * c = a * (b * c)
```

e con unità (o elemento neutro) `e` a sinistra e destra:

```
a * e = e * a = a
```

Se leggiamo `*` come composizione ed `e` come nessun effetto, queste proprietà di un monoide dicono che:

- Non c'è differenza tra comporre `c` con la composizione di `a` e `b` e comporre `a` con la composizione di `b` e `c`, purché l'ordine delle azioni sia preservato (un monoide **non è** commutativo in generale).
- La neutralità di `e` cattura il fatto che esso rappresenti una azione "pura", priva di effetti.

Una monade non è propriamente un monoide perché c'è bisogno di trasferire informazioni tra le computazioni combinate.

Per questo motivo vengono formulate **tre leggi** per rappresentare una monade, che sono:

1. `return a >>= f ≡ f a`
2. `m >>= return ≡ m`
3. `(m >>= \a -> n) >>= f ≡ m >>= (\a -> n >>= f)`

Le prime due leggi affermano che `return` si comporta come un elemento neutro (computazione pura, priva di effetti).

La terza legge afferma che `bind` è associativo (N.B.: solo se `a` non compare libera in `f`).

Queste leggi vanno **dimostrate** per ogni monade.

Si vuole ora provare a dimostrare le tre leggi per la monade `Maybe`. Riprendendo la sua definizione di `return` e `bind`:

```
return :: a -> Maybe a
return = Just

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) Nothing _ = Nothing
(>>=) (Just a) f = f a
```

Partendo dalla prima legge, si può dimostrare con:

```
return a >>= f    =    Just a >>= f    =    f a
```

Passando invece alla seconda legge, vanno dimostrati due casi possibili:

Caso `m = Nothing`:

```
m >>= return    =    Nothing    =    m
```

Caso `m = Just a`:

```
m >>= return     =    return a     =    Just a     =    m
```

Per concludere con la terze legge, che dimostra l'associatività, vanno dimostrati il lato sinistro e destro dei due casi possibili:

Lato sinistro, Caso `m = Nothing`:

```
(m >>= \a -> n) >>= f     =    (Nothing >>= \a -> n) >>= f
                  =    Nothing >>= f     =    Nothing
```

Lato sinistro, Caso `m = Just a`:

```
(m >>= \a -> n) >>= f     =    (Just a >>= \a -> n) >>= f
            =    ((\a -> n) a) >>= f     =    n >>= f
```

Lato destro, Caso `m = Nothing`:

```
m >>= (\a -> n >>= f)     =    Nothing >>= (\a -> n >>= f)
                          =    Nothing
```

Lato destro, Caso `m = Just a`:

```
m >>= (\a -> n >>= f)     =    Just a >>= (\a -> n >>= f)
             =    (\a -> n >>= f) a     =    n >>= f
```

Avendo dimostrato le tre leggi, `Maybe` viene considerata una monade.

Discorso analogo può essere fatto anche per `Counter` e `Output` (dimostrazioni sulle slide).

Viene ora fatta un'osservazione: `return` e `bind` sono **overloaded** (dipendono dalla singola monade).

L'idea di base è la definizione di una classe di **costruttori di tipo** `Monad`:

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

dove `m` è una variabile su **costruttori di tipo**.

Un ulteriore concetto utile quando si programma con le monadi è la funzione **and then** $>>$.

A volte quando si combinano due azioni di una monade, la seconda non usa il risultato della prima e solo la composizione degli effetti è importante.

Una possibile definizione di questa funzione, che si può generalizzare ad ogni monade, può essere:

```
(>>) :: Counter a -> Counter b -> Counter b
(>>) as bs = as >>= const bs
```

Un suo possibile esempio di utilizzo, nel contesto precedentemente visto di Counter, può essere:

```
evalN :: Expr -> Counter Int
evalN (Const n) = return n
evalN (Div t s) =
    evalN t >>= \m ->
    evalN s >>= \n ->
    tick >>
    return (m `div` n)
```

dove la riga `tick >>= \()` diventa solamente `tick >>`.

Un altro concetto, principalmente zucchero sintattico, è quello della notazione **do**. Riprendendo un esempio visto in precedenza:

```
evalM :: Expr -> Maybe Int
evalM (Const n) = return n
evalM (Div t s) =
    evalM t >>= \m ->
    evalM s >>= \n ->
    if n == 0 then abort
    else return (m `div` n)
```

Attraverso la notazione `do` si può semplificare in:

```
evalM :: Expr -> Maybe Int
evalM (Const n) = return n
evalM (Div t s) = do
    m <- evalM t
    n <- evalM s
    if n == 0 then abort
    else return (m `div` n)
```

Si osserva ora la possibilità di avere una monade che rappresenta computazioni con effetti molteplici di natura diversa. Questo approccio non è immediato e presenta alcune problematiche.

Il primo problema è di natura concettuale. E' infatti presenta un'ambiguità semantica. Osserviamo due esempi di semantiche differenti:

- Semantica *tradizionale*: `eval :: Expr -> Int -> (Maybe Int, Int)`
- Semantica *transazionale*: `eval :: Expr -> Int -> Maybe (Int, Int)`

A seconda di quale semantica si sceglie, si va a cambiare il significato della valutazione.

Nella prima semantica, il contatore viene incrementato anche in caso di fallimento dell'ultima valutazione.

Nella seconda, invece, in caso di fallimento dell'ultima valutazione si ottiene un `Nothing`, perdendo il valore del contatore. Si applica una logica transazionale, come nei database.

Il secondo problema è di natura ingegneristica. Creando una singola monade che combina più effetti si crea il problema del dover duplicare il codice.

Questi due problemi, con l'idea di monade osservata fino ad ora, non ci permettono di ottenere più effetti con una singola monade.

Si introduce quindi una nuova idea: invece di definire una monade per rappresentare computazioni che possono avere "certi effetti", definiamo una **trasformazione di monade** che **aggiunge** "certi effetti" a una monade già esistente.

Si osservi una possibile trasformazione `MaybeT` (N.B.: il codice visto d'ora in poi è corretto in forma logica, ma viene rifiutato dal compilatore Haskell. Per risolvere questo problema si dovrebbero usare i tipi inversi delle monadi):

```
type MaybeT m a = m (Maybe a)

instance Monad m => Monad (MaybeT m) where
    return = return . Just
    (>>=) m f = m >>= \x ->
        case x of
            Nothing -> return Nothing
            Just a -> f a

abort :: Monad m => MaybeT m a
abort = return Nothing
```

Partendo dalla definizione di `return`, volendo riscriverla in forma estesa:
`return a = return (Just a)` . Il `return` esterno fa riferimento alla funzione della monade `MaybeT`, mentre quello interno fa riferimento alla monade `m`.

Passando invece alla definizione di `bind`, vale lo stesso discorso fatto in precedenza per `return`. Le funzioni di `bind` esterna ed interna fanno riferimento a due funzioni diverse.

Volendo quindi applicare la monade `MaybeT` ad un esempio concreto:

```
type MaybeCounter = MaybeT Counter

evalM :: Expr -> MaybeCounter Int
evalM (Const n) = return n
evalM (Div t s) =
    evalM t >>= \m ->
    evalM s >>= \n ->
    tick >>= \() ->         -- ERRORE!
    if n == 0 then abort
    else return (m `div` n)
```

Non possono essere mescolate azioni di monadi diverse. `tick` fa parte della monade interna, ma non può essere richiamata in questo modo.

Per risolvere questo problema si introduce il concetto di **lifting delle operazioni**.

Quando usiamo `MaybeT` per trasformare una monade `m` in una monade `MaybeT m`, vorremmo poter trasformare computazioni di `m` (che non possono fallire) in computazioni di `MaybeT m` che non falliscono.

Questa trasformazione di computazioni si chiama **lifting**. Ogni trasformazione di monade avrà il suo lifting specifico, per cui ha senso definire una classe con una funzione `lift`.

```
class MonadT t where
    lift :: Monad m => m a -> t m a
```

Riprendendo l'esempio di `MaybeT`:

```
instance MonadT MaybeT where
    lift m = m >>= (return . Just)
```

con conseguente codice funzionante:

```
evalM :: Expr -> MaybeCounter Int
evalM (Const n) = return n
evalM (Div t s) =
    evalM t >>= \m ->
    evalM s >>= \n ->
    lift tick >>= \() ->    -- OK!
    if n == 0 then abort
    else return (m `div` n)
```

Con `MaybeT` possiamo aggiungere fallimenti a qualsiasi monade, inclusa `Counter`.

Se definiamo la trasformazione di monade `CounterT` possiamo aggiungere un contatore a qualsiasi monade, inclusa `Maybe`.

Questo porta ad avere un totale di 2 trasformazioni di monadi + 2 monadi.

Se definiamo una monade *banale* **Id** per rappresentare computazioni pure (prive di effetti), possiamo ottenere ogni combinazione come pila di trasformazioni che partono da Id.

Degli esempi possono essere `Maybe = MaybeT Id` e `Counter = CounterT Id`.

Una definizione di `Id` può essere:

```
type Id a = a

instance Monad Id where
    return = id
    (>>=) a f = f a
```

Volendo applicare questo concetto su `CounterT` (N.B.: anche questo codice non viene accettato dal compilatore Haskell, ma resta logicamente corretto):

```
type CounterT m a = Int -> m (a, Int)

instance Monad m => Monad (CounterT m) where
    return a = \x -> return (a, x)
    (>>=) m f = \x -> m x >>= \(a, y) -> f a y

tick :: Monad m => CounterT m ()
tick = \x -> return ((), x + 1)
```