**Emerging Programming Paradigms**

[ lezione 13 maggio ]

**Monadi**

Vengono introdotte a fine anni ‘90 in Haskell, derivano dalla matematica. Idealmente fino a quando un linguaggio di programmazione non ha side-effects (I/O, generatore di numeri casuali, non determinismo, controllo non locale, concorrenza, accesso a uno stato condiviso) è facile fornire una sua descrizione matematica.

L’I/O è per esempio un side effect perchè:

* il compilatore di Haskell, assumendo l’assenza di side-effect, se vede che calcoliamo la radice quadrata dello stesso numero 100 volte, salva il valore e non la computa ogni volta
* per l’input se noi leggessimo dallo stdin 100 volte, non vorremmo che il compilatore salvasse il primo valore letto perché questo potrebbe cambiare
* in output se noi scrivessimo 100 volte nello stdout, il valore di ritorno di print è sempre lo stesso (una cosa come 0 se è andato tutto bene), quindi il compilatore potrebbe togliere tutte le altre chiamate perchè tanto ritornano sempre lo stesso valore. E sarebbe sbagliato.

È possibile creare descrizioni ad-hoc per ogni side-effect possibile, ma scomodo. Tuttavia le semantiche dei side-effects possono essere viste come le istanze di un concetto della teoria delle categorie (ramo della matematica che generalizza tutto ciò che è generalizzabile), ovvero le **monadi**.

In Haskell questo concetto viene espresso sintatticamente. L’idea è che invece di programmare usando side-effects si programma una descrizione del side effects, si da una semantica del side-effect, e grazie alle monadi il programma è in grado di comporre questi side-effects lasciando il codice pulito.

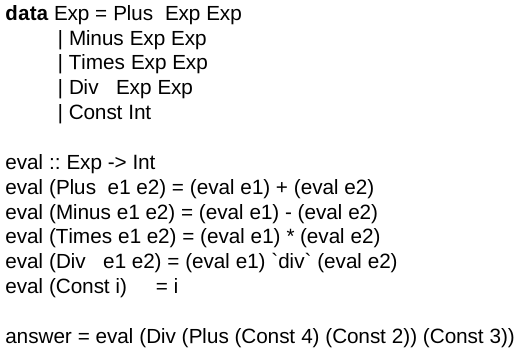
La **referential transparency** è la seguente proprietà matematica:

*f(i) può essere sostituita con o dove o è il risultato della chiamata f(i).*

Quindi l’output di una chiamata di funzione e la chiamata stessa sono del tutto indistinguibili. Un esempio in cui questo non è vero è in presenza di un side-effect di stampa a schermo, se la funzione viene invocata una volta effettuerà una stampa, ma se viene invocata due volte effettuerà due stampe, l’output non rimane quindi lo stesso. Haskell ha referential transparency anche in presenza di stampa ad esempio, Haskell è un linguaggio puro privo di side-effects.

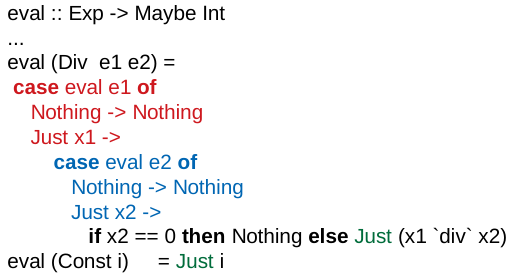
Le **monadi** possono essere viste come un meccanismo per gestire side-effects in un linguaggio funzionale senza perdere referential transparency. Ma non solo, un’altra visione un po’ più ampia, è che una monade è un meccanismo per nascondere all’utente il threading (prendere qualcosa in input, darlo in output, passarlo in giro, ecc.) di input/output aggiuntivi delle funzioni. Grazie alle monadi è possibile tenere la business logic (quello che il programma deve fare) separata da come il codice gestisce i side-effects.

Esempi:

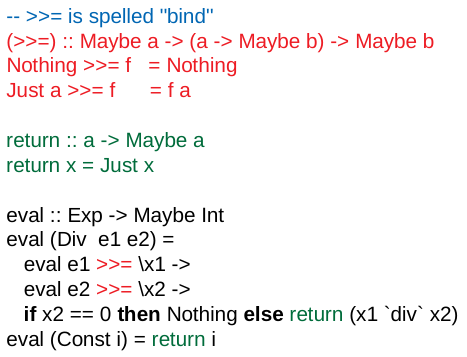


Dato un ADT Exp come indicato sopra, si scrive una funzione eval che data un’espressione restituisca un intero che è la sua semantica.

Un side-effect classico in questo caso può essere la divisione per zero. Un modo per gestire questa situazione è restituendo un option (Maybe) int, in cui se la divisione è per zero allora non c’è un output, altrimenti viene ritornato l’intero (in caso un valore viene valutato Nothing si propaga). In codice:



Il codice non è particolarmente brillante, le parti in rosso e blu si riferiscono solo a come gestire i casi di propagazione dell’errore o meno, inoltre è codice identico ripetuto.



L’operatore **bind** (>>=) introduce l’idea di passare come primo parametro il codice che precedentemente era evidenziato di rosso e come secondo parametro quello evidenziato di blu: Maybe a -> (a -> Maybe b) -> Maybe b. L’idea è che il primo argomento a (tipo generico) può esserci o meno (Maybe), se c’è si ha una funzione che genera un b che può esserci meno (Maybe) e da qui si può ottenere una funzione che genera un b oppure no.

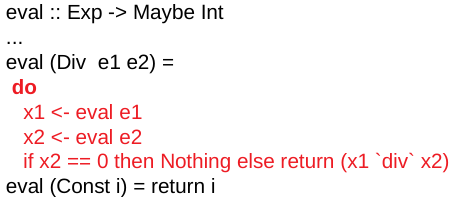
Questo viene definito per casi, se l’input Maybe a è un Nothing allora viene restituito Nothing, se l’input è un Just a allora si applica la funzione f.

Il bind è stato definito in modo che abbia due input: un eval che può esserci o meno e una funzione anonima che prende un elemento sempre di tipo eval e restituisce una funzione che ritorna un risultato o meno.

Viene introdotta (in verde) anche la funzione **return** (non il comando di interruzione return). L’obiettivo di questa funzione return prende un a qualunque e restituisce un Maybe a, mentre return di x restituisce semplicemente Just x.

Mettendo tutto insieme il codice di prima viene compresso. L’idea è che viene valutato e1 e si passa una funzione anonima che prende in input x1, si valuta e2 e si passa in input alla bind una funzione che prende in input x2 ed esegue il controllo che il denominatore non sia zero, in tal caso si restituisce x1 diviso x2. Un modo per leggere la bind è “valuto e1 e chiamo x1 il risultato (se non c’è stato side-effect, in questo caso se non c’è stato errore chiamo x1 il risultato corretto), valuto e2 e chiamo x2 il risultato e si restituisce la computazione”.

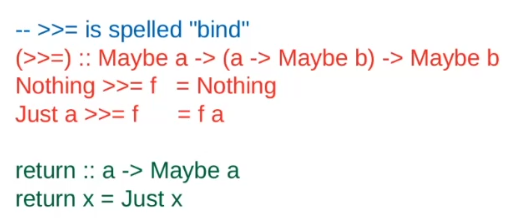
La notazione **do** è zucchero sintattico per rappresentare il codice visto sopra:



SEMBRA programmazione imperativa ma viene convertita nel codice sopra a tempo di compilazione. Il side-effect che viene esposto in questo caso è il Nothing nell’ultima riga rossa, ma non tutti i side-effects sono legati agli options.

NOTA: nel codice originale un elemento di tipo int rappresenta il *risultato* di una computazione, nel nuovo codice un elemento di tipo Maybe int *descrive* una computazione che produrrà un intero.

I tipi sono risultato di una computazione i Maybe (qualcosa) sono la descrizione di una computazione.



L’operatore di binding (>>=) permette di fare *plumbing* (o composizione) di tali descrizioni senza esporre *come* avviene la computazione.

Il Nothing descrive un nuovo modo di procedere nella computazione, ciò è stato possibile estendendo l’intero come opzionale, lasciando al binding la gestione di questo operatore (questa cosa si chiama plumbing).

Quando abbiamo parlato di Maybe abbiamo detto che Maybe o i tipi di dati optional fanno parte di una teoria più generale, questa teoria generale sono le monadi.

In Haskell è impossibile scrivere l’hello world senza una monade.

Si può generalizzare quanto visto per gestire il side effect della divisione per 0 anche per altri side effect.

Aggiungiamo adesso all’esempio precedente la radice quadrata come espressione:



La *isqrt* (integer square root) prende un intero e restituisce un intero e compone tre funzioni (il punto è la composizione di funzione) prima viene chiamata fromIntegral (che trasforma un intero in un float), poi chiama sul risultato la radice quadrata (applicabile solo a float da libreria) e infine chiama sul risultato floor per restituire un intero.

Tuttavia in matematica la radice quadrata di, ad esempio 4, è un valore non deterministico che restituisce ±2.

Per descrivere il non determinismo in Haskell è necessario descrivere eval come una funzione che prende in input un Exp e restituisce una lista di interi, ovvero tutte le possibilità.

La radice quadrata sarà implementata come mostrato sopra, come prima cosa si fa una chiamata ricorsiva, valutando e1 e assegnando il risultato a l1, che sarà una lista di risultati, e come output si darà la lista l1 di tutte le radici positive, concatenata (++) alla lista con tutte le radici cambiate di segno.

Anche qui la business logic è offuscata dalla gestione del non determinismo.

La soluzione con bind:



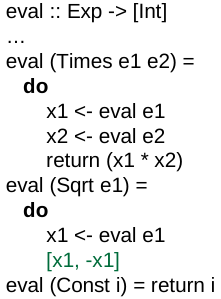
\x1 lo slash è lambda cioè una lambda che prende in input x1.

Se l’input è vuoto si ritorna la lista vuota, altrimenti se è una lista con x in testa e xs in coda, e si vuole fare bind con la funzione f, si chiama ricorsivamente il bind f su x in quanto ha tipo a (perché è la testa della lista) e ritorna una lista di b, poi si fa una chiamata ricorsiva xs bind f che ritorna nuovamente una lista b, avendo due liste di b vengono concatenate con ++.

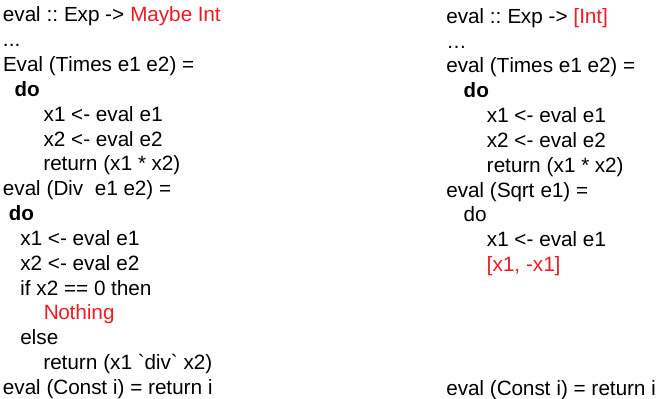
“Per ogni elemento della lista in input viene chiamata f che restituisce una lista in output e tutte queste liste vengono concatenate”.

NOTA: nell’ultimo eval (Sqrt e1) = … manca la radice quadrata isqrt.

Utilizzando la notazione do:



Le due soluzioni (division by zero e non determinismo) a confronto:



Ora l’idea è quella di trovare una generalizzazione di Maybe Int e [Int], si cerca un modo per astrarre sui costruttori di tipo Maybe e Lista tramite una generalizzazione comune che abbia dichiarato bind e return. L’elemento descritto altro non è che una **type-class**. Si vuole quindi scrivere un predicato su un costruttore di tipo, tale che predichi l’esistenza di una funzione return e di una funzione bind.

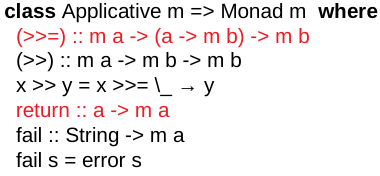
Un funtore applicativo è una via di mezzo tra un funtore e una monade.

Un funtore sono “cose” su cui ci si può mappare, tipo liste, tipi Maybe, alberi…

In haskell i funtori sono descritti dalla type-class Functor cha ha un metodo fmap con tipo  
fmap :: (a -> b) -> f a -> f b

Questo inizia a essere simile al concetto di monade perché come si può vedere si ha un valore wrappato e si prende in input una funzione che cambia il valore interno mantenendo il wrapping (e il wrapping può essere la lista, l’albero, il maybe, etc…)

Per i funtori applicativi questo è quello che si trova (qui è stato leggermente semplificato) nella libreria standard di Haskell:

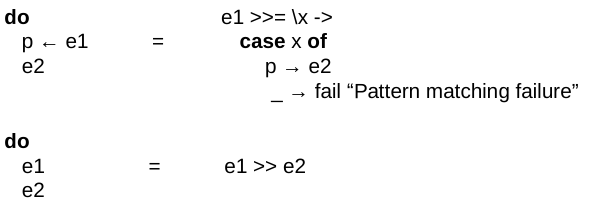


Questo si legge come:

m, che è un costruttore di tipo, gode del fatto di essere una monade:

1. se è applicativo (la forma più debole di monade nella gerarchia matematica, ma non è importante ai fini dell’esempio)
2. allora è anche una monade, se implementa:
   1. la funzione bind (>>=),
   2. la funzione degenere (>>, chiamata fish), ovvero quando si chiama una funzione solo per il suo side effect (simile al punto e virgola nei linguaggi imperativi)
   3. la return:
   4. la fail: per comporre monadi che rappresentano errori della computazione

Esempi di come viene riscritto il costrutto do:



Una monade deve implementare certe funzioni (bind, fish, return e fail), ma non come si vuole, l’implementazione deve rispettare le **leggi monadiche**. Se viene implementata una monade che non rispetta queste leggi è segno che c’è qualcosa che non va nel codice.

Leggi monadiche:

1. return x >>= f = f x

Se si vede il bind come la composizione di side effects, la return è l’assenza di side effects (o side effect identità, che non fa niente). Quindi ritornare x bind f deve essere la stessa cosa che fare f di x, ovvero se si restituisce x senza fare alcun side effect e lo si *compone* con f, allora tanto vale fare direttamente f x.

2. m >>= return = m

In modo analogo preso il risultato della m, con il suo side effect, e lo si combina con una return che non ha alcun side effect, deve essere uguale ad eseguire direttamente m.

3. (m >>= f) >>= g = m >>= \x -> f x >>= g

Ci si aspetta che la composizione sia associativa. Quindi se si parte da m e si applica f (al contenuto di m spacchettato) combinando i side effects, e successivamente g combinando i side effects deve essere uguale a fare g combinato ad una funzione che applica f a x, questo combinato ad m (si sposta l’associatività a destra). La f è stata aperta perché è g ad essere bindato con fx (il quale ritorna una monade M b), dopodichè si avrà ancora la funzione da \x ma stavolta ritorna una monade M c risultante dalla combinazione con g, la quale verrà applicata ad m.

I restanti tre sono simili ma usando il fish:

4. return x >> m = m

5. m >> return x = return x

6. (m >> n) >> I = m >> (n >> I)

visto che fish è definito dalla bind, queste valgono automaticamente se valgono le prime 3 della bind, se invece si ri-definisce il fish bisogna farle valere anche queste quando si implementa una monade.

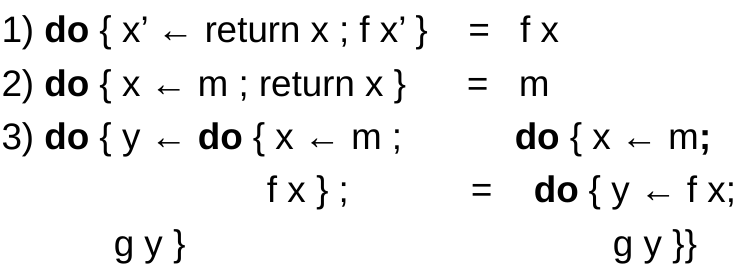
(parentesi algebrica)

Si è detto che le monadi altro non sono che un monoide nella categoria degli endofuntori. qui si vede che sono monoidi. Un monoide per l’operatore >>=

Infatti si può pensare e vedere che c’è:

* elemento neutro (return)
* chiusura (chiamata moltiplicazione che dati due monoidi ne restituisce un terzo)
* associatività (terza regola)

In sintassi do:



Dimostrazioni dati gli options:

Nothing >>= f = Nothing

(Just x) >>= f = f x

return x = Just x

Dimostrazione per la legge 1:

return x >>= f

= Just x >>= f

= f x

Dimostrazione per la legge 2:

m >>= return

= case m of

Nothing -> Nothing

| Just x -> return x

= case m of

Nothing -> Nothing

| Just x -> Just x

= m

Dimostrazione per la legge 3:

(m >>= f) >>= g

= (case m of

Nothing -> Nothing

| Just x -> f x) >>= g

= case m of

Nothing -> Nothing >>= g

| Just x -> f x >>= g

= case m of

Nothing -> Nothing

| Just x -> (\x. f x >>= g) x

= m >>= \x. f x >>= g

Esempio concettuale di monadi in pseudocodice OO:

**interface** monad {

// Il metodo bind prende in input una monade di tipo I (esempio una lista di interi) e una qualche funzione f che prende in input un elemento di tipo I (esempio un intero) e restituisce una monade di tipo O (esempio una lista di tipo float) e la bind restituisce una monade di tipo O (esempio una lista di elementi float)

monad<O> **bind**<I, O> (monad<I> i, f: I -> monad<O>);

// Il metodo return prende in input un elemento di tipo I (esempio un intero) e restituisce in output una monade con quel tipo I (esempio una lista con solo l’intero i)

monad<I> **return**<I> (I i);

}

questo in sintassi Java like è l’esempio della somma nel l’ADT exp che avevamo visto all’inizio dove m può essere una monade come Maybe.

**eval**(monad m, expr e) {

**switch** e **of**:

**case** Plus(e1, e1):

m.**bind**<Int, Int>(eval e1,

// x1 è l’elemento (o gli elementi) contenuto in e1

(x1) => m.**bind**<Int, Int>(eval e2,

// x2 è l’elemento (o gli elementi) contenuto in e2

(x2) => m.**return**(x1 + x2)

)

)

...

}

In Haskell esistono anche **Monad Transformers**, ovvero elementi che prendono in input una monade e la arricchiscono con altre funzionalità. Ad esempio la monad transformer di writer prende in input una monade e ne restituisce una uguale ma che è anche in grado di scrivere. Uguale per la reader.

Una monad transformer è una monade di ordine superiore.

**List comprehension**

Le seguenti tre espressioni sono tutte differenti sintassi per la stessa cosa:

[(x,y) | x <- [1,2,3], y <- [1,2,3], x /= y]

do x <- [1,2,3]

y <- [1,2,3]

True <- return (x /= y)

return (x, y)

[1,2,3] >>= (\ x -> [1,2,3] >>= (\y -> return (x /= y) >>=

(\r -> case r of

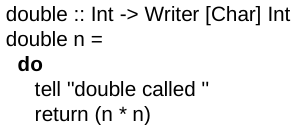
True -> return (x, y)

\_ -> fail "")))

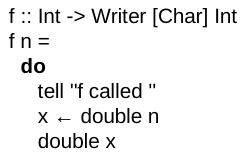
**Writer monad**

L’idea è che in qualunque momento nel codice si vuole essere in grado di fare output.

Esempio di utilizzo:



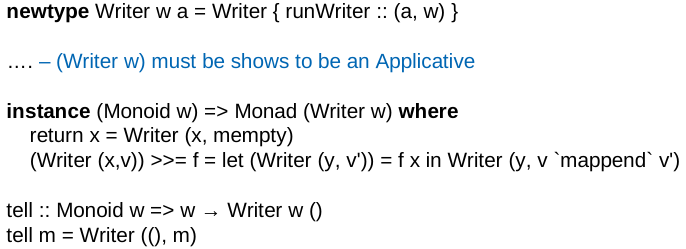
Prende in input un intero e restituisce in output la monade Writer, la quale prende due tipi: una lista di caratteri (una stringa) e un intero (questi due tipi non sono fissi, qui servivano una stringa e un intero ma si potrebbero restituire un intero e un double, e così via).





Si vuole che in output venga restituito oltre che il risultato anche tutti i side effects, in questo caso le stampe.

L’**implementazione** della monade Writer è la seguente:



Si dichiara un nuovo tipo di dato algebrico Writer w a (con w e a due variabili di tipo, a rappresenta il dato e w quello che ci stò scrivendo).

Writer { runWriter :: (a, w) } è un costruttore Writer che ha come argomento un record, con un solo campo runWriter che ha come unico elemento dentro una coppia (a, w), in breve indica che Writer è una coppia contenente il dato a e il dato w che consiste nel side effect.

Il side effect è proprio questo, fornire il dato a in output accompagnato da un side effect (ad esempio una stringa in output).

Successivamente bisogna dimostrare che è una monade, tramite instance … => Monad (Writer w) … Writer è un’applicazione parziale, in quanto Writer è un costruttore di tipo doppio, che prende un a e un w, ovvero costruisce un tipo dati due tipi in input, tuttavia anche solo con w rimane un costruttore di tipo, che viene dichiarato avere le caratteristiche di una monade.

------------- Breve excursus -------------

Un **Monoide** in matematica è un insieme S, un elemento e ed un operazione \*: (S, e, \*). In cui:

* S è un insieme
* e è un elemento di S chiamato elemento neutro
* \* è una funzione binaria su s (ovvero S x S → S) tale che:
  + e \* x = x per ogni x (e elemento neutro a sinistra)
  + x \* e = x per ogni x (e elemento neutro a destra)
  + (x \* y) \* z = x \* (y \* z)

Esempi di monoidi: (ℕ, 0, +), (ℕ, 1, \*), (Stringhe, “”, ++), ...

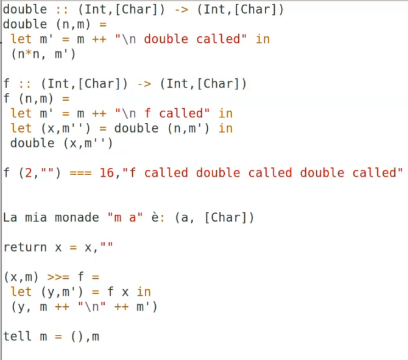
------------------------------------------

*return* x vuole fornire una rappresentazione di x senza side effects, quindi una coppia contenente x e side effect vuoto, mempty è l’elemento neutro nel Monoid w (quando composto con gli altri elementi non li altera).

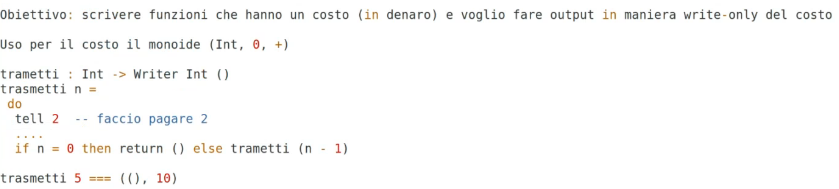
Per la *bind* abbiamo un Writer (x, v), ovvero si è già calcolato x con side effect v e vi si vuole applicare agli elementi del monoide la funzione f. Prima di tutto quindi si applica la funzione f ad x, la funzione f prende il dato e restituisce un elemento del monoide (y, v’), il quale viene “impacchettato” nel monoide Writer, lasciando come dato finale y, mentre si combina l’output con il side effect originario (mappend è l’operazione \* del monoide, così come mempty la e).

Una volta descritto Writer come un monoide è possibile implementare la funzione tell, la quale prende un messaggio m e restituisce un Writer con un unit (nulla, come in Scala) e il messaggio. Il tipo di tell è: per ogni w, supponendo sia un Monoide, si ha una funzione da w a Writer x ().

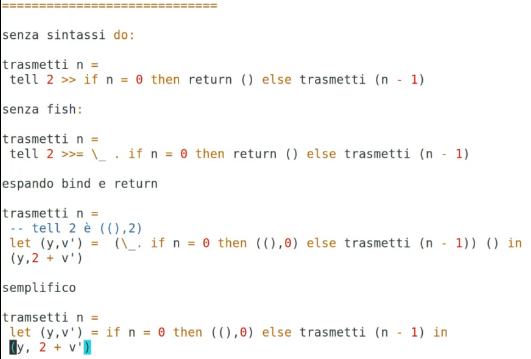
Implementazione grezza senza e con monade:



Esempio on the fly:

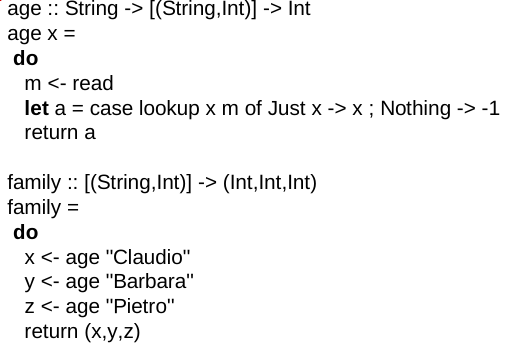


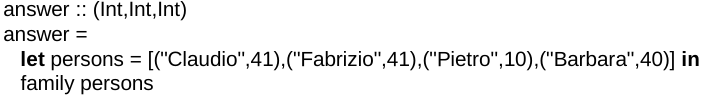
La funzione trasmetti riscritta e semplificando:



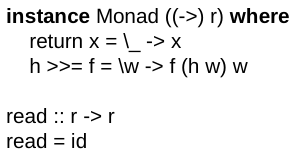
**Reader monad**

Esempio di utilizzo





L’implementazione della monade è la seguente:



Si passa il costruttore di tipo (->) r, il simbolo -> è un costruttore di tipo per le funzioni, da a in r. Quello che fa la monade è prendere il costruttore ->, prendere r e creare una funzione che va da r in r (r -> r) implementandone return e bind.

La return prende qualsiasi cosa e ritorna x, ma bind prende w e ritorna la f applicata a w e all’esito di h su w.

La funzione read è la funzione identità, dato lo stato restituisce lo stato stesso.

Riscrivendo più in dettaglio l’implementazione della monade:

return x = \stato -> x

return x è una funzione che prende in input lo stato, lo ignora e ritorna x:

tipaggio: h :: m a ovvero: h :: stato -> a

tipaggio: f :: a -> m b ovvero: f :: a -> stato -> b

tipaggio: h >> f :: m b ovvero: h >> f :: stato -> b

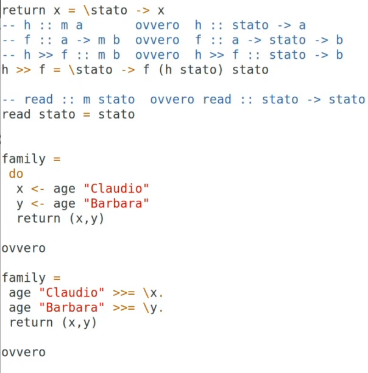
h >> f = \stato -> f (h stato)

h bind f deve dare in output una funzione da stato in qualcosa chiamare h sullo stato restituisce qualcosa che verrà dato in input alla f insieme allo stato nuovamente.

tipaggio: read :: m stato ovvero read :: stato -> stato

read stato = stato

Implementazione più esplicita:





**Riassunto Monadi** [lezione 18 Maggio]

a sono i tipi di dato, nell’ambito delle monadi consideriamo i tipi di dato come il risultato di una computazione (es. int è una computazione che ha creato un intero, bool crea un booleano e così via).

m a è un costruttore di tipo m applicato ad a (m è una **monade** quando implementa certe funzioni con certe proprietà matematiche), nell’ambito delle monadi consideriamo m a non il risultato di una computazione, bensì la descrizione di una computazione che calcola un a (es. per la Reader monad la m a è una funzione che prende uno stato e restituisce un a).

Questa non è l’unica possibile lettura delle monadi, ce ne sono tante altre, ad esempio m a può essere visto anche come un arricchimento del tipo di dato a (es. per la Writer monad la m a era un valore di tipo a assieme ad un altro valore che si sta calcolando, come il log di ciò che è successo, il costo calcolato fino ad un certo momento, ecc).

Un’altra possibile lettura utile per certe monadi è vedere m a come un contenitore di dati di tipo a (es. la List monad usata come la descrizione di una computazione non deterministica, può essere vista come un insieme (contenitore) che contiene coppie multiple di tipi di dato a).

Un altro esempio è la Maybe monad che abbiamo visto come la descrizione di una computazione che calcola un a e la descrizione di una computazione che può fallire, ovvero non trovare alcun a. Come arricchimento può essere vista come l’informazione aggiuntiva che l’operazione è andata a buon fine e il risultato è presente. Infine può essere vista come un contenitore che può o meno contenere un dato di tipo a.

Queste tre letture sono solo tre esempi, ce ne sono molte altre, le monadi astraggono contemporaneamente tanti fenomeni diversi.

Una monade può essere implementata tramite una type-class, perché sia tale devono essere presenti almeno due funzioni:

- return : a -> m a

- bind : m a -> (a -> m b) -> m b

Senso delle funzioni in base all’interpretazione della monade:

* m a è la **descrizione di una computazione** che calcola a (es. Reader monad)
  + return : lo fa in maniera semplice, priva di side effects, ecc.
  + bind : composizione di computazioni che combina i side effects (se ci sono), o altro. Ad esempio se il side effect è l’apertura di un certo numero di file, allora m a -> è un valore a insieme alla lista dei file aperti, (a -> m b) è una funzione che prende a e calcola un b aprendo dei nuovi file ad esempio, il risultato -> m b sarà quindi un b con una nuova lista di file aperti, ovvero quelli aperti in m a concatenati a quelli aperti dalla funzione (a -> m b). In questo caso si sta quindi componendo il side effect, che è l’apertura dei file.
* m a è l’**arricchimento di una computazione** che calcola un a (es. Writer monad)
  + return : è il non arricchimento (o meglio l’arricchimento con un elemento neutro)
  + bind : applica una funzione combinando gli arricchimenti. Quindi m a è un dato a arricchito, la funzione (a -> m b) crea un dato b arricchito, il risultato sarà un dato b con un arricchimento che combini quello iniziale con quello di (a -> m b).
* m a è un **contenitore di dati** di tipo a (es. List monad)
  + return : inserisce un dato nel contenitore, ottenendo un singoletto
  + bind : effettua una sorta di list comprehension sul contenitore. Quindi ad esempio m a contiene degli elementi di tipo a, (a -> m b) per ogni elemento di tipo a calcola degli elementi di tipo b, che verranno concatenati insieme in un unico contenitore -> m b.

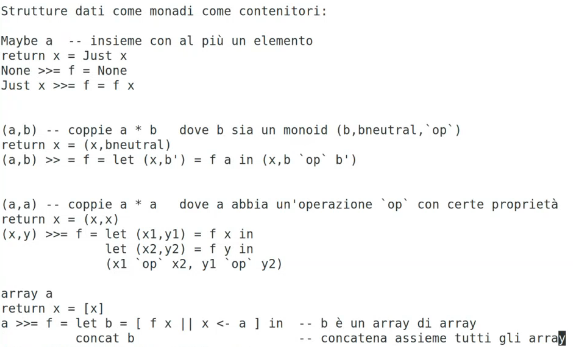
È importante che le leggi monadiche vengano rispettate, nello specifico ci dicono che:

1. la return è l’elemento neutro rispetto a bind

2. la bind è associativa.

Applicandole alle interpretazioni date sopra:

* m a come **descrizione di una computazione** che calcola un a
  + 1. la computazione return non deve avere side effects, perché deve essere neutra rispetto a fare poi il bind.
  + 2. la composizione è priva di “memoria storica” , ovvero non ha importanza in che ordine viene computato il codice, questo è un modo per lasciare libertà al compilatore (associatività).
* m a come **arricchimento di una computazione** che calcola un a
  + 1. la return non può arricchire
  + 2. la composizione degli arricchimenti deve essere associativa, questo non è sempre scontato gli arricchimenti a volte necessitano di un ordine, ma è un vincolo che va rispettato.
* m a come **contenitore di dati** di tipo a
  + 1. la return crea un singoletto
  + 2. la composizione sia priva di “memoria storica”.



Come si sviluppa una nuova monade

Di solito o le monadi sono naturali (come per le strutture dati, quasi sempre hanno le caratteristiche di una monade, in questi casi è sufficiente definire le operazioni bind, return, …) oppure se si vogliono utilizzare per descrivere computazioni con side effect allora è necessario descrivere il side effect stesso, cercando la struttura dati più opportuna per la sua implementazione.

Ad esempio per la monade **Writer**, si vuole che il *side effect* sia “put x”, ovvero produrre su un side channel una x di qualche tipo che deve essere combinata con ciò che è stato prodotto in precedenza (es. concatenare stringhe in un log, o sommare interi che rappresentano un “costo”).

Quindi writer a = (a, m) in cui m è l’output aggiuntivo.

return x = (x, mneutral) per assicurare che esista un elemento neutro e che non alteri nulla quando concatenato è necessario richiedere come condizione che m sia un monoide:

monoid m => writer a = (a, m)

Mentre per quanto riguarda la bind:

(x, m1) >>= f = let (y, m2) = f x in (y, m1 ‘op’ m2)

Si ha la coppia x e messaggio m1, si vuole fare il bind con la funzione f quindi si chiamerà f su x, la quale ritornerà una coppia y e messaggio m2, a questo punto verrà ritornato y e la combinazione dei due messaggi m1 ed m2, l’operazione ‘op’ di combinazione sarà poi definita all’interno del monoide.

Infine la put x :: writer (), ovvero la put di un messaggio x ha tipo writer di unit (in quanto utilizzata solo per il side effect) ed è implementata come put x = ((), x), si ignora il valore mettendo un unit e si tiene in considerazione solo il messaggio x. Si sfrutteranno le qualità della monade writer per concatenare x al resto ad ogni chiamata.

Per quanto riguarda la monade **Reader** invece deve restituire un valore, scelto all’inizio, che rimarrà costante per tutta l’esecuzione del programma, in sostanza legge un qualcosa che non si vuole mutare (es. un database o un file read only).

Quindi reader a = m -> a in cui m è ciò che si vuol leggere e viene restituito a.

La read avrà tipo, read :: reader m, ovvero sarà una monade reader che produrrà un tipo di dato m, ovvero il valore che si sta implicitamente “portando dietro”.

Come implementazione invece la read sarà la funzione identità, ovvero read = \x. x (prende in input una x di tipo m e restituisce x in quanto è esattamente la reader). L’implementazione della reader è:

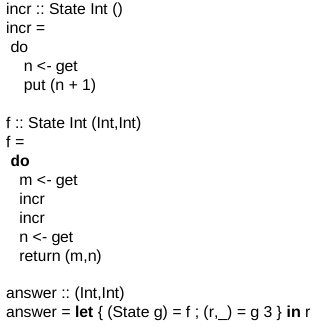
return x = \\_. x

f >>= g = \x. g (f x) x

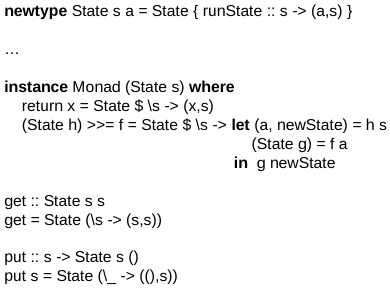
(la g è una funzione a -> m b quindi viene restituita x alla fine in quanto è m b)

Infine per quanto riguarda la monade **State**, l’idea è che è presente uno stato rappresentato da una variabile intera. La funzione incr ad esempio ha come effetto quello di incrementare la variabile di stato, il suo tipo è quindi State monad Int unit, int indica il tipo della variabile di stato e unit indica che la state monad viene usata solo per il suo side effect. L’implementazione avviene con una get per leggere il valore della variabile e una put per aggiornarne il valore incrementato di uno.

f invece è una funzione che “lavora” una variabile anonima di tipo intero e restituisce una coppia di interi. La sua implementazione avviene con una get per leggere il valore della variabile, un doppio incremento, un’ulteriore lettura e in output viene restituita la coppia con i valori dello stato prima e dopo gli incrementi. Infine in answer viene utilizzata f.



L’implementazione della State monad è la seguente:



Per leggere e scrivere questo intero, in un linguaggio funzionale, bisogna “passarlo in giro”, quindi prendere in input il precedente valore dello stato, e dare in output il nuovo valore dello stato.

**Altre monadi**

Altri esempi di monadi possono essere:

- distribuzioni probabilistiche (es. operatore di lancio di una moneta, descrive tutti i possibili output con le relative probabilità). La return ad esempio deve restituire una lista con un valore con 100% di probabilità, mentre la bind prende una lista con delle percentuali di probabilità, e le combina moltiplicandole.

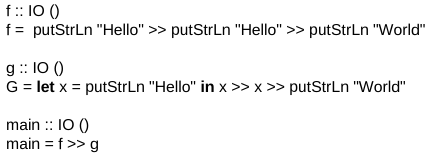
- eccezioni (tipo la Maybe monad, ma con valori trasportati).

- concorrenza e software transactional memory (ovvero i thread comunicano con la memoria condivisa e la nozione di transazione atomica implementata in maniera ottimistica).

**IO Monad**

La IO Monad gestisce l’interazione con il sistema (file, socket, timers, …). La main deve avere tipo IO (), in cui () è unit.

NOTA: un valore di tipo “IO a” non è il risultato di un’esecuzione che ha prodotto side effect, ma è la **descrizione** di tale esecuzione.



In OCaml o C la funzione g stamperebbe Hello una volta sola e alla x verrebbe assegnato unit, dopodichè farebbe “unit”, “unit” e stamperebbe “World”. Mentre in Haskell f e g fanno la stessa cosa, questo è il principio della referential transparency. Questo perché putStrLn “Hello” in Haskell non stampa “Hello” ma genera il *codice assembly per stampare* Hello.

La magia è che il runtime di Haskell contiene un interprete per un linguaggio imperativo.

Monade STM: l’idea dietro alla Software Transactional Memory è che in presenza di memoria condivisa non si scrive effettivamente sulla memoria, ma le operazioni vengono scritte su un journal e si effettua un commit tutto in una volta (simile a quando in Linux si scrive su file, in realtà si scrive l’intenzione di scrivere sul file, e quando si chiude il file se nessun altro lo sta scrivendo viene fatto il commit).