Reasoning about Composing Monads

Denis Mazzucato
Master in Computer Science
University of Padova

2019
February

Abstract

I *Monads* sono un'importante strumento per i linguaggi funzionali. Differenti *Monads* possono essere utilizzati per modellare un ampio gruppo di features di programmazione. Vedremo come questi meccanismi portino benefici in un semplice esempio. Molto spesso progetti di qualsiasi entità richiedono combinazioni di queste feature, è quindi importante avere tecniche per poter combinare varie *Monads* in un singolo *Monad*. In pratica questa non è un'operazione difficile, infatti è possibile combinare qualche feature specifica per costruire nuovi *Monads* con facilità. Le tecniche solitamente utilizzate però sono generalmente ad-hoc e sembra molto più difficile il problema di trovare una caratterizzazione generale per combinare *Monads* in maniera arbitraria. Vedremo poi tre costrutti generali per comporre *Monads*, ognuno di esso collegato alle proprie condizioni e funzioni ausiliarie. Concludendo con un esempio pratico.

Indice

1 Introduction

Recentemente, il concetto di Monade è diventato un importante e pratico strumento per i linguaggi funzionali. La ragionde di questo dipende dal fatto che i *Monads* propongono un framework uniforme per descrivere una enorme set di caratteristiche tipicamente imperative all'interno di un sistema puro.
Nella sezione 2 vedremo una possibile implementazione per una funzione di valutazione di una semplice grammatica, esplicitando tutta la computazione necessaria.
L'introduzione ai *Monads* è definita nel capitolo 3, dove definisco la struttura di un Monade e ne descrivo le proprietà.
Successivamente, la sezione 4, descrive lo stesso interprete in chiave monadica, esponendo i vantaggi dell'applicazione delle *Monads* su questo semplice esempio.
La sezione 5 pone le basi e descrive la caratterizzazione generale del concetto di *Composing Monads*, questa sezione non entra nel dettaglio dell'algebra astratta ad alto livello, ma ragione su una astrazione di un linguaggio funzionale puro. Questa è la sezione principale di questo report, definendo formalmente le tre classi di possibili tecniche per combinare *Monads* in maniera arbitraria.
Passiamo poi a definire nella sezione 6 un sistema per generalizzare la composizione dei monadi.
L'ultima sezione (capitolo 7) infine implementa i meccanismi definiti nella sezione 5 nello stesso esempio visto nei primi due capitoli.

Per questo report non assumo nessun linguaggio di programmazione, gli esempi e la sintassi utilizzata fanno riferimento a *Gofer* [1], simile ad *Haskell*. In appendice D troviamo anche un pratico esempio in *Scala* per la composizione generalizzata.

In appendice A definisco le dimostrazioni che ritengo più importanti e istruttive. Alcune dimostrazioni semplici e/o limitate sono definite tra le righe, solitamente delimitate da un semplice box. Mentre in appendice B mostro un esempio di come un *composed monad* non riesce ad esprimere il concetto di *State Transitions*, per questo esempio noteremo come un'implementazione ad-hoc sia diversa dall'equivalente rappresentazione mediante composizione.

[1] *Gofer*, un piccolo, sperimentale e semplice linguaggio puramente funzionale

2 A Simple Evaluator

2.1 Abstract Syntax

La funzione di valutazione riceverà in input espressioni generate dalla seguente grammatica:

$Expr \rightarrow Const\ Value$	Valore costante
$\mid Var\ Name$	Variabile in memoria
$\mid Expr + Expr$	Addizione tra espressioni
$\mid Trace\ String\ Expr$	Struttura di log
$Env \rightarrow [(Name, Value)]$	Ambiente di mapping delle variabili
$Name \rightarrow String$	
$Value \rightarrow Int$	

Per mantere l'esempio semplice ho ridotto al minimo le espressioni generabili nel linguaggio, permettendo solamente costanti, variabili, addizioni ed un semplice meccanismo di log.

Si noti come questa grammatica permetta comunque l'implementazione di alcune caratteristiche basilari come: la gestione degli errori per variabili sconosciute, un sistema di output e la gestione dell'accesso delle variabili nel sistema.

2.2 Implementation

Per la prima implementazione della funzione di valutazione utilizzo i seguenti tipi astratti:

$$\textbf{data}\ Error\ a = Error\ String \mid Ok\ a$$

$$\textbf{data}\ Writer\ s\ a = (s, a) \qquad \text{con s :: } Monoid\text{, vedi C}$$

Con questi due tipi riusciamo a codificare la gestione degli errori e la gestione del log, per la codifica dell'*Enviroments* definiamo semplicemente la funzione di valutazione, che riceverà in input lo stato iniziale e ritornerà un valore (se nessun errore è occorso) e un output, possibilmente vuoto.

$$eval :: Expr \rightarrow Env \rightarrow Writer\ [String]\ (Error\ Value)$$

Già dal tipo della funzione *eval* si può notare come sia poco pulito, difatti il tipo di ritorno è molto complesso dato che deve gestire ben due caratteristiche non banali, la gestione dell'output e degli errori. Ora passiamo all'implementazione dell'interprete.

$$
\begin{aligned}
eval\ (Const\ v)\ env &= ([\,],\ Ok\ a) \\
eval\ (Var\ x)\ env &= ([\,],\ lookup\ env\ x) \\
eval\ (e_0 + e_1)\ env &= \textbf{case}\ eval\ e_0\ env\ \textbf{of} \\
&\qquad (out_0,\ Error\ s) \rightarrow ([\,],\ Error\ s) \\
&\qquad (out_0,\ Ok\ v_0) \quad \rightarrow \textbf{case}\ eval\ e_1\ env\ \textbf{of} \\
&\qquad\qquad (out_1,\ Error\ s) \rightarrow ([\,],\ Error\ s) \\
&\qquad\qquad (out_1,\ Ok\ v_1) \quad \rightarrow (out_0 +\!\!+\ out_1, v_0 + v_1) \\
eval\ (Trace\ s\ e)\ env &= ([s],\ eval\ e\ env)
\end{aligned}
$$

Prima di procedere notiamo l'utilizzo della funzione *lookup* definita come segue.

$$
\begin{aligned}
lookup &:: Env \rightarrow Name \rightarrow Error\ Value \\
lookup\ [\,]\ x &= Error\ \text{"}Value\ unbounded\text{"} \\
lookup\ [(name,\ value) : xs]\ x &= \textbf{if}\ name = x\ \textbf{then}\ Ok\ value\ \textbf{else}\ lookup\ xs\ x
\end{aligned}
$$

Come possiamo notare questo interprete è complicato e necessita di esplicitare tutti i passi computazionali esponendolo ad errori. Il codice risulta meno mantenibile e ancora meno estensibile, un qualsiasi cambiamento, o richiesta di nuova *feature*, richiederebbe la modifica di quasi la totalità delle righe di codice. In un linguaggio imperativo banale l'implementazione di queste caratteristiche è un task molto semplice e non richiede particolari competenze, i *Monads* serviranno proprio per portare questa facilità d'esecuzione dentro ad un linguaggio funzionale puro.

3 Monads

Walder[2] definisce un *Monad* come un tipo costrutture unario M, con un unico parametro, insieme alle tre funzioni *map*, *unit* e *join* definite dal tipo

$$\begin{aligned} &map :: (a \to b) \to (M\ a \to M\ b) \\ &unit :: a \to M\ a \\ &join :: M\ (M\ a) \to M\ a \end{aligned}$$

Altre caratterizzazioni (equivalenti) possono essere definite per il concetto dei *Monads*, per esempio come definito in Walder[3]

$$bind :: M\ a \to (a \to M\ b) \to M\ b$$

Ragionando su questa forma è semplice notare come sia del tutto equivalente alle precedenti mediante le relazioni

$$\begin{aligned} bind\ x\ f &= join\ (map\ f\ x) \\ join\ x &= bind\ x\ id \end{aligned}$$

In ogni caso, per questo report, ho scelto di lavorare con *map*, *unit* e *join* perchè rendono più marcata la differenza tra *Monads* e *Premonads* e inoltre rendono più semplice lo svolgimento delle dimostrazioni in questo framework.

In aggiunta a queste funzioni si richiede di soddisfare una collezione di proprietà algebriche

$$\begin{aligned} map\ id &= id && (M1) \\ map\ f\ .\ map\ g &= map\ (f\ .\ g) && (M2) \\ unit\ .\ f &= map\ f\ .\ unit && (M3) \\ join\ .\ map\ (map\ f) &= map\ f\ .\ join && (M4) \\ join\ .\ unit &= id && (M5) \\ join\ .\ map\ unit &= id && (M6) \\ join\ .\ map\ join &= join\ .\ join && (M7) \end{aligned}$$

Per lo scopo del report è conveniente definire per parti un *Monad*; se M è un tipo costruttore unario diremo che

1. M è un *Functor* se c'è una funzione $map :: (a \to b) \to (M\ a \to M\ b)$ che soddisfa le leggi *(M1)* e *(M2)*.
2. M è un *Premonad* se c'è una funzione $unit :: a \to M\ a$ che soddisfa la legge *(M3)*.
3. M è un *Monad* se c'è una funzione $join :: M(M\ a) \to M\ a$ che soddisfa le leggi *(M4)*, *(M5)*, *(M6)* e *(M7)*.

Examples

Mostriamo come due linguaggi di programmazione (non sperimentali) definiscano i concetti di $Monads$.

Haskell Per Haskell un Monade è definito in modo molto simile a $Gofer$, ovvero è un oggetto con le proprietà

- $return$, corrispondente a $unit$
- $bind$, corrispondente a $join$, vedi l'equazione $bind\ x\ f = join\ (map\ f\ x)$

Scala Per scala invece un Monade è definito con le seguenti proprietà

- $unit$
- $flatMap$, corrispondente a $join$, vedi $flatMap$ è l'operatore $bind$

4 Monadic Evaluator

Ora che sappiamo cosa siano i *Monads* possiamo applicari al primo esempio nella sezione 2, in questa sezione costruiremo un valutatore monadico per rappresentare le caratteristiche volute. Senza entrare troppo nel dettaglio divido l'esempio in due parti per semplicità, nelle prossime sezione vedremo come combinare il tutto in maniera semplice ed elegante con un sistema generalizzabile.

4.1 Exceptions

Per rappresentare la gestione delle eccezioni definisco il *Monad*

$$\textbf{data}\ Error\ a = Raise\ String \mid Return\ a$$

$$\begin{aligned} map &:: (a \to b) \to Error\ a \to Error\ b \\ map\ f\ (Raise\ s) &= Raise\ s \\ map\ f\ (Return\ x) &= Return\ (f\ x) \end{aligned}$$

$$\begin{aligned} unit &:: a \to Error\ a \\ unit &= Return \end{aligned}$$

$$\begin{aligned} join &:: Error\ (Error\ a) \to Error\ a \\ join\ (Raise\ s) &= Raise\ s \\ join\ (Return\ x) &= x \end{aligned}$$

La dimostrazione la si può trovare in appendice A.1. Definiamo quindi il corrispondente interprete per la grammatica 2.1

$$\begin{aligned} eval &:: Expr \to Env \to Error\ Value \\ eval\ (Const\ v)\ env &= unit\ v \\ eval\ (Var\ x)\ env &= lookup\ env\ x \\ eval\ (e_0 + e_1)\ env &= bind\ (eval\ e_0\ env) \\ &\qquad \lambda\ v_0 \to bind\ (eval\ e_1\ env) \\ &\qquad\quad \lambda\ v_1 \to unit(v_0 + v_1) \end{aligned}$$

Nota se *Error* è *Monad* posso usare sia join che bind, per questo esempio è comodo utilizzare il bind operator. Ora passiamo ad osservare i pregi di questo esempio, notiamo subito come il codice sia più coinciso e descriva meglio l'intenzione piuttosto che la computazione. L'utilizzo di *unit* e *bind* semplifica sia l'implementazione che il ragionamento sull'interprete rendendolo più resistente a possibili errori di programmazione.

4.2 Output

Per l'output definiamo il tipo astratto *Writer* che definisce un meccanismo per tenere la traccia del log prodotto (se prodotto), richiedo inoltre che il tipo di

$$\textbf{data}\ Writer\ s\ a = (s,\ a)$$

$$map :: Monoid\ s \Rightarrow (a \to b) \to Writer\ s\ a \to Writer\ s\ b$$
$$map\ f\ (s,\ a) = (s,\ f\ a)$$

$$unit :: Monoid\ s \Rightarrow a \to Writer\ s\ a$$
$$unit\ x = (zero,\ x)$$

$$join :: Monoid\ s \Rightarrow Writer\ s\ (Writer\ s\ a) \to Writer\ s\ a$$
$$join\ (s',\ (s'',\ a)) = (s' +\!\!+ s'',\ a)$$

Definiamo, come per le eccezioni, la funzione per la grammatica 2.1

$$eval :: Expr \to Env \to Writer\ [String]\ Value$$
$$eval\ (Const\ v)\ env = unit\ v$$
$$eval\ (Var\ x)\ env = \textbf{case}\ lookup\ env\ x\ \textbf{of}$$
$$Raise\ s \to unit\ defaultValue$$
$$Ok\ x \to unit\ x$$
$$eval\ (e_0 + e_1)\ env = bind\ (eval\ e_0\ env)$$
$$\lambda\ v_0 \to bind\ (eval\ e_1\ env)$$
$$\lambda\ v_1 \to unit(v_0 + v_1)$$
$$eval\ (Trace\ s\ e)\ env = ([s],\ eval\ e\ env)$$

Per il caso delle eccezioni 4.1, i *Monads* portano gli stessi benefici osservati precedentemente. Unica nota negativa, in questo caso per l'espressione $Var\ x$ bisogna controllare che la variabile sia stata definita, non gestendo le eccezioni non abbiamo alcun modo di ritornare un caso di errore quando la variabile non è definita.

4.3 Observations on Monadic Evaluator

Da questi due esempi notiamo subito qualche osservazione.

Why Composing Monads Per questa sezione 4 ho dovuto dividere in due il semplice interprete concreto. Questa necessità è dovuta dal fatto che il monade delle eccezioni definisce un meccanismo per gestire gli errori, mentre il monade per la traccia dell'output gestisce i logs. Quindi non abbiamo raggiunto lo scopo finale, i due interpreti non sono, se presi singolarmente, equivalenti al primo 2. Possono definirsi "equivalenti" solamente se ne prendiamo l'unione degli effetti.

Per risolvere questo problema ci troviamo davanti a due strade, la prima possibilità consiste nel definire un nuovo *Monad* caratterizzato dal tipo

$$\textbf{data}\ M\ s\ a = Raise\ String \mid Return\ (s,\ a)$$

Questo tipo astratto però è definito ad-hoc e non presenta nessuna caratteristica di generalizzazione, implica che la prossima feature da aggiungere renderà inutilizzabile il tipo M e necessiterà di un nuovo tipo M' specializzato anch'esso sulla nuovo task.
La seconda possibilità suppone l'esistenza di un compositore di *Monads*

$$\textbf{data}\ MonadCompositor\ m_0\ m_1\ a = MC\ m_0\ m_1\ a$$

Anche $MonadCompositor$ è un *Monad* e passati due monadi come parametri ne ritorna la composizione naturale. Con questo framework potremmo definire l'interprete sul tipo

$$eval :: Expr \to Env \to MonadCompositor\ Error\ (Writer\ [String])$$

Con il vantaggio di dover modificare solamente poca sintassi dell'interprete monadico già definito. Inoltre se una nuova feature dovesse essere aggiunta lo sforzo per raggiungere il task sarebbe minimo utilizzando il tipo

$$MonadCompositor\ Error\ (MonadCompositor\ FeatureMonad\ (Writer\ [String]))$$

Data la sola definizione del $FeatureMonad$ che dovrà solamente occuparsi di implementare il proprio *Monad* in un ambiente chiuso.

Se questa supposizione fosse vera avremmo un meccanismo per modularizzare questi *Monads* e combinarli a piacimento tra loro. Vedremo come questa supposizione non sia corretta in maniera generale nel capitolo 5, difatti il *Monad Composer* richiede alcune assunzioni e requisiti che vedremo più avanti.

Monad Comprehensions Scrivendo le funzioni di valutazioni ci accorgiamo di come la sintassi sia accessibile solamente da chi abbia piena conoscenza del funzionamento delle monadi, quindi ad un lettore esterno il codice di *eval*, in chiave monadica, risulta comunque complesso. Per migliorare la qualità del codice sotto l'aspetto della comprensibilità ci basta notare che se M è un *Monad* possiamo definire le sue operazioni con la *comprehension notation* [2] semplificando così la definizione di metodi basati su *map*, *unit* e *join*.
Traduciamo quindi queste operazioni nella forma di *comprehension notation*

$$\begin{aligned}
[\ exp \mid x \leftarrow e\ ] &= map\ (\lambda\ x \rightarrow exp)\ e && mapComp \\
[\ exp\ ] &= unit\ exp && unitComp \\
[\ exp \mid gs,\ hs\ ] &= join\ [\ [\ exp \mid hs] \mid gs\ ] && joinComp
\end{aligned}$$

Per convenienza possiamo derivare le regole ($M\ 1 \ldots 7$) e definirne di nuove con questa notazione

$$\begin{aligned}
[\ x \mid x \leftarrow xs\ ] &= xs && compId \\
[\ f\ x \mid x \leftarrow map\ g\ e\ ] &= [\ f\ (g\ x) \mid x \leftarrow e\ ] && compMap \\
[\ f\ x \mid x \leftarrow unit\ e\ ] &= [\ f\ e\ ] && compUnit \\
[\ exp \mid x \leftarrow join\ e\ ] &= [\ exp \mid z \leftarrow e,\ x \leftarrow z\ ] && compJoin
\end{aligned}$$

Queste possono quindi essere usate per definire l'operatore *join*, vedi

$$\begin{aligned}
& join\ e \\
= \quad & [\ x \mid x \leftarrow e\ ] && (compId) \\
= \quad & [\ x \mid z \leftarrow e,\ x \leftarrow z\ ] && (compJoin)
\end{aligned}$$

Se applichiamo questa sintassi alla funzione di valutazione otteniamo il seguente risultato (solo per il caso in cui ho un'espressione del tipo somma, secondo me l'unico caso da migliorare)

$$eval\ (e_0 + e_1)\ env = [\ v_0 + v_1 \mid v_0 \leftarrow eval\ e_0\ env,\ v_1 \leftarrow eval\ e_1\ env\ ]$$

La dimostrazione:

$$\begin{aligned}
& [\ v_0 + v_1 \mid v_0 \leftarrow eval\ e_0\ env,\ v_1 \leftarrow eval\ e_1\ env\ ] = \\
& bind\ (eval\ e_0\ env)(\lambda\ v_0 \rightarrow bind\ (eval\ e_1\ env)(\lambda\ v_1 \rightarrow unit(v_0 + v_1)))
\end{aligned}$$

è stata svolta e riportata in appendice A.3. Questa nuova versione in stile *Monad comprehension* è valida per entrambi i codici, sia per l'output che per le eccezioni.

[2] la *comprehension notation* è un espressione [exp | gs] dove exp è un espressione e gs è chiamato "generatore".

5 Composing Monads

Questo è il capitolo principale dell'intero report, espone il concetto più interessante della composizione dei *Monads*, prima di procedere definiamo in maniera naturale la composizione per *Functor* e *Premonads*. Poi proviamo a trovare la composizione naturale anche per i *Monads* e daremo una dimostrazione formale di non esistenza A.2. Quindi cercheremo di classificare quali condizioni sono necessarie per comporre due *Monads* M e N.

5.1 Conditions for Composition

Supponiamo che M e N siano $Functors$, per eliminare a priori la confusione generata da quale *map* applico di volta in volta, scriverò map_M e map_N per ogni caso. Ora possiamo pensare alla composizione di M e N com un tipo costruttore che riceve a come parametro (senza restrizioni di tipo, a può avere qualiasi tipo) e costruisce il tipo $M\ (N\ a)$. Per questo nuovo tipo l'operatore *map* avrà la seguente firma.

$$map\ ::\ (a \to b) \to (M\ (N\ a) \to M\ (N\ b))$$

Fortunatamente è molto semplice trovare una funzione che soddisfi la firma utilizzando il fatto che M e N siano funtori

$$map\ =\ map_M\ .\ map_N$$

Questa funzione soddisfa le leggi $(M1)$ e $(M2)$

$$\begin{array}{lll}
(M1) & map\ id & \{map\} \\
= & map_M\ (map_N\ id) & \{M1_N\} \\
= & map_M id & \{M1_M\} \\
= & id & \\
 & & \\
(M2) & map\ f\ .\ map\ g & \{map, map\} \\
= & map_M\ (map_N\ f)\ .\ map_M\ (map_N\ g) & \{M2_M\} \\
= & map_M\ (map_N\ f\ .\ map_N\ g) & \{M2_N\} \\
= & map_M\ (map_N\ (f\ .\ g)) & \{map\} \\
= & map\ (f\ .\ g) &
\end{array}$$

La composizione di due *Premonads* arbitrari è simile, dobbiamo trovare una funzione che abbia la firma

$$unit\ ::\ a \to M\ (N\ a)$$

Come per il precedente caso viene banalmente

$$unit\ =\ unit_M\ .\ unit_N$$

Questa funzione soddisfa la legge $(M3)$

$$
\begin{array}{lll}
(M3) & unit\ .\ f & \{unit\} \\
= & unit_M\ .\ unit_N\ .\ f & \{M3_N\} \\
= & unit_M\ .\ map_N\ f\ .\ unit_N & \{M3_M\} \\
= & map_M\ (map_N\ f)\ .\ unit_M.\ unit_N & \{map, unit\} \\
= & map\ f\ .\ unit &
\end{array}
$$

Sfortunatamente lo stesso meccanismo non può essere ripetuto per comporre due *Monads* difatti dovremmo trovare una funzione dalla seguente firma

$$join\ ::\ M\ (N\ (M\ (N\ a))) \rightarrow M\ (N\ a)$$

Che soddisfi le leggi $(M4 \ldots 7)$. Come prima prova potremmo definire il *join* come composizione naturale ottenendo $join\ =\ join_M\ .\ join_N$, questa funzione però non è tipata correttamente se M e N sono diversi tra loro. Infatti non c'è nessun modo di costruire la funzione *join* con il tipo sopra descritto usando solamente le operazioni dei due *Monads*. La dimostrazione formale è data in appendice A.2.

Procedo quindi aggiungendo requisiti per poter ampliare le condizioni per la composizione, difatti è possibile generalizzare la composizione tra *Monads* richiedendo insieme ai *Monads* o *Premonads* M e N anche una funzione ausiliaria per l'implementazione di *join*.

5.2 Prod Construction

La composizione di un monade M con un premonade N è un monade se è definita una funzione

$$prod\ ::\ N\ (M(Na)) \rightarrow M\ (N\ a)$$

Con questa funzione possiamo definire

$$join\ =\ join_M\ .\ map_M\ prod$$

La funzione *prod* deve soddisfare le seguenti leggi

$$
\begin{array}{rcll}
prod\ .\ map_N\ (map\ f) & = & map\ f\ .\ prod & (P1) \\
prod\ .\ unit_N & = & id & (P2) \\
prod\ .\ map_N\ unit & = & unit_M & (P3) \\
prod\ .\ map_N\ join & = & join\ .\ prod & (P4)
\end{array}
$$

La dimostrazione che la composizione di M con N è *Monads* la si può trovare in Jones[1].

Un punto interessante è il fatto che abbiamo richiesto che N sia solamente un *Premonads* per via del fatto che tra definizione di *join* e regole $(P1 \ldots 4)$ non

si utilizza l'operatore $join_N$, stiamo quindi costruendo la condizione di *"composition of monads"* sotto condizioni meno restrittive.

Ora invece ci chiediamo quali $Monads$ possono essere ottenuti con il costruttore *prod*. Più formalemente, supponendo di avere M e N monadi ed esiste la composizione tra M e N con gli operatori *map*, *unit* e *join* che soddisfano le leggi $(M1 \ldots 7)$ tale che

$$map = map_M \ . \ map_N$$
$$unit = unit_M \ . \ unit_N$$

Quali *composite monads* definito in questo modo può essere ottenuto dal costruttore *prod*. Viene naturale definire una funzione corretta (del tipo requisito) per *prod* come segue

$$prod \ = \ join \ . \ unit_M$$

Notiamo come la funzione *join* ottenuta con il costrutto *prod* sia equivalente al *join* con cui siamo partiti

$$\begin{array}{lll} & join_M \ . \ map_M \ prod & \{prod\} \\ = & join_M \ . \ map_M \ (join \ . \ unit_M) & \{M2_M\} \\ = & join_M \ . \ map_M \ join \ . \ map_M \ unit_M & \{J1\} \\ = & join \ . \ join_M \ . \ map_M \ unit_M & \{M6_M\} \\ = & join & \end{array}$$

Questa dimostrazione dipende dal fatto che esista una proprietà "commutativa" tra *join* e $join_M$, da $(M7)$ riusciamo a definire $(J1)$ in un simile modo

$$join_M \ . \ map_M \ join \ = \ join \ . \ join_M \qquad (J1)$$

Questa condizione sembrerebbe arbitraria, ma ci possiamo accorgere di come tutti i $Monads$ ottenuti con il costrutto *prod* la possiedono

$$\begin{array}{lll} & join_M \ . \ map_M \ join & \{join\} \\ = & join_M \ . \ map_M \ (join_M \ . \ map_M \ prod) & \{M2_M\} \\ = & join_M \ . \ map_M \ join_M \ . \ map_M \ (map_M \ prod) & \{M7_M\} \\ = & join_M \ . \ join_M \ . \ map_M \ (map_M \ prod) & \{M4_M\} \\ = & join_M \ . \ map_M \ prod. \ join_M & \{join\} \\ = & join.join_M & \end{array}$$

Segue che l'insieme dei *composite Monads* che possono essere ottenuti usando il costrutto *prod* sono precisamente quelli che soddisfano $(J1)$. Mancherebbe per concludere in maniera formale la dimostrazione che $prod \ = \ join \ . \ unit_M$ rispetta le regole $(P1 \ldots 4)$ ma la dimostrazione è data in Jones[1].

5.3 Dorp Construction

Come per il caso precedente, La composizione di un premonade M con un monade N è un monade se è definita una funzione

$$dorp \;::\; M\ (N(Ma)) \to M\ (N\ a)$$

Con questa funzione possiamo definire

$$join \;=\; map_M\ join_N\ .\ dorp$$

La funzione *dorp* deve soddisfare le seguenti leggi

$$\begin{aligned}
dorp\ .\ map\ (map_M\ f) &= map\ f\ .\ dorp && (D1)\\
dorp\ .\ unit &= map_M\ unit_N && (D2)\\
dorp\ .\ map\ unit_M &= id && (D3)\\
dorp\ .\ join &= join\ .\ map\ dorp && (D4)
\end{aligned}$$

La dimostrazione che la composizione di M con N è $Monads$ la si può trovare in Jones[1].

Ora invece ci chiediamo quali $Monads$ possono essere ottenuti con il costruttore *dorp*. Più formalemente, supponendo di avere M e N monadi ed esiste la composizione tra M e N con gli operatori *map*, *unit* e *join* che soddisfano le leggi $(M1 \ldots 7)$ tale che

$$\begin{aligned}
map &= map_M\ .\ map_N\\
unit &= unit_M\ .\ unit_N
\end{aligned}$$

Quali *composite monads* definito in questo modo può essere ottenuto dal costruttore *dorp*. Viene naturale definire una funzione corretta (del tipo requisito) per *dorp* come segue

$$dorp \;=\; join\ .\ map\ (map_M\ unit_N)$$

Notiamo come la funzione *join* ottenuta con il costrutto *dorp* sia equivalente al *join* con cui siamo partiti

$$\begin{aligned}
& map_M\ join_N\ .\ dorp && \{dorp\}\\
= \;& map_M\ join_N\ .\ join\ .\ map\ (map_M\ unit_N) && \{J2\}\\
= \;& join\ .\ map\ (map_M\ join_N)\ .\ map\ (map_M\ unit_N) && \{M2\}\\
= \;& join\ .\ map\ (map_M\ join_N\ .\ map_M\ unit_N) && \{M2_M\}\\
= \;& join\ .\ map\ (map_M\ (join_N\ .\ unit_N)) && \{M5_N\}\\
= \;& join\ .\ map\ (map_M\ id) && \{M1_M\}\\
= \;& join\ .\ map\ id && \{M1\}\\
= \;& join
\end{aligned}$$

Questa dimostrazione dipende dal fatto che esista una proprietà "commutativa" tra $join$ e $join_N$, da $(M4)$ riusciamo a definire $(J2)$ in un simile modo

$$join\ .\ map\ (map_M\ join_N)\ =\ map_M\ join_N\ .\ join \qquad (J2)$$

Questa condizione sembrerebbe arbitraria, ma ci possiamo accorgere di come tutti i $Monads$ ottenuti con il costrutto $dorp$ la possiedono

$$\begin{array}{lll} & join\ .\ map\ (map_M\ join_N) & \{join\} \\ = & map_M\ join_N\ .\ dorp\ .\ map\ (map_M\ join_N) & \{D1\} \\ = & map_M\ join_N\ .\ map\ join_N\ .\ dorp & \{map, M2_M\} \\ = & map_M\ (join_N\ .\ map_N\ join_N)\ .\ dorp & \{M7_N\} \\ = & map_M\ (join_N\ .\ join_N)\ .\ dorp & \{M2_M\} \\ = & map_M\ join_N\ .\ map_M\ join_N\ .\ dorp & \{join\} \\ = & map_M\ join_N\ .\ join & \end{array}$$

Segue che l'insieme dei *composite Monads* che possono essere ottenuti usando il costrutto $dorp$ sono precisamente quelli che soddisfano $(J2)$. Mancherebbe per concludere in maniera formale la dimostrazione che $dorp\ =\ join\ .\ map\ (map_M\ unit_N)$ rispetta le regole $(D1 \ldots 4)$ ma la dimostrazione è data in Jones[1].

5.4 Swap Construction

Possiamo anche definire la composizione di due $Monads$ senza richiedere che uno dei due sia un $Premonad$. La composizione di un monade M con un monade N è un monade se è definita una funzione

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

Con questa funzione possiamo definire

$$join\ =\ map_M\ join_N\ .\ join_M\ .\ map_M\ swap$$

Oppure in maniera equivalente

$$join\ =\ join_M\ .\ map_M\ (map_M\ join_N\ .\ swap)$$

Per definire le regole $(S1 \ldots 4)$ definiamo prima la relazione tra i costrutti $dorp$ e $prod$ con $swap$, relazione definita da:

$$prod = map_M\ join_N\ .\ swap$$
$$dorp = join_M\ .\ map_M\ swap$$

La funzione $swap$ deve soddisfare le seguenti leggi

$$swap\ .\ map_N\ (map_M\ f)\ =\ map_M\ (map_N\ f)\ .\ swap \qquad (S1)$$
$$swap\ .\ unit_N\ =\ map_M\ unit_N \qquad (S2)$$
$$swap\ .\ map_N\ unit_M\ =\ unit_M \qquad (S3)$$
$$prod\ .\ map_N\ dorp\ =\ dorp\ .\ prod \qquad (S4)$$

La dimostrazione che la composizione di M con N è $Monads$ la si può trovare in Jones[1].

Prima di tutto possiamo già notare come la definizione di $join$ coincide con la definizione di $join$ ottenuta sia con il costruttore $prod$ che con il costruttore $dorp$.

Per il costruttore $prod$

$$\begin{array}{lll} & join_M \ . \ map_M \ prod & \{prod\} \\ = & join_M \ . \ map_M \ (map_M \ join_N \ . \ swap) & \{join\} \\ = & join & \end{array}$$

Per il costruttore $dorp$

$$\begin{array}{lll} & map_M \ join_N \ . \ dorp & \{dorp\} \\ = & map_M \ join_N \ . \ join_M \ . \ map_M \ swap & \{join\} \\ = & join & \end{array}$$

Come per i precedenti costrutti, ci chiediamoinoltre quali $Monads$ possono essere ottenuti con il costruttore $swap$. Più formalemente, supponendo di avere M e N monadi ed esiste la composizione tra M e N con gli operatori map, $unit$ e $join$ che soddisfano le leggi $(M1 \ldots 7)$ tale che

$$\begin{aligned} map &= map_M \ . \ map_N \\ unit &= unit_M \ . \ unit_N \end{aligned}$$

Quali *composite monads* definito in questo modo può essere ottenuto dal costruttore $swap$. Viene naturale definire una funzione corretta (del tipo requisito) per $swap$ come segue

$$swap \ = \ join \ . \ unit_M \ . \ map_N \ (map_M \ unit_N)$$

Oppure in maniera del tutto equivalente

$$swap \ = \ join \ . \ map \ (map_M \ unit_N) \ . \ unit_M$$

Possiamo notare come si possa definire l'operatore $swap$ anche in termini di $prod$ e $dorp$

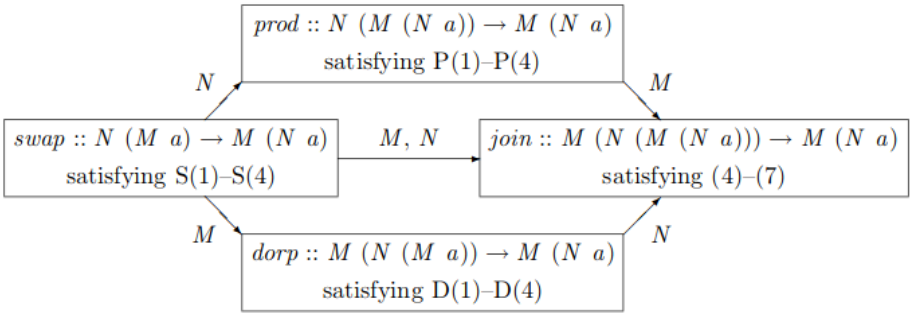
$$\begin{aligned} swap &= prod \ . \ map_N \ (map_M \ unit_N) \\ swap &= dorp \ . \ unit_M \end{aligned}$$

La dimostrazione è simile alle precedenti, vedi 5.2 e 5.3. Segue che l'insieme dei *composite Monads* che possono essere ottenuti usando il costrutto $swap$ sono precisamente quelli che soddisfano entrambe $(J1)$ e $(J2)$.

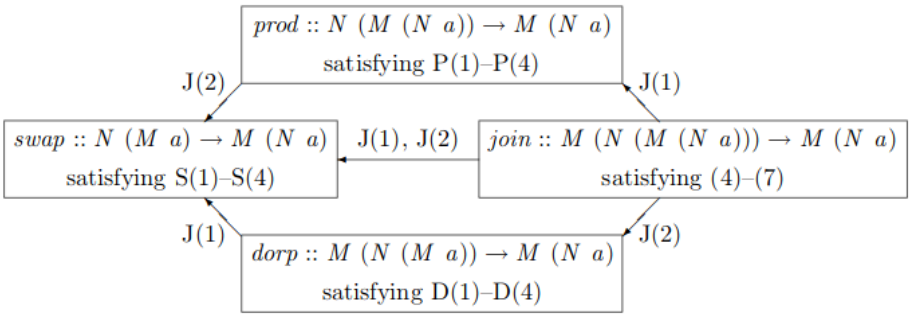
5.5 Summary

Per concludere includo due grafici che spiegano molto bene le relazioni che intercorrono tra i vari costruttori, grafici presi da Jones[1].

Il primo descrive la relazione tra i differenti costruttori per comporre monadi. I nodi rappresentano gli operatori di cui si dispone, mentre gli archi definiscono le restrizioni di essere monade. Per esempio un arco $A \xrightarrow{N} B$ significa che se mi trovo in A per passare a B devo "costringere" N ad essere monade.



Il secondo descrive la relazione per passare da monadi ai tre costruttori. Questa volta gli archi descrivono proprietà richieste per ottenere il costrutto.



6 General Framework for composition

In questa sezione verranno definite una serie di classi per la generazione di un framework per la composizione di monadi. Dal precedente capitolo 5, definiremo una serie di classi e funzioni per poter tipare correttamente le operazioni necessarie e definire finalmente il tipo di dato astratto per la composizione generalizzata. Le classi saranno implementate in sintassi Gofer per permettere un migliore ragionamento e in Scala per avere esmpio concreti.

Abbiamo già visto come non sia possibile definire la composizione che riceva due *Monads* arbitrari. La prossima migliore opzione è di fissare uno dei due componenti (della composizione) e permettere all'altro componente di variare dentro ad una famiglia di differenti Monads. Come questo accada sarà mostrato con qualche esempio nella sezione 6.6, prima definiamo il framework.

6.1 Representing Functors, Premonads and Monads

Vediamo velocemente come si rappresentano i tipi costruttori base.

$$\begin{aligned} &\textbf{\textit{class}}\ Functor\ f\ \textbf{\textit{where}} \\ &\quad map :: (a \to b) \to (f\ a \to f\ b) \end{aligned}$$

$$\begin{aligned} &\textbf{\textit{class}}\ Functor\ m \Rightarrow Premonad\ m\ \textbf{\textit{where}} \\ &\quad unit :: a \to m\ a \end{aligned}$$

$$\begin{aligned} &\textbf{\textit{class}}\ Premonad\ m \Rightarrow Monad\ m\ \textbf{\textit{where}} \\ &\quad join :: m\ (m\ a) \to m\ a \end{aligned}$$

Con $\Longrightarrow$ si intende la relazione di *subtyping*, ovvero la classe a sinistra di $\Longrightarrow$ è sotto classe di quella definita a sinistra.

Listing 1: Functor Premonad and Monad Traits

```
1  trait Functor[F[_]]{
2    def map[A,B](ma:F[A])(f:(A) => B):F[B]
3  }
4  trait Premonad[M[_]] extends Functor[M]{
5    def unit[A](a:A):M[A]
6  }
7  trait Monad[M[_]] extends Functor[M]{
8    // come il join operator
9    def flatten[A](mm:M[M[A]]):M[A]
10   // come il bind operator, Scala defisce cosi' il Monad
11   def flatMap[A,B](ma:M[A])(f:(A)=>M[B]) = flatten(map(ma)(f))
12 }
```

6.2 Composition Construction

Per descrivere la composizione di *Functors* e *Premonads* utilizziamo le definizioni

$$
\begin{aligned}
&mapC :: (Functor\ f,\ Functor\ g) \implies (a \to b) \to (f\ (g\ a) \to f\ (g\ b)) \\
&mapC = map\ .\ map \\
&unitC = (Premonad\ f,\ Premonad\ g) \implies a \to f\ (g\ a) \\
&unitC = unit\ .\ unit
\end{aligned}
$$

Queste due firme sono del tipo corretto per essere utilizzate come definizione di funtore e premonade. Tuttiavia, non possono essere utilizzate direttamente come istanze di funtore (e premonade) dato che non c'è modo di definire una classe $(f\ .\ g)$ istanza di *Functor* o *Premonad* o *Monad*. Precisamente, non c'è nulla nel tipo dell'espressione $f\ (g\ x)$ che indichi a quale costruzione è destinato. Per evitare questa ambiguità, definiamo un costruttore c per ogni *composition constructions* con l'intenzione che $c\ f\ g\ x$ sia la composizione $f\ (g\ a)$, identificando quindi la costruzione utilizzata.

$$
\begin{aligned}
&\textbf{\textit{class}}\ Composer\ c\ \textbf{\textit{where}} \\
&\quad open :: c\ f\ g\ x \to f\ (g\ x) \\
&\quad close :: f\ (g\ x) \to c\ f\ g\ x
\end{aligned}
$$

Ora proceidamo a definire le istanze di *Functor* e *Premonad* per i tipi compositi , nota che queste definizioni sono *general purpose*, quindi non necessitano di essere ridefinite ogni volta.

$$
\begin{aligned}
&\textbf{\textit{instance}}\ (Composer\ c\ Functor\ f,\ Functor\ g) \implies Functor\ (c\ f\ g)\ \textbf{\textit{where}} \\
&\quad map\ f = close\ .\ mapC\ f\ .\ open \\
&\textbf{\textit{instance}}\ (Composer\ c\ Premonadm,\ Premonadn) \\
&\quad \implies Premonad\ (c\ m\ n)\ \textbf{\textit{where}} \\
&\quad unit = close\ .\ unitC
\end{aligned}
$$

Listing 2: Natural map and unit operator

```
implicit def premonadsCompose[M[_],N[_]]
  (implicit m:Monad[M], n:Monad[N])
  = new Premonad[({type MN[x] = M[N[x]]})#MN]{
    def map[A,B](ma:M[N[A]])(f: (A) => B)
      = m.map(ma)(n.map(_)(f))
    def unit[A](a: A) = m.unit(n.unit(a))
}
```

Le prossime definizioni saranno utilizzate molteplici volte nelle sezioni seguenti. Il loro scopo è di convertire funzioni per il *join* di composizioni nella forma equivalente usando un'istanza c della classe *Composer*.

$$
\begin{aligned}
&wrap :: (Composer\ c, Functor\ m, Functor\ n) \implies \\
&\quad (m\ (n\ (m\ (n\ a))) \to m\ (n\ a)) \to \\
&\quad (c\ m\ n\ (c\ m\ n\ a) \to c\ m\ n\ a) \\
&wrap\ j = close\ .\ j\ .\ mapC\ open\ .\ open
\end{aligned}
$$

In special modo le prossime due funzioni servono a fornire un modo per racchiudere la computazione di un componente dentro il *composite monad*.

$$right :: (Composer\ c,\ Premonad\ f) \implies g\ a \to c\ f\ g\ a$$
$$right = close\ .\ unit$$

$$left :: (Composer\ c,\ Functor\ f,\ Premonad\ g) \implies f\ a \to c\ f\ g\ a$$
$$left = close\ .\ map\ unit$$

6.3 Programming Prod Construction

Creaiamo un'istanza del *Composer* sopra definito per il costruttore *prod*

$$\textbf{\textit{data}}\ PComp\ f\ g\ x\ =\ PC\ (f\ (g\ x))$$
$$\textbf{\textit{instance}}\ Composer\ PComp\ \textbf{\textit{where}}$$
$$open\ (PC\ x)\ = x$$
$$close = PC$$

La costruzione richiede lei stessa la funzione *prod* come definita

$$\textbf{\textit{class}}\ (Monad\ m,\ Premonad\ n) \implies PComposable\ m\ n\ \textbf{\textit{where}}$$
$$prod :: n\ (m\ (n\ a)) \to m\ (n\ a)$$

Listing 3: Prod Constructor

```
trait PComposable[M[_],N[_]]{
  def M:Monad[M]
  def N:Premonad[N]
  def prod[A](nm:N[M[N[A]]]):M[N[A]]
}
```

Notiamo come le richieste imposte specifichino per n solamente l'appartenenza alla classe dei *Premonads*, mentre per m si richiede di essere un *Monads*

$$joinP :: (Pcomposable\ m\ n) \implies m\ (n\ (m\ (n\ a))) \to m\ (n\ a)$$
$$joinP = join\ .\ map\ prod$$

Finalmente possiamo definire che *PComp* sia un *Monad*

$$\textbf{\textit{instance}}\ PComposable\ m\ n \implies Monad\ (PComp\ m\ n)\ \textbf{\textit{where}}$$
$$join = wrap\ joinP$$

Listing 4: PComposable is Monad

```
implicit def monadPCompose[M[_],N[_]]
  (implicit m:Monad[M], n:Monad[N], p:PComposable[M,N])
  = new Monad[({type MN[x] = M[N[x]]})#MN]{
    def flatten[A](mnmn:M[N[M[N[A]]]])
      = m.flatten(m.map(mnmn)(p.prod(_)))
}
```

6.4 Programming Dorp Construction

Come per il caso precedente, procediamo in maniera simile. Creaiamo un'istanza del *Composer* sopra definito per il costruttore *dorp*

$$\begin{aligned} &\textbf{\textit{data}}\ DComp\ f\ g\ x\ =\ DC\ (f\ (g\ x)) \\ &\textbf{\textit{instance}}\ Composer\ DComp\ \textbf{\textit{where}} \\ &\qquad open\ (DC\ x)\ = x \\ &\qquad close = DC \end{aligned}$$

La costruzione richiede lei stessa la funzione *dorp* come definita

$$\begin{aligned} &\textbf{\textit{class}}\ (Premonad\ m,\ Monad\ n)\ \Longrightarrow\ DComposable\ m\ n\ \textbf{\textit{where}} \\ &\qquad dorp :: m\ (n\ (m\ a)) \rightarrow m\ (n\ a) \end{aligned}$$

Listing 5: Dorp Constructor

```
1  trait DComposable[M[_],N[_]]{
2    def M:Premonad[M]
3    def N:Monad[N]
4    def dorp[A](nm:M[N[M[A]]]):M[N[A]]
5  }
```

Notiamo come le richieste imposte specifichino per m solamente l'appartenenza alla classe dei *Premonads*, mentre per n si richiede di essere un *Monads*

$$\begin{aligned} &joinD :: (DComposable\ m\ n)\ \Longrightarrow\ m\ (n\ (m\ (n\ a))) \rightarrow\ (n\ a) \\ &joinD = join\ .\ map\ dorp \end{aligned}$$

Finalmente possiamo definire che *DComp* sia un *Monad*

$$\begin{aligned} &\textbf{\textit{instance}}\ DComposable\ m\ n\ \Longrightarrow\ Monad\ (DComp\ m\ n)\ \textbf{\textit{where}} \\ &\qquad join = wrap\ joinD \end{aligned}$$

Listing 6: DComposable is Monad

```
1  implicit def monadDCompose[M[_],N[_]]
2    (implicit m:Monad[M], n:Monad[N], d:DComposable[M,N])
3    = new Monad[({type MN[x] = M[N[x]]})#MN]{
4      def flatten[A](mnmn:M[N[M[N[A]]]])
5        = m.flatten(m.map(mnmn)(d.dorp(_)))
6  }
```

6.5 Programming Swap Construction

Anche questo costruttore molto simile ai casi precedenti. Creaiamo un'istanza del *Composer* sopra definito per il costruttore *swap*

$$\begin{gathered}\textbf{\textit{data}}\ SComp\ f\ g\ x\ =\ SC\ (f\ (g\ x))\\ \textbf{\textit{instance}}\ Composer\ SComp\ \textbf{\textit{where}}\\ open\ (SC\ x)\ = x\\ close = SC\end{gathered}$$

La costruzione richiede lei stessa la funzione *swap* come definita

$$\begin{gathered}\textbf{\textit{class}}\ (Premonad\ m,\ Monad\ n)\implies SComposable\ m\ n\ \textbf{\textit{where}}\\ swap :: n\ (m\ a)\to m\ (n\ a)\end{gathered}$$

Listing 7: Swap Constructor

```
trait SComposable[M[_],N[_]]{
  def M:Monad[M]
  def N:Monad[N]
  def swap[A](nm:N[M[A]]):M[N[A]]
}
```

Notiamo come le richieste imposte specifichino sia per m che per n l'appartenenza alla classe dei *Monads*, diversamente dai casi precedenti

$$\begin{gathered}joinS :: (SComposable\ m\ n)\implies m\ (n\ (m\ (n\ a)))\to\ (n\ a)\\ joinS = map\ join\ .\ join\ .\ map\ swap\end{gathered}$$

Finalmente possiamo definire che *SComp* sia un *Monad*

$$\begin{gathered}\textbf{\textit{instance}}\ SComposable\ m\ n\implies Monad\ (SComp\ m\ n)\ \textbf{\textit{where}}\\ join = wrap\ joinS\end{gathered}$$

Listing 8: SComposable is Monad

```
implicit def monadSCompose[M[_],N[_]]
  (implicit m:Monad[M], n:Monad[N], s:SComposable[M,N])
  = new Monad[({type MN[x] = M[N[x]]})#MN]{
    def flatten[A](mnmn:M[N[M[N[A]]]])
      = m.map(m.flatten(m.map(mnmn)(s.swap(_))))(n.flatten(_))
}
```

Inoltre possiamo notare come un *SComposable* sia egli stesso un *PComposable* e *DComposable*, questa definizione mostra come un costruttore *prod* e *dorp* possa sempre essere derivato da una definizione di *swap*

$$\begin{gathered}\textbf{\textit{instance}}\ SComposable\ m\ n\implies PComposable\ m\ n\ \textbf{\textit{where}}\\ prod = map\ join\ .\ swap\\ \textbf{\textit{instance}}\ SComposable\ m\ n\implies DComposable\ m\ n\ \textbf{\textit{where}}\\ dorp = map\ join\ .\ swap\end{gathered}$$

6.6 Some Examples

Reader

Il Monade *Reader* è un costruttore del tipo $(r \ \rightarrow)$ che mappa per ogni tipo a alla funzione del tipo $r \rightarrow a$, la sua strutture è definita come segue

$$\begin{aligned}
&\textbf{\textit{instance}}\ Functor\ (r \rightarrow)\ \textbf{\textit{where}} \\
&\quad map\ f\ g\ = f\ .\ g \\
&\textbf{\textit{instance}}\ Premonad\ (r \rightarrow)\ \textbf{\textit{where}} \\
&\quad unit\ x\ y\ = x \\
&\textbf{\textit{instance}}\ Monad\ (r \rightarrow)\ \textbf{\textit{where}} \\
&\quad join\ f\ x\ = f\ x\ x
\end{aligned}$$

Questo caso combineremo un qualsiasi Monade n con il monade *Reader* usando il costruttore *dorp*

$$\begin{aligned}
&\textbf{\textit{instance}}\ DComposable\ (r \rightarrow)\ n\ \textbf{\textit{where}} \\
&\quad dorp\ m\ r\ =\ [\ g\ r \mid g \leftarrow m\ r\ ]
\end{aligned}$$

Writer

Il *Writer Monad* invece è un Monade che descrive programmi che producono sia valori che output in maniera parallela. Inoltre questo Monade assume che il parametro fisso s sia un $Monoid$, vedi sezione C.

$$\begin{aligned}
&\textbf{\textit{instance}}\ Functor\ (Writer\ s)\ \textbf{\textit{where}} \\
&\quad map\ f\ (Result\ s\ a)\ = Result\ s\ (f\ a) \\
&\textbf{\textit{instance}}\ Monoid\ s \implies Premonad\ (Writer\ s)\ \textbf{\textit{where}} \\
&\quad unit\ = Result\ zero \\
&\textbf{\textit{instance}}\ Monoid\ s \implies Monad\ (Writer\ s)\ \textbf{\textit{where}} \\
&\quad join\ (Result\ s\ (Result\ t\ x)) = Result\ (add\ s\ t)\ x
\end{aligned}$$

Questo Monade è un esempio di come si utilizza il costrutto *swap* per combinare un qualsiasi Monade

$$\begin{aligned}
&\textbf{\textit{instance}}\ SComposable\ (Writer\ s)\ n\ \textbf{\textit{where}} \\
&\quad swap\ (Result\ s\ m) = [\ Result\ s\ a \mid a \leftarrow m\ ]
\end{aligned}$$

Maybe

Il *Maybe Monad* invece è un Monade che descrive programmi che producono non sempre valori corretti. Molto simile al monade *Error*.

$$
\begin{aligned}
&\textbf{\textit{instance}}\ Functor\ Maybe\ \textbf{\textit{where}}\\
&\quad map\ f\ (Just\ x)\ = Just\ (f\ x))\\
&\quad map\ f\ (Nothing)\ = Nothing\\
&\textbf{\textit{instance}}\ Premonad\ Maybe\ \textbf{\textit{where}}\\
&\quad unit\ = Just\\
&\textbf{\textit{instance}}\ Monad\ Maybe\ \textbf{\textit{where}}\\
&\quad join\ (Just\ m) = m\\
&\quad join\ Nothing = Nothing
\end{aligned}
$$

Questo Monade è un esempio di come si utilizza il costrutto *prod* per combinare un qualsiasi Monade

$$
\begin{aligned}
&\textbf{\textit{instance}}\ PComposable\ m\ Maybe\ \textbf{\textit{where}}\\
&\quad prod\ (Just\ m) = \ m\\
&\quad prod\ Nothing = \ [\ Nothing\ ]
\end{aligned}
$$

Listing 9: Composing Maybe

```
implicit def MonadMaybeSComposable[M[_]]
  (implicit monadM: Monad[M])
  = new SComposable[M,Maybe]{
    def M = implicitly[Monad[Maybe]]
    def N = monadM
    def prod[A](nm: Maybe[M[A]]) = nm match{
      case Just(ma) => monadM.map(ma)(Maybe(_))
      case Nothing => monadM.unit(Nothing)
    }
}
```

7 Composed Monad Interpreter

Ora riprendiamo il semplice interprete definito nella sezione 4, definiamolo però in termini di monade combinato. Ricordando che la funzione di valutazione dovrà gestire l'accesso a variabili dentro ad un *enviroments*, potrebbe produrre dell'output e richiede la gestione degli errori. Questo capitolo fa affidamento alle ricerche di Steele[4], Hughes[5] e Jones[6]. Tutto questo è coperto dalla seguente definizione di dato astratto:

$$\textbf{\textit{type}}\ M\ a\ =\ Env\ \rightarrow Writer\ [String]\ (Error\ a)$$

Questa definizione può essere espressa nel seguente modo come composizione tra monadi

$$\textbf{\textit{type}}\ M\ a\ =\ DComp\ (Env\ \rightarrow)\ (SComp\ (Writer\ [String])\ Error)a$$

Inoltre notiamo come, utilizzando $Gofer$, questa definizione risulti "brutta" da vedere. Potremmo ragionevolmente aspettarci di poter definire M il una maniera migliore. In un contesto adatto potremmo definire M in maniera del tutto equivalente nel seguente modo

$$M\ a\ =\ (Env\ \rightarrow)\ .\ Writer\ [String]\ .\ Error\ a$$

Utilizzando le funzioni *general purpose* $left$ e $right$, vedi sezione 6.2, possiamo includere la computazione di ogni componente dentro all'intero $MonadM$ composto.

$$\begin{aligned}
&inError :: Error\ a \rightarrow M\ a\\
&inError = right\ .\ right\\
\\
&inReader :: (Env \rightarrow a) \rightarrow M\ a\\
&inReader = left\\
\\
&inWriter :: Writer\ [String]\ a \rightarrow M\ a\\
&inWriter = right\ .\ left
\end{aligned}$$

Possiamo anche notare da queste definizioni che si espone un concetto di associatività all'interno di M, ossia l'operatore di composizione associa a destra. Quindi le funzioni per accedere alla posizione del proprio monade all'interno di M devono percorre l'albero generato dalla composizione fino al proprio nodo.

La definizione di $eval$ è definita da

$$\begin{aligned}
eval &:: Expr \rightarrow M\ Value\\
eval\ (Const\ v) &= [unit\ v]\ \text{(in maniera equivalente } [[v]])\\
eval\ (Var\ n) &= [x \mid r \leftarrow inReader\ (lookup\ n),\ x \leftarrow inError\ r]\\
eval\ (e_0 + e_1) &= [x + y \mid x \leftarrow eval\ e_0,\ y \leftarrow eval\ e_1]\\
eval\ (Trace\ m\ e) &= [x \mid x \leftarrow eval\ e,\\
&\qquad () \leftarrow inWriter\ (Result\ [m\ ++\ "="\ ++\ x]\ ())]
\end{aligned}$$

Osserviamo come questa versione sia finalmente la soluzione preferibile. Propone un codice elegante e difficilmente si incorre il rischio di commettere errori.

8 Summary

Ora invece cerchiamo di capire dato un ADT [3], come potremmo comporlo con altri monadi. Definiamo quindi una struttura generale per gli ADT, notiamo prima qualche osservazione per restringere la costruzione di questi tipi di dato astratti

- Non faremo distinzioni tra coppie e sequenze in quanto un ADT definito mediante coppia e uno mediante sequenza sono equivalenti

  $$\textbf{\textit{data}}\ A\ a = A\ (a,\ b_1,\ \ldots\ ,\ b_n) \equiv \textbf{\textit{data}}\ B\ a = B\ a\ b_1\ \ldots\ b_n$$

  Con i parametri $b_1,\ \ldots\ ,b_n$ fissati, è facile vedere come in ogni situazione in cui necessito di A possa utilizzare B data la funzione biettiva di conversione

  $$\begin{aligned} \gamma &:: A \to B \\ \gamma\ (A\ (x,\ y_1,\ \ldots\ ,\ y_n)) &= B\ x\ y_1\ \ldots\ \ y_n \\ \gamma^{-1}\ (B\ x\ y_1\ \ldots\ \ y_n) &= A\ (x,\ y_1,\ \ldots\ ,\ y_n) \end{aligned}$$

- Ogni ADT sarà unario per semplicità. Nota come questa non sia una restrizione ma solamente una semplificazione, in quanto, ogni costruttore non unario può essere $curryfied$ e diventarlo. Considerando semplicemente un parametro alla volta.

La struttura che ho trovato più generale possibile per caratterizzare tutti i tipi di ADT è la seguente

$$\textbf{\textit{data}}\ A\ a = A_0\ {a_0}^*\ \mid\ \ldots\ \mid\ A_n\ {a_n}^*$$

Dove:

- A_i descrivono i vari costruttore utilizzati, inoltre $\mid A_i\ \forall i \mid > 0$ non può esistere il caso in cui un tipo di dato astratto non abbia nessun costruttore
- ${a_i}^*$ è una lista di parametri (anche vuota), $\forall i,\ k.\ {a_i}^k$ è a oppure un tipo fissato

Notiamo come i principali ADT incontrati fino ad ora rispettino questa caratterizzazione:

$$\begin{aligned} Maybe\ a = Just\ a \quad &\mid Nothing \\ A\ a = A_0\ a \quad &\mid A_1 \end{aligned}$$

$$\begin{aligned} Error\ s\ a = Raise\ s \quad &\mid Return\ a \\ B\ a = A_0\ s \quad &\mid A_1\ a \end{aligned}$$

$$\begin{aligned} (s \to)\ a &= Reader\ (s \to a) \\ A\ a &= A_0\ C\ a \end{aligned}$$

$$\begin{aligned} ZipList\ a &= Z\ [\ a\ ] \\ A\ a &= A_0\ D\ a \end{aligned}$$

[3] Abstract Data Type, tipo di dato astratto

Con B, C e D definiti come:

- $B \equiv (A\ a_0)$ con a_0 che diventa parametro fissato
- $C \equiv (s \rightarrow)$ per descrivere le realzioni
- $D \equiv \lambda x \rightarrow [\ x\ ]$ per rappresentare l'operatore di *binding* per liste

Definiamo inoltre il **tipo di dato astratto base**, ovvero un *ADT* che non può essere scomposto in altri *ADT* di inferiore struttura. In altri termini, un *ADT* base non è rappresentabile mediante la composizione di più *ADT* in ogni sua parte.

Dato quindi un *ADT* **base** possiamo

- pre-comporlo [4] ad m, utilizzando il costruttore *prod*, se m dispone di una funzione di aggregazione oppure se in A non ci sono multiple apparizioni di a parametro
- post-comporlo [5] ad m, utilizzando il costruttore *dorp*, se m dispone di una funzione di aggregazione oppure se in A non ci sono multiple apparizioni di a parametro
- pre-comporlo utilizzando il costruttore *swap*, richiedendo all'arbitrario monade di possedere la proprietà commutativa

Inoltre possiamo osservare che anche le Monadi che per ogni costruttore utilizzano al massimo una volta il parametro a possono essere sempre composte senza limitazioni, vedi *Maybe* o *Error*.

Dare una dimostrazione formale di queste proprietà richiede l'esistenza di funzioni ausiliarie di commutazione che porterebbero alla costruzione di *prod* e *dorp*, la cui esistenza porterebbe alla definizione di *join*. In particolare richiediamo la proprietà commutativa (1) e quella "aggregativa" (2)

$$[\ exp \mid x \leftarrow v,\ y \leftarrow u\ ] = [\ exp \mid y \leftarrow u,\ x \leftarrow v\ ] \tag{1}$$

$$\exists f.\ f :: a_0 \rightarrow \cdots \rightarrow a_n \rightarrow b \tag{2}$$

Non stiamo quindi concludendo che due monadi qualsiasi sono componibili tra di loro dato che in questa lista riassuntiva richiediamo sempre di essere a conoscenza della struttura di almeno uno dei due monadi. Difatti stiamo solo ristrutturando in maniera equivalente le conclusioni del capitolo 6.
Inoltre non sempre la composizione di due monadi è la modalità preferibile di composizione, nella sezione B vediamo un caso in cui la composizione è semanticamente diversa al monade che stiamo cercando.

A Proofs

A.1 Error is a Monad

Dimostriamo che il tipo astratto *Error* 4.1 è un *Functor*, *Premonad* e *Monad*

Dim:

Dimostriamo che il tipo astratto con le operazioni di *map*, *unit* e *join* rispetta le leggi definite per *Functor*, *Premonad* e *Monad*, $M1 \dots 7$

$$\begin{aligned}
&\textbf{data}\ Error\ a = Raise\ String \mid Return\ a \\
\\
&map :: (a \to b) \to Error\ a \to Error\ b \\
&map\ f\ (Raise\ s) = Raise\ s \\
&map\ f\ (Return\ x) = Return\ (f\ x) \\
\\
&unit :: a \to Error\ a \\
&unit = Return \\
\\
&join :: Error\ (Error\ a) \to Error\ a \\
&join\ (Raise\ s) = Raise\ s \\
&join\ (Return\ x) = x
\end{aligned}$$

Per la dimostrazione procedo per casi sui costrutti *Raise* e *Return*

$$\begin{array}{lcll}
(Raise) & & map\ id\ (Raise\ s) & \{map\} \\
& = & (Raise\ s) & \{id^{-1}\} \\
& = & id\ (Raise\ s) & \\
\\
(Return) & & map\ id\ (Return\ x) & \{map\} \\
& = & Return\ (id\ x) & \{id\} \\
& = & Return\ x &
\end{array}$$

$$
\begin{array}{lll}
(Raise) & (map\ f\ .\ map\ g)\ (Raise\ s) & \{composition\} \\
= & map\ f\ (map\ g\ (Raise\ s)) & \{map\} \\
= & map\ f\ (Raise\ s) & \{map\} \\
= & Raise\ s & \{map^{-1}\} \\
= & map\ (f\ .\ g)\ (Raise\ s) &
\end{array}
$$

$$
\begin{array}{lll}
(Return) & (map\ f\ .\ map\ g)\ (Return\ s) & \{composition\} \\
= & map\ f\ (map\ g\ (Return\ s)) & \{map\} \\
= & map\ f\ (Return\ (g\ s)) & \{map\} \\
= & Return\ (f\ (g\ s)) & \{composition^{-1}\} \\
= & Return\ ((f\ .\ g)\ s) & \{map^{-1}\} \\
= & map\ (f\ .\ g)\ (Return\ s) &
\end{array}
$$

$$
\begin{array}{lll}
(Raise) & (unit\ .\ f)\ (Raise\ s) & \{composition\} \\
= & unit\ (f\ (Raise\ s)) & \{unit\} \\
= & Return\ (f\ (Raise\ s)) & \{map^{-1}\} \\
= & map\ f\ (Return\ (Raise\ s)) & \{unit^{-1}\} \\
= & map\ f\ (unit\ (Raise\ s)) & \{composition^{-1}\} \\
= & (map\ f\ .\ unit)\ (Raise\ s) &
\end{array}
$$

$$
\begin{array}{lll}
(Return) & (unit\ .\ f)\ (Return\ s) & \{composition\} \\
= & unit\ (f\ (Return\ s)) & \{unit\} \\
= & Return\ (f\ (Return\ s)) & \{unit^{-1}\} \\
= & Return\ (f\ (unit\ s)) & \{composition^{-1}\} \\
= & Return\ ((f\ .\ unit)\ s) & \{map^{-1}\} \\
= & (map\ f\ .\ unit)\ (Return\ s) &
\end{array}
$$

$$
\begin{array}{lll}
(Raise) & (join\ .\ map\ (map\ f))\ (Raise\ s) & \{composition\} \\
= & join\ (map\ (map\ f)\ (Raise\ s)) & \{map\} \\
= & join\ (Raise\ s) & \{join\} \\
= & Raise\ s & \{map^{-1}\} \\
= & map\ f\ (Raise\ s) & \{join^{-1}\} \\
= & map\ f\ (join\ (Raise\ s)) & \{composition^{-1}\} \\
= & map\ f\ .\ join\ (Raise\ s) &
\end{array}
$$

$$
\begin{array}{lll}
(Return) & (join\ .\ map\ (map\ f))\ (Return\ s) & \{composition\} \\
= & join\ (map\ (map\ f)\ (Return\ s)) & \{map\} \\
= & join\ (Return\ (map\ f\ s)) & \{join\} \\
= & map\ f\ s & \{join^{-1}\} \quad (*) \\
= & map\ f\ (join\ (Return\ s)) & \{composition^{-1}\} \\
= & map\ f\ .\ join\ (Return\ s) &
\end{array}
$$

$$
\begin{array}{lll}
(Raise) & (join\ .\ unit)\ (Raise\ s) & \{composition\} \\
= & join\ (unit\ (Raise\ s)) & \{unit\} \\
= & join\ (Return\ (Raise\ s)) & \{join\} \\
= & Raise\ s & \{id^{-1}\} \\
= & id\ (Raise\ s) &
\end{array}
$$

$$
\begin{array}{lll}
(Return) & (join\ .\ unit)\ (Return\ s) & \{composition\} \\
= & join\ (unit\ (Return\ s) & \{unit\} \\
= & join\ (Return\ (Return\ s) & \{join\} \\
= & Return\ s & \{id^{-1}\} \\
= & id\ (Return\ s) &
\end{array}
$$

$$
\begin{array}{lll}
(Raise) & (join\ .\ map\ unit)\ (Raise\ s)) & \{composition\} \\
= & join\ (map\ unit\ (Raise\ s)) & \{map\} \\
= & join\ (Raise\ s) & \{join\} \\
= & Raise\ s & \{id^{-1}\} \\
= & id\ (Raise\ s) &
\end{array}
$$

$$
\begin{array}{lll}
(Return) & (join\ .\ map\ unit)\ (Return\ s) & \{composition\} \\
= & join\ (map\ unit\ (Return\ s)) & \{map\} \\
= & join\ (Return\ (unit\ s)) & \{unit^{-1}\} \\
= & join\ (unit\ (unit\ s)) & \{composition^{-1}\} \\
= & join\ .\ unit\ (unit\ s) & \{M5\} \\
= & id\ (unit\ s) & \{unit\} \\
= & id\ (Return\ s) &
\end{array}
$$

$$
\begin{array}{lll}
(Raise) & (join\ .\ map\ join)\ (Raise\ s)) & \{composition\} \\
= & join\ (map\ join\ (Raise\ s)) & \{map\} \\
= & join\ (Raise\ s) & \{join^{-1}\} \\
= & join\ (join\ (Raise\ s)) & \{composition^{-1}\} \\
= & join\ .\ join\ (Raise\ s) &
\end{array}
$$

$$
\begin{array}{lll}
(Return) & (join\ .\ map\ join)\ (Return\ s) & \{composition\} \\
= & join\ (map\ join\ (Return\ s)) & \{map\} \\
= & join\ (Return\ (join\ s)) & \{join\} \\
= & join\ s & \{join^{-1}\} \quad (*) \\
= & join\ (join\ (Return\ s)) & \{composition^{-1}\} \\
= & join\ .\ join\ (Return\ s) &
\end{array}
$$

Osservazione $(*)$, utilizzando la definizione inversa di $join$ richiedo che il suo parametro s sia di tipo $s :: Error$

A.2 Natural Join doesn't exist

Dimostriamo che non esiste il *join* naturale tra *Monads* generici 5.1. In parole, è impossibile costruire la funzione *join* per la composizione di due *Monads* utilizzando esclusivamente gli operatori dei monadi.
Supponiamo di lavorare con due *Monads* dati M e N. Nel caso più generale un termine ben tipato può essere costruito utilizzando solo gli operatori *map*, *unit* e *join*. Precisamente possiamo costruire solo termini generati dalla seguenti regole

$$a \xrightarrow{id} a \qquad \frac{b \xrightarrow{f} c \quad a \xrightarrow{g} b}{a \xrightarrow{f \cdot g} c}$$

$$\frac{a \xrightarrow{f} b}{M\ a \xrightarrow{map_M\ f} M\ b} \qquad \frac{a \xrightarrow{f} b}{N\ a \xrightarrow{map_N\ f} N\ b}$$

$$a \xrightarrow{unit_M} M\ a \qquad a \xrightarrow{unit_N} N\ a$$

$$M\ (M\ a) \xrightarrow{join_M} M\ a \qquad N\ (N\ a) \xrightarrow{join_N} N\ a$$

Queste regole comprendono il più piccolo set per poter modellare adeguatamente le operazioni in maniera naturale. Utilizzando questa caratterizzazione mostrerò che non c'è modo di costruire un termine del tipo $M\ (N\ (M\ (N\ a))) \to M\ (N\ a)$ e quindi che non esiste il *join* naturale per la composizione dei *Monads*.

Per convenienza utilizzerò le seguenti convenzioni:

- scriverò la stringa $MNMNX$ al posto dell'espressione $M\ (N\ (M\ (N\ X)))$,inoltre per l'i-esimo elemento di una qualsiasi stringa X utilizzerò la notazione $(X)_i$, ad esempio, $(MNMMX)_1 = M$, $(MNMMX)_5 = X$ e $(MNMMX)_6 = \emptyset$
- userò la notazione $rd\ X$ per la stringa ottenuta rimuovendo tutti i duplicati adiacenti da X, per esempio, $rd\ MMNMNNX = MNMNX$
- $\| X \|$ è l'operatore che ritorna $|rd\ X|$, ossia la cardinalità di $rd\ X$

Notiamo quindi che se le regole base definissero solo funzioni del tipo

$$X \to Y \implies \begin{cases} \| X \| < \| Y \| & if\ (X)_1 \neq (Y)_1 \\ \| X \| \leq \| Y \| & if\ (X)_1 = (Y)_1 \end{cases} \qquad (*)$$

allora potremmo conseguire che la funzione *join* non è ottenibile essendo che la funzione del tipo $MNMNX \to MNX$ ha come proprietà

$$(MNMNX)_1 = M = (MNX)_1 \wedge \| MNMNX \| > \| MNX \|$$

E quindi non rispetta $(*)$.

Dim:

Procedo per induzione sull'altezza h dell'albero di derivazione generato dalle regole definite.

Caso base: $(h = 0)$ nessuna regola mi permette di procedere senza eseguire almeno un passo quindi il predicato $X \rightarrow_0 Y$ è falso. Implica che la proprietà $(*)$ è vacuamente vera.

Caso induttivo: $(h \rightarrow h+1)$
La proprietà $(*)$ da dimostrare diventa quindi

$$X \rightarrow_{h+1} Y \implies \begin{cases} \| X \| < \| Y \| & if\ (X)_1 \neq (Y)_1 \\ \| X \| \leq \| Y \| & if\ (X)_1 = (Y)_1 \end{cases} \qquad (**)$$

L'ipotesi induttiva invece è così definita

$$X \rightarrow_{\leq h} Y \wedge (X)_1 \neq (Y)_1 \implies \| X \| < \| Y \| \qquad (IP1)$$
$$X \rightarrow_{\leq h} Y \wedge (X)_1 = (Y)_1 \implies \| X \| \leq \| Y \| \qquad (IP2)$$

Procedo per casi per ogni regola definita

- **[id]**

$$\frac{\| X \| \leq \| X \|}{X \xrightarrow{id}_{h+1} X}$$

  In questo caso è ovvio che $(X)_1 = (X)_1$
  E dato che l'operatore $\leq$ gode della proprietà riflessiva quindi ho dimostrato $(**)$ per l'identità

- **[composition]**

$$\frac{\exists Z. \quad Z \xrightarrow{f}_{\leq h} Y \quad X \xrightarrow{g}_{\leq h} Z}{X \xrightarrow{f \cdot g}_{h+1} Y} \quad \begin{matrix} (1) \\ (2) \end{matrix}$$

  Procediamo per casi su (1) e (2):

  - $(Z)_1 = (Y)_1 \wedge (X)_1 = (Z)_1$, quindi applico l'ipotesi induttiva $(IP2)$ a (1) e (2) e ottengo che $\| Z \| \leq \| Y \| \wedge \| X \| \leq \| Z \|$.
    Dato che l'operatore $\leq$ gode della proprietà riflessiva ho che vale quindi $\| X \| \leq \| Y \|$.
    Per concludere basta osservare che $(Z)_1 = (Y)_1 \wedge (X)_1 = (Z)_1 \implies (X)_1 = (Y)_1$ e quindi devo provare la proprietà $\| X \| \leq \| Y \|$ in $(**)$.

  - $(Z)_1 = (Y)_1 \wedge (X)_1 \neq (Z)_1$, quindi applico l'ipotesi induttiva $(IP2)$ a (1) e $(IP1)$ a (2) e ottengo che $\| Z \| \leq \| Y \| \wedge \| X \| < \| Z \|$.
    Concatenando (1) e (2) ottengo $\| X \| < \| Z \| \leq \| Y \|$ che implica $\| X \| < \| Y \|$.

Per concludere basta osservare che $(Z)_1 = (Y)_1 \wedge (X)_1 \neq (Z)_1 \implies (X)_1 \neq (Y)_1$ e quindi devo provare la proprietà $\| X \| < \| Y \|$ in $(**)$.

- $(Z)_1 \neq (Y)_1 \wedge (X)_1 = (Z)_1$, duale al caso precedente.

- $(Z)_1 \neq (Y)_1 \wedge (X)_1 \neq (Z)_1$, quindi applico l'ipotesi induttiva $(IP1)$ a (1) e (2) e ottengo che $\| Z \| < \| Y \| \wedge \| X \| < \| Z \|$.
  Dato che l'operatore $<$ gode della proprietà riflessiva ho che vale quindi $\| X \| < \| Y \|$.
  Ho concluso dimostrando $(**)$ dato che senza interessarci a quanto vale l'uguaglianza tra $(X)_1$ e $(Y)_1$ ho comunque già valida l'ipotesi più forte, ovvero $\| X \| < \| Y \|$.

Per tutti i quattro casi vale $(**)$ quindi ho dimostrato $(**)$ per la composizione.

- **[map]** dimostro per map_M e ottengo anche il duale map_N

$$\frac{X \overset{f}{\longrightarrow}_{\overset{(1)}{\leq} h} Y}{MX \overset{map_M\ f}{\longrightarrow}_{h+1} MY}$$

Ora notiamo che vale $(MX)_1 = M = (MY)_1$ quindi la proprietà da dimostrare in tutti i casi è la meno restrittiva, ossia $\| MX \| \leq \| MY \|$. Procedo per casi su (1):

- $(X)_1 = (Y)_1$, quindi applico l'ipotesi induttiva $(IP2)$ e ottengo $\| X \| \leq \| Y \|$.
  Ora mi trovo davanti a due sottocasi:
  - $(X)_1 = M (\implies (Y)_1 = M)$, allora ho che

$$\| MX \| = \| MMX' \| = \| MX' \| = \| X \|$$
$$\| MY \| = \| MMY' \| = \| MY' \| = \| Y \|$$

    quindi data l'ipotesi induttiva ottengo che

$$\| MX \| = \| X \| \leq_{ip.ind} \| Y \| = \| MY \|$$

  - $(X)_1 = N (\implies (Y)_1 = N)$, allora ho che

$$\| MX \| = \| MNX' \| \overset{(a)}{=} 1 + \| NX' \| = 1 + \| X \|$$
$$\| MY \| = \| MNY' \| \overset{(a)}{=} 1 + \| NY' \| = 1 + \| Y \|$$

    quindi data l'ipotesi induttiva ottengo che

$$\| MX \| = 1 + \| X \| \leq_{ip.ind} 1 + \| Y \| = \| MY \|$$

    Il passaggio (a) è giustificato dal fatto che $\| MNX' \| = |rd\ MNX'| = 1 + |rd\ NX'|$

– $(X)_1 \neq (Y)_1$, quindi applico l'ipotesi induttiva $(IP1)$ e ottengo $\| X \| < \| Y \|$.
Ora mi trovo davanti a due sottocasi:

* $(X)_1 = M(\implies (Y)_1 = N)$, allora ho che

$$\| MX \| = \| MMX' \| = \| MX' \| = \| X \|$$
$$\| MY \| = \| MNY' \| = 1 + \| NY' \| = 1 + \| Y \|$$

quindi data l'ipotesi induttiva ottengo che

$$\| MX \| = \| X \| <_{ip.ind} \| Y \| < 1 + \| Y \| = \| MY \|$$

* $(X)_1 = N(\implies (Y)_1 = M)$, allora ho che

$$\| MX \| = \| MNX' \| = 1 + \| NX' \| = 1 + \| X \|$$
$$\| MY \| = \| MMY' \| = \| MY' \| = \| Y \|$$

quindi data l'ipotesi induttiva ottengo che

$$\| MX \| = 1 + \| X \| <_{ip.ind} 1 + \| Y \|$$
$$\implies 1 + \| X \| \leq \| Y \| = \| MY \|$$

Concludo quindi di aver dimostrato $(**)$ per l'operatore *map*.

- **[unit]** dimostro per $unit_M$ e ottengo anche il duale $unit_N$

$$X \xrightarrow{unit_M}_{h+1} MX$$

Mi trovo davanti a due casi:

– $(X)_1 = M \implies \| MX \| = \| MMX' \| = \| MX' \| = \| X \|$
Per riflessività di $\leq$ ottengo

$$\| X \| = \| MX \| \leq \| MX \|$$

Inoltre essendo che $(X)_1 = M = (MX)_1$ allora ho dimostrato $(**)$ per questo caso.

– $(X)_1 = N \implies \| MX \| = \| MNX' \| = 1 + \| NX' \| = 1 + \| X \|$
Quindi ottengo

$$\| X \| < 1 + \| X \| = \| MX \|$$

Inoltre essendo che $(X)_1 = N \neq M = (MX)_1$ allora ho dimostrato $(**)$ per questo caso.

Concludo quindi di aver dimostrato $(**)$ per l'operatore *unit*.

- **[join]** dimostro per $join_M$ e ottengo anche il duale $join_N$

$$MMX \xrightarrow{join_M}_{h+1} MX$$

Per l'operatore di *join* concludo direttamente osservando le due proprietà

$$(MMX)_1 = M = (MX)_1$$
$$\| MMX \| = \| MX \| \leq \| MX \|$$

Quindi concludo anche per quest'ultimo caso che la proprietà $(**)$ è valida per l'operatore di *join*.

Questa dimostrazione prova quindi che, con la grammatica sopra definita, la funzione $join$ non è ottenibile essendo funzioni del tipo $MNMNX \to MNX$ hanno come proprietà

$$(MNMNX)_1 = M = (MNX)_1 \wedge \parallel MNMNX \parallel \; > \; \parallel MNX \parallel$$

Non ottenibili in questo contesto dato che non soddisfano $(*)$.

Conseguenze: come conseguenze possiamo notare che gli operatori map e $unit$ possono essere generati in quanto rispettano $(**)$

$$\begin{aligned} (MNX)_1 = M = (MNX)_1 \wedge \parallel MNX \parallel \leq \parallel MNX \parallel \quad & (map_M \; . \; map_N \; f) \\ \parallel X \parallel < \parallel MNX \parallel \quad & (unit_M \; . \; unit_N) \end{aligned}$$

Mentre nessuno tra $prod$, $dorp$ e $swap$ può essere costruito.

A.3 Monad Comprehensions Equivalence

Dimostriamo la seguente proprietà, mostrata nella sezione 4.3

$$[\ v_0 + v_1 \mid v_0 \leftarrow eval\ e_0\ env,\ v_1 \leftarrow eval\ e_1\ env\ ] =$$
$$bind\ (eval\ e_0\ env)(\lambda\ v_0 \rightarrow bind\ (eval\ e_1\ env)(\lambda\ v_1 \rightarrow unit(v_0 + v_1)))$$

Per semplicità utilizzerò le seguenti definizioni durante la dimostrazione, essendo equazioni non modifico la semantica della proprietà da dimostrare

$$E_i = eval\ e_i\ env \qquad \forall i \in [0,\ 1]$$

La proprietà da dimostrare diventa

$$[\ v_0 + v_1 \mid v_0 \leftarrow E_0,\ v_1 \leftarrow E_1\ ] =$$
$$bind\ E_0\ (\lambda\ v_0 \rightarrow bind\ E_1\ (\lambda\ v_1 \rightarrow unit\ (v_0 + v_1)))$$

Dim:

Prima di cominciare bisogna definire una regola che segue direttamente da un'osservazione delle regole $(M5)$ e $(M6)$

$$map\ (f\ .\ unit)\ =\ map\ f\ .\ unit \qquad (Sub)$$

Questa proprietà si prova facilmente con la seguente dimostrazione

$$\begin{array}{lll} & map\ (f\ .\ unit) & \{M2\} \\ = & map\ f\ .\ map\ unit & \{(*)\} \\ = & map\ f\ .\ unit & \end{array}$$

$$(*)\ join\ .\ unit \overset{(M5)}{=} id \overset{(M6^{-1})}{=} join\ .\ map\ unit$$
$$\Rightarrow unit\ =\ map\ .\ unit$$

Ora invece possiamo procedere alla dimostrazione principale

$$
\begin{array}{lll}
 & [\ v_0 + v_1 \mid v_0 \leftarrow E_0,\ v_1 \leftarrow E_1\ ] & \{joinComp\} \\
= & join\ [\ [\ v_0 + v_1 \mid v_1 \leftarrow E_1\ ] \mid v_0 \leftarrow E_0\ ] & \{mapComp\} \\
= & join\ [\ map\ (v_0+)\ E_1 \mid v_0 \leftarrow E_0\ ] & \{mapComp\} \\
= & join\ (map\ (\lambda\ v_0 \rightarrow & \\
 & \quad map\ (v_0+)\ E_1)\ E_0) & \{id\} \\
= & join\ (map\ (\lambda\ v_0 \rightarrow & \\
 & \quad (map\ (v_0+)\ .\ id)\ E_1)\ E_0) & \{M5^{-1}\} \\
= & join\ (map\ (\lambda\ v_0 \rightarrow & \\
 & \quad (map\ (v_0+)\ .\ join\ .\ unit)\ E_1)\ E_0) & \{M4^{-1}\} \\
= & join\ (map\ (\lambda\ v_0 \rightarrow & \\
 & \quad (join\ .\ map\ (map\ (v_0+))\ .\ unit)\ E_1)\ E_0) & \{Sub\} \\
= & join\ (map\ (\lambda\ v_0 \rightarrow & \\
 & \quad (join\ .\ map\ (map\ (v_0+)\ .\ unit))\ E_1)\ E_0) & \{M4^{-1}\} \\
= & join\ (map\ (\lambda\ v_0 \rightarrow & \\
 & \quad (join\ .\ map\ (unit\ .\ (v_0+)))\ E_1)\ E_0) & \{composition\} \\
= & join\ (map\ (\lambda\ v_0 \rightarrow & \\
 & \quad join\ (map\ (unit\ .\ (v_0+))\ E_1))\ E_0) & \{bind^{-1}\} \\
= & join\ (map\ (\lambda\ v_0 \rightarrow & \\
 & \quad bind\ E_1\ (unit\ .\ (v_0+)))\ E_0) & \{bind^{-1}\} \\
= & bind\ E_0\ (\lambda\ v_0 \rightarrow bind\ E_1\ (unit\ .\ (v_0+))) & \{composition\} \\
= & bind\ E_0\ (\lambda\ v_0 \rightarrow bind\ E_1\ (\lambda\ v_1 \rightarrow unit\ (v_0 + v_1))) & \{E_0\ e\ E_1\} \\
= & bind\ (eval\ e_0\ env)\lambda\ v_0 \rightarrow & \\
 & \quad bind\ (eval\ e_1\ env)\lambda\ v_1 \rightarrow & \\
 & \quad\quad unit\ (v_0 + v_1) &
\end{array}
$$

B State Transitions

In questa sezione mostro il motivo per cui non sempre la composizione ha l'effetto sperato. Il *Monad* per lo *State transitions* difatti rappresenta un esempio pratico in cui, dato un *Monad* M, la composizione generalizzata non è ciò che ci aspettiamo.

L'*ADT* per lo *state transitions*

$$\textbf{\textit{data}}\ State\ s\ a = ST(s \to (a,\ s))$$

Difatti il monade per le transizioni di stato è definito

$$\begin{gathered}
\textbf{\textit{instance}}\ Functor\ (State\ s)\ \textbf{\textit{where}} \\
map\ f\ (ST\ x)\ = ST(\lambda s \to \textbf{\textit{let}}\ (s',\ x) = st\ s\ \textbf{\textit{in}}\ (s',\ f\ x)) \\
\textbf{\textit{instance}}\ Premonad\ (State\ s)\ \textbf{\textit{where}} \\
unit\ x = ST(\lambda s \to (s,\ x)) \\
\textbf{\textit{instance}}\ Monad\ (State\ s)\ \textbf{\textit{where}} \\
join\ (ST\ m) = ST(\lambda s \to \textbf{\textit{let}}\ (s',\ ST\ m') = m\ s\ \textbf{\textit{in}}\ m'\ s')
\end{gathered}$$

Possiamo notare come lo *State* possa essere definito in termini di *Monad Composition* tra *Reader* e $Writer$. Difatti possiamo comporre lo *State* in questo modo, $State\ s\ a = (s \to)\ .\ (Writer\ s)\ a$. Oppure mediante il framework della sezione 6, $\textbf{\textit{data}}\ State\ s\ a = DComp\ (s \to)\ (Writer\ s)\ a$.

Questo nuovo *Monad* non è equivalente allo *State* definito sopra, difatti nel caso *unit* il monade composto ritorna, dato un qualsiasi s lo *zero* del *Monoid* s, inoltre richiede appunto che s sia *Monoid*.

Potremmo quindi creare un caso specifico per combinare un qualsiasi *Monad* con lo *State Transitions*, prima di tutto definiamo il suo tipo, sarà generico in m

$$\textbf{\textit{data}}\ StateM\ m\ s\ a = STM(s \to m\ (a,\ s))$$

Poi istanziamolo ad essere un monade

$$\begin{gathered}
\textbf{\textit{instance}}\ Monad\ m \implies Functor\ (StateM\ m\ s)\ \textbf{\textit{where}} \\
map\ f\ (STM\ xs)\ = STM(\lambda s \to [(s',\ f\ x) \mid (s',\ x) \leftarrow xs\ s]) \\
\textbf{\textit{instance}}\ Monad\ m \implies Premonad\ (StateM\ m\ s)\ \textbf{\textit{where}} \\
unit\ xs = STM(\lambda s \to [(s,\ x)]) \\
\textbf{\textit{instance}}\ Monad\ m \implies Monad\ (StateM\ m\ s)\ \textbf{\textit{where}} \\
join\ (STM\ xss) = STM(\lambda s \to [(s'',\ x) \mid (STM\ xs,\ s') \leftarrow xss\ s, \\
(s'',\ x) \leftarrow xs\ s'])
\end{gathered}$$

Ora riusciamo a comporre un qualsiasi *Monads* con il sistema di *State Transitions* , notiamo però che abbiamo creato un sistema di composizione generalizzata ad hoc. Quindi non sfruttiamo il framework precedentemente creato.

Una spiegazione a questo fatto è che la composizione di monadi generata dal nostro framework non riesce ad esprimere la relazione che intercorre tra i componenti del monade composto. Questa spiegazione, istanziata al nostro caso, in particolare non esprime l'equivalenza tra il parametro in input al *Reader Monad* con l'output prodotto nel *Writer Monad*.

Steele [4], prova a definire un meccanismo generico anche per gli *ADT* che hanno queste caratteristiche di relazioni tra monadi composti. Utilizza un sistema di pseudomonadi per riuscire a comporre monadi lasciando il monade arbitrario non più ai "lati" della composizione ma al "centro", non approfondirò questo punto all'interno di questo report.

C Monoid

Un *Monoid* è un tipo di dato astratto definito dalla seguente dichiarazione di classe

```
class Monoid s where
  zero :: s
  add :: s → s → s
```

Dato che nel contesto di questo report un *Monoid* può essere utilizzato esclusivamente per operare sui *Monads* richiedo che *add* sia associativa a destra e sinistra con *zero*. Possibili *Monoid* sono le liste, le funzioni e i numeri interi

```
instance Monoid [a] where
  zero = [ ]
  add = (++ )

instance Monoid (a → a) where
  zero = id
  add = (.)

instance Monoid Int where
  zero = 0
  add = (+)
```

D Scala Code

Come richiesto in questa sezione inserirò degli *snippets* di codice *Scala* per la composizione generalizzata di monadi. Non essendo pratico di *scala* sono ricorso agli esempi definiti da Omura [8] e Milewski [7]. Questo esempio ricrea il framework generalizzato per la composizione simile a quello visto nella sezione 6.

Listing 10: Functor and Monad Traits

```
1  trait Functor[F[_]]{
2    def map[A,B](ma:F[A])(f:(A) => B):F[B]
3  }
4  // per l'esempio unisco la definizione di Monad e Premonad
5  trait Monad[M[_]] extends Functor[M]{
6    def unit[A](a:A):M[A]
7    // come il join operator
8    def flatten[A](mm:M[M[A]]):M[A]
9    // flatMap puo' essere utile negli esempi pratici
10   // derivata da map e join
11   def flatMap[A,B](ma:M[A])(f:(A)=>M[B]) = flatten(map(ma)(f))
12 }
```

Per l'esempio considererò una coppia di monadi composta con il costruttore *swap*, vedi sezione 5.4

Listing 11: Swap Constructor

```
1  trait SComposable[M[_],N[_]]{
2    def M:Monad[M]
3    def N:Monad[N]
4    def swap[A](nm:N[M[A]]):M[N[A]]
5  }
```

Definiamo qualche istanza di *Monad*

Listing 12: Type class instances

```
1  implicit val OptionMonad = new Monad[Option]{
2    def map[A, B](ma: Option[A])(f: (A) => B) = ma.map(f)
3    def unit[A](a: A) = Option(a)
4    def flatten[A](mm: Option[Option[A]]) = mm.flatMap(identity)
5  }
6  implicit val ListMonad = new Monad[List]{
7    def map[A, B](ma: List[A])(f: (A) => B) = ma.map(f)
8    def unit[A](a: A) = List(a)
9    def flatten[A](mm: List[List[A]]) = mm.flatten
10 }
```

Definiamo anche la sintassi per *map* e $flatmap$ per le monadi scala, essenzialmente serve per poterle utilizzare dentro al ciclo for.

Listing 13: Composer

```
case class Mval[M[_]:Monad,A](v:M[A]){
  def map[B](f:(A)=>B) = implicitly[Monad[M]].map(v)(f)
  def flatMap[B](f:(A)=>M[B])
    = implicitly[Monad[M]].flatMap(v)(f)
}
implicit def v2M[M[_]:Monad, A](v:M[A]) = Mval(v)
def compose[M[_],N[_],A](mna:M[N[A]])
  (implicit mn:Monad[({type MN[x] = M[N[x]]})#MN])
  :Mval[({type MN[x] = M[N[x]]})#MN,A]
  = Mval[({type MN[x] = M[N[x]]})#MN,A](mna)
```

Quindi passiamo a definire istanze di *SComposable* come *Monad*

Listing 14: SComposable is Monad

```
implicit def monadSCompose[M[_],N[_]]
  (implicit m:Monad[M], n:Monad[N], s:SComposable[M,N])
  = new Monad[({type MN[x] = M[N[x]]})#MN]{
    def map[A,B](ma:M[N[A]])(f: (A) => B)
      = m.map(ma)(n.map(_)(f))
    def unit[A](a: A) = m.unit(n.unit(a))
    def flatten[A](mnmn:M[N[M[N[A]]]])
      = m.map(m.flatten(m.map(mnmn)(s.swap(_))))(n.flatten(_))
}
```

Ora definiamo la composizione generalizzata mediante *swap* per le due istanze definite

Listing 15: Composing List

```
  implicit def MonadListSComposable[M[_]]
    (implicit monadM:Monad[M])
    = new SComposable[M,List] {
    def M = monadM
    def N = implicitly[Monad[List]]
    def swap[A](nm: List[M[A]]):M[List[A]] = nm match {
      case List() => monadM.unit(List())
      case x::xs => for {
         y <- x;
         ys <- swap(xs)
       } yield y::ys
    }
  }
```

Listing 16: Composing Optional

```
implicit def MonadOptionSComposable[M[_]]
  (implicit monadM: Monad[M])
  = new SComposable[M,Option]{
    def M = monadM
    def N = implicitly[Monad[Option]]
    def swap[A](nm: Option[M[A]]) = nm match{
      case Some(ma) => monadM.map(ma)(Option(_))
      case None => monadM.unit(None)
    }
}
```

Per finire concludo con la definizione di un semplice main per testare il codice appena scritto

Listing 17: Main

```
// outputs: List(Some(4), Some(5), Some(8), Some(9))
println(for {
   i<- compose(List(Option(1), None, Option(5)))
   j<- compose(List(Option(3), Option(4)))
  } yield i+j)

// outputs: Some(List(4, 5, 8, 9))
println(for {
   i<- compose(Option(List(1, 5)))
   j<- compose(Option(List(3, 4)))
  } yield i+j)
```

Riferimenti bibliografici

[1] M. P. Jones, L. Duponcheel, *Composing Monads**, Research Report YALEU/DCS/RR-1004, December 1993

[2] P. Walder, *Mathematical Structures in Computer Science*, Nice, France, June 1994

[3] P. Walder, *Monads fo functional programming*, University of Glasgow, Scotland

[4] L. Steele, *Building Interpreters by Composing Monads*, Thinking Machines Corporation, Cambridge, Massachusetts

[5] J. Hughes, *Why Functional programming Matters*, The University, Glasgow

[6] L. P. Jones, D. R. Lester, *Implementing Functional Languages: a tutorial*, University of Glasgow, University of Manchester

[7] Bartosz Milewski, `https://github.com/typelevel/CT_from_Programmers.scala/tree/master/src/main/tut`

[8] Shingo Omura, `https://github.com/everpeace/composing-monads`