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 impoerative 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].

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:

$$
\begin{array}{rll}
\mathit{Expr} \rightarrow & \mathit{Const\ Value} & \text{Valore costante} \\
| & \mathit{Var\ Name} & \text{Variabile in memoria} \\
| & \mathit{Expr} + \mathit{Expr} & \text{Addizione tra espressioni} \\
| & \mathit{Trace\ String\ Expr} & \text{Struttura di log} \\
\mathit{Env} \rightarrow & [(\mathit{Name}, \mathit{Value})] & \text{Ambiente di mapping delle variabili} \\
\mathit{Name} \rightarrow & \mathit{String} & \\
\mathit{Value} \rightarrow & \mathit{Int} &
\end{array}
$$

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:

$$
\begin{array}{ll}
\textbf{data}\ \mathit{Error\ a} = \mathit{Error\ String} \ | \ \mathit{Ok\ a} & \\
\textbf{data}\ \mathit{Writer\ s\ a} = (s, a) & \text{con s :: } \mathit{Monoid}\text{, vedi C}
\end{array}
$$

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.

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

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)*.

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*

$$\mathbf{data}\ Exception\ a = Raise\ String \mid Return\ a$$

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

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

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

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

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

Nota se *Exception* 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

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

$$\begin{aligned} map &:: Monoid\ s \Rightarrow (a \to b) \to Writer\ s\ a \to Writer\ s\ b \\ map\ f\ (s,\ a) &= (s,\ f\ a) \end{aligned}$$

$$\begin{aligned} unit &:: Monoid\ s \Rightarrow a \to Writer\ s\ a \\ unit\ x &= (zero,\ x) \end{aligned}$$

$$\begin{aligned} join &:: Monoid\ s \Rightarrow Writer\ s\ (Writer\ s\ a) \to Writer\ s\ a \\ join\ (s',\ (s'',\ a)) &= (s' +\!\!+\ s'',\ a) \end{aligned}$$

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

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

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

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

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

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

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 $(M\ 1 \dots 7)$ 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 $(M2)$

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

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 $(M3)$

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

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

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

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

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

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

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

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

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

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

$$dorp \ :: \ M \ (N(Ma)) \rightarrow M \ (N \ a)$$

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:

$$\begin{aligned} prod &= map_M\ join_N\ .\ swap \\ dorp &= join_M\ .\ map_M\ swap \end{aligned}$$

La funzione *swap* deve soddisfare le seguenti leggi

$$\begin{array}{rcll} swap\ .\ map_N\ (map_M\ f) & = & map_M\ (map_N\ f)\ .\ swap & (S1) \\ swap\ .\ unit_N & = & map_M\ unit_N & (S2) \\ swap\ .\ map_N\ unit_M & = & unit_M & (S3) \\ prod\ .\ map_N\ dorp & = & dorp\ .\ prod & (S4) \end{array}$$

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 $(M1 \dots 7)$ tale che

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

Quali *composite monads* definito in questo modo può essere ottenuto dal costruttore *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*

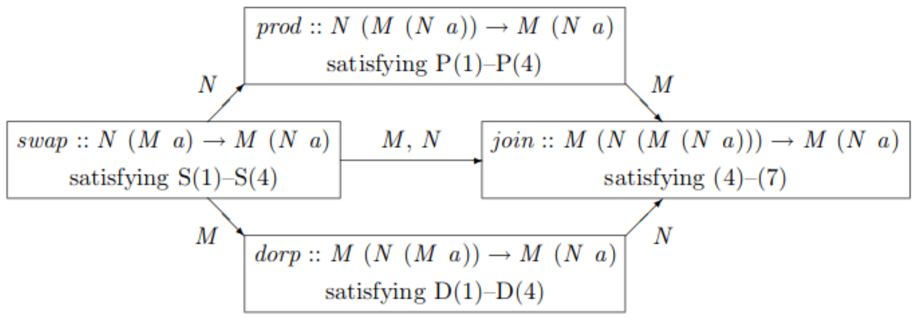
$$swap = prod \ . \ map_N \ (map_M \ unit_N)$$
$$swap = dorp \ . \ unit_M$$

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 $(J2)$.

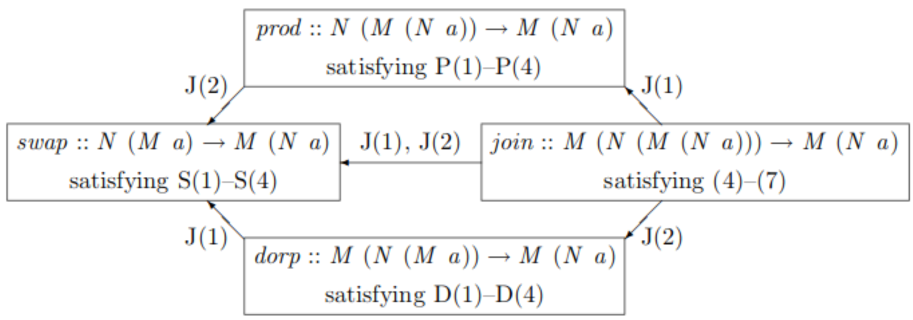
5.5 Summary

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 di essere codificate in un qualsiasi linguaggio funzionale.

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.

$$\begin{array}{l} \textbf{\textit{class}}\ Functor\ f\ \textbf{\textit{where}} \\ map :: (a \to b) \to (f\ a \to f\ b) \end{array}$$

$$\begin{array}{l} \textbf{\textit{class}}\ Functor\ m \Rightarrow Premonad\ m\ \textbf{\textit{where}} \\ \qquad unit :: a \to m\ a \end{array}$$

$$\begin{array}{l} \textbf{\textit{class}}\ Premonad\ m \Rightarrow Monad\ m\ \textbf{\textit{where}} \\ \qquad join :: m\ (m\ a) \to m\ a \end{array}$$

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

6.2 Composition Construction

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

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

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.

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

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 $Composer$.

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

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

$$\begin{aligned} &right :: (Composer\ c,\ Premonad\ f) \implies g\ a \to c\ f\ g\ a \\ &right = close\ .\ unit \end{aligned}$$

$$\begin{aligned} &left :: (Composer\ c,\ Functor\ f,\ Premonad\ g) \implies f\ a \to c\ f\ g\ a \\ &left = close\ .\ map\ unit \end{aligned}$$

6.3 Programming Prod Construction

6.4 Programming Dorp Construction

6.5 Programming Swap Construction

6.6 Some Examples

Reader

Writer

Error

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. Tutto questo è coperto dalla seguente definizione di dato astratto:

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

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

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

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

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

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

$$\begin{aligned}
&inError :: Error\ a \to M\ a\\
&inError = right\ .\ right
\end{aligned}$$

$$\begin{aligned}
&inReader :: (Env \to a) \to M\ a\\
&inReader = left
\end{aligned}$$

$$\begin{aligned}
&inWriter :: Writer\ [String]\ a \to M\ a\\
&inWriter = right\ .\ left
\end{aligned}$$

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

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

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

A Proofs

A.1 Exception is a Monad

Dimostriamo che il tipo astratto *Exception* 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*, $M1 \ldots 7$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Osservazione $(*)$, utilizzando la definizione inversa di *join* richiedo che il suo parametro s sia di tipo $s :: \mathit{Exception}$

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

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

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

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

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

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, $rd\ MMNMNNX = MNMNX$
- $\parallel X \parallel$ 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

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

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

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

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

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]**

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

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

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

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

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

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 $(**)$

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

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

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

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

Ora invece possiamo procedere alla dimostrazione principale

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

B State Transformer

C Monoid

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

$$
\begin{array}{l}
\textbf{class}\ \text{Monoid s}\ \textbf{where} \\
\quad zero :: s \\
\quad add :: s \to s \to s
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{instance}\ \text{Monoid [a]}\ \textbf{where} \\
\qquad zero = [\,] \\
\qquad add = (+\!\!+) \\
\\
\textbf{instance}\ \text{Monoid (a} \to \text{a)}\ \textbf{where} \\
\qquad zero = id \\
\qquad add = (.) \\
\\
\textbf{instance}\ \text{Monoid Int}\ \textbf{where} \\
\qquad zero = 0 \\
\qquad add = (+)
\end{array}
$$

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