**ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA**
**CAMPUS DI CESENA**

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

# Gestione degli effetti in linguaggi di programmazione funzionale: tecniche di modellazione e interpretazione

Tesi di laurea in
PARADIGMI DI PROGRAMMAZIONE E SVILUPPO

*Relatore*
**Prof. Mirko Viroli**

*Candidato*
**Giacomo Cavalieri**

*Correlatori*
**Dott. Roberto Casadei**
**Dott. Gianluca Aguzzi**

Anno Accademico 2021-2022

*Alla mia famiglia e ai miei amici*

# Abstract

Nel corso degli anni i linguaggi di programmazione imperativi "mainstream" – come Java, Python e JavaScript – sono stati fortemente influenzati dal paradigma di programmazione funzionale, adottando e riconoscendo l'utilità di numerose tecniche come funzioni anonime, tipi di dato algebrici (ADT), *pattern matching*, strutture dati immutabili, ecc.

Un approccio che ad oggi rimane prettamente confinato al mondo della programmazione funzionale pura è quello che consiste nel modellare esplicitamente gli effetti delle funzioni, separando in maniera netta – grazie al supporto automatico del *type system* – codice puro da quello che può presentare side effect.

In linguaggi di programmazione imperativi che non supportano nativamente questi meccanismi, è possibile ottenere una separazione analoga utilizzando architetture come quella "esagonale". Tuttavia, il rispetto di queste buone pratiche di programmazione è lasciato interamente all'autodisciplina del programmatore e non è imposto automaticamente dal compilatore. La risultante presenza incontrollata di side effect ha l'effetto di creare reti di dipendenze invisibili che rendono complesso ragionare sulle proprietà del codice, testarlo o effettuare *refactoring*.

L'obiettivo di questa tesi è fornire una visione complessiva delle principali tecniche che possono essere adottate per modellare e rendere esplicita la presenza dei side effect nelle funzioni: *Monad Transformer Library* (MTL) e *free monad*. Infine, viene mostrato un approccio emergente basato sull'utilizzo degli effetti algebrici. Tutti i meccanismi analizzati sono introdotti in maniera graduale accompagnando la trattazione con diversi esempi e frammenti di codice che evidenziano i benefici che queste tecniche possono apportare.

# Indice

**Abstract** **i**

**1 Introduzione** **1**
1.1 Criticità dei side effect . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.1.1 Complessità del refactoring . . . . . . . . . . . . . . . . . . . 2
1.1.2 Difficoltà nel testare funzioni impure . . . . . . . . . . . . . . 3
1.2 Obiettivi della tesi . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

**2 Modellazione dei side effect** **7**
2.1 Linguaggi funzionali puri e side effect . . . . . . . . . . . . . . . . . . 7
2.1.1 Esempi di side effect modellati esplicitamente . . . . . . . . 7
2.1.2 Svantaggi della modellazione esplicita . . . . . . . . . . . . . 10
2.2 Modellazione degli effetti tramite monadi . . . . . . . . . . . . . . . . 12
2.2.1 Cos'è una monade? . . . . . . . . . . . . . . . . . . . . . . . 13
2.2.2 Encoding di una monade . . . . . . . . . . . . . . . . . . . . 14
2.2.3 Esempi di monadi . . . . . . . . . . . . . . . . . . . . . . . . 15
2.2.4 Zucchero sintattico per codice monadico . . . . . . . . . . . 18
2.2.5 Vantaggi nell'uso delle monadi . . . . . . . . . . . . . . . . . 20
2.3 Input e output puri . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.3.1 Modello di valutazione lazy . . . . . . . . . . . . . . . . . . . 22
2.3.2 I/O monadico in Haskell . . . . . . . . . . . . . . . . . . . . 23
2.3.3 I/O monadico in Scala . . . . . . . . . . . . . . . . . . . . . 26

**3 Stack di monadi** **31**
3.1 Problemi nel comporre più side effect . . . . . . . . . . . . . . . . . . 31
3.1.1 Parsing monadico . . . . . . . . . . . . . . . . . . . . . . . . 31
3.1.2 Duplicazione di codice . . . . . . . . . . . . . . . . . . . . . 33
3.2 Monad transformer . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
3.2.1 Encoding di un monad transformer . . . . . . . . . . . . . . 35
3.2.2 Esempi di monad transformer . . . . . . . . . . . . . . . . . 37
3.2.3 Rimozione della duplicazione di codice . . . . . . . . . . . . 41

3.3 Stack di monadi . . . 41
3.3.1 Composizione di monadi . . . 42
3.3.2 Importanza dell'ordine nella composizione . . . 43
3.3.3 Miglioramento del parsing monadico . . . 44

**4 Monad Transformer Library** **47**
4.1 Problemi nell'uso dei monad transformers . . . 47
4.1.1 Lifting manuale delle operazioni . . . 47
4.1.2 Principio di privilegio minimo . . . 49
4.1.3 Violazione dell'incapsulamento . . . 50
4.2 L'approccio MTL . . . 50
4.2.1 Idea alla base di MTL . . . 51
4.2.2 Composizione di più effetti . . . 53
4.2.3 Interpretazione della computazione . . . 53
4.3 MTL come effect system . . . 56
4.3.1 Definizione di effetti arbitrari . . . 57

**5 Free Monad** **61**
5.1 Implementazione di una Free Monad . . . 61
5.1.1 Descrizione astratta di un programma monadico . . . 62
5.2 Interpretazione di una free monad . . . 65
5.2.1 Interpretazione delle istruzioni di un programma . . . 65
5.2.2 Interpretazione di un'intero programma . . . 66
5.2.3 Ispezione dell'AST di una free monad . . . 66
5.3 Composizione di più DSL . . . 70
5.3.1 Composizione modulare di linguaggi . . . 70
5.4 Free monad come effect system . . . 75
5.4.1 Definizione di effetti arbitrari . . . 76

**6 Effetti Algebrici** **79**
6.1 Il linguaggio Koka . . . 80
6.1.1 Sintassi di base . . . 80
6.1.2 Tipi di dato algebrici e pattern matching . . . 82
6.1.3 Effect system . . . 83
6.2 Idea alla base degli effetti algebrici . . . 85
6.2.1 Modellazione degli effetti . . . 85
6.2.2 Interpretazione degli effetti . . . 86
6.2.3 Interpretazione degli effetti come eccezioni ripristinabili . . . 89
6.3 Definizione di effetti arbitrari . . . 92
6.3.1 Meccanismi di controllo di flusso complessi . . . 93
6.4 Confronto con l'approccio monadico . . . 96

6.4.1 Boilerplate ed inferenza . . . . . . . . . . . . . . . . . . . . . 96
6.4.2 Stile di programmazione . . . . . . . . . . . . . . . . . . . . . 98

**7 Conclusioni 101**

**A Dimostrazioni delle leggi monadiche 103**
A.1 Dimostrazione per la monade identità . . . . . . . . . . . . . . . . . . 103
A.2 Dimostrazione per la monade Optional . . . . . . . . . . . . . . . . . 104
A.3 Dimostrazione per la monade State . . . . . . . . . . . . . . . . . . . 106

**B Basi del linguaggio Haskell 109**
B.1 Sintassi di base . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
B.1.1 Definizione e invocazione di funzioni . . . . . . . . . . . . . 109
B.2 Tipi di dato algebrici e pattern matching . . . . . . . . . . . . . . . . 110
B.3 Type classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111
B.3.1 Uso delle type class per fornire un contesto alle funzioni . . . 112

# Capitolo 1

# Introduzione

La ragion d'essere di un qualunque programma è quella di produrre un qualche effetto tangibile sul mondo esterno: che sia scrivere dei dati su disco, inviare un messaggio sulla rete, stampare dei caratteri a schermo ecc. Linguaggi dalla natura imperativa come C, Java o Scala forniscono delle funzioni apposite per ottenere tali effetti; in Scala, per esempio, si potrebbe implementare la seguente funzione per apporre una stringa al contenuto di un file su disco[1]:

```
def appendToFile(file: File, line: String): Unit =
  println(f"Appending $line to $file")
  val writer = FileWriter(file, true)
  try writer.write(f"$line\n")
  finally writer.close
```

Lo scopo di funzioni come `appendToFile`, `write` e `println` non è quello di produrre un risultato – il loro valore di ritorno è sempre `Unit` – ma di mettere in atto dei side effect.

Più in generale, con side effect si intende una qualunque interazione della funzione con un ambiente diverso da quello locale. Possono essere quindi considerati side effect il modificare una variabile globale o un parametro passato per riferimento, lanciare un'eccezione, l'effettuare operazioni di input e output – come mostrato precedentemente in `appendToFile` – o il chiamare una funzione che a sua volta presenta dei side effect [1].

Le funzioni che presentano side effect sono spesso anche dette *impure* mentre funzioni che non hanno side effect sono anche dette *pure* o caratterizzate da *trasparenza referenziale*.

[1] In questa funzione così come in tutte le altre funzioni Scala a seguire viene adottata la sintassi introdotta da Scala 3 [41].

## 1.1 Criticità dei side effect

Il fatto che una funzione possa avere interazioni con il mondo esterno presenta però alcune criticità. Prendiamo ad esempio una funzione Scala `f : Int => Int`[2]; a una prima osservazione potrebbe risultare sorprendente, ma in generale non varrà l'uguaglianza per cui `f(1) + f(1) = 2 * f(1)`. Si consideri la seguente implementazione di `f`:

```
var counter = 0
def f(x: Int): Int =
  counter = counter + 1
  x + counter
```

In questo caso `2 * f(1) = 4` ma `f(1) + f(1) = 5`. Infatti `f` legge e modifica una variabile globale dalla quale dipende il suo valore di ritorno; per via di questo side effect ogni chiamata successiva alla funzione, a parità di argomento, produrrà un valore differente. Dall'esempio è possibile intuire come, in presenza di side effect, sia più difficile ragionare sul comportamento delle funzioni: per poterlo comprendere non è sufficiente osservare il corpo della funzione in analisi ma è necessario conoscere il contesto nel quale questa viene invocata.

Al contrario, quando si ha a che fare con una funzione pura è possibile manipolarla ed effettuare refactoring del codice in maniera *equazionale*: è sempre possibile sostituire alla chiamata di funzione il suo valore di ritorno senza il rischio di alterare la semantica del programma. Se `f` è una funzione pura allora `f(1) + f(1) = 2 * f(1)`.

### 1.1.1 Complessità del refactoring

*L'ordine* e il *numero delle invocazioni* di funzioni con side effect ha importanza nel determinare il comportamento complessivo di un programma e – come mostrato nell'esempio della funzione `f` – modificare uno di questi due fattori può comportarne uno stravolgimento. Questa mancanza di trasparenza referenziale porta il codice ad avere una maggiore resistenza al cambiamento: è più difficile per il programmatore – così come per strumenti automatici – effettuare del refactoring avendo la certezza di mantenere inalterata la semantica del programma originale [2].

Per utilizzare le efficaci parole di Robert Martin "I side effect sono bugie. La tua funzione promette di fare una cosa, ma in realtà fa anche qualcos'altro *di*

[2] La notazione `f : Int => Int` indica una funzione chiamata `f` che prende come argomento un valore di tipo `Int` e restituisce un valore di tipo `Int`.

*nascosto.* [...] Sono falsità che spesso risultano in strani accoppiamenti temporali e comportano dipendenze nell'ordine delle funzioni"[3].

### 1.1.2 Difficoltà nel testare funzioni impure

Le stesse motivazioni che rendono difficile ragionare e compiere refactoring di codice con side effect ne complicano anche la fase di *unit testing*. Per testare una funzione impura in isolamento è necessario che vengano effettuate apposite operazioni prima e dopo lo svolgimento di ciascun test per garantire che il risultato non possa cambiare a seconda del loro ordine di esecuzione.

Immaginando di dover testare il comportamento di `f`:

```
def test1: Unit =
  val oldCounter = counter
  counter = 0 // set up
  f(1) shouldBe 2
  counter = oldCounter // tear down

def test2: Unit =
  val oldCounter = counter
  counter = 1 // set up
  f(1) shouldBe 3
  counter = oldCounter // tear down
```

Si può notare come la dipendenza implicita introdotta dal side effect di `f` renda necessario dover modificare opportunamente lo stato globale `counter` prima e dopo l'esecuzione di ogni test. Non solo, tali test non potranno nemmeno essere mandati in esecuzione simultaneamente[4] in quanto l'*interleaving* delle loro operazioni potrebbe portare a fallimenti imprevedibili. Due esempi di interleaving che portano a risultati differenti dei test sono riportati alle Tabelle 1.1 e 1.2.

Quest'ultimo esempio fornisce anche un ottimo spunto per riflettere sull'ulteriore difficoltà nella possibilità di parallelizzare codice che presenta side effect. Poiché i side effect introducono dipendenze temporali nascoste, non è possibile garantire che l'esecuzione di funzioni impure in parallelo – e quindi con ordini di esecuzione che potrebbero variare a seconda dell'interleaving dei processi – dia sempre lo stesso risultato. Quindi non potranno essere parallelizzate in maniera automatica; sarà invece necessario ricorrere a euristiche [4] o estensioni del linguaggio [5] per poter capire se una funzione è pura o meno e poter sfruttare automaticamente i vantaggi

[3]Traduzione dal testo originale: "Side effects are lies. Your function promises to do one thing, but it also does other *hidden* things. [...] They are [...] mistruths that often result in strange temporal couplings and order dependencies" [3, p. 44].

[4]Si potrebbe ovviare a questo problema introducendo un meccanismo di *locking* della risorsa condivisa per gestire l'esecuzione parallela dei test. Tuttavia, questa strategia aggiungerebbe una complessità accidentale non indifferente per testare una funzione semplice come `f`!

| `test1` | `test2` |
| --- | --- |
| `oldCounter = counter` | |
| `counter = 0` | |
| `f(1) shouldBe 2` | |
| | `oldCounter = counter` |
| | `counter = 1` |
| | `f(1) shouldBe 3` |
| `counter = oldCounter` | |
| | `counter = oldCounter` |

Tabella 1.1: Esempio di interleaving delle operazioni dei test `test1` e `test2` che porterebbe al successo di entrambi i test.

| `test1` | `test2` |
| --- | --- |
| `oldCounter = counter` | |
| | `oldCounter = counter` |
| `counter = 0` | |
| | `counter = 1` |
| `f(1) shouldBe 2` | |
| `counter = oldCounter` | |
| | `f(1) shouldBe 3` |
| | `counter = oldCounter` |

Tabella 1.2: Esempio di interleaving delle operazioni dei test `test1` e `test2` che porterebbe al fallimento dell'asserzione fatta da `test1`.

offerti dai moderni processori multi-core senza dover intervenire sulla struttura del programma originale.

## 1.2 Obiettivi della tesi

Seguire buone pratiche di programmazione può limitare le problematiche associate alla presenza di side effect. Tuttavia, è interessante osservare come gli stessi linguaggi di programmazione possano aiutare il programmatore a rendere espliciti gli effetti delle funzioni.

Questa rassegna è diretta al programmatore che voglia approfondire il tema della gestione esplicita dei side effect e non abbia particolari conoscenze pregresse; l'unico prerequisito necessario è una certa familiarità con il linguaggio Scala, necessaria a comprendere gli esempi riportati.

La seguente rassegna affronta i diversi approcci che possono essere adottati per modellare gli effetti delle funzioni; partendo da una descrizione del concetto di monade e dei suoi impieghi, sono poi analizzati meccanismi più sofisticati come MTL e free monad.

Queste tecniche sono ormai ampiamente diffuse nell'ambito dei linguaggi di programmazione funzionali puri – come Haskell – e stanno iniziando a guadagnare terreno anche in linguaggi di programmazione più diffusi fra cui Python e TypeScript [6, 42, 43]. In particolare, uno dei campi applicativi di maggior rilievo in cui queste tecniche possono essere adottate con successo è quello della concorrenza strutturata: framework come *ts-effect* [42] e *ZIO* [44] sono prominenti esempi di come un approccio funzionale puro possa permettere di affrontare la realizzazione di sistemi concorrenti complessi in maniera modulare, componibile ed espressiva.

La trattazione si conclude con l'analisi di un approccio emergente basato su effetti algebrici, mostrando come questo possa risolvere diversi problemi legati all'adozione delle tecniche monadiche e semplificare la diffusione della gestione esplicita degli effetti nel mainstream [7].

Sicuramente l'adozione di un approccio basato sulla programmazione funzionale pura e la gestione esplicita dei side effect rappresenta un forte cambiamento rispetto alla più diffusa programmazione ad oggetti. Questo nuovo paradigma potrebbe richiedere tempo prima di essere recepito o influenzare il mainstream; rimane evidente il fatto che quello verso cui sembra puntare la ricerca da ormai diverso tempo è un futuro "orientato agli effetti". In conclusione, l'obiettivo del seguente elaborato è quello di fornire una visione d'insieme delle tecniche che, con grande probabilità, influenzeranno maggiormente l'evoluzione dei linguaggi nei prossimi anni.

# Capitolo 2

# Modellazione dei side effect

Date le criticità evidenziate nella sezione precedente, diverse pratiche di buona programmazione suggeriscono di ridurre al minimo le funzioni che presentano side effect [3, p. 44] e, quando inevitabili, di renderli espliciti nel nome della funzione [3, p. 313].

Tuttavia, si possono individuare altre tecniche più sofisticate per tracciare i side effect delle funzioni. Queste nascono nel contesto dei linguaggi funzionali puri, perciò è necessario comprendere come questa classe di linguaggi possa conciliare la presenza di side effect con la loro natura puramente funzionale.

## 2.1 Linguaggi funzionali puri e side effect

Un linguaggio funzionale si dice *puro* se le sue funzioni sono referenzialmente trasparenti. Questa totale assenza di side effect sembra in netto contrasto con la possibilità di scrivere codice di una qualche utilità: come può un programma puro interagire con il mondo esterno, leggere il valore di uno stato globale o lanciare eccezioni?

La soluzione consiste nella possibilità di "simulare" la presenza di side effect tramite opportune modifiche ai tipi delle funzioni mantenendole pure. In seguito sono riportati alcuni esempi di questo approccio in Scala[1].

### 2.1.1 Esempi di side effect modellati esplicitamente

#### Eccezioni

Le eccezioni sono un meccanismo di controllo del flusso che permette di interrompere l'esecuzione di una funzione e di ritornare il controllo al chiamante. Tuttavia, una

[1]Sebbene Scala sia un linguaggio impuro può essere utilizzato come se fosse un linguaggio puro evitando di ricorrere ai meccanismi che fornisce per produrre side effects.

funzione che ne fa uso non avrà trasparenza referenziale:

```
def divBy(n: Int): Int =
  n match
    case 0 => throw Exception("n = 0")
    case _ => 10 / n
```

`divBy` non può essere assimilata a una funzione $divBy : \mathbb{Z} \to \mathbb{Z}$ in termini matematici; infatti, per certi input – in questo caso 0 – la funzione non restituisce un valore appartenente al proprio codominio ma lancia un'eccezione.

Per rendere esplicito il fatto che la funzione possa terminare in maniera anomala per determinati input si può estenderne il codominio: ritornando all'analogia matematica si può pensare ad `divBy` come a una funzione $divBy : \mathbb{Z} \to \mathbb{Z} \cup \{\texttt{Error}\}$. L'equivalente implementazione Scala sarebbe:

```
enum Result:
  case Ok(value: Int)
  case Error

def safeDivBy(n: Int): Result =
  n match
    case 0 => Error
    case _ => Ok(10 / n)
```

In questo caso è evidente dal tipo di ritorno di `safeDivBy` che questa può fallire restituendo un valore di tipo `Error`. La funzione non nasconde questo comportamento tramite il meccanismo delle eccezioni, lo rende invece evidente nel proprio tipo `Int => Result`. Il grande vantaggio di rendere esplicita la possibilità di fallimento nel tipo di ritorno sta nel fatto che il programmatore dovrà obbligatoriamente gestire anche il caso in cui la computazione fallisca o il compilatore solleverà un errore a tempo di compilazione:

```
def useSafeDivBy(n: Int): Int =
  // safeDivBy(n) + safeDivBy(n + 1) darebbe un errore
  // a tempo di compilazione dato che safeDivBy(x)
  // ha come tipo Result e non Int
  safeDivBy(n) match
    case Error       => 0
    case Ok(value1) => safeDivBy(n + 1) match
      case Error       => 0
      case Ok(value2) => value1 + value2
```

### Lettura e modifica di uno stato globale

Dal momento in cui una funzione legge una variabile globale perde la propria trasparenza referenziale: il suo risultato non dipende più dai soli valori passati in input ma da un nuovo input *nascosto*, lo stato a cui accede. La soluzione per

rendere esplicita questa dipendenza è passare come parametri di tutti gli elementi necessari al funzionamento della funzione, senza affidarsi alla definizione in uno *scope* esterno di variabili globali a cui accedere implicitamente.

Questa semplice trasformazione permette di rimuovere il side effect che consiste nella lettura di uno stato globale. Sfortunatamente, non è sufficiente per modellare anche la *modifica* di una variabile globale. In questo caso può essere utile tornare all'analogia con le funzioni matematiche. Prendiamo come esempio la funzione `f` definita in precedenza:

```
var counter = 0
def f(x: Int): Int =
  counter = counter + 1
  x + counter
```

Questa prende in input un numero, ha il side effect di incrementare un contatore globale e poi ne legge il valore per aggiungerlo all'input. Sebbene il tipo di `f` sia `Int => Int`, a causa dei suoi side effect questa funzione non può essere modellata come una funzione matematica $f : \mathbb{Z} \to \mathbb{Z}$: a parità di input non darà sempre lo stesso output.

Dato che la funzione necessita di leggere una variabile definita esternamente, questa dovrà essergli passata in come input esplicitamente:

```
def wrongF(x: Int, counter: Int): Int =
  // Come modificare lo stato di current?
  val newCounterState = counter + 1
  x + newCounterState
```

Rimane il problema di come poter modellare la modifica dello stato globale in modo che tale effetto si ripercuota anche su chiamate successive. Infatti, il side effect di aumentare il valore del contatore fa parte della logica applicativa di `f` e rimuoverlo comporterebbe un cambiamento nella sua semantica.

La soluzione consiste nel trasformare la funzione in modo tale che restituisca il valore del nuovo stato così che possa essere utilizzato per chiamate successive:

```
def betterF(x: Int, counter: Int): (Int, Int) =
  val newCounter = counter + 1
  val result = x + newCounter
  (result, newCounter)
```

Quindi lo stato dovrà essere passato in maniera esplicita da una chiamata a funzione alla successiva:

```
def useBetterF: (Int, Int) =
  val startingCounter = 1
  val (result1, counter1) = betterF(1, startingCounter)
  // Il nuovo stato counter1 viene passato alla seconda
  // chiamata di betterF
  val (result2, counter2) = betterF(1, counter1)
```

```
  // Il nuovo stato counter2 viene passato alla terza
  // chiamata di betterF
  val (result3, finalCounter) = betterF(1, counter2)
  (result1 + result2 + result3, finalCounter)
```

La funzione non solo prende in input lo stato a cui deve accedere ma restituisce in output la nuova versione dello stato modificato[2], a questo punto è nuovamente possibile modellarla come una funzione matematica $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$ che, per ogni coppia di valori ricevuti in input, darà sempre lo stesso risultato.

In questo modo diventa più facile testare il comportamento della funzione:

```
def testBetterF(): Unit =
  val counter = 0
  val (result, newCounter) = betterF(1, counter)
  result shouldBe 2
  newCounter shouldBe 1
```

Il test è completamente autonomo, non sono più necessarie operazioni preliminari di preparazione dello stato: l'output di `betterF`, infatti, dipende unicamente dai valori passati in input e non da una qualche variabile globale che potrebbe essere utilizzata e modificata anche da altri test.

### 2.1.2 Svantaggi della modellazione esplicita

Con le trasformazioni elencate in precedenza è possibile mantenere la trasparenza referenziale delle funzioni modellandone i side effect in maniera esplicita. Questo ha il vantaggio di aiutare il programmatore nella composizione delle funzioni rendendo chiaro, a tempo di compilazione, quale potrebbe essere il loro comportamento.

Nonostante ciò, come forse si è già potuto intuire da alcuni degli esempi riportati, questo approccio rende necessario lo scrivere codice spesso molto più verboso. In seguito sono riportati due esempi di questo problema e delle soluzioni ad hoc che possono essere adottate per porvi rimedio.

#### Gestione del fallimento di una funzione

Consideriamo la seguente funzione:

```
def halve(n: Int): Option[Int] =
  n % 2 match
    case 0 => Some(n / 2)
    case _ => None
```

[2]In un linguaggio con strutture dati mutabili come Scala la funzione potrebbe anche non restituire la nuova versione dello stato ma modificare lo stato ricevuto come argomento per riferimento. Tuttavia, come descritto in precedenza anche la modifica degli argomenti è un side effect; questo approccio non risolverebbe il problema del poter tracciare esplicitamente gli effetti di una funzione.

Nel caso in cui il numero passato come input sia pari ne restituirà la metà, altrimenti fallirà (il side effect del fallimento è reso esplicito tramite l'uso del tipo `Option`). Per realizzare una funzione analoga che, se l'input è divisibile per 8 ne restituisca il risultato della divisione, un programmatore potrebbe comporre insieme più chiamate a `halve`:

```
def eighth(n: Int): Option[Int] =
  halve(n) match
    case None       => None
    case Some(half) => halve(half) match
      case None         => None
      case Some(fourth) => halve(fourth)
```

Nonostante la semplicità della funzione, si può osservare come il dover gestire in maniera esplicita il fallimento di ogni chiamata ad `halve` renda il codice più verboso e meno leggibile. Nella funzione si mescolano la logica applicativa – il dover dividere ripetutamente per due – e la gestione del possibile fallimento della divisione intera – il pattern matching sul risultato.

Il problema può essere risolto osservando una struttura comune a tutti i passaggi: se uno qualunque dei risultati intermedi restituisce `None` allora la funzione fallisce immediatamente restituendo `None`, come se si fosse verificata un'eccezione; altrimenti, si prosegue continuando a dividere il valore ottenuto. Questo comportamento può essere fattorizzato in un'apposita funzione[3]:

```
extension [A](a: Option[A])
  def andThen[B](f: A => Option[B]): Option[B] =
    a match
      case None         => None
      case Some(result) => f(result)
```

che permetterà di scrivere la funzione `eighth` in maniera molto più chiara:

```
def eighth(n: Int): Option[Int] =
  halve(n).andThen(halve).andThen(halve)
```

La funzione è solo interessata dalla logica applicativa: dividere tre volte il valore in input. La gestione del fallimento è delegata ad `andThen`[4] che, in caso di fallimento, restituisce immediatamente `None`.

### Gestione della lettura e modifica di uno stato globale

Come già discusso, è possibile modificare una funzione che accede e modifica uno stato globale in una funzione pura; in modo che riceva lo stato globale come input

[3] In Scala per poter utilizzare `andThen` come una funzione infissa questa deve essere dichiarata come *extension method* [45] degli oggetti di tipo `Option`.

[4] In Scala è già definito per `Option` il metodo `flatMap` con lo stesso comportamento di `andThen` mostrato nell'esempio. Per questi esempi si preferisce utilizzare `andThen` in quanto rende più chiaro il ruolo della funzione.

e ne restituisca la nuova versione in output. Lo svantaggio di tale approccio sta nel fatto che il programmatore dovrà gestire manualmente il passaggio dello stato fra le diverse chiamate; ciò, oltre ad aumentare il *boilerplate*, rende più probabile compiere errori come dimenticare di passare lo stato aggiornato a una chiamata successiva:

```
...
val (res1, state1) = statefulFunction1(initialState)
val (res2, state2) = statefulFunction2(res1, state1)
val (res3, state3) = statefulFunction3(res2, state2)
statefulFunction4(res3, state2)
// bug: statefulFunction4 ha ricevuto come input lo
// stato precedente a quello aggiornato da
// statefulFunction3!
...
```

Anche in questo caso si può ottenere una soluzione che permetta di separare la logica applicativa dalla gestione manuale del passaggio dello stato fra un chiamata e la successiva:

```
extension [A, S](f: S => (A, S))
  def andThen[B](g: A => S => (B, S)): S => (B, S) =
    state0 =>
      val (result, state1) = f(state0)
      g(result)(state1)
```

Grazie a questa funzione è possibile riscrivere la funzione dell'esempio precedente in maniera più chiara:

```
...
statefulFunction1
  .andThen(statefulFunction2)
  .andThen(statefulFunction3)
  .andThen(statefulFunction4)
  .apply(initialState)
...
```

La gestione del *threading* dello stato fra una chiamata e la successiva è delegato ad `andThen`; in questo modo è possibile indicare la sequenza di operazioni che si vogliono compiere e lasciare che il boilerplate relativo al passaggio dello stato fra una chiamata e la successiva venga gestito automaticamente.

## 2.2 Modellazione degli effetti tramite monadi

Osservando le soluzioni adottate nella sezione precedente si può osservare come a queste sottende un meccanismo comune. Infatti in entrambi i casi è stata definita una funzione `andThen` per permettere di combinare in sequenza più operazioni con

side effect. Il risultato ottenuto è una descrizione dichiarativa della sequenza di operazioni da svolgere; il particolare meccanismo con cui le operazioni vengono concatenate – gestione del fallimento prematuro, o passaggio implicito dello stato ad ogni passaggio – viene delegato alla funzione `andThen`.

Questo meccanismo può essere catturato dall'astrazione delle monadi.

### 2.2.1 Cos'è una monade?

Il concetto di monade, nato nell'ambito della teoria delle categorie [8], venne utilizzato da Eugenio Moggi come mezzo per strutturare la semantica denotazionale di aspetti di un programma come lo stato mutabile, la gestione delle eccezioni e delle continuazioni [9]. Fu poi Philip Wadler, ispirato dal lavoro di Moggi e Michael Spivey [10], a intuire che questa stessa tecnica potesse essere sfruttata direttamente per *strutturare* un programma funzionale – non solo per descriverne la semantica come fatto da Moggi [11, 12].

Una monade è una tripla `(M, pure, flatMap)` dove:

- `M` è un costruttore di tipi; ovvero prende in input un tipo `A` e restituisce un tipo `M[A]`. Un valore di tipo `M[A]` può essere interpretato come una computazione che restituisce un valore di tipo `A` e può avere un qualche side effect
- `pure` è una funzione polimorfa con tipo `A => M[A]`[5]
- `flatMap` è una funzione polimorfa con tipo `(M[A], A => M[B]) => M[B]`[6]; rappresenta la combinazione in sequenza di due computazioni che possono presentare side effect

Inoltre è richiesto che valgano le seguenti *leggi monadiche*:

- (Identità sinistra) `pure(a) >>= f = f(a)`
- (Identità destra) `m >>= pure = m`
- (Associatività) `(m >>= f) >>= g = m >>= (x => f(x) >>= g)`

Le prime due leggi servono a garantire che `pure` sia l'elemento neutro per l'operazione di concatenazione `flatMap`: `pure` può essere quindi visto come l'operazione che trasforma un valore di tipo `A` in un valore di tipo `M[A]` senza compiere alcun side effect. La terza legge, garantisce che `flatMap` sia associativa, dunque scrivere `m >>= f >>= g` è equivalente a scrivere `(m >>= f) >>= g` o `m >>= (x => f(x) >>= g)`.

[5] `pure` è spesso anche indicato come `return`.

[6] `flatMap` è anche indicato come `bind` o `>>=`. Nella sua versione `>>=` viene generalmente utilizzato come operatore binario infisso: vale a dire che `m >>= f` è equivalente a indicare `flatMap(m, f)`.

### 2.2.2 Encoding di una monade

È possibile esprimere le soluzioni ad hoc adottate alla Sezione 2.1.2 in termini del concetto di monade, garantendo un'interfaccia uniforme per diversi side effect. Tuttavia, prima di poter generalizzare gli esempi precedenti è necessario capire come il concetto di monade possa essere implementato in un linguaggio di programmazione.

#### Encoding in Haskell

Come descritto in precedenza una monade è composta da tre elementi fondamentali: un costruttore di tipo e due funzioni `return` e `>>=`. In Haskell è possibile esprimere direttamente tale concetto tramite l'uso di una *type class*[7]:

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

La dichiarazione di una type class può essere interpretata come un predicato su un tipo o, come in questo caso, su un costruttore di tipi. Quindi, la precedente definizione può essere letta come: *"Un generico costruttore di tipi `m` è una monade se esistono due funzioni `return` con tipo `a -> m a` e `>>=` con tipo `m a -> (a -> m b) -> m b`"*. Dato un concreto costruttore di tipi si può istanziare la type class `Monad` fornendo l'implementazione delle funzioni che questa dichiara:

```
instance Monad Maybe where
  return = Just
  Nothing >>= f = Nothing
  Just x  >>= f = f x
```

Interpretando la type class come un predicato, definire un'istanza consiste nel provare che lo specifico tipo – in questo caso `Maybe` – soddisfa tale predicato e viene fornita come dimostrazione l'implementazione delle funzioni `return` e `>>=`.

#### Encoding in Scala

In Scala è possibile codificare una type class sfruttando il passaggio implicito di parametri [13], i *context bound* [46] e il supporto agli *higher-kinded type*.

Per definire una type class è innanzitutto necessario indicarne i metodi utilizzando un'interfaccia; nel caso della type class per le monadi:

```
trait Monad[M[_]]:
  def pure[A](a: A): M[A]
  extension [A](m: M[A]) def flatMap[B](f: A => M[B]): M[B]
```

Il *trait* deve essere generico su un costruttore di tipi `M[_]`, quindi è necessario che il linguaggio supporti gli higher-kinded type: vale a dire, dev'essere possibile

[7] Per maggiori dettagli sul meccanismo delle type class si rimanda all'Appendice B

poter indicare – com'è stato fatto in questo caso con `M` – costruttori di tipi generici come parametri di tipo.

Per istanziare una type class è necessario implementare un'istanza dell'interfaccia per uno specifico costruttore di tipi. In Scala è possibile fare ciò fornendo un'implementazione come istanza implicita [47]:

```
given Monad[Option] with
  def pure[A](a: A): Option[A] = Some(a)
  extension [A](m: Option[A])
    def flatMap[B](f: A => Option[B]): Option[B] =
      m match
        case None    => None
        case Some(a) => f(a)
```

Infine è possibile sfruttare il passaggio implicito di parametri per poter specificare che una porzione di codice necessita di utilizzare i metodi di una certa type class:

```
def useMonadInstance[M[_]](using instance: Monad[M]) =
  instance.pure("Hello, World!")
```

In questo caso la funzione richiede che le venga passata l'implementazione della type class di monade per il generico tipo `M` in modo da poterne utilizzare il metodo `pure`. Scala offre anche zucchero sintattico per poter esprimere in maniera concisa questo genere di argomenti impliciti: i context bound. Un context bound permette di specificare la necessità di avere un parametro implicito passato come argomento della funzione senza assegnargli un nome; il codice appena mostrato può essere riscritto come segue:

```
def useMonadInstance[M[_]: Monad] =
  summon[Monad[M]].pure("Hello, World!")
```

La notazione `M[_]: Monad` può quindi essere letta come *"Dato un generico costruttore di tipi `M` che sia una monade"*.

### 2.2.3 Esempi di monadi

Gli esempi mostrati alla Sezione 2.1.2 presentano una struttura comune: si definisce una struttura dati che descrive il risultato di un'operazione con il possibile side effect e si definisce poi una funzione – negli esempi chiamata `andThen` – che permette di combinare in sequenza passaggi intermedi.

Si può osservare come in entrambi i casi la funzione `andThen` abbia la stessa firma descritta per `flatMap`: infatti, i casi mostrati in precedenza non sono altro che esempi di monadi. In seguito viene formalizzata la definizione di monade per entrambi gli esempi riportando in aggiunta l'implementazione di `pure`. Inoltre viene mostrata la definizione di una monade banale che non esegue alcun side effect.

### La monade identità

La più semplice monade possibile è quella che non applica alcun side effect. Tale monade può essere definita come segue:

```
final case class Identity[A](a: A)

object Identity:
  given Monad[Identity] with
    def pure[A](a: A): Identity[A] = Identity(a)

    extension [A](m: Identity[A])
      def flatMap[B](f: A => Identity[B]): Identity[B] = f(m.a)
```

- `Identity` è il costruttore di tipi: preso un tipo `A` restituisce un tipo `Identity[A]` che rappresenta una computazione che produce un valore di tipo `A` senza attuare alcun side effect
- `pure` permette di trasformare un valore di tipo `A` in uno di tipo `Identity[A]`. In questo caso corrisponde alla funzione identità che non modifica il valore di tipo `A`
- `flatMap` permette di mettere in sequenza valori di tipo `Identity`; dato che la monade non introduce alcun side effect corrisponde all'applicazione di funzione

Una dimostrazione del rispetto delle leggi monadiche è riportata in Appendice A.1.

L'utilità di una monade che non introduce alcun side effect sarà resa evidente nel Capitolo 3.

### La monade Option

Per gestire il fallimento prematuro di una funzione era stato sfruttato il costruttore di tipi `Option` e la funzione `andThen` per concatenare in sequenza passaggi intermedi e propagare il fallimento. Si può mostrare come `Option` sia una monade implementando l'operazione `>>=` come `andThen`:

```
enum Option[+A]:
  case Some(a: A)
  case None

given Monad[Option] with
  def pure[A](a: A): Option[A] = Some(a)
  extension [A](m: Option[A])
    def flatMap[B](f: A => Option[B]): Option[B] =
      m match
        case None    => None
        case Some(a) => f(a)
```

- `Option` è il costruttore di tipi: preso un tipo `A`, restituisce un tipo `Option[A]` che rappresenta una computazione che può fallire o produrre un valore di tipo `A`
- `pure` permette di trasformare un valore di tipo `A` in uno di tipo `Option[A]` senza avere side effect. In questo caso il side effect sarebbe fallire restituendo `None`; quindi `pure(a)` restituisce `Some(a)`
- `flatMap` permette di concatenare in sequenza passaggi intermedi propagando il fallimento

Come descritto in precedenza, perché `Option` sia effettivamente una monade deve rispettare le tre leggi monadiche; una dimostrazione è riportata all'Appendice A.2.

### La monade State

È possibile definire un'istanza di monade anche per l'esempio dello stato globale mutabile mostrato alla Sezione 2.1.2; per fare ciò può tornare utile definire prima un'apposita struttura che incapsula una funzione che prende in input lo stato e restituisce il risultato e lo stato aggiornato:

```
final case class State[S, A](runState: S => (A, S))
```

Un problema nel definire l'istanza di monade per `State` sta nel fatto che questo è un costruttore di tipi che accetta in input *due tipi* `S` e `A` – quello dello stato manipolato e quello del risultato – e restituisce un tipo `State[S, A]`. Per poter essere un costruttore valido secondo la definizione di monade deve prendere in input un solo tipo; per ovviare a tale problema si può fissare uno dei due tipi e lasciare l'altro libero: in questo caso si è scelto di fissare il tipo dello stato `S` e lasciare libero di variare il tipo del risultato[8]:

```
given [S]: Monad[State[S, _]] with
  def pure[A](a: A): State[S, A] = State(s => (a, s))
  extension [A](m: State[S, A])
    def flatMap[B](f: A => State[S, B]): State[S, B] =
      State(state0 =>
        val (result1, state1) = m.runState(state0)
        f(result1).runState(state1)
      )
```

[8] In Scala 3 si può ottenere questa applicazione parziale del costruttore di tipi utilizzando il meccanismo delle *type lambda* [48]; quindi per fissare il tipo `S` e lasciare libero `A` in `State` si può utilizzare la seguente sintassi: `[A] =>> State[S, A]`. Nel codice riportato nell'esempio viene utilizzata una sintassi abbreviata che utilizza il carattere `_` come segnaposto nella lambda a livello di tipi in maniera analoga a come viene utilizzato per le lambda a livello di termini [49]. Tale sintassi non è ancora stata adottata come default in Scala 3 ma verrà introdotta in futuro, al momento è disponibile utilizzando l'estensione del compilatore `"-Ykind-projector:underscores"` [50].

- `State[S, _]` è il costruttore di tipi: preso un tipo `A` restituisce un tipo `State[S, A]` che rappresenta una computazione che può modificare uno stato globale di tipo `S` e restituisce un valore di tipo `A`
- `pure` permette di trasformare un valore di tipo `A` in uno di tipo `State[S, A]` senza avere side effect. Il side effect sarebbe modificare lo stato globale, quindi in questo caso lo stato globale viene restituito inalterato e il risultato è il valore passato in input
- `flatMap` permette di concatenare in sequenza operazioni che operano su uno stato globale mutabile di tipo `S` e passa in automatico la sua versione aggiornata da un una chiamata alla successiva

Come mostrato nell'Appendice A.3 l'istanza di monade per `State` rispetta le tre leggi monadiche.

Nel caso della monade `State` alcune operazioni di base sono quelle che permettono di leggere o modificare il valore dello stato mutabile:

```
def get[S]: State[S, S] = State(s => (s, s))
def set[S](s: S): State[S, Unit] = State(_ => ((), s))
```

Queste funzioni possono essere utilizzate come base per ottenere operazioni più complesse. Un esempio più articolato che combina queste operazioni è dato dalla seguente funzione `incrementCounter`: questa ha come tipo `State[Int, String]` dunque può accedere a uno stato mutabile di tipo `Int` e produce come risultato una stringa:

```
def incrementCounter: State[Int, String] =
  get.flatMap(counter =>
    set(counter + 1).flatMap(_ =>
      get.flatMap(newCounter =>
        Monad.pure(f"counter is: $newCounter")
      )
    )
  )
```

Come prima azione accede allo stato globale con `get`, ne aumenta il valore con `set` e infine legge nuovamente il valore dopo averlo modificato con un'ultimo `get`. Il risultato è una stringa contenente il nuovo valore appena letto. Si noti come, per mettere in sequenza le operazioni di lettura e scrittura sia necessario utilizzare `flatMap`. Per poter eseguire la computazione con uno specifico stato iniziale sarà sufficiente utilizzare il metodo `runState`:

```
incrementCounter.runState(0) // -> ("counter is: 1", 1)
incrementCounter.runState(2) // -> ("counter is: 3", 3)
```

### 2.2.4 Zucchero sintattico per codice monadico

Osservando l'esempio precedente è possibile notare come la messa in sequenza delle operazioni tramite l'uso di `flatMap` può rendere il codice più complesso da leggere:

infatti, ad ogni concatenazione successiva aumenta il livello di annidamento del codice portando alla cosiddetta *pyramid of doom*. Scrivere codice di questo genere diventerebbe impraticabile molto rapidamente anche per brevi sequenze di funzioni concatenate con `flatMap`.

Per questo motivo, linguaggi come Haskell e Scala forniscono dello zucchero sintattico che permette di scrivere codice monadico in modo più leggibile e con un aspetto più "imperativo".

### ***For comprehension*** **in Scala**

Per risolvere questo problema, Scala fornisce la *for comprehension* [51]; per esempio la funzione `incrementCounter` mostrata in precedenza può essere scritta in modo equivalente come segue:

```
def incrementCounterForComprehension: State[Int, String] =
  for
    counter    <- State.get
    _          <- State.set(counter + 1)
    newCounter <- State.get
  yield f"counter is: $newCounter"
```

Il codice appare come una sequenza di operazioni imperative. In realtà, il compilatore Scala traduce il codice in una sequenza di chiamate a `flatMap` e `pure`. Le regole adottate per il *desugaring* possono essere descritte ricorsivamente come segue[9]:

```
for {name <- expr} yield res = expr.map(name => res)

for {name <- expr; exprs} yield res =
  expr.flatMap(name => for {exprs} yield res)
```

La funzione `map` utilizzata nel desugaring di una singola espressione è semanticamente equivalente, per una monade, alla seguente composizione di di `flatMap` e `pure`: `m.map(f) = m.flatMap(x => pure(f(x)))`.

### ***Do notation*** **in Haskell**

Haskell adotta una soluzione analoga fornendo la cosiddetta *do notation*; il precedente codice Scala preso ad esempio potrebbe essere implementato analogamente in Haskell come segue:

```
incrementCounter = do
  counter <- get
  set (counter + 1)
```

[9]In realtà nella *for comprehension* sarebbe possibile utilizzare anche altre espressioni che non siano nella forma `name <- expr` ma, per semplicità, non verranno considerate in questa sezione.

```
  newCounter <- get
  return ("counter is:" ++ show newCounter)
```

Anche in questo caso, il compilatore traduce il codice in una serie di chiamate a `>>=` e `return`. In particolare, le regole per la traduzione sono le seguenti:

```
do { expr } = expr

do { name <- expr; exprs } =
  expr >>= \name -> do { exprs }

do { expr; exprs } = expr >>= \_ -> do { exprs }
```

Quindi, la forma senza zucchero sintattico di `incrementCounter` sarebbe:

```
incrementCounter =
  get >>= (\counter ->
    set (counter + 1) >>= (\_ ->
      get >>= (\newCounter ->
        return ("counter is:" ++ show newCounter))))
```

### 2.2.5 Vantaggi nell'uso delle monadi

#### Separazione del codice impuro dal codice puro

Un primo importante vantaggio sta nella possibilità di esprimere a livello di type system quali funzioni presentano side effect e quali no. Questo è un aiuto fondamentale per il programmatore: infatti, per capire se una funzione è pura è sufficiente analizzarne il tipo, senza dover cercare di capirlo dal suo nome – che potrebbe non rispecchiare l'effettivo comportamento della funzione – o ispezionandone il corpo. Consideriamo come esempio le seguenti funzioni:

```
def incrementCounter: State[Int, ()] = ...
def first[A](xs: List[A]): Option[A] = ...
def double(n: Int): Int = ...
```

Senza bisogno di conoscerne le implementazioni si può capire che la prima funzione può modificare uno stato mutabile di tipo intero e che la seconda funzione può fallire non producendo alcun valore. Inoltre è possibile capire che la terza funzione non ha side effect: non potrà fallire, modificare uno stato globale o effettuare operazioni di input o output; il suo output sarà determinato unicamente dal valore dei suoi parametri[10].

[10] In realtà, dato che in Scala è comunque possibile scrivere codice impuro la funzione `double` potrebbe avere side effect; quanto detto vale sotto l'assunzione che il programmatore stia modellando esplicitamente i side effect tramite l'approccio descritto. In altri linguaggi come Haskell, invece, è il compilatore stesso a fare in modo che questa regola venga rispettata rendendo impossibile lo scrivere funzioni che non siano pure. Perciò, si avrebbe la certezza che una funzione come `double` non possa avere side effect.

### Side effect come entità di prima classe del linguaggio

Utilizzare il concetto di monade come astrazione unificante delle diverse tipologie di side effect ha un ulteriore vantaggio: le azioni con side effect sono valori di prima classe che possono essere combinati in maniera modulare e astraendo dallo specifico tipo di effetti.

È possibile definire l'equivalente di strutture di controllo imperative tramite funzioni generiche; per poter ripetere più volte gli effetti di un'azione è possibile implementare una funzione come segue:

```
extension [A, M[_]: Monad](m: M[A])
  def >>[B](other: M[B]) = m.flatMap(_ => other)

def repeat[M[_]: Monad, A]
  (times: Int)
  (action: M[A]): M[Unit] =
  times match
    case 0 => pure(())
    case n => action >> repeat(n-1)(action)
```

Se invece si volesse implementare una versione del ciclo `for` che permetta di ripetere un qualunque side effect per tutti gli elementi di una lista si potrebbe implementare la seguente funzione:

```
def forLoop[A, M[_]: Monad]
  (as: List[A])
  (f: A => M[Unit]): M[Unit] =
  as match
    case Nil      => pure(())
    case a :: as => f(a) >> forLoop(as)(f)
```

`forLoop` prende in input una lista e una funzione che, dato un elemento della lista, determina l'azione da intraprendere; il risultato sarà un'unica computazione che esegue tutti i side effect dati dall'applicazione della funzione a ciascun elemento della lista. Poiché la funzione è generica sul tipo della monade è possibile utilizzarla per qualsiasi tipo di side effect! Questa è una tecnica molto potente che lascia al programmatore la libertà di inventare le proprie strutture di controllo senza essere doversi limitare a quelle predefinite dal linguaggio [14].

## 2.3 Input e output puri

Nelle precedenti sezioni è stato mostrato come sia possibile "simulare" la presenza di side effect – come eccezioni e modifica di uno stato mutabile – con opportune modifiche al tipo delle funzioni in modo da rendere esplicito il fatto che queste possano avere side effect. Inoltre, è stato evidenziato come il concetto di monade

permetta di fornire un meccanismo comune per la modellazione e messa in sequenza di tali side effect.

Tuttavia, fino ad ora è stato tralasciato un side effect fondamentale: l'esecuzione di input e output. Chiaramente una funzione Scala potrebbe effettuare input e output semplicemente utilizzando le funzioni standard fornite dal linguaggio:

```
def addTo(x: Int): Int =
  val y = scala.io.StdIn.readInt() // side effect!
  x + y
```

Tuttavia, dalla sola analisi del tipo della funzione `sum : Int => Int` non è possibile capire se questa interagirà con il mondo esterno o meno.

Per comprendere come sia possibile tracciare tale side effect a livello di tipi verrà preso come esempio paradigmatico il linguaggio Haskell; successivamente verrà mostrato come le stesse intuizioni possano essere applicate in Scala.

### 2.3.1 Modello di valutazione *lazy*

Haskell è un linguaggio funzionale con strategia di valutazione lazy (anche detta *call-by-need*): ciò significa che gli argomenti delle funzioni vengono valutati solo se strettamente necessario e non sono valutati prima di essere passati alla funzione. Si consideri per esempio la seguente funzione:

```
lazy :: Int -> Int -> Int
lazy x y = x * 3
```

Quando la funzione viene chiamata, anziché valutare i suoi argomenti prima di eseguire il corpo della funzione, vengono allocati due *thunk* che rappresentano le espressioni da valutare. Sarà poi il corpo della funzione, in base al bisogno, a stabilire di quali thunk forzare la valutazione. Nell'esempio specifico valutato solo il thunk del primo argomento. Si considerino le possibili chiamate alla funzione `lazy`:

```
lazy (2 + 3) (expensiveFunction 2)
lazy (2 + 3) (1 + undefined)
```

Nel primo esempio il thunk che rappresenta l'espressione `expensiveFunction 2` non verrà mai valutato; allo stesso modo nella seconda chiamata il thunk dell'espressione `1 + undefined`[11] non sarà valutato; il risultato sarà 15 in entrambi i casi.

Grazie a questa strategia di valutazione è possibile definire direttamente operatori con *short-circuiting* come `&&` e `||`:

[11]In Haskell il valore `undefined` è un valore speciale che comporta il crash dell'applicazione nel momento in cui viene valutato. In questo caso, dato che si trova in un thunk che verrà scartato senza essere valutato non verrà sollevata alcuna eccezione.

```
(&&) :: Bool -> Bool -> Bool
x && y = case x of
  True  -> y
  False -> False

(||) :: Bool -> Bool -> Bool
x || y = case x of
  True  -> True
  False -> y
```

Le due funzioni hanno una struttura simile: inizialmente viene forzata la valutazione del primo argomento tramite pattern matching sul suo valore. In seguito viene restituito il secondo argomento o un valore predefinito in base al risultato del pattern matching. In entrambi i casi non viene forzata la valutazione del secondo argomento attuando la logica di short-circuiting che ci si potrebbe aspettare dagli operatori logici `&&` e `||`: il risultato di `False && undefined` sarà `False` e il programma non terminerà con un'eccezione dato che l'espressione `undefined` non sarà valutata.

#### Incompatibilità di *laziness* e side effect

Un aspetto negativo della strategia lazy è che può diventare estremamente complesso capire l'ordine con il quale le espressioni vengono valutate. Infatti, come mostrato negli esempi precedenti la valutazione dei thunk viene forzata solo quando strettamente necessario. Per questo motivo sarebbe pressoché impossibile riuscire a compiere in una sequenza prevedibile i side effect delle funzioni [14].

È proprio questa caratteristica che ha fatto sì che Haskell rimanesse un linguaggio puro e ha portato all'invenzione dell'input e output monadico: "forse il più grande beneficio della laziness non è la laziness in sè, quanto il fatto che ci abbia forzato a rimanere puri, motivando così una grande quantità di lavoro sulle monadi"[12].

### 2.3.2 I/O monadico in Haskell

La soluzione adottata da Haskell per permettere di effettuare operazioni di I/O è quello di fornire un nuovo costruttore di tipi chiamato `IO` che sia una monade. Un valore di tipo `IO a` modella una computazione che produce un valore di tipo `a` e può avere il side effect di interagire con il sistema, per esempio effettuando operazioni di input o output.

Il programmatore potrà definire nuove operazioni combinando le funzioni di libreria fornite dal linguaggio sfruttando l'interfaccia delle monadi:

[12]Traduzione dal testo originale: "[...] perhaps the biggest single benefit of laziness is not laziness per se, but rather that laziness kept us pure, and thereby motivated a great deal of productive work on monads [...]" [15]

```
echo :: IO ()
echo = do
  line <- getLine
  putStrLn line
```

Partendo dal tipo della funzione si può comprendere come questa rappresenti una computazione che una volta eseguita restituisce un valore di tipo `()` e che può effettuare I/O. La funzione è implementata mettendo in sequenza due operazioni più semplici: prima viene letta una riga dallo *standard input* e il contenuto letto viene stampato sullo *standard output* tale e quale. È interessante osservare come la do notation nasconda l'applicazione di `>>=` e `pure` dando al codice un tipico aspetto imperativo. Nonostante l'apparente "imperatività" è fondamentale ricordare che i valori di tipo `IO` – come `getLine` e `putStrLn` – non sono funzioni che provocano side effect ma descrivono i side effect che devono avere luogo. L'equivalente versione senza zucchero sintattico è:

```
echo :: IO ()
echo = getLine >>= putStrLn
```

Il punto d'ingresso di ogni programma Haskell è la funzione `main`:

```
main :: IO ()
main = echo
```

Quindi un programma non è altro che una struttura dati immutabile che *descrive* la sequenza di operazioni che il *runtime system* del linguaggio deve eseguire a tempo d'esecuzione.

### Separazione di codice puro e impuro

Grazie all'approccio appena mostrato non è possibile mescolare inavvertitamente codice puro e codice con side effect – proprio come per i casi di eccezioni e stato mutabile mostrati in precedenza. Per esempio, consideriamo come potrebbe essere riscritta la funzione impura mostrata all'inizio della sezione; semplicemente leggere un valore intero da standard input non è possibile:

```
readInt :: IO Int
readInt = fmap read getLine

addTo :: Int -> Int
addTo x = let y = readInt in x + y
```

Il codice mostrato non compilerebbe in quanto `x` ha tipo `Int` mentre `readInt` è un valore di tipo `IO Int`: non è un valore intero bensì una computazione che produrrà un intero. Per far sì che `addTo` possa utilizzare una funzione impura come `readInt`, è necessario rendere esplicito a livello di tipi il fatto che anche `addTo` sia impura:

```
addTo :: Int -> IO Int
```

```
addTo x = do
  y <- getInt
  pure (x + y)
```

### Programmi come valori di prima classe

Come già descritto, un valore di tipo `IO a` non è altro che una struttura dati che descrive una sequenza di computazioni per produrre un valore di tipo `a`. Ciò permette di passare programmi come valori di prima classe e costruire una ricca serie di funzioni generiche che operano su programmi e producono nuovi programmi in output. Per esempio la funzione

```
forever :: IO a -> IO b
forever action = action >> forever action
```

prende in input un programma e restituisce un programma che lo esegue in loop all'infinito.

Un ulteriore esempio può essere la funzione `retry` definita come segue:

```
retry :: Int -> (a -> Bool) -> IO a -> IO (Maybe a)
retry 0 _ _ = pure Nothing
retry times shouldRetry action = do
  result <- action
  if shouldRetry result
    then retry (times - 1) shouldRetry action
    else pure (Just result)
```

Questa restituisce in output un programma che ripete fino a un massimo numero di volte un programma passato in input secondo una certa logica di ripetizione definita dal predicato `shouldRetry`. Queste funzioni possono essere combinate in programmi più complessi[13]:

```
URLToResource :: String -> IO (Response ByteString)
URLToResource url = httpLBS (fromString url)

shouldRetry :: Response a -> Bool
shouldRetry response =
  let statusCode = getResponseStatusCode response
   in statusCode `elem` [500, 503]

main :: IO ()
main = forever $ do
  let times = 10
  putStrLn "URL of the resource: "
  url <- getLine
  result <- retry times shouldRetry (URLToResource url)
  case result of
```

[13] Nell'esempio per effettuare le richieste a un server viene utilizzata la libreria *http-conduit* [52]

```
    Nothing -> putStrLn "Failed after 10 retries"
    Just _  -> putStrLn "Got a response"
```

Sfruttando la funzione `retry` è possibile definire un programma che ripete una richiesta HTTP fino a un massimo di 10 volte in caso di errore 500 o 503[14]. Utilizzando la funzione `forever` è possibile fare in modo che il programma continui a chiedere input al programmatore all'infinito.

Un ulteriore vantaggio dato dal fatto che `IO` è una monade sta nella possibilità di sfruttare codice generico sul tipo di monade:

```
void :: Monad m => m a -> m ()
void = m >>= (\_ -> pure ())

sequence :: Monad m => [m a] -> m [a]
sequence [] = pure []
sequence (m:ms) = do
  a  <- m
  as <- sequence ms
  pure (a:as)

main :: IO ()
main =
  let messages = ["message1", "message2", "message3"]
  in void (sequence (map putStrLn messages))
```

In realtà le stesse funzioni `forever` e `retry` possono essere definite in maniera generica rispetto al tipo di effetto in considerazione:

```
forever :: Monad m => m a -> m b
retry :: Monad m
  => Int
  -> (a -> Bool)
  -> m a
  -> m (Maybe a)
```

L'implementazione è tralasciata in quanto identica a quanto riportato nell'esempio di codice mostrato in precedenza: l'unico cambiamento necessario per rendere la funzione più generica è stato quello di sostituire nel tipo `IO` con una generica monade `m`.

### 2.3.3 I/O monadico in Scala

In Haskell la monade `IO` deve essere implementata come un tipo di dato opaco con un supporto speciale del compilatore. Haskell infatti è un linguaggio puro con una

[14] Ai fini dell'esempio la funzione `retry` è piuttosto semplice e non implementa logiche complesse per attendere prima di ripetere una richiesta evitando di oberare il server. Il package *retry* [53] implementa una funzione analoga a quella mostrata con la possibilità di specificare delle *policy* per stabilire la strategia con cui riprovare l'azione.

modalità di valutazione lazy che, come mostrato in precedenza, è incompatibile con la presenza di side effect. In Scala non è necessario un supporto diretto del compilatore in quanto è già possibile definire computazioni che svolgono side effect; perciò una semplice implementazione[15] della monade `IO` potrebbe essere la seguente:

```
final case class IO[A](unsafeRun: () => A)

object IO:
  given Monad[IO] with
    def pure[A](a: A): IO[A] = IO(() => a)
    extension [A](m: IO[A])
      def flatMap[B](f: A => IO[B]): IO[B] =
        IO(() => f(m.unsafeRun()).unsafeRun())
```

- `IO` è un costruttore di tipi che preso in input un valore di tipo `A` restituisce un valore di tipo `IO[A]`. Un valore con questo tipo rappresenta una computazione che, quando eseguita, può effettuare input o output e restituisce un valore di tipo `A`
- `pure` permette di trasformare un valore di tipo `A` in uno di tipo `IO[A]`. Non introduce alcun side effect e restituisce semplicemente il valore passato in input
- `flatMap` permette di concatenare in sequenza due operazioni che possono svolgere I/O. Il risultato sarà una singola computazione che esegue i side effect di ciascuna in sequenza

Le funzioni impure della libreria standard di Scala possono quindi essere espresse in termini di `IO`:

```
def putStrLn(s: String): IO[Unit] = IO(() => println(s))
def getLine: IO[String] = IO(() => scala.io.StdIn.readLine())
```

`putStrLn` e `getLine` sono valori di tipo `IO`; vale a dire strutture dati immutabili che descrivono come eseguire un side effect nel momento in cui la computazione verrà interpretata. Si noti la differenza fra una computazione `IO` e l'equivalente versione impura:

```
val res1 = putStrLn("Hello, World!")
// res1 : IO[Unit]
```

[15] L'implementazione vuole unicamente mostrare come si possa immaginare l'implementazione della monade `IO` in un linguaggio come Scala. Tuttavia, un'implementazione simile non è *stack safe*: interpretare un'azione `IO` ottenuta componendo molti blocchi di base può portare a un errore di *stack overflow*. Questo problema può essere risolto complicando l'implementazione della monade sfruttando una tecnica nota come *trampolining* [16] ed è l'approccio adottato da librerie come Cats Effect [54].

```
val res2 = println("Hello, World!")
// res2 : Unit
// -> "Hello, World!"
```

La valutazione di `putStrLn` produce un valore di tipo `IO[Unit]` senza alcun side effect; d'altro canto, la valutazione di `println` produce un valore di tipo `Unit` e ha il side effect di stampare in output la stringa.

L'unico modo per poter estrarre un valore dalla monade `IO` interpretandone il contenuto è tramite l'uso di `unsafeRun`:

```
val program = for
  _ <- putStrLn("Line 1")
  _ <- putStrLn("Line 2")
yield ()

program.unsafeRun()
// -> Line 1
// -> Line 2
```

Il metodo è stato chiamato `unsafeRun` a suggerire l'eccezionalità nella sua invocazione. Infatti, l'intero programma dovrebbe essere descritto all'interno della monade `IO` per poter poi essere eseguito nel main del programma con un'unica chiamata a `unsafeRun`.

I vantaggi ottenuti grazie a questo approccio sono gli stessi già descritti nella Sezione 2.3.2: è possibile separare chiaramente il codice impuro dal codice puro; i programmi diventano entità di prima classe, possono essere presi come input e restituiti in output e diventa possibile sfruttare tutte le funzioni generiche sul tipo di monade per comporre programmi complessi. Per esempio, come mostrato al Listato 2.1, il codice mostrato in precedenza per effettuare delle richieste a un server può essere scritto in Scala in maniera molto simile[16].

[16] Nell'esempio per effettuare le richieste a un server viene utilizzata la libreria *Requests-Scala* [55]

```
extension[A](m: IO[A])
  def forever[B]: IO[B] = m.flatMap(_ => m.forever)
  def retry(times: Int, shouldRetry: A => Boolean):
    IO[Option[A]] =
    times match
      case 0 => IO.pure(None)
      case n => m.flatMap{ result =>
        if shouldRetry(result)
        then m.retry(n-1, shouldRetry)
        else IO.pure(Some(result))
      }

def urlToResource(url: String): IO[Try[Response]] =
  IO(() => Try(requests.get(url)))

def shouldRetry(response: Try[Response]): Boolean =
  response match
    case Failure(exception: RequestFailedException) =>
      val statusCode = exception.response.statusCode
      List(500, 503).contains(statusCode)
    case _ => false

@main def main: Unit =
  val times = 10
  val step = for
    _ <- putStrLn("URL of the resource: ")
    url <- getLine
    result <- URLToResource(url)
                .retry(times, shouldRetry)
    _ <- result match
      case None => putStrLn("Failed after 10 retries")
      case Some(_) => putStrLn("Got a response")
  yield ()
  step.forever.unsafeRun()
```

Listato 2.1: Esempio di codice monadico che incapsula i side effect all'interno della monade IO per implementare una politica di *retry* per le richieste HTTP.

# Capitolo 3

# Stack di monadi

Nel capitolo precedente si è mostrato come sia possibile implementare semplici monadi per poter modellare la presenza di diversi side effect. Tuttavia, seguendo tale approccio non è evidente come sia possibile gestire contemporaneamente più side effect. Infatti ogni monade descritta permette di modellare un singolo side effect per volta: uno stato mutabile per `State`, la possibilità di fallimenti con `Option` e la capacità di effettuare input e output con `IO`.

In questo capitolo verrà introdotto il concetto di *monad transformer*: un meccanismo che permette di unire monadi elementari combinandone le caratteristiche.

## 3.1 Problemi nel comporre più side effect

Per rendere evidenti le problematiche dell'approccio mostrato nel capitolo precedente verrà preso ad esempio il *parsing* monadico di stringhe. Nell'implementare un'istanza di monade per un *parser* si potrà osservare come il codice sia essenzialmente una ripetizione di quello di alcune monadi di base già descritte.

### 3.1.1 Parsing monadico

Un parser può essere modellato come una funzione che consuma una stringa (o una sua porzione) producendo un risultato come un *Abstract Syntax Tree* (AST). Nel caso in cui la stringa non sia conforme al linguaggio descritto dal parser questo processo potrebbe fallire senza produrre alcun valore. Chiaramente la modellazione riportata è una semplificazione che cattura solamente l'essenza di un parser tralasciando altri aspetti come la possibilità di restituire più risultati – nel caso in cui la grammatica del parser sia ambigua – o di indicare la posizione nella stringa dove si è verificato un errore.

Nella definizione riportata, per quanto semplice, è possibile individuare due tipologie di side effect già incontrate in precedenza: la stringa su cui opera il parser è uno stato mutabile e l'operazione di parsing potrebbe fallire. Il modello di parser così descritto può essere rappresentato tramite un'apposita classe:

```
final case class Parser[A](parse: String => Option[(A, String)])
```

Un parser è modellato come una funzione che prende in input la stringa da consumare e restituisce il risultato del parsing che consiste nella coppia composta dalla porzione non consumata della stringa e dal risultato vero e proprio. Inoltre, dato che il processo di parsing potrebbe fallire, il risultato viene modellato come un valore opzionale.

Per esempio un parser che fallisce sempre può essere definito come:

```
def fail[A]: Parser[A] = Parser(_ => None)
```

Un parser che fallisce in caso di stringhe vuote e altrimenti ne consuma il primo carattere restituendolo come risultato può essere implementato come:

```
def char: Parser[Char] =
  Parser(string => string.headOption.map((_, string.tail)))
```

È possibile definire per la classe `Parser` un'istanza di monade che rispetta le leggi monadiche:

```
given Monad[Parser[_]] with
  def pure[A](a: A): Parser[A] = Parser(s => Some(a, s))
  extension [A](p: Parser[A])
    def flatMap[B](f: A => Parser[B]): Parser[B] =
      Parser(string0 =>
        p.parse(string0) match
          case None => None
          case Some((result, string1)) =>
            f(result).parse(string1)
      )
```

- `pure` è l'operazione di costruzione di un parser che non attua alcun side effect: restituisce il valore `a` senza consumare la stringa e senza fallire
- `flatMap` permette di concatenare fra loro dei parser ottenendo il seguente comportamento: se uno dei parser fallisce allora fallisce l'intera sequenza, altrimenti verranno usati i due parser in sequenza e il secondo consumerà la stringa rimanente non consumata dal primo

Grazie a quest'interfaccia monadica è possibile definire parser complessi in maniera modulare, combinando fra loro blocchi elementari tramite l'uso di `flatMap` [17]. Per esempio un parser che consuma esattamente due caratteri dalla testa di una stringa può essere definito come:

```
def consumeTwo: Parser[(Char, Char)] =
  for
    c1 <- char
    c2 <- char
  yield (c1, c2)
```

Per combinare fra loro parser non è necessario conoscere i dettagli implementativi della classe `Parser` ma è sufficiente sapere che questo è una monade. In questo modo se l'implementazione di `Parser` dovesse cambiare – come verrà mostrato in seguito – non sarà necessario riscrivere i parser ottenuti combinando elementi di base in questo modo.

Inoltre, poiché `Parser` è una monade, è possibile sfruttare tutte le funzioni generiche definite in precedenza per le monadi. L'esempio precedente può essere generalizzato ad un parser che consuma esattamente $n$ caratteri dalla testa di una stringa:

```
def consumeN(n: Int): Parser[List[Char]] =
  List.fill(n)(char).sequence
```

### 3.1.2 Duplicazione di codice

Osservando la definizione di `Parser` e la sua istanza di monade si può notare che questa ha una struttura simile a quella di `State` e `Option` viste al capitolo precedente:

```
final case class Parser[A](
  parse: String => Option[(A, String)])
final case class State[S, A](
  run:   S      =>        (A, S))
```

Ciò non dovrebbe stupire: come descritto in precedenza, `Parser` gestisce uno stato mutabile e può fallire nel processo di parsing. Dovendo ottenere questi effetti la struttura della monade `Parser` presenterà delle similarità con quella di `State` e `Option`. Ciò è ancora più evidente se si osservano le implementazioni di `pure` e `flatMap`. La funzione `pure` non fa altro che applicare l'effetto neutro sia di `State` che di `Option`:

```
def pure[A](a: A): Option[A]  = Some(a)
def pure[A](a: A): State[S, A] = State(s => (a, s))
def pure[A](a: A): Parser[A]  = Parser(s => Some((a, s)))
```

Anche nel caso di `flatMap` è presente una ripetizione della logica di base che contraddistingue `State` e `Option`:

```
def flatMap[B](f: A => Parser[B]): Parser[B] =
  Parser(string0 =>
    p.parse(string0) match
      case None => None
      case Some((result, string1)) =>
```

```
        f(result).parse(string1)
  )
```

Il pattern matching più esterno attua la logica di short-circuiting della monade `Option`, restituendo il valore `None` nel caso in cui il parsing fallisca. Mentre la gestione dello stato – in questo caso il passaggio della nuova stringa `string1` al secondo parser – è analoga al threading effettuato dalla monade `State`.

L'implementazione di `Parser` presenta quindi una significativa duplicazione di codice: l'istanza di monade è essenzialmente ottenuta combinando le due implementazioni di `State` e `Option`. Il problema scaturisce dalla necessità di dover combinare più side effect in un'unica monade andando a replicare la struttura delle monadi di base che forniscono gli effetti desiderati.

Per rendere ancora più evidente il problema si immagini di dover estendere il parser aggiungendo la possibilità di effettuare *logging* su standard output. In questo caso la monade `Parser` dovrebbe fornire anche le funzionalità della monade `IO`:

```
final case class Parser[A](
  parse: String => (() => Option[(A, String)]))

def flatMap[B](f: A => Parser[B]): Parser[B] =
  Parser(string0 =>
    val unsafeRun1 = p.parse(string0)
    unsafeRun1() match
      case None => None
      case Some((result, string1)) =>
        val unsafeRun2 = f(result).parse(string1)
        unsafeRun2())
```

L'implementazione di `flatMap` è ulteriormente complicata dalla necessità di dover gestire l'esecuzione delle computazioni con side effect; inoltre, si può nuovamente notare come l'implementazione consiste nella combinazione delle implementazioni delle tre monadi viste in precedenza: `State`, `IO` e `Option`.

Una soluzione ideale consentirebbe di definire `Parser` come una combinazione di monadi elementari, ciascuna delle quali fornisce la possibilità di gestire uno specifico effetto senza doverne implementare nuovamente la logica. Per risolvere tale problema è possibile sfruttare il concetto di monad transformer che permette di comporre in maniera modulare più monadi per gestire molteplici effetti contemporaneamente.

## 3.2 Monad transformer

Un monad transformer è una coppia (`T`, `lift`) dove:

- `T` è un costruttore di tipi che prende in input un costruttore di tipi `M`, un tipo `A` e restituisce un tipo `T[M, A]`

- Se `M` è una monade, allora `T[M, _]` è una monade
- `lift` è una funzione polimorfa con tipo `M[A] => T[M, A]` (dove `M` è una monade). Questa funzione serve a portare nella monade `T[M, _]` valori che si trovano nella monade `M`

Inoltre è richiesto che valgano alcune leggi[1]:

- `lift(pure`$_m$` x) = pure`$_{tm}$`(x)`
- `lift(m >>=`$_m$` f) = lift(m) >>=`$_{tm}$` (x => lift(f(x)))`

La prima legge serve a garantire che `lift` sia l'operazione neutra: un valore senza side effect nella monade `M` – indicato come `pure`$_m$`(x)` – non avrà alcun side effect se portato nella monade `T[M, _]` tramite l'operazione di *lifting*. La seconda legge indica che effettuare il lifting di una sequenza di operazioni – vale a dire `lift(m >>=`$_m$` f)` – è equivalente a effettuare il lifting delle singole operazioni e poi metterle in sequenza all'interno della monade `T[M, _]`.

È possibile interpretare un monad transformer `T` come una monade di ordine superiore: ovvero una monade parametrizzata su una generica monade `M`. L'idea alla base di un *transformer* è che questo permette di arricchire la monade parametro aggiungendovi ulteriori effetti [18].

### 3.2.1 Encoding di un monad transformer

L'*encoding* di un transformer può essere effettuato in maniera analoga a quanto mostrato per le monadi: è sufficiente definire un costruttore di tipi `T` che prenda in input un costruttore di tipi `M` e un tipo `A`. Inoltre è possibile definire tramite il meccanismo delle type class la funzione `lift` che ha tipo `M[A] -> T[M, A]`.

#### Encoding in Haskell

Secondo la definizione fornita un monad transformer deve possedere una funzione in grado di portare un valore da una generica monade `m` alla monade composta `t m`. Questo vincolo può essere espresso in Haskell tramite la seguente definizione:

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

Il *constraint* `Monad m =>` indica la necessità che `m` sia una monade perché la definizione sia valida.

Per poter istanziare un transformer sarà sufficiente definire un'implementazione per la funzione `lift` e per l'interfaccia di monade:

[1]Per chiarezza vengono riportate a pedice le indicazioni su quale monade appartengano le operazioni `pure` e `>>=`: `m` a pedice indica che l'operazione appartiene alla monade `M`, mentre `tm` a pedice indica che l'operazione fa riferimento alla monade `T[M, _]`.

```
data T = ...

instance MonadTransformer T where
  lift = ...

instance Monad m => Monad (T m) where
  return = ...
  (>>=) = ...
```

Sfruttando l'interpretazione delle type class come predicati, la seconda istanza equivale a provare che `T m` è una monade fornendo come dimostrazione le definizioni delle funzioni `return` e `>>=`. La differenza rispetto a quanto visto per le monadi è che per poter effettuare questa dimostrazione (e poter implementare i metodi richiesti) si assume che il generico tipo `m` sia una monade. Questo vincolo è espresso tramite il constraint `Monad m =>` nella definizione dell'istanza.

Imporre tale limitazione non è in contrasto con la definizione fornita precedentemente di monad transformer: infatti, si è specificato che `t m` deve essere una monade qualora anche `m` lo sia.

### Encoding in Scala

L'encoding in Scala è analogo a quello in Haskell:

```
trait MonadTransformer[T[_[_], _]]:
  def lift[M[_]: Monad, A](m: M[A]): T[M, A]
```

L'istanza per uno specifico tipo `T` sarà definita fornendo delle *given instance* per le type class di monade e transformer:

```
final case class T[M[_], A]()

given MonadTransformer[T] with
  def lift[M[_]: Monad, A](m: M[A]): T[M, A] = ???

given transformerInstance[M[_]: Monad]: Monad[T[M, _]] with
  def pure[A](a: A): T[M, A] = ???
  extension [A](t: T[M, A])
    def flatMap[B](f: A => T[M, B]): T[M, B] = ???
```

Inoltre, per comodità, è possibile definire `lift` come extension method applicabile su una qualunque monade; questa definizione verrà riutilizzata spesso nei prossimi esempi:

```
extension [A, M[_]: Monad](m: M[A])
  def lift[T[_[_], _]: MonadTransformer]: T[M, A] =
    summon[MonadTransformer[T]].lift(m)
```

### 3.2.2 Esempi di monad transformer

#### Il transformer `OptionT`

È possibile definire un transformer che permette di arricchire una monade con il side effect del fallimento prematuro della computazione:

```
final case class OptionT[M[_], A](runOptionT: M[Option[A]])
```

`OptionT` inserisce all'interno della monade `M` un valore di tipo `Option[A]`: graficamente l'applicazione del transformer può essere visualizzata come mostrato in Figura 3.1.

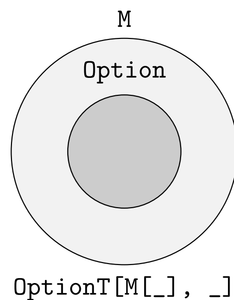


Figura 3.1: Rappresentazione grafica dell'applicazione del transformer `OptionT` a una generica monade `M`

È possibile implementare un'istanza di monad transformer per `OptionT`:

```
given MonadTransformer[OptionT] with
  def lift[M[_]: Monad, A](m: M[A]): OptionT[M, A] =
    OptionT(m.map(Some(_)))
```

Per rispettare le leggi dei monad transformer la funzione `lift` non aggiunge alcun side effect limitandosi a inserire il valore contenuto nella monade `M` all'interno di `Some`; questo non comporterà il fallimento con short-circuiting della computazione risultante.

Inoltre, se `M` è una monade, allora `OptionT[M, _]` è una monade:

```
given optionTMonad[M[_]: Monad]: Monad[OptionT[M, _]] with
  def pure[A](a: A): OptionT[M, A] =
    OptionT(Monad.pure(Some(a)))

  extension [A](t: OptionT[M, A])
    def flatMap[B](f: A => OptionT[M, B]): OptionT[M, B] =
      OptionT(
        t.runOptionT.flatMap(option =>
          option match
            case Some(a) => f(a).runOptionT
            case None    => Monad.pure(None)
        )
```

```
    )
```

- `pure` inserisce un valore `a` nella monade `OptionT` senza alcun side effect: innanzitutto il valore `a` viene inserito all'interno di `Some` – l'elemento che non comporta il side effect del fallimento. In seguito viene sfruttato il fatto che `M` sia una monade e si inserisce `Some(a)` al suo interno senza side effect utilizzando la funzione `pure`
- `flatMap` permette di mettere in sequenza computazioni che possono fallire prematuramente mantenendo anche gli effetti della monade parametro `M`: in caso di fallimento questo viene propagato all'interno della monade `M`

Grazie a `OptionT` è possibile arricchire una qualunque monade preesistente con il side effect del fallimento prematuro della computazione. Per esempio una computazione che effettua input e output e che può fallire prematuramente può essere espressa come:

```
def fail[M[_]: Monad, A]: OptionT[M, A] =
  OptionT(Monad.pure(None))

def failAndIO: OptionT[IO, Unit] =
  for
    _ <- IO.putStrLn("Hello, world!").lift
    _ <- OptionT.fail: OptionT[IO, Unit]
    _ <- IO.putStrLn("Unreachable").lift
  yield ()
```

Da questo semplice esempio è possibile notare alcuni aspetti fondamentali: semplicemente leggendo il tipo del valore `failAndIO` è possibile comprendere come questo modelli una computazione che può sia fallire che effettuare operazioni di input e output. Inoltre, per poter eseguire gli effetti della monade più interna – in questo caso la monade `IO` – è necessario effettuarne il lifting all'interno della monade `OptionT` sfruttandone le proprietà di monad transformer. Poiché `lift` rispetta le leggi dei monad transformer il comportamento è quello atteso: una computazione trasformata con `lift` mantiene i side effect della monade sottostante senza aggiungerne di nuovi.

Per poter eseguire concretamente la computazione ed estrarre un valore dalla monade `OptionT` è sufficiente sfruttare `runOptionT` che permetterà di accedere direttamente a una computazione eseguibile nella monade `IO`:

```
@main def main: Unit = failAndIO.runOptionT.unsafeRun()
// -> "Hello, world!"
```

### Il transformer StateT

È possibile definire un transformer il cui compito è quello di aggiungere ad una qualsiasi monade la possibilità di simulare la presenza di uno stato mutabile in maniera simile a quanto fatto dalla monade `State`:

```
final case class StateT[S, M[_], A](runStateT: S => M[(A, S)])
```

La struttura di `StateT` è analoga a quella di `State`, con la differenza che il valore di ritorno ottenuto dalla computazione è contenuto all'interno di un generico `M`. Graficamente l'applicazione del transformer può essere visualizzata come mostrato in Figura 3.2.

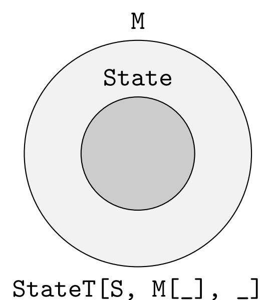


Figura 3.2: Rappresentazione grafica dell'applicazione del transformer `StateT` a una generica monade `M`

L'istanza di monad transformer di `StateT` è:[2]

```
type StateTFixS[S] = [M[_], A] =>> StateT[S, M, A]
given [S]: MonadTransformer[StateTFixS[S]] with
  def lift[M[_]: Monad, A](m: M[A]): StateT[S, M, A] =
    StateT(s => m.map((_, s)))
```

Poiché `lift` non può aggiungere alcun side effect alla monade la sua implementazione si limita a restituire lo stato mutabile senza modificarlo.

Anche l'istanza di monade sarà analoga a quella utilizzata da `State` con la differenza che sarà necessario effettuare le computazioni con stato all'interno della generica monade `M` sulla quale è parametrizzato il transformer:

```
given stateTMonad[M[_]: Monad, S]: Monad[StateT[S, M, _]] with
  def pure[A](a: A): StateT[S, M, A] =
    StateT(s => Monad.pure((a, s)))
```

[2]La type lambda `StateTFixS` serve a ottenere un costruttore di tipi della forma richiesta per essere un transformer fissando il tipo dello stato gestito dalla monade. Infatti, se `S` non fosse fissato allora `StateT` sarebbe un costruttore di tipi che richiede tre tipi generici, mentre per poter essere un transformer è richiesto che sia generico su due soli tipi.

```
extension [A](t: StateT[S, M, A])
  def flatMap[B](f: A => StateT[S, M, B]): StateT[S, M, B] =
    StateT(state0 =>
      t.runStateT(state0)
        .flatMap(result =>
          val (a, state1) = result
          f(a).runStateT(state1)
        )
    )
```

- `pure` permette di creare una computazione che non presenta alcun side effect. In questo caso l'effetto neutro consiste nel non modificare lo stato globale e nel non applicare gli effetti della monade sottostante
- `flatMap` permette di mettere in sequenza operazioni monadiche che possono avere l'effetto di modificare uno stato mutabile condiviso. L'implementazione non fa altro che effettuare il threading dello stato fra le computazioni monadiche che avvengono nella generica monade `M`

Le operazioni di base di `get` e `set` viste per la monade `State` possono essere implementate in anche per il generico `StateT`:

```
def get[S, M[_]: Monad]: StateT[S, M[_], S] =
  StateT(s => Monad.pure((s, s)))

def set[S, M[_]: Monad](state: S): StateT[S, M[_], Unit] =
  StateT(_ => Monad.pure(((), state)))
```

Grazie a queste funzioni è possibile definire delle computazioni alle quali aggiungere il side effect della modifica di uno stato globale:

```
def stateAndIO: StateT[String, IO, Unit] =
  for
    state <- StateT.get[String, IO]
    _     <- IO.putStrLn(f"Current state: $state").lift
    _     <- IO.putStrLn("Setting new state").lift
    _     <- StateT.set[String, IO]("bar")
  yield ()
```

Analizzando il tipo di `stateAndIO` è possibile comprendere come la computazione modellata possa effettuare sia input e output che modificare uno stato globale: infatti la monade di base `IO` è stata arricchita dal transformer `StateT` ottenendo una monade composta che modella i side effect di entrambi. Per poter interpretare la computazione modellata ottenendo i side effect desiderati è sufficiente sfruttare `runStateT` fornendo il valore iniziale dello stato mutabile:

```
@main def main: Unit =
  stateAndIO.runStateT("foo").unsafeRun()
// -> Current state: foo
// -> Setting new state
```

### 3.2.3 Rimozione della duplicazione di codice

È possibile osservare come l'implementazione del transformer per lo stato mutabile sia molto simile a quella della normale monade `State`. Anche le loro istanze di monade sono piuttosto simili effettuando il threading dello stato mutabile fra computazioni successive. È possibile eliminare tale duplicazione di codice definendo la monade `State` in termini della monade `StateT`:

```
type State[S, A] = StateT[S, Identity, A]
```

`State` viene espressa come la monade identità – che non presenta side effect – arricchita dalla possibilità di modificare uno stato globale. In questo modo non è più necessario implementare manualmente l'istanza di monade per `State` che sarà derivata automaticamente: infatti, poiché `Identity` è una monade, anche `StateT[S, Identity, _]` lo sarà. Le operazioni della monade `State` possono essere implementate in termini di quelle della monade `StateT`:

```
def get[S]: State[S, S] = StateT.get
def set[S](state: S): State[S, Unit] = StateT.set(state)
```

In questo modo è comunque possibile scrivere codice in termini della monade `State` nascondendo il fatto che questa è stata implementata sfruttando il transformer `StateT`; per esempio il codice mostrato nella Sezione 2.2.4 per incrementare il valore di un contatore rimarrà inalterato:

```
def incrementCounter: State[Int, String] =
  for
    counter    <- State.get
    _          <- State.set(counter + 1)
    newCounter <- State.get
  yield f"counter is: $newCounter"
```

## 3.3 Stack di monadi

Come mostrato negli esempi riportati nella sezione precedente è possibile comporre gli effetti di più monadi tramite l'uso dei monad transformer. Ciascun transformer viene utilizzato per aggiungere la possibilità di effettuare uno specifico side effect a una monade di base: `StateT` permette di aggiungere il side effect della modifica di uno stato, `OptionT` permette di modellare il fallimento esplicito di una computazione. Semplicemente osservando il transformer impiegato su una monade di base è possibile comprendere quali siano gli effetti aggiunti:

```
def program1: StateT[String, IO, Int] = ...
def program2: OptionT[IO, Int] = ...
```

`program1` modella una computazione che può sia effettuare input e output che modificare uno stato mutabile di tipo stringa. Dal tipo di `program2`, invece, è

possibile dedurre che la computazione può fallire e che può effettuare input e output.

### 3.3.1 Composizione di monadi

Negli esempi mostrati fino ad ora è sempre stato applicato un singolo transformer a una monade di base. La monade risultante da questa composizione può avere due tipologie di side effect: quello della monade di base e quello aggiunto dal transformer.

Inoltre è possibile applicare più transformer ad una sola monade di base in modo da ottenere uno stack di monadi che possegga molteplici tipologie di side effect. Infatti, come emerge dalla definizione fornita in precedenza, un transformer è a sua volta una monade e nulla vieta di applicarvi un ulteriore transformer. Si consideri per esempio la seguente monade definita utilizzando una serie di transformer:

```
type Program[A] = OptionT[StateT[String, IO, _], A]
```

La monade `Program` così definita permette di modellare una computazione che potrebbe fallire, modificare uno stato mutabile ed effettuare I/O. Dato che l'applicazione di transformer a una monade produce a sua volta una monade è possibile comporre un programma di tipo `Program` sfruttando lo zucchero sintattico della for comprehension:

```
def program: Program[Int] =
  for
    s <- IO.getLine.lift[StateTFixS[String]].lift[OptionT]
    _ <-
      if s == "fail"
      then OptionT.fail: Program[Unit]
      else StateT.set[String, IO](s).lift[OptionT]
  yield s.length
```

La computazione descritta da questa porzione di codice legge una riga dallo standard input; se la stringa letta è `"fail"` allora la computazione fallisce, altrimenti viene modificato lo stato mutabile impostandolo al valore della stringa. È possibile utilizzare ciascuna delle azioni fornite dalle monadi di base – in questo caso `fail`, `set` e `getLine` – previo un opportuno lifting per portarle nel livello corretto dello *stack* di monadi. In particolare:

- L'operazione di lettura `getLine` richiede un duplice lifting: poiché `IO` è alla base dello stack di monadi il primo `lift` servirà a portarlo nella monade `StateT`; in questo modo è possibile ottenere un valore di tipo `StateT[String, IO, String]`. Per poter portare tale valore dentro al transformer `OptionT` è necessario applicare un'ultima volta l'operazione di lifting
- L'operazione di modifica dello stato richiede l'applicazione di `lift` una sola volta dato che `StateT` è nel secondo livello dello stack di monadi

- L'operazione di fallimento non richiede alcun lifting: `fail` è già un valore di tipo `OptionT` e pertanto può essere utilizzato direttamente

Per poter effettivamente eseguire una computazione descritta in questo modo è sufficiente utilizzare i metodi `runStateT` e `runOptionT`. Questi permettono di rimuovere i livelli dello stack di monadi fino ad arrivare ad una computazione che possa essere direttamente eseguita:

```
@main def runProgram: (Option[Int], String) =
  program.runOptionT
    .runStateT("initial state")
    .unsafeRun()
```

Lo scopo della prima chiamata a `runOptionT` è rimuovere il primo livello dello stack di monadi, ottenendo una computazione di tipo `StateT[String, IO, Option[String]]`. La seconda chiamata a `runStateT` – fornendo il valore iniziale dello stato mutabile – rimuove il secondo livello dello stack di monadi, ottenendo una computazione di tipo `IO[(Option[Int], String)]`. Infine, la chiamata a `unsafeRun` permette di eseguire la computazione ottenuta.

### 3.3.2 Importanza dell'ordine nella composizione

Come forse si sarà potuto intuire dall'esempio riportato in precedenza l'ordine con il quale vengono applicati i transformer svolge un ruolo fondamentale. In generale, modificare l'ordine delle monadi che compongono uno stack va a modificare la semantica del programma che si sta prendendo in considerazione. Per esempio, si considerino i seguenti stack:

```
type Program1[A] = StateT[String, OptionT[IO, _], A]
type Program2[A] = OptionT[StateT[String, IO, _], A]
```

In `Program2` è stato invertito l'ordine con cui sono stati applicati i due transformer. I due stack possono essere visualizzati come mostrato in Figura 3.3:

- `Program1` rappresenta un programma che può effettuare I/O e potrebbe fallire. Nel caso in cui non fallisca la computazione può modificare uno stato mutabile; nel caso in cui invece la computazione fallisca allora sicuramente non potrà aver modificato lo stato globale
- `Program2` rappresenta un programma che può effettuare I/O e modificare uno stato mutabile; il programma potrebbe non produrre alcun risultato fallendo ma avrebbe modificato in ogni caso lo stato globale

Quindi un valore di tipo `Program1` può essere interpretato come una computazione che gestisce il proprio stato in maniera transazionale: si può accedere a uno stato mutabile ma in caso di fallimento della computazione non avviene alcune modifica.

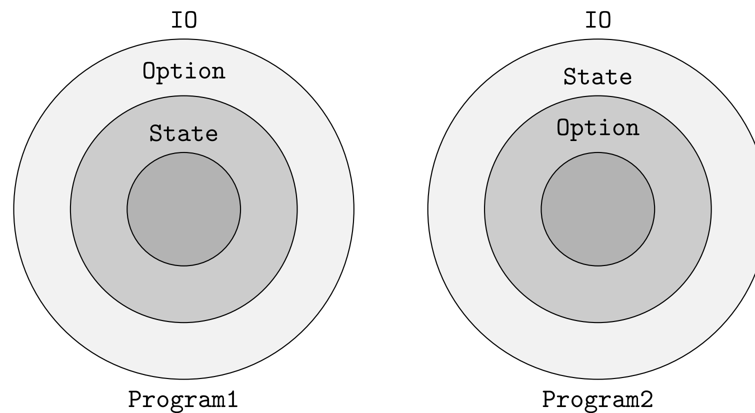


Figura 3.3: Rappresentazione grafica degli stack di monadi corrispondenti ai tipi `Program1` e `Program2`

Al contrario, un valore di tipo `Program2` potrebbe non produrre alcun risultato ma aver comunque modificato lo stato globale. Ciò è reso evidente dai tipi dei risultati ottenuti eseguendo le computazioni una volta eliminati i diversi livelli degli stack di monadi:

```
val program1: Program1[Int] = ???
val result1: Option[(Int, String)] =
  program1.runStateT("initial state").runOptionT.unsafeRun()

val program2: Program2[Int] = ???
val result2: (Option[Int], String) =
  program2.runOptionT.runStateT("initial state").unsafeRun()
```

L'esecuzione del primo programma ha come risultato un valore di tipo `Option[(Int, String)]` – dove il valore di tipo `Int` è il risultato vero e proprio della computazione mentre quello di tipo `String` rappresenta il nuovo stato. Quindi, nel caso in cui la computazione fallisca, non verrà prodotta una nuova versione dello stato mutabile che potrebbe essere stato modificato. D'altro canto, l'esecuzione del secondo programma produce un valore di tipo `(Option[Int], String)`: in questo caso, anche se la computazione dovesse fallire non producendo un risultato di valore `Int`, sarebbe sempre presente una nuova versione dello stato mutabile.

### 3.3.3 Miglioramento del parsing monadico

Come mostrato in precedenza nella Sezione 3.1.1 è possibile definire un parser con un'interfaccia monadica. Tuttavia si è potuto osservare come l'implementazione fornita presentasse una forte duplicazione di codice per reimplementare la logica

delle monadi `State` e `Option`. Grazie all'uso dei monad transformer è possibile ridefinire il tipo di un parser per eliminare completamente il codice duplicato:

```
opaque type Parser[A] = OptionT[State[String, _], A]
```

In questo modo non è più necessario dover derivare manualmente l'istanza di monade per `Parser`: questa sarà ottenuta in automatico in virtù del fatto che un `Parser` non è altro che uno stack di monadi.

Tuttavia, il fatto che `Parser` sia implementato come uno stack di monadi rimane un dettaglio implementativo che è opportuno nascondere a chi fa uso di tale tipo. Per esempio si può definire un semplice extension method che permetta di effettuare il parsing di una stringa andando a rimuovere i livelli dello stack di monadi in maniera analoga a quanto mostrato in precedenza:

```
extension [A](parser: Parser[A])
  def parse(string: String): (Option[A], String) =
    parser.runOptionT.runState(string)
```

Per esporre operazioni di base, come il fallimento del parsing o la lettura dello stato mutabile è possibile definire alcune funzioni che nascondano l'uso dei transformer che compongono lo stack:

```
def fail[A]: Parser[A] = OptionT.fail
def get: Parser[String] = State.get.lift[OptionT]
def set(string: String): Parser[Unit] =
  State.set(string).lift[OptionT]
```

L'operazione di parsing di un singolo carattere dovrà quindi essere ridefinita in termini di queste operazioni di base:

```
def char: Parser[Char] =
  for
    string <- get
    result <- string.headOption match
      case Some(char) =>
        for _ <- set(string.tail)
        yield char
      case None => fail
  yield result
```

Le funzioni `consumeTwo` e `consumeN` mostrate in precedenza non necessitano di alcuna modifica in quanto sono state definite sulla base dell'interfaccia monadica di `Parser` che, nella trasformazione dell'implementazione in uno stack di monadi, non è stata modificata.

# Capitolo 4

# Monad Transformer Library

Per quanto i transformer possano essere uno strumento efficace per modellare la presenza di diversi side effect, il loro utilizzo è generalmente associato a una serie di problematiche che possono rendere il codice poco leggibile e difficile da mantenere.

In questo capitolo viene descritto il meccanismo MTL come alternativa all'uso diretto dei monad transformer. Questo approccio ha il significativo vantaggio di permettere di descrivere in maniera *astratta* i side effect per modificarne l'interpretazione in base alle necessità. Infine, viene mostrato come MTL possa essere utilizzato per poter descrivere classi di effetti arbitrari.

## 4.1 Problemi nell'uso dei monad transformers

### 4.1.1 Lifting manuale delle operazioni

Come mostrato negli esempi riportati al capitolo precedente, perché un transformer possa sfruttare gli effetti della monade che va ad arricchire deve ricorrere all'operazione di lifting. Questa operazione permette di portare all'interno del transformer un valore dalla monade base senza aggiungere ulteriori side effect. Dunque, scrivendo codice monadico all'interno di un transformer il programmatore si troverà spesso a dover ricorrere all'operazione di lifting:

```
def manualLifting: StateT[Int, OptionT[IO, _], String] =
  for
    _ <- IO.putStrLn("test").lift[OptionT].lift[StateTFixS[Int]]
    _ <- OptionT.fail[IO, Any].lift[StateTFixS[Int]]
  yield "result"
```

Il tipo che descrive tutti i side effect della funzione `manualLifting` è `StateT[Int, OptionT[IO, _], String]`; alla base dello stack si trova la monade `IO` a cui viene aggiunta la possibilità di fallimento grazie al transformer `OptionT` e di modificare uno stato mutabile. Per poter eseguire l'operazione di stampa in output il cui

tipo è `IO[Unit]` è quindi necessario trasformarla in un valore compatibile con lo stack utilizzato. In questo caso la prima operazione di lifting trasforma il tipo in `OptionT[IO, Unit]`; l'operazione è applicata un'ultima volta per portare il valore nello stack descritto.

Come è possibile osservare il codice è molto verboso e introduce la necessità di inserire diverse chiamate a `lift` il cui unico scopo è quello di far combaciare i tipi delle operazioni. Il risultato sarà un codice poco leggibile dove la logica applicativa viene offuscata dalla presenza di numerose operazioni di lifting. Infatti, specialmente in stack composti da diversi transformer, è necessario ricorrere a più operazioni di lifting per ogni singola azione con side effect. Questo comporta un significativo sbilanciamento fra le porzioni di codice effettivamente rilevanti – che codificano la logica applicativa – e il boilerplate necessario per poter utilizzare correttamente lo stack di monadi.

Inoltre, il codice scritto utilizzando uno specifico stack di monadi sarà "viscoso": opporrà maggiore resistenza al cambiamento della struttura dello stack di monadi comportando la riscrittura di diverse porzioni di codice. Per esempio, aggiungendo o rimuovendo un ulteriore transformer allo stack utilizzato sarà necessario aggiungere o rimuovere da ogni operazione la chiamata al metodo `lift`. Immaginando di rimuovere il transformer `StateT` dalla funzione `manualLifting` mostrata in precedenza il codice dovrà essere trasformato in:

```
def manualLifting: OptionT[IO, String] =
  for
    _ <- IO.putStrLn("test").lift[OptionT]
    _ <- OptionT.fail[IO, Any]
  yield "result"
```

In questo caso il cambiamento nel tipo di ritorno della funzione ha comportato il dover modificare tutte le operazioni coinvolte rimuovendo le chiamate a `lift`.

Anche il semplice riordinare gli elementi dello stack di monadi comporta la necessità di apportare modifiche al codice. Sempre riprendendo l'esempio di `manualLifting` si immagini di dover cambiare lo stack di monadi in modo da avere un tipo `OptionT[StateT[Int, IO, _], String]`. In questo caso il codice dovrà essere modificato come segue:

```
def manualLifting: OptionT[StateT[Int, IO, _], String] =
  for
    _ <- IO.putStrLn("test").lift[StateTFixS[Int]].lift[OptionT]
    _ <- OptionT.fail[StateT[Int, IO, _], Any]
  yield "result"
```

Si noti come sia stato necessario modificare l'ordine nelle annotazioni dei tipi del lifting relativo all'operazione di stampa in output.

### 4.1.2 Principio di privilegio minimo

Una possibile soluzione al dover intervallare operazioni relative alla logica applicativa con le operazioni di lifting può consistere nel fissare lo stack da utilizzare. Una volta stabiliti i side effect di cui l'applicazione avrà bisogno – e quindi quale stack di monadi sia necessario utilizzare – si può realizzare primitive di base che restituiscano valori all'interno di tale stack e comporre queste per ottenere programmi complessi. Si consideri il seguente esempio: si vuole realizzare un programma che legga un file CSV, ne parsi il contenuto e calcoli la somma dei valori contenuti in una specifica colonna. Immaginando un'API per poter effettuare parsing di file CSV il risultato potrebbe essere il seguente:

```
type App = OptionT[IO, _]
def mainAction: App[Int] = for
  rawData <- readFile("data.csv") // : App[String]
  csv     <- parseCSV(rawData) // : App[CSV]
  column  <- csv.getIntColumn("column") // : App[List[Int]]
yield column.sum
```

La logica applicativa viene espressa in maniera concisa e leggibile. Ciò è reso possibile dal fatto che ogni operazione intermedia restituisce un valore all'interno della monade `App` e quindi non è necessario effettuare lifting delle operazioni per permettere ai tipi di combaciare.

Questo vantaggio dal punto di vista della leggibilità del codice viola però il *principio di privilegio minimo*: il programmatore, per poter esprimere in maniera concisa la logica applicativa, è costretto a incapsulare ogni valore intermedio all'interno dell'intero stack di monadi dell'applicazione anche qualora ciò non sia necessario. Per esempio, l'operazione di parsing del contenuto del file potrebbe fallire nel caso in cui questo sia mal formato; tuttavia, il parsing non richiede di effettuare operazioni di input e output. Nonostante ciò, `parseCSV` – per poter essere composto con le altre operazioni – restituisce un valore di tipo `OptionT[IO, CSV]`. Vale a dire che l'operazione di parsing potrebbe potenzialmente effettuare input e output anche se dal punto di vista logico ciò non ha senso. Lo stesso ragionamento vale per la chiamata al metodo `getIntColumn` che per esempio potrebbe fallire nel caso in cui non sia presente una colonna col nome desiderato; non c'è alcun motivo per cui tale operazione dovrebbe poter effettuare operazioni di input e output.

Dunque, il tipo di ciascuna delle funzioni intermedie è meno generale di quanto non sia effettivamente necessario per esprimerne il comportamento. Le operazioni `parseCSV` e `getIntColumn` non effettuano input e output ma il tipo di ritorno non lo impedisce; solo un'analisi dell'implementazione del corpo delle funzioni può permettere di capire se svolgono o meno uno dei determinati effetti previsti dallo stack di monadi utilizzato.

### 4.1.3 Violazione dell'incapsulamento

Un ulteriore problema che deriva dall'uso dei monad transformer sta nel fatto che il codice che utilizza tali stack di monadi è fortemente legato alla specifica modellazione del side effect che viene utilizzata. Essenzialmente viene violato il *principio di incapsulamento* per cui sarebbe necessario programmare basandosi su un'interfaccia piuttosto che su una specifica implementazione [3, p. 94].

Nell'esempio mostrato in precedenza la possibilità di fallimento viene espressa tramite l'uso del transformer `OptionT`. Vale a dire che, una volta interpretata, se la computazione dovesse fallire ritornerà un valore `None`. In questo modo, il programma è legato alla specifica modellazione del side effect del fallimento che viene fornita dal transformer `OptionT`. In circostanze differenti si potrebbe voler modificare il modo in cui un side effect viene implementato. Per esempio, nel caso del side effect del fallimento, si potrebbe sfruttare la monade `IO` per descrivere una computazione che fallisca con un'eccezione una volta interpretata:

```
def fail[A]: IO[A] = IO(() => throw Exception())
```

Questo è un possibile modo di ottenere il side effect del fallimento sfruttando direttamente la monade `IO` senza dover ricorrere al transformer `OptionT`. Nella descrizione della logica applicativa non importa quale sia la concreta implementazione utilizzata per descrivere tale side effect; l'aspetto rilevante è che il fallimento comporti l'interruzione della computazione.

Codificare le operazioni in maniera diretta sfruttando uno specifico stack di monadi crea un accoppiamento fra la descrizione della logica applicativa e la sua effettiva implementazione.

## 4.2 L'approccio MTL

Monad Transformer Library (MTL) è una libreria Haskell nata con l'obiettivo di semplificare la gestione di stack di monad transformer [56] permettendo di comporre in maniera modulare computazioni che modellano la presenza di side effect. Questa libreria è diventata nel tempo uno standard *de facto* per la realizzazione di sistemi complessi nell'ecosistema di Haskell: circa il 30% dei pacchetti presenti su Hackage[1] ne dipendono direttamente [19] ed è il sesto pacchetto più utilizzato su Stackage[2] [20].

L'approccio di MTL è stato poi adottato anche in altri linguaggi; per esempio in Scala nell'ecosistema di Typelevel è stata realizzata la libreria *Cats MTL* [57].

[1] Hackage è l'archivio di software open source della comunità Haskell utilizzato per pubblicare librerie e programmi.

[2] Stackage è un'infrastruttura utilizzata per mantenere degli *snapshot* di pacchetti Hackage compatibili fra loro per ottenere delle *build* stabili.

### 4.2.1 Idea alla base di MTL

L'obiettivo di MTL è di permettere la descrizione di computazioni con side effect in maniera flessibile, modulare e facilmente componibile, con la possibilità di cambiare senza difficoltà l'implementazione sottostante utilizzata per gestire i side effect. Questo obiettivo può essere raggiunto sfruttando nuovamente il meccanismo delle type class per definire *famiglie* di effetti, ciascuna delle quali è caratterizzata da una serie di primitive che permettono di esprimere le operazioni che possono essere effettuate [21].

#### Modifica di uno stato mutabile

Nello stile MTL il side effect del cambiamento di uno stato mutabile può essere catturato dalla seguente interfaccia:

```
trait State[M[_], S]:
  def get: M[S]
  def set(s: S): M[Unit]
```

L'interfaccia descrive, per un generico costruttore di tipi `M[_]`, le operazioni che questo deve fornire per permettere di gestire uno stato mutabile. Sfruttando il meccanismo dei parametri impliciti di Scala è possibile definire una funzione che accetta come parametro il contesto `M[_]` che implementa l'interfaccia `State`:

```
def update[S, M[_]: Monad](
  f: S => S
)(using S: State[M, S]): M[Unit] =
  for
    state <- S.get
    _     <- S.set(f(state))
  yield ()
```

Inoltre, è possibile modificare la funzione per rendere in maniera più "dichiarativa" i side effect che `M[_]` deve avere. Innanzitutto, è necessario definire il seguente *type alias*:

```
type HasState[S] = [M[_]] =>> State[M, S]
```

In questo modo, sfruttando lo zucchero sintattico che Scala offre per definire context bound, è possibile modificare la firma della funzione precedente:

```
def update[S, M[_]: Monad: HasState[S]](f: S => S): M[Unit] = ???
```

Nella definizione del generico `M[_]` è presente una lista di type class che ne descrivono le funzionalità; la definizione può essere letta come: *"Dato un qualunque `M` che sia una monade e abbia uno stato di tipo `S`"*. Infine, si può aggiungere al *companion object* di `State` un metodo che permetta di ottenere il parametro passato implicitamente:

```
object State:
  def apply[M[_]: HasState[S], S]: State[M, S] =
    summon[State[M, S]]
```

In questo modo il corpo della funzione potrà essere implementato come:

```
def update[S, M[_]: Monad: HasState[S]](f: S => S): M[Unit] =
  for
    state <- State[M, S].get
    _     <- State[M, S].set(f(state))
  yield ()
```

Dunque, `M[_]` può essere interpretato come il generico "contesto" all'interno del quale si svolge la computazione descritta; i context bound permettono di vincolare il tipo `M` indicando i side effect che dev'essere possibile svolgere all'interno di tale contesto.

### Fallimento di una computazione

In maniera analoga a quanto mostrato per lo stato mutabile, è possibile definire in astratto il side effect del fallimento di una computazione:

```
trait Fail[M[_]]:
  def fail[A]: M[A]
```

Anche in questo l'interfaccia definisce per un generico contesto `M[_]` le operazioni rilevanti per permettere di descrivere il fallimento di una computazione.

Un'operazione che richiede tale side effect può ricevere implicitamente in input una generica monade `M[_]` che implementi tale interfaccia:

```
def divide[M[_]: Monad](dividend: Int, divisor: Int)(using
  F: Fail[M]
): M[Int] =
  if divisor == 0 then F.fail
  else Monad.pure(dividend / divisor)
```

Anche in questo caso è possibile sfruttare lo zucchero sintattico di Scala per definire in maniera concisa il context bound:

```
object Fail:
  def apply[M[_]: Monad: Fail]: Fail[M] = summon[Fail[M]]

def divide[M[_]: Monad: Fail](
  dividend: Int,
  divisor: Int
): M[Int] =
  if divisor == 0 then Fail[M].fail
  else Monad.pure(dividend / divisor)
```

### 4.2.2 Composizione di più effetti

Il grande vantaggio offerto da MTL sta nella possibilità di combinare fra loro side effect differenti con grande semplicità. Si consideri la seguente computazione:

```
def effects[M[_]: HasState[Int]: Fail: Monad]: M[String] =
  for
    state    <- State[M, Int].get
    newState <- divide(10, state)
    _        <- State[M, Int].set(newState)
  yield f"The result is $newState"
```

Si possono osservare alcuni aspetti interessanti:

- La computazione sfrutta due diversi effetti che sono resi espliciti tramite l'uso dei context bound: la definizione del tipo generico `M[_]` può essere letta come *"Dato un qualunque `M` che permetta di modificare uno stato di tipo `S`, che permetta di far fallire la computazione e che sia una monade"*. Mentre i primi due requisiti sono fondamentali per poter utilizzare i side effect, richiedere che `M` sia una monade ha lo scopo di poter accedere all'operazione di `flatMap` per mettere in sequenza più operazioni
- Non viene concretizzato il tipo che permetterà di interpretare i diversi effetti, tutta la definizione è basata su un generico `M[_]`; la semantica delle operazioni potrà essere stabilita in un secondo momento in maniera indipendente. Perciò, a differenza dei transformer, la descrizione della computazione e di come questa verrà eseguita sono disaccoppiate

Inoltre viene eliminata la rigidità data dai monad transformer nel comporre operazioni con tipi differenti: in un contesto `M[_]` è possibile eseguire operazioni che richiedono entrambi gli effetti, solo uno di essi o nessuno di essi. Nell'esempio precedente viene utilizzata la funzione `divide` che richiede solo la capacità di fallire; ciò è possibile dato che è uno degli effetti forniti nel contesto della funzione `effect`. Non è necessario applicare operazioni di lifting nè modificare artificialmente il tipo di ritorno delle funzioni per semplificarne la composizione come veniva fatto nel caso dei transformer.

### 4.2.3 Interpretazione della computazione

Una computazione descritta secondo lo stile MTL definisce in astratto quali sono le operazioni che dovranno essere eseguite; quindi, per poterla interpretare è necessario fornire un tipo concreto che implementi tutte le operazioni richieste. L'implementazione viene sempre fornita come istanza di una type class in maniera analoga a quanto fatto per le monadi. Per esempio il transformer `OptionT` può essere usato come meccanismo concreto per interpretare computazioni che possono fallire:

```
given optionCanFail[M[_]: Monad]: Fail[OptionT[M, _]] with
  def fail[A]: OptionT[M, A] = OptionT.fail
```

Può essere utile riprendere l'interpretazione delle type class come predicati: fornire un'istanza equivale a dimostrare che `OptionT` può fallire e la dimostrazione è data dall'implementazione del metodo `fail`. Tuttavia, come già descritto in precedenza, `OptionT` non è l'unica monade che permette di modellare una computazione che fallisca; il fallimento può essere modellato anche nella monade `IO` tramite l'uso di eccezioni:

```
given ioCanFail: Fail[IO] with
  def fail[A]: IO[A] = IO(() => throw Exception())
```

Una volta definite tali istanze sarà possibile sfruttarle per interpretare una qualunque computazione che richieda il side effect del fallimento:

```
def interpretDivide: Unit =
  val res1: Int = divide[IO](10, 20).unsafeRun()
  val res2: Option[Int] =
    divide[OptionT[Identity, _]](10, 20).runOptionT
```

Utilizzare un diverso interprete per la computazione si riduce a indicare quale monade utilizzare come concreta implementazione del contesto `M[_]`. Nel primo caso viene utilizzata la monade `IO` e, se la computazione dovesse fallire, verrà sollevata un'eccezione; nel secondo caso viene utilizzato `OptionT` e in caso di fallimento il risultato sarà `None`.

L'istanza per l'effetto della modifica di uno stato è simile a quella del fallimento; in questo caso si mostra come il transformer `StateT` permetta di ottenere tale effetto:

```
given stateHasState[S, M[_]: Monad]: State[StateT[S, M, _], S]
  with
  def get = StateT.get
  def set(s: S) = StateT.set(s)
```

### Integrazione di stack di monadi e MTL

Le istanze fornite fino a questo momento permettono di ottenere semplici effetti ma non sono sufficienti per interpretare computazioni che necessitino di più effetti contemporaneamente: per esempio il semplice `OptionT` non può essere utilizzato per interpretare computazioni che richiedono uno stato mutabile. Una computazione simile richiede necessariamente uno stack di monadi che abbia al proprio interno una monade in grado di gestire il cambiamento di stato (come per esempio `StateT`), in quel caso l'intero stack potrà essere sfruttato per interpretare la computazione:

```
given transformerHasState[
  T[_[_], _]: MonadTransformer,
```

```
  S, M[_]: Monad: HasState[S]]: State[T[M, _], S] with
  def get = State[M, S].get.lift[T]
  def set(s: S) = State[M, S].set(s).lift[T]
```

In questo caso la definizione dell'istanza è più complessa:

- L'istanza viene definita per un generico transformer `T` e una generica monade `M`; se `M` può gestire uno stato mutabile di tipo `S`, allora anche `T` con `M` al proprio interno potrà farlo
- Per poter ottenere lo stato globale, `T` delega alla monade `M` l'operazione di `get` che viene poi inserita all'interno del transformer tramite `lift`. L'operazione di `set` è implementata in maniera analoga

Le due istanze `stateHasState` e `transformerHasState` appena mostrate permettono di ottenere un'istanza di `State` per un qualunque stack di monadi che abbia `StateT` al proprio interno: `stateHasState` funge da caso base per la ricorsione mentre `transformerHasState` permette di applicare tutti i `lift` necessari automaticamente indipendentemente dalla composizione dello stack di monadi.

Si consideri il seguente stack: `Transformer1[Transformer2[StateT[Int, IO, _], _], _]` (si supponga che `Transformer1` e `Transformer2` siano due monad transformer). Il compilatore Scala potrà derivare in automatico un'istanza di `HasState[Int]` per l'intero stack secondo il seguente procedimento:

1. `StateT[Int, IO, _]` ha un'istanza di `State` che può essere ottenuta tramite `stateHasState`
2. Poiché `Transformer2` è un transformer e `StateT[Int, IO, _]` è una monade con un'istanza di `State` allora anche `Transformer2[StateT[Int, IO, _], _]` ha un'istanza di `State` che può essere sintetizzata grazie a `transformerHasState`
3. Poiché `Transformer1` è un transformer e `Transformer2[StateT[Int, IO, _], _]` è una monade con un'istanza di `State` – generata al passaggio precedente – allora anche `Transformer1[Transformer2[StateT[Int, IO, _], _], _]` ha un'istanza di `State` che può essere sintetizzata grazie a `transformerHasState`

Nel concreto una chiamata a `get` in questo stack sarà interpretata come `StateT.get.lift[Transformer2].lift[Transformer1]`; tutti i `lift` saranno applicati automaticamente grazie alle istanze generate dal compilatore.

Il caso di fallimento è analogo a quanto osservato per `State`; se un transformer ha al proprio interno una monade che permette di ottenere tale side effect allora può a sua volta fornire un'istanza di `Fail`:

```
given transformerCanFail[
  T[_[_],_]: MonadTransformer,
  M[_]: Monad: Fail]: Fail[T[M, _]] with
  def fail[A] = Fail[M].fail.lift[T]
```

Una volta definite tali istanze una computazione complessa come `effects` mostrata nella Sezione 4.2.2 può essere interpretata specificando uno stack contenente i transformer necessari:

```
def interpretEffects: Unit =
  type Stack1 = OptionT[StateT[Int, Identity, _], _]
  val res1: (Option[String], Int) =
    effects[Stack1].runOptionT.runStateT(1)

  type Stack2 = StateT[Int, OptionT[Identity, _], _]
  val res2: Option[(String, Int)] =
    effects[Stack2].runStateT(1).runOptionT

  type Stack3 = StateT[Int, IO, _]
  val res3: (String, Int) =
    effects[Stack3].runStateT(1).unsafeRun()
```

Il compilatore fornirà implicitamente le istanze necessarie a interpretare la computazione. A seconda dello stack utilizzato si potrà dare una diversa interpretazione degli effetti specificati:

- Nel primo risultato ogni modifica allo stato prima del fallimento della computazione sarebbe comunque valida: infatti, il tipo di ritorno non incapsula anche lo stato `Int` all'interno di `Option`
- Nel secondo caso lo stato è gestito in maniera "transazionale": se la computazione fallisce allora lo stato non viene modificato
- Nel terzo caso l'interruzione della computazione viene ottenuta tramite le eccezioni e interpretarla potrebbe comportare il fallimento del programma a *runtime*

## 4.3 MTL come effect system

MTL può essere visto come un primo esempio di *effect system*: un sistema formale che permette di descrivere in maniera concisa i side effect di una computazione, le sue azioni osservabili dall'esterno [22, p. 943].

Per esempio, il meccanismo delle eccezioni di Java rappresenta un rudimentale effect system che permette di tracciare una sola tipologia di side effect: il lancio di un'eccezione [22, p. 985]. Anche in Scala, tramite l'uso delle *capability* [58], è stato definito un sistema di effetti che permette di verificare staticamente la presenza di eccezioni [23]. Attualmente è possibile abilitare questo meccanismo di cattura delle eccezioni come feature sperimentale; tuttavia, diverse ricerche sono attualmente in atto per permettere l'utilizzo delle capability per gestire qualunque tipologia di effetto [24, 59].

La versione originale della libreria MTL include diverse classi per trattare un'ampia gamma di side effect comuni: il fallimento di una computazione, la lettura di uno stato immutabile, la modifica di uno stato mutabile, ecc.

Tuttavia, l'approccio MTL è molto generale e permette di definire in *user space* quali sono gli effetti rilevanti per un determinato dominio applicativo, lasciando ampia libertà al programmatore di scegliere il livello di astrazione più adatto. Un effect system può diventare quindi uno strumento di design per la realizzazione di sistemi orientati agli effetti.

### 4.3.1 Definizione di effetti arbitrari

È possibile definire type class per classi di effetti arbitrari e al livello di granularità più adatto al problema da risolvere. Si immagini di dover realizzare una porzione di codice che necessita di interfacciarsi con un database contenente degli utenti; ai fini dell'esempio un utente viene modellato in maniera semplificata come:

```
final case class UserId(id: Int)
final case class User(id: UserId, name: String, age: Int)
```

Le operazioni che si vogliono implementare sono il recupero di un utente in base al suo identificativo, il salvataggio di un utente e la sua cancellazione.

Sicuramente tali operazioni potrebbero essere descritte direttamente all'interno della monade `IO`, stabilendo una connessione con il database ed eseguendo le query SQL corrispondenti. Tuttavia, tale approccio rende più difficile la realizzazione di test unitari: nell'esecuzione delle query queste cercheranno di stabilire una connessione con il database e potrebbero alterarne lo stato se non opportunamente gestite. Una soluzione comunemente adottata consiste nell'attuare *dependency injection* simulando il comportamento del database senza effettivamente stabilire una connessione, oppure connettendo i test a un database di test separato da quello in produzione.

#### Definizione degli effetti

Secondo lo stile MTL vengono innanzitutto definite le operazioni rilevanti per il dominio: primitive astratte che definiscono le operazioni che si possono eseguire sul database. La codifica mostrata in precedenza per gli effetti del fallimento e dello stato mutabile può essere utilizzata anche in questo caso:

```
trait UserStore[M[_]]:
  def get(userId: UserId): M[Option[User]]
  def save(user: User): M[Unit]
  def delete(userId: UserId): M[Unit]
```

L'interfaccia definisce un insieme minimale e ortogonale di operazioni di base che si vuole poter compiere nell'interazione con il database contenente gli utenti. Queste operazioni possono poi essere composte per ottenere logiche più complesse:

```
def updateOrDelete[M[_]: UserStore]
  (user: User)(f: User => User | Delete): M[Unit] =
  f(user) match
    case updatedUser: User => UserStore[M].save(updatedUser)
    case _: Delete         => UserStore[M].delete(user.id)

def updateOrDelete[M[_]: Monad: UserStore]
  (userId: UserId)(f: User => User | Delete): M[Unit] =
  UserStore[M].get(userId).flatMap {
    case Some(user) => updateOrDelete(user)(f)
    case None       => Monad.pure(())
  }
```

Lo scopo di `updateOrDelete` è di permettere di aggiornare un utente a partire dal suo identificativo, oppure di cancellarlo secondo la logica stabilita da una funzione passata come parametro. Tale funzione è definita in astratto per un qualunque tipo `M[_]` che permetta di ottenere le operazioni di `UserStore`.

### Interpretazione in un ambiente di produzione

Il vantaggio dato dall'encoding MTL delle operazioni sta nella possibilità di modificare come queste vengono interpretate in base alla necessità. Per esempio le operazioni possono essere interpretate in un ambiente di produzione dove viene effettivamente stabilita una connessione con una database. In questo caso, lo stack utilizzato per interpretare la computazione dovrà sicuramente poter permettere l'esecuzione di I/O e gestire uno stato che contenga la connessione al database:

```
type ProductionRunner = StateT[Runtime, IO, _]
def getRuntime: ProductionRunner[Runtime] = StateT.get
```

In questo caso l'interprete adottato per un ambiente di produzione può manipolare uno stato – nell'esempio chiamato `Runtime` – che contiene la connessione al database.

Una volta stabilito lo stack da utilizzare è sufficiente definire un'istanza di `UserStore`:

```
given UserStore[ProductionRunner] with
  def get(userId: UserId) = for
    runtime <- getRuntime
    connection = runtime.connection
    // use the database connection to perform the SELECT query
    user <- ??? : ProductionRunner[Option[User]]
  yield user
```

L'implementazione recupera la connessione a partire dal runtime e la sfrutta per comunicare con il database, l'effettiva implementazione in questo caso non è rilevante ed è stata tralasciata.

L'idea fondamentale sta nella possibilità di disaccoppiare la *descrizione* della logica applicativa dalla sua effettiva *interpretazione*. Modificare l'interprete non comporta cambiamenti nelle funzioni descritte in astratto e viceversa.

### Interpretazione in un ambiente di test

Per mostrare come sia possibile modificare l'interprete in base alle necessità si consideri la seguente porzione di codice:

```
def updateAge[M[_]: Monad: UserStore](userId: UserId) =
  updateOrDelete(userId) { user =>
    if user.age < 18
    then Delete
    else user.copy(age = user.age + 1)
  }
```

La funzione, preso un identificativo di un utente, lo elimina se questo è minorenne o ne aumenta l'età di un anno. Come è possibile osservare dalla firma del metodo, questo necessita di poter effettuare gli effetti descritti in astratto da `UserStore`.

Per poter testare la correttezza della logica applicativa – vale a dire che utenti minorenni siano effettivamente cancellati – potrebbe non essere pratico realizzare e connettersi ad un database di test. In questo caso è possibile definire un interprete che simuli il comportamento del database mantenendo un insieme di utenti in memoria senza necessità di effettuare operazioni di input e output. L'unico elemento di cui il runtime ha bisogno è una mappa degli utenti:

```
final case class Runtime(users: Map[UserId, User])
type TestRunner = State[Runtime, _]
```

L'interprete `TestRunner` non è altro che la monade `State` che manipola tale mappa. L'implementazione delle operazioni risulta essere molto semplice:

```
given UserStore[TestRunner] with
  def get(userId: UserId) = State.get.map(_.users.get(userId))
  def save(user: User) = updateUsers(_ + (user.id -> user))
  def delete(userId: UserId) = updateUsers(_ - userId)

def updateUsers(
  update: Map[UserId, User] => Map[UserId, User]
): TestRunner[Unit] =
  State.update { runtime =>
    val newUsers = update(runtime.users)
    runtime.copy(users = newUsers)
  }
```

Diventa quindi possibile testare la logica applicativa senza dover effettuare alcuna operazione di input e output; il test stabilisce lo stato iniziale del sistema e verifica che la logica dell'operazione eseguita sia corretta:

```
def testUpdateAgeDeletesUnderageUsers: Unit =
  val user = User(UserId(1), "Giacomo", 12)
  val runtime = Runtime(Map(user.id -> user))
  val finalRuntime =
    updateAge[TestRunner](user.id).runStateT(runtime)._2
  assert(finalRuntime.users.isEmpty)
```

# Capitolo 5

# Free Monad

Fino ad ora si è osservato come sia possibile modellare i side effect tramite l'uso di monadi, sia ricorrendo ai *monad transformers* che all'approccio MTL. In quest'ultimo caso la computazione viene descritta tramite l'uso di metodi astratti definiti in un'interfaccia; l'interpretazione concreta di tali chiamate a metodo può essere definita in un secondo momento assegnando la semantica desiderata a ciascuna di esse.

L'approccio adottato dalle free monad consiste nel descrivere la computazione tramite un AST che permette di comporre in sequenza più operazioni definite in maniera astratta. In questo modo è possibile dare semantica alle operazioni definendo interpreti che, attraversando l'AST, possono tradurre le operazioni astratte in una versione "eseguibile".

## 5.1 Implementazione di una Free Monad

L'AST descritto tramite una free monad ha l'obiettivo di catturare la struttura sintattica di una computazione monadica [25]. Come già descritto in precedenza, l'interfaccia delle monadi permette di descrivere una computazione come una sequenza di passi successivi che, una volta terminati, restituiscono un valore.

In seguito verrà formalizzata tale definizione arrivando in maniera graduale alla definizione di una free monad. L'encoding adottato è quello delle cosiddette *Free Operational Monad* [25]; tuttavia esistono svariate tecniche per ottenere risultati analoghi [26–28].

### 5.1.1 Descrizione astratta di un programma monadico

#### Istruzioni di un programma monadico

In generale si può considerare un programma monadico come definito da una serie di istruzioni appartenenti ad un insieme $I$ messe in sequenza fra loro. Ogni istruzione, una volta eseguita, può restituire un valore differente; si consideri per esempio il seguente programma definito nella monade `State`:

```
def program[S](f: S => S): State[S, String] =
  for
    state <- get
    newState = f(state)
    _ <- set(newState)
  yield "end"
```

Le operazioni di base di cui si compone sono `get` e `set`; mentre la prima restituisce un valore di tipo `S`, la seconda restituisce `Unit`. Quindi, in questo caso, l'insieme delle istruzioni sarebbe $I = \{get, set\}$; questo insieme può essere codificato in Scala con una semplice enumerazione:

```
enum StateDSL[S, A]:
  case Get[S]() extends StateDSL[S, S]
  case Set[S](s: S) extends StateDSL[S, Unit]
```

Un valore di tipo `StateDSL[S, A]` rappresenta una singola istruzione che opera su uno stato di tipo `S` e restituisce un valore di tipo `A`:

- `Get` opera su uno stato di tipo `S` e lo restituisce, quindi ha tipo `StateDSL[S, S]`
- `Set` ha tipo `StateDSL[S, Unit]` in quanto, come descritto in precedenza, ha l'effetto di modificare lo stato `S` ma non restituisce alcun valore d'interesse

#### Esecuzione di istruzioni in un programma monadico

Come descritto all'inizio del capitolo una free monad cattura in una struttura dati la struttura sintattica di un programma monadico. Un tipo di dato che rappresenti l'AST di un programma monadico deve quindi permettere di rappresentare l'esecuzione di una singola istruzione.

Per disaccoppiare l'AST dallo specifico tipo di istruzioni eseguite un programma verrà definito in maniera generica rispetto alle istruzioni che può eseguire:

```
enum Program[I[_], A]: ...
```

Il tipo generico `I` rappresenta il set di istruzioni che possono essere eseguite dal programma, mentre `A` rappresenta il tipo del valore di ritorno ottenuto dalla sua esecuzione. Per esempio, un valore di tipo `Program[StateDSL[Int, _], String]` rappresenterà un programma monadico che può eseguire operazioni su uno stato di tipo `Int` e restituisce un valore di tipo `String`.

Per poter costruire un programma che esegue una singola istruzione può essere definito un caso ad hoc dell'enumerazione e un corrispondente *smart constructor*:

```
enum Program[I[_], A]:
  case Instruction[I[_], A](instruction: I[A])
    extends Program[I, A]

object Program:
  def fromInstruction[I[_], A](instruction: I[A]): Program[I, A] =
    Instruction(instruction)
```

Questa definizione permette di creare semplici programmi che eseguono una singola istruzione:

```
def get[S] = Program.fromInstruction(StateDSL.Get())
def set[S](state: S) =
  Program.fromInstruction(StateDSL.Set(state))
```

Chiaramente un programma che esegue una singola istruzione non è di particolare interesse; per poter combinare semplici istruzioni e realizzare programmi complessi è necessario espandere ulteriormente la definizione di `Program`.

### Restituzione di un valore puro

Un'ulteriore caratteristica di ogni computazione monadica – espressa dal metodo `pure` dell'interfaccia delle monadi – sta nella possibilità di poter inserire un qualunque valore all'interno della sequenza delle computazioni senza aggiungervi ulteriori side effect. Tuttavia, data la definizione attuale di `Program`, un programma monadico può limitarsi a permettere l'esecuzione di una singola istruzione appartenente al set `I` specificato. Dunque l'AST di `Program` deve essere esteso per permettere di ritornare un valore senza eseguire istruzioni:

```
enum Program[I[_], A]:
  case Instruction[I[_], A](instruction: I[A])
    extends Program[I, A]
  case Return[I[_], A](value: A) extends Program[I, A]

object Program:
  def fromValue[I[_], A](value: A): Program[I, A] =
    Return(value)
```

Il nuovo costruttore `Return` permette di definire programmi che restituiscono valori senza alcun side effect, ovvero senza eseguire nessuna delle operazioni dell'insieme `I`. Per esempio

```
val program: Program[StateDSL[Int, _], String] =
  Program.fromValue("result")
```

Restituisce il valore `"result"` senza modificare lo stato mutabile tramite le operazioni definite nello `StateDSL`.

### Messa in sequenza di programmi

L'aspetto più importante di una monade sta nella possibilità di mettere in sequenza operazioni tramite l'uso di `flatMap`. Anche l'AST di `Program` deve esprimere questo concetto per poter catturare la struttura inerentemente sequenziale di un programma monadico. Per questo motivo viene definito un ultimo costruttore:

```
enum Program[I[_], A]:
  case Instruction[I[_], A](instruction: I[A])
      extends Program[I, A]
  case Return[I[_], A](value: A) extends Program[I, A]
  case Then[I[_], A, B](
    program: Program[I, A],
    continuation: A => Program[I, B]
  ) extends Program[I, B]
```

Il costruttore `Then` permette di catturare la messa in sequenza di operazioni monadiche: il primo argomento è il primo programma da eseguire che produrrà un valore di tipo `A`; il secondo argomento è una continuazione che, preso il valore prodotto dal primo programma, restituisce un secondo programma da eseguire.

È possibile definire un extension method per rendere più facile la costruzione di programmi complessi:

```
extension [I[_], A](program: Program[I, A])
  def andThen[B](continuation: A => Program[I, B]) =
    Then(program, continuation)
```

Per esempio il programma con stato mostrato alla sezione precedente può essere ora espresso come:

```
def program[S](f: S => S): Program[StateDSL[S, _], String] =
  get.andThen { state =>
    val newState = f(state)
    set(newState).andThen { _ =>
      Program.fromValue("end")
    }
  }
```

È possibile osservare come `Program` rispetti per costruzione l'interfaccia delle monadi, indipendentemente dal tipo di istruzioni che utilizza: i costruttori `Result` e `Then` equivalgono rispettivamente alle operazioni `pure` e `flatMap`:

```
given programIsMonad[I[_]]: Monad[Program[I, _]] with
  def pure[A](a: A): Program[I, A] = Return(a)
  extension [A](program: Program[I, A])
    override def flatMap[B](
      continuation: A => Program[I, B]
    ): Program[I, B] = Then(program, continuation)
```

In questo modo è possibile definire programmi nella monade `Program` sfruttando lo zucchero sintattico della for comprehension; l'esempio precedente può essere riscritto in maniera più chiara come:

```
def program[S](f: S => S): Program[StateDSL[S, _], String] =
  for
    state <- get
    newState = f(state)
    _ <- set(newState)
  yield "end"
```

# 5.2 Interpretazione di una free monad

## 5.2.1 Interpretazione delle istruzioni di un programma

I valori di tipo `Program` mostrati fino ad ora non sono altro che degli AST che rappresentano sotto forma di struttura dati un programma monadico. Questa struttura dati può essere attraversata e interpretata per assegnare una semantica alle istruzioni di cui si compone. Prima di poter definire come interpretare un programma monadico è necessario definire come interpretarne le singole istruzioni; perciò si introduce il concetto di interprete[1]:

```
trait Interpreter[F[_], G[_]]:
  def apply[A](f: F[A]): G[A]

type ~>[F[_], G[_]] = Interpreter[F, G]
```

Un interprete, dato un generico insieme di istruzioni `F` permette di trasformare – tramite il metodo `apply` – una qualunque istruzione di tipo `F[A]` in un valore `G[A]`. Il tipo `G[_]` è generico e potrebbe per esempio essere un secondo insieme di istruzioni o una monade.

Per esempio, è possibile definire un interprete che trasforma le istruzioni dello `StateDSL` in azioni concrete all'interno del transformer `StateT`:

```
def pureInterpreter[S] =
  new (StateDSL[S, _] ~> StateT[S, Identity, _]):
    def apply[A](s: StateDSL[S, A]): StateT[S, Identity, A] =
      s match
        case Get()  => StateT.get
        case Set(s) => StateT.set(s)
```

L'interprete si limita a tradurre le istruzioni di `Get` e `Set` nelle corrispondenti operazioni del transformer.

---

[1] Questo concetto è noto in letteratura come trasformazione naturale[29]. Nella trattazione si preferisce utilizzare il termine "interprete" in quanto rende in maniera efficace il suo utilizzo nell'interpretazione delle operazioni di una free monad.

### 5.2.2 Interpretazione di un'intero programma

È possibile sfruttare gli interpreti che traducono singole istruzioni per interpretare interi programmi definiti all'interno della monade `Program`.

L'unico accorgimento necessario è che l'interprete traduca ciascuna istruzione in un tipo che rispetti l'interfaccia di monade. Grazie a questo vincolo è possibile mettere automaticamente in sequenza le operazioni una volta che sono state tradotte:

```
extension [I[_], A](program: Program[I, A])
  def interpret[M[_]: Monad](interpreter: I ~> M): M[A] =
    program match
      case Return(value)            => Monad.pure(value)
      case Instruction(instruction) => interpreter(instruction)
      case Then(program, continuation) =>
        for
          a <- program.interpret(interpreter)
          continuationProgram = continuation(a)
          result <- continuationProgram.interpret(interpreter)
        yield result
```

- `Return` corrisponde ad una chiamata di `pure` nella generica monade `M`
- `Instruction` permette di eseguire una singola istruzione che viene direttamente interpretata dall'interprete fornito in input
- `Then` rappresenta la composizione in sequenza di due programmi. Il primo programma viene interpretato ottenendo un valore di tipo `A`. Tale valore viene fornito in input alla continuazione per ottenere il secondo programma che, ricorsivamente, viene interpretato producendo il risultato finale

### 5.2.3 Ispezione dell'AST di una free monad

Il metodo mostrato nella sezione precedente permette di dare semantica al programma stabilendo come ciascuna delle istruzioni debba essere interpretata. Tuttavia, il vantaggio dato dalla rappresentazione della computazione come AST sta nella possibilità di poter ispezionare il programma per avere un controllo più fine sulla sua interpretazione.

È infatti possibile estrarre dal programma la prima istruzione che questo deve eseguire e la continuazione che determina come l'esecuzione deve procedere. Per fare ciò è possibile definire la seguente struttura:

```
enum ProgramView[I[_], A]:
  case Return[I[_], A](value: A) extends ProgramView[I, A]
  case Then[I[_], A, B](
    instruction: I[A],
    continuation: A => Program[I, B]
  ) extends ProgramView[I, B]
```

`ProgramView` permette di avere una vista uniforme di un programma limitando i casi possibili a due sole opzioni: il programma esegue un'istruzione e prosegue con una data continuazione, oppure il programma termina restituendo un valore.

Per poter estrarre una `ProgramView` a partire da un programma è possibile implementare il seguente metodo:

```
extension [I[_], A](program: Program[I, A])
  @tailrec def next: ProgramView[I, A] = program match
  case Return(value) => ProgramView.Return(value)
  case Instruction(instruction) =>
    ProgramView.Then(instruction, Return(_))
  case Then(program, f) =>
    program match
      case Return(value) => f(value).next
      case Instruction(instruction) =>
        ProgramView.Then(instruction, f)
      case Then(program, g) =>
        program.andThen(x => g(x).andThen(f)).next
```

I diversi casi del pattern matching coprono tutte le possibili conformazioni di un programma:

- Se il programma restituisce un valore questo viene mappato nella vista corrispondente
- Se il programma esegue un'istruzione questa viene inserita nella vista come prossima istruzione e la continuazione si limita a restituire il valore dell'istruzione
- Nel caso vengano composti più programmi in sequenza è necessario osservare la composizione del primo programma:
  - Se si limita a restituire un valore questo viene fornito alla continuazione e viene restituita la prima istruzione del programma ottenuto
  - Se si limita a eseguire un'istruzione questa viene tradotta nella vista corrispondente che racchiude al proprio interno l'istruzione e la continuazione
  - Se è a sua volta la composizione sequenziale di due programmi, allora viene restituita la prima operazione del programma più interno

Questo meccanismo rende possibile realizzare funzioni che interpretano un programma ispezionandone di volta in volta la successiva istruzione. Per esempio, è possibile realizzare una funzione che esegue una computazione che fa uso dello `StateDSL`:

```
extension [S, A](program: Program[StateDSL[S, _], A])
  @tailrec def runWithState(state: S): (S, A) =
    program.next match
      case ProgramView.Return(value) => (state, value)
```

```
    case ProgramView.Then(instruction, continuation) =>
      instruction match
        case Get() => continuation(state).runWithState(state)
        case Set(s) => continuation(()).runWithState(s)
```

La funzione ottiene la prima istruzione da eseguire e nel caso in cui sia una `Get` fornisce alla continuazione lo stato corrente. Nel caso in cui l'istruzione sia invece `Set`, la continuazione viene ripresa fornendole il valore `Unit`. Sul programma ottenuto viene ricorsivamente chiamato il metodo `runWithState` modificando lo stato fornito.

### A poor man's concurrency (free) monad

È possibile sfruttare la possibilità di ispezionare l'AST di un programma descritto in questo modo per realizzare interpreti più complessi.

Questa sezione riprende l'eccellente esempio mostrato in [30] e illustra come sia possibile implementare una *"poor man's concurrency monad"* utilizzando la free monad appena mostrata. In particolare, la possibilità di accedere alla continuazione che determina come deve procedere la computazione rende piuttosto semplice l'implementazione di una forma di concorrenza in *user space*.

Il linguaggio preso in considerazione è il seguente:

```
type Concurrent[A] = Program[ConcurrentDSL, A]
enum ConcurrentDSL[A]:
  case YieldControl extends ConcurrentDSL[Unit]
  case Stop extends ConcurrentDSL[Nothing]
  case Perform[A](action: () => A) extends ConcurrentDSL[A]
  case Fork(process: Concurrent[Unit])
      extends ConcurrentDSL[Unit]
```

Le operazioni mostrate sono sufficienti per descrivere un meccanismo di *multithreading* cooperativo:

- `Fork` permette di creare un nuovo *thread* che esegue il programma specificato
- `YieldControl` permette a un thread di segnalare esplicitamente allo *scheduler* di voler mettere in pausa la propria esecuzione lasciando il controllo ad altri thread
- `Stop` permette a un thread di terminare prematuramente la propria esecuzione
- `Perform` viene utilizzato come meccanismo per eseguire una qualunque azione con side effect; la funzione che produce il valore di tipo `A` può eseguire side effect arbitrari

È possibile definire alcuni smart constructor per permettere di costruire più semplicemente un programma concorrente:

```
def yieldControl = Program.fromInstruction(YieldControl)
def stop = Program.fromInstruction(Stop)
```

```
def perform[A](action: => A) =
  Program.fromInstruction(Perform(() => action))
def fork(program: Concurrent[Unit]) =
  Program.fromInstruction(Fork(program))
```

Un esempio di programma ottenuto a partire da queste operazioni di base potrebbe essere il seguente:

```
val program =
  perform(println("forking"))
    >> fork(thread("1", "hello world"))
    >> yieldControl
    >> perform(println("thread1 - ending"))

def thread(name: String, message: String) =
  perform(println(f"thread $name - $message"))
    >> yieldControl
    >> perform(println(f"thread $name - ending"))
```

Per poter stabilire l'effettiva semantica delle operazioni ed eseguire il programma è necessario definire una funzione che lo interpreti.

In questo esempio il comportamento desiderato è quello di realizzare uno scheduler che esegua i thread in maniera cooperativa: un thread esegue le proprie operazioni ininterrotto fino a quando non termina o rende esplicito il voler cedere il controllo tramite una chiamata a `yield`. Solo in seguito a una chiamata a `yield` lo scheduler stabilirà il nuovo thread da mandare in esecuzione mettendo in pausa l'esecuzione di quello attualmente attivo. Il metodo per eseguire un programma secondo questa politica di *scheduling* è il seguente:

```
extension (thread: Concurrent[Unit])
  def runSingleThreadedCooperative: Unit =
    runThreads(List(thread))

  private def runThreads(threads: List[Concurrent[Unit]]): Unit =
    threads match
      case Nil => ()
      case thread :: threads =>
        runInstruction(thread.next, threads)
```

Per semplificare l'implementazione dello scheduler, questo utilizza una coda in cui si trovano i thread da eseguire e nel momento in cui avviene lo `yield` il thread corrente viene messo in fondo alla coda e viene dato il controllo a quello in testa alla coda.

L'intera logica di esecuzione delle istruzioni è codificata nella funzione mostrata al Listato 5.1. A seconda dell'istruzione che viene eseguita, il comportamento è il seguente:

- `Perform`: viene eseguita l'azione incapsulata nel costruttore e il risultato ottenuto è dato come input alla continuazione per ottenere il programma

che contiene la continuazione del thread corrente. Questo programma viene inserito in testa alla coda dei thread da eseguire in quanto la semantica scelta stabilisce che un thread possa essere interrotto solo se esegue esplicitamente `yield`. Si noti come sarebbe possibile modificare questa politica interrompendo con *preemption* un thread ad ogni operazione: sarebbe sufficiente inserire il programma che rappresenta la continuazione in fondo alla coda così da dare precedenza ad altri thread

- `Stop`: interrompe l'esecuzione del thread. In questo caso non viene aggiunta alcuna continuazione alla lista di thread da eseguire. Si noti inoltre un aspetto interessante: poiché `Stop` ha tipo `ConcurrentDSL[Nothing]`, in questo ramo del pattern matching la continuazione necessiterebbe di un valore di tipo `Nothing` per poter generare il programma che rappresenta la continuazione del thread corrente. Poiché non c'è modo di produrre un valore concreto di tipo `Nothing`[2] è impossibile ottenere la continuazione e l'unica operazione sensata è quella di terminare l'esecuzione del thread
- `YieldControl`: in questo caso la continuazione del thread corrente viene messa in fondo alla lista dei thread
- `Fork`: è l'unica operazione che permette di accrescere il numero di programmi contenuti nella lista dei thread da eseguire. La continuazione del thread corrente viene messa in cima alla lista in modo che possa continuare l'esecuzione mentre il thread di cui è stato effettuato il *fork* viene messo in fondo

## 5.3 Composizione di più DSL

### 5.3.1 Composizione modulare di linguaggi

Negli esempi mostrati fino ad ora è sempre stato utilizzato un programma il cui insieme di istruzioni è definito da una sola enumerazione. Tuttavia, sarebbe desiderabile poter combinare fra loro più linguaggi di base per poter descrivere programmi complessi.

Se il programmatore fosse obbligato a definire l'intero insieme di operazioni in una sola enumerazione allora questa finirebbe per essere l'equivalente di una *god interface* con moltissimi metodi che mescolano operazioni appartenenti a diversi ambiti.

Si immagini di realizzare un programma che deve leggere e scrivere dal terminale e permettere di effettuare logging delle operazioni che esegue. Sicuramente sarebbe possibile definire un solo set di istruzioni che racchiuda tutte le funzionalità richieste:

[2] Tecnicamente sarebbe possibile creare un valore di tipo `Nothing` (per esempio `???` o una qualunque eccezione) ma questo comporterebbe una terminazione anomala del programma o la sua divergenza senza poter eseguire in ogni caso la continuazione

```
def runInstruction(
  instruction: ProgramView[ConcurrentDSL, Unit],
  threads: List[Concurrent[Unit]]
): Unit =
  instruction match
    case ProgramView.Return(_) => runThreads(threads)
    case ProgramView.Then(instruction, continuation) =>
      instruction match
        case Perform(action) =>
          val result = action()
          val newThreads = continuation(result) +: threads
          runThreads(newThreads)
        case Stop =>
          runThreads(threads)
        case YieldControl =>
          val newThreads = threads :+ continuation(())
          runThreads(newThreads)
        case Fork(process) =>
          val newThreads =
            continuation(()) +: threads :+ process
          runThreads(newThreads)
```

Listato 5.1: Implementazione della funzione `runInstructions`.

```
enum LogLevel:
  case Info, Warning, Error

enum LogAndConsole[A]:
  case Log(level: LogLevel, msg: String)
    extends LogAndConsole[Unit]
  case GetLine() extends LogAndConsole[String]
  case PrintLine(msg: String) extends LogAndConsole[Unit]
```

Tuttavia, il linguaggio `LogAndConsole` unisce due funzionalità differenti: la gestione del logging e la gestione del terminale. Se un programma dovesse avere bisogno unicamente di accedere al side effect del logging dovrebbe comunque essere definito in termini di `LogAndConsole`. Idealmente, dovrebbe essere possibile definire i due DSL separatamente per poterli combinare in un secondo momento, se necessario.

### Iniezione di istruzioni in un linaguaggio generico

Si considerino due degli smart constructor mostrati come esempi nelle sezioni precedenti:

```
def get[S]: Program[StateDSL, S] = ...
def stop: Program[ConcurrentDSL, Nothing] = ...
```

I tipi di questi programmi sono troppo specifici per poter essere utilizzati nel comporre programmi complessi che fanno uso di più DSL contemporaneamente. Infatti, ciascuno limita l'insieme di istruzioni a cui il programma può accedere a quelle di uno specifico DSL: `StateDSL` nel primo e `ConcurrentDSL` nel secondo.

Il problema sta nella definizione della funzione `Program.fromInstruction` utilizzata per creare gli smart constructor. Questa funzione, infatti, restituisce un programma le cui istruzioni devono essere tutte appartenenti al tipo dell'istruzione specificata. È possibile definire uno smart constructor più generico che permette di "iniettare" un'istruzione all'interno di un linguaggio più ampio; per fare ciò è possibile sfruttare nuovamente il concetto di interprete introdotto in precedenza:

```
def inject[I[_], A, I2[_]](instruction: I[A])(using T: I ~> I2) =
  Program.fromInstruction(T(instruction))
```

La funzione utilizza implicitamente un interprete `T` che permette di trasformare un'istruzione dell'insieme `I` in un'istruzione dell'insieme `I2`. L'istruzione passata in input viene quindi trasformata in un'istruzione del secondo insieme sfruttando tale interprete. In questo modo è possibile realizzare degli smart constructor che non limitano il set di istruzioni a cui il programma può accedere:

```
def stop[I[_]](using ConcurrentDSL ~> I): Program[I, Nothing] =
  Program.inject(ConcurrentDSL.Stop())
```

Il costruttore stabilisce che, dato un qualunque insieme di istruzioni `I` per il quale sia possibile inserirvi istruzioni di tipo `ConcurrentDSL`, è possibile creare un programma che utilizza le istruzioni di `I`. La notazione può essere ulteriormente alleggerita definendo un type alias per gli interpreti:

```
type With[F[_]] = [G[_]] =>> F ~> G
```

In questo modo, `With` può essere utilizzato per descrivere in maniera dichiarativa le istruzioni che un generico linguaggio deve poter offrire:

```
def stop[I[_]: With[ConcurrentDSL]]: Program[I, Nothing] =
  Program.inject(ConcurrentDSL.Stop())
```

La firma del metodo può essere letta come *"Dato un qualunque linguaggio `I` che ha a disposizione le operazioni del `ConcurrentDSL`, restituisco un programma che può utilizzare le istruzioni di `I` e restituisce `Nothing` quando termina"*.

### Esempio di un programma che combina più DSL

Sfruttando le definizioni appena mostrate è possibile definire un programma che combina le funzionalità di più DSL. Si consideri nuovamente l'esempio di un programma che deve poter interagire con il terminale ed effettuare logging.

Gli effetti richiesti dal programma possono quindi essere descritti da due DSL distinti:

```
enum LogDSL[A]:
  case Log(logLevel: LogLevel, msg: String) extends LogDSL[Unit]

enum ConsoleDSL[A]:
  case GetLine() extends ConsoleDSL[String]
  case PrintLine(msg: String) extends ConsoleDSL[Unit]
```

La differenza fondamentale rispetto agli altri esempi mostrati fino ad ora sta nella definizione degli smart constructor: questi utilizzeranno la funzione `inject` mostrata in precedenza per non fissare a priori il set di istruzioni da utilizzare. Per esempio nel caso del `LogDSL` il risultato sarà:

```
object Log:
  def log[I[_]: With[LogDSL]](logLevel: LogLevel, msg: String) =
    Program.inject(LogDSL.Log(logLevel, msg))
```

In questo modo è possibile utilizzare entrambi gli insiemi di istruzioni in un unico programma ottenuto tramite la for comprehension e i vincoli imposti dai singoli costruttori si accumuleranno definendo una lista di DSL che il programma può utilizzare:

```
def echo[I[_]: With[ConsoleDSL]: With[LogDSL]]
  : Program[I, Unit] =
  for
    line <- Console.getLine
    _ <-
      if (line == "quit")
      then Log.log(LogLevel.Info, "quitting")
      else
        Console.printLine(line)
          >> Log.log(LogLevel.Info, f"echoed a line")
          >> echo
  yield ()
```

Il programma `echo` può utilizzare un qualunque insieme di istruzioni `I` purché permetta di utilizzare le istruzioni di base del `LogDSL` e del `ConsoleDSL`. Per poter interpretare ed eseguire il programma è necessario un ultimo passo: concretizzare il linguaggio `I` in un tipo specifico che rispetti i vincoli imposti.

### Concretizzazione di un linguaggio generico

Per permettere di combinare fra loro più set di istruzioni può essere conveniente definire un coprodotto[3] che, combinando due linguaggi, permette di avere istruzioni provenienti dall'uno o dall'altro [26]. Sfruttando il supporto diretto agli *union type* [60] di Scala 3 è possibile utilizzare la seguente definizione:

[3] Un coprodotto è spesso indicato anche col nome di *sum type* o unione disgiunta. In Scala 3 può essere ottenuto utilizzando `sealed trait` e `case class` o delle enumerazioni.

```
type :|[F[_], G[_]] = [A] =>> F[A] | G[A]
```

Grazie a questo type alias è piuttosto semplice definire linguaggi composti da un numero arbitrario di DSL:

```
enum LogDSL[A]: ...
enum ConsoleDSL[A]: ...
enum FileSystemDSL[A]: ...

type ComposedDSL[A] =
  (LogDSL :| ConsoleDSL :| FileSystemDSL)[A]
val program: Program[ComposedDSL, Int] = ...
```

Un programma definito tramite il linguaggio `ComposedDSL` può utilizzare le istruzioni provenienti da ciascuno dei componenti di base dell'unione. Infatti, espandendo i diversi type alias il tipo `ComposedDSL` non sarà altro che una serie di opzioni rappresentate da uno union type:

```
ComposedDSL[A] =
= (LogDSL :| ConsoleDSL :| FileSystemDSL)[A]
= ((LogDSL :| ConsoleDSL) :| FileSystemDSL)[A]
= (LogDSL :| ConsoleDSL)[A] | FileSystemDSL[A]
= (LogDSL[A] | ConsoleDSL[A]) | FileSystemDSL[A]
= LogDSL[A] | ConsoleDSL[A] | FileSystemDSL[A]
```

Il programma `echo` mostrato in precedenza potrebbe usare come linguaggio concreto `ConsoleDSL :| LogDSL`. Non rimane che trovare un modo per generare in automatico le istanze di `Interpreter` necessarie per inserire le operazioni dei singoli DSL all'interno del linguaggio composto.

Una prima osservazione fondamentale sta nel fatto che un linguaggio `F` può sempre essere interpretato in un linguaggio `F :| G`:

```
given left[F[_], G[_]]: (F ~> (F :| G)) with
  def apply[A](f: F[A]) = f
```

Una qualunque istruzione di tipo `F[A]` è automaticamente un sottotipo di `(F :| G)[A]`; infatti, `(F :| G)[A]` equivale all'unione `F[A] | G[A]`.

La sola istanza `left` non è sufficiente per generare in automatico gli interpreti necessari a combinare più linguaggi. Infatti permette solo di inserire `F` in un'unione che contiene `F` come primo elemento. Una generalizzazione di quest'istanza permette di inserire `F` in un'unione il cui primo elemento possa a sua volta contenere `F`:

```
given leftProduct[F[_], G[_], H[_]](using
  T: F ~> G
): (F ~> (G :| H)) with
  def apply[A](f: F[A]) = T(f)
```

Dati tre linguaggi `F`, `G` e `H` se è possibile interpretare `F` nel linguaggio `G` allora è anche possibile definire un interprete per `F` nel linguaggio `G :| H`.

Questa coppia di definizioni è sufficiente perché il compilatore possa automaticamente generare qualunque istanza di interprete per linguaggi ottenuti componendo un numero arbitrario di DSL tramite il combinatore `(:|)`.

Al Listato 5.2 è mostrato come sia possibile definire un interprete che permette di eseguire un programma con un set di istruzioni composto da `ConsoleDSL` e `LogDSL`.

```
@tailrec def interpret[A](
  program: Program[LogDSL :| ConsoleDSL, A]
): A =
  program.next match
    case ProgramView.Return(a) => a
    case ProgramView.Then(instruction, continuation) =>
      instruction match
        case ConsoleDSL.PrintLine(msg) =>
          println(msg)
          interpret(continuation(()))
        case ConsoleDSL.GetLine() =>
          val line = scala.io.StdIn.readLine()
          interpret(continuation(line))
        case LogDSL.Log(logLevel, msg) =>
          println(f"[$logLevel] $msg")
          interpret(continuation(()))
```

Listato 5.2: Esempio di interpretazione di un programma composto da più DSL. Il pattern matching sulle istruzioni permette di gestire istruzioni provenienti da entrambi i DSL di base.

## 5.4 Free monad come effect system

Il meccanismo delle free monad introdotto in questo capitolo è sicuramente un esempio di effect system in quanto permette di descrivere e interpretare effetti arbitrati. Inoltre, è stato mostrato come sia possibile modellare separatamente e comporre fra loro DSL che descrivono effetti pertinenti a domini differenti.

I programmi realizzati tramite l'uso di questo meccanismo non sono altro che strutture dati che descrivono in maniera astratta quali effetti devono avere luogo. L'interpretazione degli effetti avviene in un secondo momento e può assegnare un significato alle diverse istruzioni in base alle necessità del contesto in cui si sta lavorando.

### 5.4.1 Definizione di effetti arbitrari

Anche in questo caso, così come per l'approccio MTL, è possibile definire qualunque tipo di effetto sia ritenuto rilevante ai fini della computazione; la scelta del livello di dettaglio con cui questi vengono descritti è lasciato a discrezione dell'autore del DSL.

In seguito viene ripreso l'esempio mostrato per MTL nella Sezione 4.3.1 reimplementandolo in maniera analoga nello stile delle free monad. Gli effetti da modellare sono il recupero di un utente, il salvataggio e la cancellazione:

```
enum UserStoreDSL[A]:
  case Get(userId: UserId) extends UserStoreDSL[Option[User]]
  case Save(user: User) extends UserStoreDSL[Unit]
  case Delete(userId: UserId) extends UserStoreDSL[Unit]
```

Il DSL è analogo a quanto definito per MTL; una differenza importante sta nella necessità, nel caso delle free monad, di dover definire dei costruttori per ogni operazione del DSL:

```
object UserStore:
  def get[I[_]: With[UserStoreDSL]](userId: UserId) =
    Program.inject(Get(userId))

  def save[I[_]: With[UserStoreDSL]](user: User) =
    Program.inject(Save(user))

  def delete[I[_]: With[UserStoreDSL]](userId: UserId) =
    Program.inject(Delete(userId))
```

Come già mostrato in precedenza, è possibile codificare programmi complessi in termini delle operazioni di base. La logica applicativa sarà basata sull'interfaccia astratta definita dall'insieme di operazioni che compongono il DSL:

```
def updateOrDelete[I[_]: With[UserStoreDSL]]
  (user: User)(f: User => User | Delete): Program[I, Unit] =
  f(user) match
    case updatedUser: User => UserStore.save(updatedUser)
    case _: Delete         => UserStore.delete(user.id)
```

L'implementazione è essenzialmente identica a quella mostrata per l'esempio di MTL; l'unica sostanziale differenza sta nel tipo di ritorno delle funzioni. Da un lato MTL richiede di definire una generica monade `M` all'interno della quale sarà incapsulato il valore di ritorno; l'approccio delle free monad, invece, restituisce un valore di tipo `Program` parametrizzato su un generico DSL `I`. L'implementazione di `Program` fa si che questo sia automaticamente una monade; la firma del metodo non dovrà quindi occuparsi di specificare alcun vincolo di questo tipo.

Nell'approccio MTL è necessario indicare esplicitamente il rispetto dell'interfaccia di monade per poter mettere in sequenza le diverse operazioni tramite l'uso di

`flatMap`. In questo caso, invece, una volta che un'operazione è stata incapsulata all'interno di `Program` il programmatore guadagna automaticamente la possibilità di combinarla in sequenza con altri programmi.

### Interpretazione e testing dei programmi

Potendo definire interpreti arbitrari per le istruzioni di un programma è piuttosto facile realizzare dei *mock* che permettono di testare effetti complessi senza dover complicare significativamente il processo di test della logica applicativa.

A differenza dell'approccio MTL, l'interpretazione può essere definita come un'analisi della sequenza di istruzioni del programma. In questo caso, quindi, non è necessario introdurre esplicitamente il concetto di monade che rimane un dettaglio implementativo sfruttato da `Program`.

Nel caso di MTL, poiché il vincolo di monade è reso esplicito all'interno della firma del metodo, per implementare un interprete è necessario avere un tipo di dato che rispetti l'istanza di monade introducendo la complessità di dover lavorare esplicitamente con i monad transformer. In questo caso, invece, l'interpretazione può essere implementata come una semplice funzione ricorsiva che di passo in passo assegna semantica all'istruzione corrente e interpreta la continuazione.

Si consideri per esempio la funzione `updateAge` già mostrata nel caso di MTL:

```
def updateAge[I[_]: With[UserStoreDSL]](userId: UserId) =
  updateOrDelete(userId) { user =>
    if user.age < 18
    then Delete
    else user.copy(age = user.age + 1)
  }
```

Al Listato 5.3 è possibile osservare l'implementazione di un interprete che implementa il mock di un database in memoria e testa la corretta implementazione della logica applicativa.

```
extension [A](program: Program[UserStoreDSL, A])
  @tailrec
  def runMocked(users: Map[UserId, User]): Map[UserId, User] =
    program.next match
      case ProgramView.Return(value) => users
      case ProgramView.Then(instruction, continuation) =>
        instruction match
          case Get(userId) =>
            continuation(users.get(userId)).runMocked(users)
          case Save(user) =>
            val updatedUsers = users + ((user.id, user))
            continuation(()).runMocked(updatedUsers)
          case Delete(userId) =>
            continuation(()).runMocked(users - userId)

def testUpdateAgeDeletesUnderageUsers: Unit =
  val user = User(UserId(1), "Giacomo", 12)
  val users = Map(user.id -> user)
  val finalUsers = updateAge(user.id).runMocked(users)
  assert(finalUsers.isEmpty)
```

Listato 5.3: Esempio di un interprete di test per un programma che usa lo il DSL per l'accesso a un database di utenti. In questo caso l'uso di una mappa degli utenti permette di testare semplicemente la logica applicativa delle operazioni senza dover ricorrere a un vero e proprio database

# Capitolo 6

# Effetti Algebrici

Tutti gli approcci mostrati fino ad ora fanno uso del concetto di monade basandosi sulla fondamentale osservazione che una computazione può essere modellata tramite un'opportuna monade per descriverne in maniera pura i side effect. Partendo dalla base teorica del lavoro di Moggi, Haskell è stato il primo linguaggio a introdurre il concetto di monade come elemento chiave per la gestione dei side effect.

A dimostrazione dell'efficacia di questo approccio, sono state sviluppate librerie ed estensioni per poter sfruttare la programmazione monadica anche in altri linguaggi come OCaml [61], Scala [44, 62] e Kotlin [63].

Tuttavia, il concetto di monade non è l'unico approccio possibile per modellare i side effect: in particolare, gli *effetti algebrici* sono un'alternativa che sta ricevendo grande attenzione. Questi permettono – come nel caso delle monadi – di modellare effetti come le eccezioni, il non determinismo, e la mutabilità[31].

Inoltre, è stato mostrato come questo meccanismo sia una generalizzazione di un'ampia gamma di costrutti di programmazione per il controllo di flusso: eccezioni, iteratori, *async-await* [32], *coroutines*, e *green thread* [7].

Non sorprende dunque che ci sia un crescente interesse per l'adozione di questo meccanismo:

- Nello standard WebAssembly è in corso la valutazione se adottare gli effetti algebrici come meccanismo unificante per la compilazione efficiente di meccanismi di controllo come le eccezioni e i generatori [43]
- Sono state realizzate diverse librerie sia in Scala [64] che in Haskell [65–67] per poter utilizzare gli effetti algebrici in alternativa all'approccio MTL
- Nella sua versione 5.0, OCaml supporta nativamente gli effetti algebrici [33]
- Sono stati sviluppati numerosi linguaggi di ricerca per esplorare l'implementazione degli effetti algebrici: Eff [68], Effekt [69], Koka [70] e Unison [71]

## 6.1 Il linguaggio Koka

Prima di poter proseguire nella trattazione degli effetti algebrici è necessario dare un'introduzione al linguaggio di riferimento utilizzato per gli esempi che saranno riportati: Koka[70].

Koka è un linguaggio di ricerca con supporto agli effetti algebrici; alcune delle sue caratteristiche chiave sono:

- Tipizzazione statica e *type inference*
- Supporto agli effetti algebrici e *effect inference* per minimizzare la necessità di annotazioni
- Un cuore funzionale composto da pochi costrutti quanto più generali e componibili possibile: funzioni di prima classe, ADT, effetti algebrici, e *handler*
- Uso del metodo Perceus [34] per gestire automaticamente la memoria tramite *reference counting*. Questa tecnica elimina il bisogno di avere un *garbage collector* o un runtime system ottenendo ottime prestazioni. Inoltre, l'uso di Perceus rende possibile programmare secondo uno stile che i suoi autori definiscono *Functional But In Place*: grazie all'analisi del *reference counter* è possibile ottimizzare algoritmi puramente funzionali – che utilizzano strutture dati immutabili – in modo tale da eseguire mutazioni *in place* in maniera trasparente al programmatore [72]

### 6.1.1 Sintassi di base

#### Dichiarazione di funzioni e zucchero sintattico

Per dichiarare una funzione viene utilizzata la parola chiave `fun`, mentre per dichiarare una funzione anonima la parola chiave da utilizzare è `fn`:

```
fun main() {
  repeat(10, fn() {
    println("Hello, Koka!")
  })
}
```

Nell'esempio viene chiamata la funzione `repeat` con due argomenti: una funzione anonima che descrive l'azione da compiere e un numero che indica quante volte ripetere l'azione.

La sintassi Koka è molto simile a quella di linguaggio come C e Java; tuttavia, il linguaggio adotta una serie di regole e zucchero sintattico per rendere la sintassi più concisa.

In particolare, secondo la regola della *brace elision*, ogni blocco di codice indentato viene automaticamente considerato come compreso fra parentesi graffe. Dunque, come in linguaggi come Python e Haskell, gli spazi bianchi sono significativi.

Sfruttando questo zucchero sintattico è possibile riscrivere l'esempio come:

```
fun main()
  repeat(10, fn()
    println("Hello, Koka!")
  )
```

Inoltre, in Koka – così come in Swift e Kotlin – è disponibile zucchero sintattico per semplificare la scrittura delle *trailing lambda*: qualora l'ultimo argomento di una funzione sia una funzione anonima, questa può seguire direttamente la chiamata a funzione.

L'esempio precedente può quindi essere riscritto come:

```
fun main()
  repeat(10) fn()
    println("Hello, Koka!")
```

Infine, se la funzione anonima non presenta argomenti allora si può omettere di scrivere `fn()`:

```
fun main()
  repeat(10)
    println("Hello, Koka!")
```

Grazie a queste regole è semplice definire funzioni che appaiano simili a *keyword* del linguaggio e costrutti di controllo di flusso che normalmente richiederebbero un supporto diretto del compilatore. Per esempio la funzione `for` potrebbe essere implementata come:

```
fun for(start: int, end, action)
  if start > end then () else
    action(start)
    for(start + 1, end, action)

fun main()
  for(1, 10, fn(i) { println(i) })
```

Poiché Koka attua *Tail Call Optimization* il codice generato sarà stack safe e simile a quello di un classico ciclo `for`.

### La parola chiave `with`

Un'altra regola sintattica che verrà utilizzata frequentemente negli esempi a seguire è l'uso della parola chiave `with`. Questa permette di eliminare l'annidamento di funzioni che prendono in input delle funzioni anonime:

```
with x <- f(a1,...,an)
<body>
```

Viene trasformata in `f(a1,...,an, fn(x){ <body> })`. Inoltre, se la funzione anonima non prende argomenti, è possibile omettere `x <-`.

Perciò, l'esempio precedente può essere riscritto come:

```
fun main()
  with repeat(10)
  println("Hello, Koka!")
```

Nel caso della funzione `for` mostrata in precedenza la parola chiave `with` può essere utilizzata come segue:

```
fun main()
  with i <- for(1, 10)
  println(i)
```

#### Dot selection

In maniera analoga a Go, in Koka è possibile usare la sintassi generalmente associata alla chiamata di un metodo per poter invocare una funzione: `a1.f(a2,...,an)` non è altro che zucchero sintattico per `f(a1,a2,...,an)`. Si osservi per esempio la seguente porzione di codice:

```
fun main()
  println(show(map([1, 2, 3], fn(i) { i + 1 })))
```

Può essere riscritta come:

```
fun main()
  [1, 2, 3].map(fn(i) { i + 1 }).show.println
```

### 6.1.2 Tipi di dato algebrici e pattern matching

Koka permette di definire ADT, l'equivalente delle enumerazioni Scala, tramite la parola chiave `type` e separando i diversi costruttori su linee differenti:

```
type color
  Red
  Green
  Blue
```

Inoltre un ADT può essere parametrizzato su tipi generici e i suoi costruttori possono avere campi differenti l'uno dall'altro. Per esempio si potrebbe definire un tipo di dato opzionale tramite la seguente dichiarazione:

```
type optional<a>
  Some(value: a)
  Empty
```

Avendo a disposizione un valore di tipo `optional`, questo potrebbe essere uno qualunque dei due costruttori. L'unico modo per poter determinare quale dei costruttori sia e accedere ai suoi campi è tramite pattern matching.

Si consideri per esempio la funzione `map` che, preso in input un valore opzionale e una funzione `f`, la applica al suo contenuto se questo è definito:

```
fun map(option, f)
  match option
    Some(value) -> Some(f(value))
    Empty       -> Empty

fun main()
  val option = Some(1)
  option.map(fn(i) { i + 2 }) // Some(3)
```

Si noti come non è stato necessario annotare il tipo dei parametri della funzione `map`, nè il suo tipo di ritorno: l'algoritmo di inferenza di Koka si basa sul type system di Hindley e Milner ed è quasi sempre possibile inferire automaticamente il tipo più generico possibile per le funzioni senza necessità di inserire annotazioni.

In questo caso il tipo della funzione viene inferito essere `forall<a,b> (option : optional<a>, f : (a) -> b) -> optional<b>`.

### 6.1.3 Effect system

Koka dispone anche di un effect system basato sui *row types* [35] che permette di tracciare staticamente tutti i side effect che può avere una funzione.

In generale una funzione avrà come tipo $(\tau_1, \ldots, \tau_n) \to \langle \epsilon_1, \ldots, \epsilon_m \rangle\ \tau'$ a indicare che prende come argomenti i valori di tipo $\tau_1, \ldots, \tau_n$, produce un valore $\tau'$ e può avere i side effect $\epsilon_1, \ldots, \epsilon_m$.

Per esempio, in Koka, la funzione `println` ha tipo `(string) -> <console> ()`: prende in input una stringa, produce un valore `unit` e ha il solo side effect di interagire con la console.

Una funzione senza side effect avrà come tipo $(\tau_1, \ldots, \tau_n) \to \langle\rangle\ \tau'$ che viene abbreviato in $(\tau_1, \ldots, \tau_n) \to\ \tau'$. Per esempio la funzione `show` ha tipo `(int) -> string` e non presenta alcun side effect.

#### Propagazione e inferenza degli effetti

L'unico momento in cui può verificarsi un side effect è quando una funzione viene invocata. Per questo, se una funzione ha side effect $\epsilon_1, \ldots, \epsilon_m$ questi possono avere luogo solo se la funzione è invocata.

Se una funzione invoca una funzione con side effect allora produrrà i suoi effetti e di conseguenza il suo tipo deve rispecchiare tale comportamento. Si consideri per esempio:

```
fun print-twice()
  println("Hello")
  println("World!")
```

La funzione print-twice invoca due volte la funzione println; poiché questa funzione ha l'effetto di interagire con la console allora anche print-twice avrà lo stesso effetto: il tipo inferito per la funzione infatti è () -> <console> ().

Inoltre, è possibile definire funzioni generiche sullo specifico tipo di effetto che possono avere. Si consideri per esempio la funzione repeat utilizzata in precedenza: uno dei suoi argomenti è una funzione da ripetere un determinato numero di volte. Questa funzione potrebbe presentare qualunque tipo di side effect e il comportamento desiderato è che repeat ripeta più volte gli effetti della funzione.

L'implementazione di repeat potrebbe essere la seguente:

```
fun repeat(n, action)
  if n <= 0 then () else
    action()
    repeat(n - 1, action)
```

In questo caso l'azione deve essere una funzione generica sui side effect che può compiere: il suo tipo sarà forall<e, a> () -> e a.

Poiché repeat invoca la funzione action nel proprio corpo allora avrà i suoi stessi side effect. Complessivamente il tipo inferito sarà: forall<a,e> (n : int, action : () -> e a) -> e (). È reso esplicito nel tipo della funzione che, se questa viene invocata con un'azione che presenta dei side effect e, allora avrà gli stessi effetti.

### Accumulazione degli effetti

Se una funzione utilizza al proprio interno funzioni che possono avere side effect differenti, questi sono automaticamente accumulati nel tipo risultante. Si consideri il seguente esempio:

```
fun log-action(action)
  println("invoking action")
  action()
  println("action invoked")
```

log-action prende in input una funzione che potrebbe avere presentare qualunque side effect, stampa un messaggio ed esegue la funzione.

Complessivamente, log-action deve presentare non solo tutti i possibili side effect della generica funzione presa come parametro ma anche il side effect dell'interazione con la console dovuto all'uso di println.

Il tipo inferito sarà quindi forall<a,e> (action: () -> e a) -> <console|e> (). È possibile osservare come la funzione include nei propri effetti sia il concreto effetto console che tutti i possibili generici effetti e del proprio parametro. La notazione

`<e1|e>` serve a indicare che una funzione può avere tutti gli effetti `e1` e anche tutti i generici effetti `e`.

## 6.2 Idea alla base degli effetti algebrici

Nella teoria elaborata da Plotkin e Power, un effetto viene modellato tramite un insieme di operazioni e una serie di equazioni che ne descrivono le proprietà [36, 37].

Le operazioni di un effetto determinano le modalità con cui questo può verificarsi; possono quindi essere considerate come dei costruttori di un effetto [38]. Per poter interpretare un effetto e assegnarvi semantica vengono invece utilizzati degli appositi handler, o distruttori. Questi specificano come l'esecuzione del programma deve proseguire al verificarsi di ciascuna operazione di uno specifico effetto.

### 6.2.1 Modellazione degli effetti

#### Fallimento di una computazione

In Koka la parola chiave `effect` definisce un effetto e i suoi costruttori sono definiti tramite la parola chiave `ctl`.

Si consideri l'esempio già utilizzato del fallimento di una computazione: questo effetto può essere ottenuto con una sola operazione che sarà chiamata `fail`:

```
effect failure
  ctl fail(): a
```

La dichiarazione permette di definire un effetto chiamato `failure` con una sola operazione `fail()`. Questa operazione non prende in input alcun valore e restituisce un valore di un qualsiasi tipo. In automatico sarà generata una funzione `fail` con tipo `forall<a> () -> failure a`.

Grazie all'inferenza dell'effect system, una funzione che fa uso di `fail` nel proprio corpo avrà automaticamente fra i propri effetti quello del fallimento:

```
fun div(n, m)
  match m
    0 -> fail()
    _ -> n / m
```

Il tipo inferito per `div` sarà `(n : int, m : int) -> failure int`. Semplicemente osservando il tipo della funzione è possibile capire che questa potrebbe fallire; l'effect system fa in modo che non sia possibile ignorare questo aspetto.

#### Lettura e scrittura di uno stato mutabile

Seguendo la stessa strategia evidenziata per l'effetto del fallimento è possibile definire un effetto per modellare la presenza di uno stato mutabile:

```
effect state<s>
  ctl get(): s
  ctl set(new-state: s): ()
```

In questo caso l'effetto viene definito rispetto a un generico tipo `s` che rappresenta il tipo dello stato mutabile.

- `get` avrà come tipo `() -> state<s> s`: una sua invocazione restituisce un valore di tipo `s` e ha il side effect di accedere a uno stato mutabile
- `set` avrà come tipo `(new-state: s) -> state<s> ()`: prende in input il nuovo stato di tipo `s` e non restituisce alcun valore, inoltre ha il side effect di accedere a uno stato mutabile `s`

Queste operazioni di base possono essere combinate per realizzare funzioni più complesse, per esempio:

```
fun update(f)
  val current-state = get()
  val new-state = f(current-state)
  set(new-state)
```

In questo caso la funzione prende in input una funzione `f` che viene utilizzata per trasformare lo stato e aggiornarlo.

Poiché `update` utilizza al proprio interno le operazioni `get` e `set` avrà nel proprio tipo l'effetto dello stato: il tipo inferito è `(f : (s) -> e s) -> <state<s>|e> ()`.

Si noti come il tipo di `f` è stato inferito come il più generico possibile: vale a dire che questa funzione potrebbe avere un qualunque side effect `e`. Per questo motivo il tipo di `update` combina sia l'effetto dello stato che i possibili ulteriori effetti provenienti dall'invocazione di `f`.

### 6.2.2 Interpretazione degli effetti

Fino a questo momento gli effetti sono stati descritti in maniera astratta definendo una serie di operazioni; queste definiscono un'interfaccia attraverso la quale è possibile generare l'effetto desiderato.

Per poter dare una semantica alle operazioni di un effetto è necessario definire un handler. Gli handler, detti anche distruttori, sono un meccanismo duale rispetto alle operazioni: mentre l'invocazione di quest'ultime arricchisce il tipo di una funzione aggiungendovi effetti, gli handler permettono di "scaricare" gli effetti dal tipo di una funzione.

In generale, un handler in grado di dare semantica alle operazioni dell'effetto `e1` sarà una funzione che prende in input una computazione con tale effetto e lo rimuove dalla lista dei suoi effetti.

### Interpretazione degli effetti come eccezioni

Per poter definire un handler in Koka viene utilizzata la parola chiave `handle` a cui viene passata come argomento la computazione della quale devono essere gestiti gli effetti.

Il corpo di un handler viene definito tramite pattern matching sulle possibili operazioni dell'effetto che si sta interpretando.

Operativamente, quando viene invocata un'operazione il risultato è simile al sollevamento di un'eccezione in un linguaggio come Java o Scala: il normale flusso di controllo viene interrotto e il controllo passa all'handler più vicino in grado di gestire lo specifico effetto.

Per esempio, considerando la funzione `div` mostrata in precedenza, una possibile gestione del fallimento potrebbe essere la seguente:

```
fun main()
  handle(fn() div(10, 0).println)
    ctl fail() -> println("Exception: divided by zero")
```

Nel caso in cui venga chiamata l'operazione `fail` il controllo della computazione sarà interrotto e l'handler eseguirà la porzione di codice associata all'operazione `fail`.

In questo caso un blocco `handle(...){...}` può essere assimilato al blocco `try-catch` di linguaggi che supportano le eccezioni. Per esempio, si potrebbe gestire il fallimento restituendo un valore di default in caso di eccezione:

```
fun default(value, computation)
  handle(computation)
    ctl fail() -> value
```

Il tipo di `default` è `forall<a,e> (value : a, computation : () -> <failure|e> a) -> e a`:

- La computazione restituisce un valore di tipo `a` e potrebbe avere, fra gli altri effetti, anche quello del fallimento
- `value` è un valore di default con lo stesso tipo del valore di ritorno di `computation` che viene restituito in caso di fallimento
- Il risultato non ha più l'effetto del fallimento fra i propri effetti in quanto tutte le sue operazioni (in questo caso solo `fail`) sono state gestite nell'handler. Rimangono comunque i possibili altri effetti `e` della computazione in quanto l'handler gestisce solo le operazioni del fallimento

In caso si verifichi l'effetto del fallimento la computazione viene interrotta e l'handler restituisce immediatamente il valore di default; in caso contrario il risultato sarà quello restituito normalmente dalla computazione:

```
fun main()
  default(-1, { div(10, 0) }).println // -1
  default(-1, { div(10, 2) }).println // 5
```

### Cattura del risultato di una computazione

Un handler può anche gestire esplicitamente il caso in cui una computazione termini normalmente senza chiamare alcuna operazione.

In Koka questo viene fatto aggiungendo un ramo al pattern matching:

```
fun to-option(computation)
  handle(computation)
    return(value) -> Some(value)
    ctl fail()     -> Empty
```

Il ramo `return(value)` viene utilizzato se la computazione gestita dall'handler termina senza chiamare l'operazione di `fail`; in questo caso alla variabile `value` viene assegnato il valore di ritorno della computazione.

La funzione quindi permette di rimuovere l'effetto del fallimento di una funzione trasformando il suo valore di ritorno in un valore opzionale:

```
fun main()
  to-option { div(10, 0) } // Empty
  to-option { div(10, 2) } // Some(5)
```

Se non viene specificato il caso `return` – come negli esempi mostrati in precedenza – allora l'handler restituisce semplicemente il valore della computazione.

### Definizione di strutture di controllo

Grazie a questo meccanismo è possibile definire funzioni più complicate.

Molti linguaggi di programmazione forniscono, tramite supporto diretto del compilatore, le eccezioni come meccanismo per far terminare prematuramente una computazione. Inoltre, è possibile definire dei blocchi `finally` per stabilire delle azioni che devono essere intraprese quando una computazione dentro ad un blocco `try-catch` termina, con un'eccezione o normalmente.

Avendo a disposizione gli effetti è possibile definire blocchi `finally` come semplici funzioni di libreria senza bisogno di aggiungere supporto speciale del compilatore o nuove parole chiave al linguaggio:

```
fun finalizer(fin, computation)
  handle(computation)
    return(_)  -> ()
```

```
    ctl fail() -> ()
  fin()
```

La computazione viene gestita dall'handler il quale ignora il valore di ritorno e, in caso di eccezione, restituisce `()`. In entrambi i casi, dopo la gestione della computazione viene eseguita la funzione di finalizzazione:

```
fun main()
  with finalizer { print("Finalizer") }
  print("Start,")
  print(div(10, 0).show ++ ",")
  print("End")
```

In questo caso l'output sarà `"Start,5,Finalizer"` in quanto l'operazione di divisione per 0 comporta il fallimento della computazione.

### 6.2.3 Interpretazione degli effetti come eccezioni ripristinabili

Si consideri adesso l'esempio dello stato: un handler per l'effetto `state` dovrà stabilire come dare semantica alle operazioni di `get` e `set`.

In questo caso, pensare agli effetti come semplici eccezioni risulta essere limitante. Per capire il problema, si osservi la seguente computazione che ricorre all'effetto dello stato:

```
fun increment-counter()
  val counter: int = get()
  println("counter is " ++ counter.show)
  set(counter + 1)
```

Se la chiamata all'operazione `get` si limitasse a interrompere l'esecuzione della funzione, non sarebbe possibile eseguire i comandi successivi. Invece, il comportamento desiderato sarebbe quello di ottenere lo stato corrente, stampare un messaggio e modificare lo stato con un nuovo valore.

In realtà, gli effetti sono un meccanismo più generale delle semplici eccezioni. Infatti, non solo possono permettere di interrompere il normale flusso d'esecuzione del codice (come mostrato negli esempi per il side effect del fallimento) ma possono anche ripristinare il flusso d'esecuzione nel punto in cui è stata invocata un'operazione.

#### Ripristino del flusso di controllo

Quando viene chiamata l'operazione di un effetto e il flusso di controllo viene passato all'handler, questo ha la possibilità di far riprendere la normale esecuzione della funzione utilizzando la funzione `resume`.

Il tipo della funzione `resume` cambierà in base all'effetto invocato. Si consideri per esempio l'effetto di leggere valori di configurazione:

```
effect configuration
  ctl get-name(): string
  ctl get-int(): int

fun print-config()
  val name = get-name()
  println("Name is: " ++ name)
  val n = get-int()
  println("Int is: " ++ n.show)
```

Quindi, quando si verifica l'effetto `get-name`, per poter riprendere la normale computazione è necessario fornire in un qualche modo una stringa che sarà il valore di ritorno atteso dalla funzione `get-name`. In maniera analoga, quando si verifica l'effetto `get-int` sarà necessario fornire un valore intero per poter far riprendere la computazione.

Un handler per questo effetto potrebbe, per esempio, fornire sempre gli stessi valori:

```
fun constant-config(computation)
  handle(computation)
    ctl get-name() -> resume("Koka")
    ctl get-int()  -> resume(11)
```

A seconda del ramo del pattern matching sarà presente in scope una funzione `resume` con tipo differente: nel primo caso richiede una stringa mentre nel secondo un valore intero. L'effetto in entrambi i casi è quello di riprendere il normale flusso di esecuzione dando come valore di ritorno dell'operazione il valore passato alla funzione `resume`:

```
fun main()
  with constant-config
  print-config()
  // Name is: Koka
  // Int is: 11
```

### Handler per lo stato mutabile

Tornando all'esempio dello stato mutabile, è possibile osservare come, per riprendere la computazione in seguito ad una chiamata a `get`, sia necessario fornire un valore del tipo dello stato. Un possibile handler potrebbe essere il seguente:

```
fun initial-state(initial-state, computation)
  var state := initial-state
  handle(computation)
    return(x) -> (state, x)
```

```
    ctl get() -> resume(state)
    ctl set(new-state) ->
      state := new-state
      resume(())
```

La funzione utilizza internamente una variabile per tenere traccia dello stato attuale: quando viene chiamato `get` l'esecuzione viene ripresa con il valore attuale dello stato; invece alla chiamata di `set` si modifica lo stato attuale e poi viene fatta ripartire la computazione fornendo alla funzione `resume` il valore `()` – vale a dire il tipo di ritorno atteso dalla funzione `set`.

### Uso delle variabili in Koka

È possibile osservare come, per poter implementare l'handler per lo stato mutabile, si sia fatto uso di una variabile. Questo potrebbe apparire strano data la natura funzionale di Koka, tuttavia il linguaggio introduce la possibilità di utilizzare variabili con alcune limitazioni che permettono di mantenere il codice puro.

La scelta di Koka è quella di permettere la definizione di variabili all'interno del corpo di una funzione a condizione che queste non escano dallo scope lessicale della funzione all'interno della quale sono dichiarate. Per esempio il seguente codice risulterebbe in un errore da parte del compilatore:

```
fun variable()
  var x:= 0
  fn() { x := 1; x }
  // error: a reference to a local variable
  // escapes its scope
```

Le variabili non sono elementi di prima classe e non possono essere passate come argomenti alle funzioni o restituite come risultato di una funzione; ogni riferimento a una variabile equivale automaticamente a leggerne il contenuto:

```
fun variables()
  var x := 0
  println(show(x)) // Read value referenced by x
  x := x + 1
  x // Returns the immutable value referenced by x
```

Se, come nell'esempio, viene restituita una variabile in realtà si restituisce il suo contenuto immutabile e non un riferimento mutabile.

Questa limitazione permette di usare le variabili in maniera opportunistica all'interno del corpo di una funzione per poterla implementare in maniera efficiente. Tuttavia, l'uso di variabili non è mai osservabile dall'esterno della funzione che continua a comportarsi come una funzione pura. Per esempio si potrebbe realizzare un'implementazione efficiente per calcolare l'n-esimo numero della sequenza di Fibonacci:

```
fun fibonacci(n)
  var x := 0
  var y := 1
  repeat(n)
    val temp = y
    y := x + y
    x := temp
  x
```

I controlli del type system di Koka garantiscono che le variabili non possano sopravvivere oltre lo scope sintattico della funzione all'interno della quale sono dichiarate. Per questo la funzione `fibonacci`, nonostante la mutazione di una variabile, sarà comunque considerata una funzione pura con tipo `(int) -> <> int`.

## 6.3 Definizione di effetti arbitrari

Il meccanismo degli effetti algebrici è sufficientemente generale da permettere di definire effetti di qualunque tipo per modellare la logica applicativa secondo il livello di dettaglio più adatto.

Si consideri un'ultima volta l'esempio dell'accesso a un database riportato alle Sezioni 4.3.1 e 5.4.1, questo effetto può essere definito in Koka come:

```
effect user-store
  ctl get(id: user-id): optional<user>
  ctl save(user: user): ()
  ctl delete(id: user-id): ()
```

Queste operazioni possono essere direttamente utilizzate come normali funzioni e i tipi e gli effetti saranno inferiti automaticamente:

```
fun update-or-delete(user-id, f)
  match get(user-id)
    Nothing    -> ()
    Just(user) -> match f(user)
      Update(updated) -> save(updated)
      Delete          -> delete(user-id)

fun update-age(user-id)
  with user <- update-or-delete(user-id)
  if user.age < 18 then Delete else
    Update(user(age = user.age + 1))
```

Come ci si può aspettare, la funzione `update-age` avrà tipo `(user-id) -> user-store ()`. Se si provasse a invocare questa funzione direttamente nel `main`, il compilatore solleverebbe un'eccezione dato che l'effetto dello `user-store` non è stato gestito da un opportuno handler. Infatti, l'unico effetto che il `main` può avere è quello di effettuare input e output; qualunque altro effetto deve essere prima stato

gestito da un opportuno handler. In questo modo si ha la garanzia che non ci possano essere effetti che inavvertitamente sono stati ignorati.

Anche in questo caso diventa facile realizzare un handler che usi una lista di utenti in memoria per poter testare la logica applicativa della funzione `update-age`; per esempio:

```
fun mocked-users(initial-users, computation)
  fun has-id(id)
    fn (user) user.user-id.value == id.value

  var users := initial-users
  handle(computation)
    return(_) -> users
    ctl get(id) ->
      val user = users.find(has-id(id))
      resume(user)
    ctl save(user) ->
      users := users.remove(has-id(user.user-id))
      users := Cons(user, users)
      resume(())
    ctl delete(id) ->
      users := users.remove(has-id(id))
      resume(())
```

L'handler modifica internamente una lista di utenti in base alle operazioni dello `user-store`. Anche in questo caso l'utilizzo della variabile non è osservabile esternamente e la funzione è pura.

Questo permette di scrivere test completamente deterministici che non necessitano di collegarsi effettivamente ad alcun database:

```
fun test-update-age-removes-underage-users()
  val users = [User(User-id(1), "Koka", 17)]
  val final-users = mocked-users(users) { update-age(User-id(1)) }
  assert("underage user was removed", final-users.is-empty)
```

### 6.3.1 Meccanismi di controllo di flusso complessi

Gli effetti algebrici sono una generalizzazione delle eccezioni; come mostrato dagli esempi riportati, è stato possibile definire in maniera concisa meccanismi analoghi al `try-catch-finally` di linguaggi di programmazione come Java che invece richiederebbero un supporto specifico da parte del compilatore.

Inoltre, grazie all'utilizzo degli effetti algebrici è possibile implementare in user space altre tecniche di controllo di flusso complesse come generatori e concorrenza strutturata (nello stile *async-await*) [32].

Si consideri l'esempio dei generatori: numerosi linguaggi come JavaScript, TypeScript, Python e C# hanno introdotto la possibilità di definire generatori tramite le cosiddette *generator function* [73–75] e sintassi ad hoc.

Tuttavia, per poter utilizzare tale meccanismo, è richiesto un supporto diretto da parte del compilatore il quale dovrà trasformare il corpo delle generator function in macchine a stati [76]. Inoltre, nell'implementare tali trasformazioni, è fondamentale prestare attenzione al modo in cui potrebbero interferire con altri meccanismi come le eccezioni e l'asincronia.

Nel caso degli effetti non è necessaria alcuna estensione al linguaggio in quanto tutti questi meccanismi possono essere generalizzati da opportuni effetti algebrici e handler.

Il compilatore quindi dovrà supportare e ottimizzare un singolo meccanismo – quello degli effetti – e tutti gli altri potranno essere definiti sulla base di questo come librerie.

### Effetto del non determinismo

Per mostrare il potere espressivo degli effetti algebrici prenderemo in considerazione un ultimo esempio: quello del non determinismo.

L'effetto della scelta non deterministica può essere modellato come segue:

```
effect choice<a>
  ctl or(one: a, other: a): a
```

L'effetto `choice` rappresenta una computazione che può non deterministicamente scegliere una delle opzioni a disposizione. Per esempio, è possibile modellare il lancio di una moneta come:

```
type coin
  Heads
  Tails

fun flip()
  Heads.or(Tails)
```

Il tipo inferito per `flip` è `() -> choice<coin> coin`. Questa funzione può essere utilizzata per accumulare i risultati di lanci di più monete in una lista:

```
fun fill(n, action)
  if n <= 0 then [] else
    Cons(action(), fill(n - 1, action))

fun flips(times)
  fill(times) { flip() }
```

L'utilità di definire l'effetto del non determinismo in maniera astratta sta nella possibilità di poter scegliere la semantica più adatta allo specifico caso. In questo

caso si possono realizzare diversi handler; per esempio un primo handler potrebbe restituire un risultato costante:

```
fun constant-coin(coin: coin, computation)
  handle(computation)
    ctl or(_, _) -> resume(coin)
```

Questa funzione può essere utile per verificare il comportamento di funzioni che si basano sull'effetto del non determinismo e testarne i possibili percorsi di esecuzione in maniera deterministica.

Un handler più "tradizionale" potrebbe invece interpretare l'effetto della scelta restituendo casualmente una delle due opzioni:

```
fun real-coin(computation)
  handle(computation)
    ctl or(one, other) ->
      resume(if random-bool() then one else other)
```

In questo caso si usa la funzione di libreria `random-bool` che aggiunge l'effetto `ndet` che permette di sfruttare un generatore di numeri pseudocasuali predefinito. È interessante osservare come un handler possa aggiungere effetti alla computazione che sta gestendo, nell'esempio viene rimosso l'effetto `choice` sostituendolo con quello predefinito `ndet`.

Da questo punto di vista un handler può essere visto come una sorta di compilatore che traduce chiamate ad operazioni astratte in operazioni sempre più di basso livello fino ad ottenere una funzione eseguibile direttamente dal `main` (vale a dire con il solo effetto `io` che può essere gestito automaticamente dal compilatore).

### Ripristinare l'esecuzione più volte

In tutti gli esempi mostrati fino ad ora l'esecuzione della funzione che ha invocato un effetto è stata ripristinata al più una sola volta. Tuttavia, `resume` è una funzione e in quanto elemento di prima classe del linguaggio potrebbe essere invocata più volte, passata come argomento a una funzione, ecc.

Riprendendo l'esempio del non determinismo, è possibile ripristinare la computazione più volte con tutte le possibili alternative fra cui viene effettuata la scelta:

```
fun all-outcomes(computation)
  handle(computation)
    return(x) -> [x]
    ctl or(one, other) -> resume(one) ++ resume(other)
```

In questo caso la computazione viene ripristinata una prima volta utilizzando come valore il primo fra cui scegliere, viene poi ripristinata una seconda volta utilizzando il secondo valore. I risultati ottenuti sono poi concatenati fra loro.

Grazia alla possibilità di riprendere la computazione più volte con valori differenti è possibile accumulare tutti i possibili risultati del lancio di una moneta:

```
fun main()
  all-outcomes { 3.flips }
  // [
  //   [Heads, Heads, Heads],
  //   [Heads, Heads, Tails],
  //   [Heads, Tails, Heads],
  //   ...
  //   [Tails, Tails, Tails]
  // ]
```

Si noti la semplicità con cui è possibile cambiare l'handler per modificare la semantica delle operazioni senza cambiare in alcun modo la definizione delle operazioni di base come `flip` e `flips`.

## 6.4 Confronto con l'approccio monadico

Gli effetti algebrici rappresentano un'alternativa interessante all'utilizzo delle monadi per la modellazione dei side effect.

Nella definizione di una monade l'attenzione viene posta sulle operazioni di `pure` e `flatMap` dovendo indicare come può avvenire la concatenazione delle computazioni e sono queste definizioni a dare semantica alle diverse istruzioni.

L'approccio degli effetti, al contrario, si concentra direttamente sulla definizione e interpretazione delle operazioni tramite l'uso di handler. L'attenzione è posta su come interpretare ciascun effetto e non su come devono essere concatenate le operazioni. Sotto questo punto di vista, gli effetti algebrici presentano una forte similarità con le free monad: in entrambi i casi si definisce una serie di istruzioni che devono essere interpretate per dare semantica al programma.

Nel caso delle free monad il programmatore non deve definire esplicitamente il metodo `flatMap` – e quindi come concatenare le operazioni – che viene fornito automaticamente dall'uso del tipo `Program`. Anche in quel caso l'interpretazione di un programma avviene interpretandone le singole istruzioni.

### 6.4.1 Boilerplate ed inferenza

Un aspetto sotto il quale le free monad e gli effetti algebrici differiscono sta nella quantità di boilerplate necessaria per poter utilizzare ciascun approccio.

Nel caso delle free monad è necessario definire, per ciascuna delle operazioni di uno specifico linguaggio degli smart constructor. Si consideri nuovamente l'esempio dello stato mutabile:

```
enum StateDSL[S, A]:
  case Get[S]() extends StateDSL[S, S]
  case Set[S](s: S) extends StateDSL[S, Unit]

def get[S]: State[S, S] = Program.fromInstruction(Get())
def set[S](s: S): State[S, Unit] =
  Program.fromInstruction(Set(s))
```

Inoltre, è stato necessario prestare particolare attenzione nella realizzazione di un meccanismo che permettesse di usare insiemi di istruzioni differenti in una stessa funzione. Per rendere possibile questa combinazione è stato descritto il meccanismo delle iniezioni che, sfruttando il passaggio di istanze implicite, permettono di unire operazioni di più effetti in un linguaggio composto.

Nel caso di Koka la definizione di un effetto è minimale e non richiede alcun boilerplate:

```
effect state<s>
  ctl get(): s
  ctl set(new-state: s): ()
```

Inoltre, l'effect system permette di tracciare automaticamente tutti i side effect delle funzioni e non è necessario alcun passo aggiuntivo per poter combinare funzioni che presentano side effect differenti:

```
fun mixed-side-effects()
  println("Side effect!")
  fail()
```

Il tipo della funzione viene inferito come `() -> <console, failure> ()`. Il meccanismo è estremamente semplice da spiegare: tutti i side effect delle singole funzioni invocate dentro al corpo di una funzione sono accumulati nel suo tipo.

Si faccia il confronto con l'equivalente versione Scala che fa uso di free monad dove le annotazioni di tipo sono necessarie:

```
def mixedSideEffects[I[_]: With[ConsoleDSL]: With[FailDSL]]() =
  for
    _ <- println("Side effect!")
    _ <- fail()
  yield ()
```

Per poter comprendere l'esempio un programmatore deve essere a conoscenza di meccanismi del linguaggio non banali come il passaggio di istanze implicite, i context bound e gli higher-kinded type.

Il problema sorge dal fatto che linguaggi come Scala non sono stati pensati per tracciare gli effetti delle funzioni come elementi di prima classe nel proprio type system. È quindi necessario implementare un'infrastruttura – come MTL e free monad – in termini di funzionalità preesistenti del linguaggio che permetta di arricchirlo con le funzionalità desiderate. Per esempio, durante la trattazione si è

fatto uso estensivo di istanze implicite e passaggio di parametri implicito per poter codificare le monadi e vincolare gli effetti di una funzione.

Chiaramente, poiché il linguaggio non nasce con tale obiettivo, la semplicità di utilizzo non potrà essere paragonabile a quella di un linguaggio pensato appositamente per tracciare i side effect come Koka.

### 6.4.2 Stile di programmazione

Un'altra fondamentale differenza fra effetti algebrici e approccio monadico sta nello stile del codice: avendo modellato gli effetti tramite monadi l'unico modo per poterli concatenare consiste nell'usare `flatMap`:

```
def flatMapExample: State[Int, Unit] =
  State.get.flatMap { state =>
    State.set(f(state)).flatMap { _ =>
      Monad.pure(())
    }
  }
```

Il codice risultante è generalmente poco leggibile e per questo motivo viene fornito lo zucchero sintattico della for comprehension:

```
def forComprehensionExample: State[Int, Unit] =
  for
    state <- State.get
    _     <- State.set(f(state))
  yield ()
```

Il codice assume nuovamente un aspetto "imperativo". Tuttavia è importante che questo non tragga in inganno: non è possibile scrivere `State.set(f(State.get))` anche se è quello che ci si potrebbe aspettare di poter fare. Infatti `get` non è un semplice valore ma una computazione all'interno della monade `State`.

Si consideri invece l'approccio adottato nel caso degli effetti algebrici:

```
fun direct-style(f)
  set(f(get()))
```

Le operazioni `set` e `get` non sono altro che normali funzioni arricchite da annotazioni relative ai side effect: il codice risultante è esattamente quello che potrebbe scrivere un programmatore "imperativo" che non è mai stato esposto al concetto degli effetti algebrici.

La differenza chiave sta nel fatto che Koka traccia automaticamente gli effetti delle funzioni. L'effect system può essere visto come una sorta di rete di sicurezza che guida il programmatore nella corretta gestione degli effetti.

Allo stesso tempo viene ridotto il carico cognitivo e non è aggiunto alcun overhead sintattico come nel caso della for comprehension. Questo è un vantaggio

da non sottovalutare in quanto rende il linguaggio più semplice da leggere e scrivere riducendone la barriera d'adozione.

# Capitolo 7

# Conclusioni

La presenza di side effect arbitrari nel codice è un elemento che complica notevolmente la capacità di ragionare sulle sue proprietà ed effettuare refactoring. Partendo da questa osservazione chiave, l'obiettivo di questa tesi è stato indicare diversi approcci che possono essere adottati per rendere espliciti i side effect nei tipi delle funzioni. In particolare, si è mostrato come il concetto di monade possa essere utilizzato per modellare semplici side effect come il fallimento di una computazione o la presenza di uno stato mutabile.

Successivamente, osservando alcune limitazioni di questo approccio, è stato introdotto il concetto di monad transformer come meccanismo per la composizione di più monadi. Un innegabile beneficio introdotto da questo approccio sta nella possibilità di riutilizzare il codice delle monadi di base per poterle assemblare in monadi più complesse.

Uno svantaggio dell'approccio dei monad transformer sta nel fatto che la loro adozione lega fortemente il codice ad una specifica implementazione dei side effect rendendo più difficile il testare porzioni di codice con side effect. Per questo motivo sono stati analizzati gli approcci più sofisticati di MTL e delle free monad. Entrambi permettono di definire in maniera astratta e modulare gli effetti che una computazione può avere per poi stabilirne l'implementazione in un secondo momento.

Potendo definire effetti arbitrari, diventa facile scegliere il livello di astrazione più adatto a descrivere in maniera efficace la logica applicativa. Gli effetti possono quindi essere utilizzati come strumento di design per riflettere sulle operazioni che un sistema deve mettere a disposizione. Inoltre, grazie a queste tecniche, è possibile separare nettamente la dichiarazione degli effetti dalla loro specifica interpretazione rendendo facile il testare porzioni di codice con side effect mettendo in atto dependency injection.

Infine, è stato mostrato l'approccio degli effetti algebrici come alternativa a quello monadico per la rappresentazione esplicita dei side effect delle funzioni.

Questo approccio, basato su solide basi teoriche, concentra la propria attenzione sulla definizione di operazioni – che definiscono in astratto come un effetto può avere luogo – e handler – che vengono utilizzati per dare semantica alle singole operazioni. Questa netta distinzione fra operazioni e loro interpretazione, in maniera analoga al caso di MTL e delle free monad, rende più facile testare codice che presenta side effect definendo interpreti ad hoc in base alla necessità.

Gli effetti algebrici, oltre a prestarsi bene a modellare i side effect delle funzioni, sono un potente meccanismo che generalizza diversi costrutti di controllo di flusso complessi come eccezioni, iteratori e concorrenza strutturata tramite *async-await*. Sebbene i linguaggi che supportano gli effetti algebrici siano ancora principalmente solo linguaggi di ricerca, l'approccio sembra essere molto promettente e sta ricevendo crescente attenzione. Infatti, i benefici rispetto all'uso delle monadi sono diversi: nel caso degli effetti algebrici, l'attenzione è posta sulla definizione delle operazioni piuttosto che su come le singole azioni debbano essere concatenate tramite `flatMap`, rendendo più facile definire e testare effetti arbitrari senza avere boilerplate. Inoltre, è possibile adottare uno stile di programmazione diretto anziché dover ricorrere allo zucchero sintattico della for comprehension. Infine, l'approccio presenta una barriera d'ingresso più bassa in quanto non è richiesto al programmatore di conoscere terminologie e meccanismi complessi come monadi, higher-kinded type e type class.

# Appendice A

# Dimostrazioni delle leggi monadiche

## A.1 Dimostrazione per la monade identità

Alla Sezione 2.2.3 è stata data una definizione di monade per `Identity`. In seguito è riportata una dimostrazione del rispetto delle leggi monadiche elencate nella Sezione 2.2.1 per la definizione fornita[1].

Dimostrazione dell'identità sinistra, ovvero che `pure(a).flatMap(f) = f(a)`:

`pure(a).flatMap(f) =` *Definizione di `pure`*

`a.flatMap(f) =` *Definizione di `flatMap`*

`f(a)` □

Dimostrazione dell'identità destra, ovvero che `m.flatMap(pure) = m`:

`m.flatMap(pure) =` *Definizione di `flatMap`*

`pure(m) = m` □ *Definizione di `pure`*

Dimostrazione dell'associatività, ovvero che `(m.flatMap(f)).flatMap(g) = m.flatMap (x => f(x).flatMap(g))`:

`(m.flatMap(f)).flatMap(g) =` *Definizione di `flatMap`*

[1] Nei passaggi della dimostrazione, così come in quelle successive, sono utilizzati i nomi canonici utilizzati per l'implementazione Scala; quindi, si utilizza `flatMap` anziché `>>=` e `pure` anziché `return`

`f(m).flatMap(g) =` *Definizione di `flatMap`*

`g(f(m)) =` *Composizione di funzione*

`(x => g(f(x)))(m)` *Definizione di `flatMap`*

`m.flatMap(x => g(f(x))) =` *Definizione di `flatMap`*

`m.flatMap(x => f(x).flatMap(g))` □

## A.2 Dimostrazione per la monade Optional

Alla Sezione 2.2.3 è stata data una definizione di monade per `Option`. In seguito è riportata una dimostrazione del rispetto delle leggi monadiche per la definizione fornita.

Dimostrazione dell'identità sinistra, ovvero che `pure(a).flatMap(f) = f(a)`:

`pure(a).flatMap(f) =` *Definizione di `pure`*

`Some(a).flatMap(f) =` *Definizione di `flatMap`*

`f(a)` □

Dimostrazione dell'identità destra, ovvero che `m.flatMap(pure) = m`:

Procedo per casi su `m`:

*Se `m = None`*

`m.flatMap(pure) =` *Per ipotesi `m = None`*

`None.flatMap(pure) =` *Definizione di `flatMap`*

`None = m` *Per ipotesi `None = m`*

*Se `m = Some(a)`*

`m.flatMap(pure) =` *Per ipotesi `m = Some(a)`*

| | |
|---|---|
| `Some(a).flatMap(pure) =` | *Definizione di `flatMap`* |
| `pure(a) =` | *Definizione di `pure`* |
| `Some(a) = m` □ | *Per ipotesi `Some(a) = m`* |

Dimostrazione dell'associatività, ovvero che `(m.flatMap(f)).flatMap(g) = m.flatMap (x => f(x).flatMap(g))`:

Procedo per casi su `m`:

*Se `m = None`*

| | |
|---|---|
| `(m.flatMap(f)).flatMap(g) =` | *Per ipotesi `m = None`* |
| `(None.flatMap(f)).flatMap(g) =` | *Definizione di `flatMap`* |
| `None.flatMap(g) =` | *Definizione di `flatMap`* |
| `None =` | *Definizione di `flatMap`* |
| `None.flatMap(x => f(x).flatMap(g)) =` | *Per ipotesi `None = m`* |
| `m.flatMap(x => f(x).flatMap(g))` | |

*Se `m = Some(a)`*

| | |
|---|---|
| `(m.flatMap(f)).flatMap(g) =` | *Per ipotesi `m = Some(a)`* |
| `(Some(a).flatMap(f)).flatMap(g) =` | *Definizione di `flatMap`* |
| `f(a).flatMap(g) =` | *Applicazione di funzione* |
| `(x => f(x).flatMap(g))(a) =` | *Definizione di `flatMap`* |
| `Some(a).flatMap(x => f(x).flatMap(g)) =` | *Per ipotesi `Some(a) = m`* |
| `m.flatMap(x => f(x).flatMap(g))` □ | |

## A.3 Dimostrazione per la monade State

Alla Sezione 2.2.3 è stata data una definizione di monade per `State`. In seguito è riportata una dimostrazione del rispetto delle leggi monadiche per la definizione fornita.

Dimostrazione dell'identità sinistra, ovvero che `pure(a).flatMap(f) = f(a)`:

```
pure(a).flatMap(f) =                      Definizione di pure

State(s => (a, s)).flatMap(f) =           Sia m = State(s => (a, s))

m.flatMap(f) =                            Definizione di flatMap

State(s0 =>
  val (res1, s1) = m.runState(s0)
  f(res1).runState(s1)) =                 Definizione di runState

State(s0 =>
  val (res1, s1) = (s => (a, s))(s0)
  f(res1).runState(s1)) =                 Applicazione di funzione

State(s0 =>
  val (res1, s1) = (a, s0)
  f(res1).runState(s1)) =                 Pattern matching su una tupla

State(s0 => f(a).runState(s0)) =          Sia f(a) = State(g)

State(s0 => State(g).runState(s0)) =      Definizione di runState

State(s0 => g(s0)) =                      η-riduzione

State(g) =                                Per definizione di f(a)

f(a)  □
```

Dimostrazione dell'identità destra, ovvero che `m.flatMap(pure) = m`:

```
m.flatMap(pure) =                         Definizione di flatMap

State(s0 =>
  val (res1, s1) = m.runState(s0)
```

```
  pure(res1).runState(s1)) =             Definizione di pure

State(s0 =>
  val (res1, s1) = m.runState(s0)
  State(s => (res1, s))
    .runState(s1)) =                     Definizione di runState

State(s0 =>
  val (res1, s1) = m.runState(s0)
  (s => (res1, s))(s1)) =                Applicazione di funzione

State(s0 =>
  val (res1, s1) = m.runState(s0)
  (res1, s1)) =                          Eliminazione pattern matching

State(s0 => m.runState(s0)) =            Sia m = State(g)

State(s0 => State(g).runState(s0)) =     Definizione di runState

State(s0 => g(s0)) =                     η-riduzione

State(g) =                               Per definizione di m

m  □
```

Dimostrazione dell'associatività, ovvero che `(m.flatMap(f)).flatMap(g) = m.flatMap (x => f(x).flatMap(g))`:

```
(m.flatMap(f)).flatMap(g) =              Definizione di flatMap

State(s0 =>
  val (res2, s2) =
    (m.flatMap(f)).runState(s0)
  g(res2).runState(s2)) =                Definizione di flatMap

State(s0 =>
  val (res2, s2) = (State(s =>
    val (res1, s1) = m.runState(s)
    f(res1).runState(s1)
    .runState(s0)
  g(res2).runState(s2)) =                Definizione di runState
```

```
State(s0 =>
  val (res2, s2) =
    val (res1, s1) = m.runState(s0)
    f(res1).runState(s1)))
  g(res2).runState(s2)) =
```
*Estrazione dichiarazioni locali*

```
State(s0 =>
  val (res1, s1) = m.runState(s0)
  val (res2, s2) = f(res1).runState(s1)
  g(res2).runState(s2)) =
```
*Definizione di `State`*

```
State(s0 =>
  val (res1, s1) = m.runState(s0)
  State(s =>
    val (res2, s2) = f(res1).runState(s)
    g(res2).runState(s2))
    .runState(s1)) =
```
*Definizione di `flatMap`*

```
State(s0 =>
  val (res1, s1) = m.runState(s0)
  (f(res1).flatMap(g))
    .runState(s1)) =
```
*Applicazione di funzione*

```
State(s0 =>
  val (res1, s1) = m.runState(s0)
  (x => f(x).flatMap(g))(res1)
    .runState(s1)) =
```
*Definizione di `flatMap`*

```
m.flatMap(x => f(x).flatMap(g))
```
□

# Appendice B

# Basi del linguaggio Haskell

Haskell è un linguaggio di programmazione funzionale puro, basato su un modello di valutazione lazy e con inferenza di tipi statica.

L'obiettivo di questa appendice è fornire alcuni elementi di base di Haskell per rendere chiari gli esempi riportati nei Capitoli 2 e 3 al lettore che non abbia familiarità con il linguaggio.

## B.1 Sintassi di base

### B.1.1 Definizione e invocazione di funzioni

Haskell adotta una sintassi simile a quella dei linguaggi nella famiglia ML. In particolare, per dichiarare una funzione è sufficiente indicarne il nome, una lista di argomenti e il corpo con l'implementazione:

```
exampleFunction arg1 arg2 = ... {* function body *}
```

Il type system di Haskell è basato sul type system di Hindley e Milner e, nella maggior parte dei casi, è in grado di inferire automaticamente il tipo delle funzioni senza bisogno di annotazioni. Per esempio, data la seguente funzione:

```
hello name = "Hello, " ++ name ++ "!"
```

Il tipo della inferito è `String -> String`: la funzione prende in input una stringa e restituisce una stringa. L'operatore `++` viene utilizzato per la concatenazione di stringhe e di liste.

Se si volesse indicare esplicitamente il tipo di una funzione è possibile aggiungere un'annotazione nel seguente modo:

```
hello :: String -> String
hello name = "Hello, " ++ name ++ "!"
```

Si consideri una funzione con più argomenti:

```
sum :: Int -> Int -> Int
sum x y = x + y
```

In questo caso il tipo della funzione è `Int -> Int -> Int`: i tipi degli argomenti sono separati da `->` e l'ultimo tipo indicato è il tipo di ritorno della funzione. Quindi, la funzione `sum` prende in input due argomenti (`x` e `y`) di tipo intero e restituisce un valore intero.

Per poter invocare una funzione è sufficiente fornirle i parametri separandoli con degli spazi; a differenza di linguaggi come C e Scala l'invocazione di funzione non richiede la scrittura di parentesi tonde che ne racchiudano gli argomenti. Per esempio per invocare la funzione `hello` è sufficiente scrivere `hello "Haskell"` e il risultato della sua invocazione sarà `"Hello, Haskell!"`. Invece, l'invocazione di `sum` può avvenire passandole due argomenti di tipo intero: `sum 1 2 = 3`.

Inoltre, qualunque funzione abbia due soli argomenti può sempre essere invocata come se fosse un operatore infisso inserendola fra `:

```
invokeSum =
  let result1 = sum 1 2
      result2 = 1 `sum` 2
   in (result1, result2)
```

Le due invocazioni daranno lo stesso risultato. La sintassi `let ... in ...` serve a definire dei valori locali al corpo di una funzione. Il risultato di un blocco *let-in* è il valore indicato dopo la parola chiave `in`; in questo caso sarà la tupla `(result1, result2)`.

## B.2 Tipi di dato algebrici e pattern matching

Haskell permette di definire ADT tramite la parola chiave `data`; in maniera analoga alle enumerazioni Scala, un ADT può essere generico e i suoi costruttori avere campi differenti:

```
data Option a
  = Some a
  | None
```

A differenza di Scala, i tipi generici sono denotati con lettere minuscole e vengono semplicemente indicati dopo il nome del tipo. In questo caso è stato dichiarato un tipo `Option` generico su un tipo `a` con due costruttori:

- Il costruttore `Some` richiede un solo argomento di tipo `a`
- Il costruttore `None` non richiede alcun argomento

Per poter accedere ai campi di un costruttore è necessario procedere per pattern matching, in Haskell questo è possibile con la parola chiave `case`:

```
mapOption :: Option a -> (a -> b) -> Option b
mapOption option f =
  case option of
    Some value -> Some (f value)
    None -> None
```

La funzione `mapOption` prende in input un valore opzionale di tipo `Option a` – dove `a` è un tipo generico – e una funzione `a -> b`. Nel corpo della funzione viene fatto pattern matching sul valore opzionale e, nel caso in cui corrisponda al costruttore `Some`, viene applicata la funzione `f` al valore contenuto al suo interno.

Si confronti la funzione con l'equivalente versione Scala:

```
def mapOption[A, B](option: Option[A])(f: A => B): Option[B] =
  match option
    case Some(value) => Some(f(value))
    case None => None
```

In Scala è necessario indicare anticipatamente quali sono i tipi generici della funzione mentre in Haskell è possibile utilizzarli senza una previa indicazione.

Un esempio di utilizzo della funzione `mapOption` potrebbe essere il seguente:

```
useMapOption =
  mapOption (Some 1) (\value -> value + 1)
```

La sintassi `\arg0 arg1 ... argn -> ...` permette di definire una funzione anonima.

## B.3 Type classes

Haskell dà supporto al *polimorfismo ad hoc* tramite l'uso delle *type classes* [39]. Una type class permette di specificare una serie di funzioni che devono essere definite per un tipo generico:

```
class Show s where
  show :: s -> String
```

In questo caso è stata dichiarata la type class `Show` e un qualunque tipo generico `s` voglia implementarla deve fornire un'implementazione della funzione `show` che trasforma un tipo `s` in una stringa.

Una type class può essere interpretata come un predicato espresso su un tipo: per esempio, la type class `Show` può essere letta come *"Dato un generico tipo `s`, per questo vale il predicato `Show` se esiste una funzione `show` in grado di convertire `s` in una stringa"*.

Istanziare una type class equivale quindi a dimostrare che uno specifico tipo rispetta il predicato fornendo come dimostrazione l'implementazione delle funzioni definite nella type class. Riprendendo l'esempio di `Show` è possibile definire un'istanza per il tipo `Option String`:

```
instance Show (Option String) where
  show option =
    case option of
      Some string -> "Some " ++ string
      None -> "None"
```

Una volta fornita l'istanza per uno specifico tipo sarà possibile utilizzare la funzione `show` su valori di quel tipo:

```
main = println (show (Some "Hello"))
-- "Some Hello"
```

## B.3.1 Uso delle type class per fornire un contesto alle funzioni

In Haskell è possibile sfruttare il meccanismo delle type class per poter vincolare i tipi generici delle funzioni. Si consideri la seguente type class:

```
class Eq a where
  (==) :: a -> a -> Bool
```

Se un tipo `a` fornisce un'istanza di `Eq` sarà possibile confrontarne gli elementi tramite la funzione `==`. Il linguaggio fornisce una serie di implementazioni standard per alcuni tipi di base come `Int` e `String`.

Si immagini di dover realizzare una funzione che permetta di confrontare due valori opzionali: questi sono uguali solo se sono entrambi `None`, oppure nel caso in cui siano entrambi definiti e gli elementi contenuti al loro interno siano uguali:

```
areEqual :: Option a -> Option a -> Bool
areEqual optional1 optional2 =
  case (optional1, optional2) of
    (None, None) -> True
    (Just x, Just y) -> ...
    _ -> False
```

Per poter confrontare i due valori `x` e `y` è necessario poter utilizzare una funzione come `==`. Tuttavia, `==` può essere definito solo per un tipo concreto che fornisca un'istanza della type class `Eq`.

Per poter implementare la funzione `areEqual` è possibile esprimere un vincolo sul tipo generico `a` nel seguente modo:

```
areEqual :: (Eq a) => Option a -> Option a -> Bool
areEqual optional1 optional2 =
  case (optional1, optional2) of
    (None, None) -> True
    (Just x, Just y) -> x == y
    _ -> False
```

Si osservi come è cambiato il tipo della funzione: prima della lista dei tipi degli argomenti è stato specificato un *vincolo* sul generico tipo `a`. La notazione `(Eq a) =>` impone che la funzione `areEqual` possa essere invocata solo su valori opzionali parametrizzati su un tipo per cui sia disponibile un'istanza della classe `Eq`.

Riprendendo l'interpretazione delle type class come predicato si può interpretare leggere il vincolo come *"se i valori di tipo `a` rispettano il predicato `Eq` allora è possibile definire una funzione `areEqual` che confronta valori opzionali parametrizzati su `a`"*.

Si noti come, una volta imposto il vincolo, all'interno del corpo della funzione è possibile utilizzare la funzione `==` per confrontare i valori `x` e `y` di tipo `a`.

# Riferimenti Bibliografici


[1] Saleh M. Alnaeli, Amanda D. Ali. Taha e Tyler Timm. «On the prevalence of function side effects in general purpose open source software systems». In: *2016 IEEE 14th International Conference on Software Engineering Research, Management and Applications (SERA)*. 2016, pp. 141–148. DOI: 10.1109/SERA.2016.7516139.

[2] Jiachen Yang et al. «Towards purity-guided refactoring in Java». In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2015, pp. 521–525. DOI: 10.1109/ICSM.2015.7332506.

[3] Robert C. Martin e James O. Coplien. *Clean code: a handbook of agile software craftsmanship*. Prentice Hall, 2009. ISBN: 9780132350884.

[4] Raffi Khatchadourian et al. «Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams». In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 619–630. DOI: 10.1109/ICSE.2019.00072.

[5] Tim Süß et al. «Pure Functions in C: A Small Keyword for Automatic Parallelization». In: *International Journal of Parallel Programming* 49.1 (mag. 2020), pp. 1–24. DOI: 10.1007/s10766-020-00660-4.

[6] Du Phan, Neeraj Pradhan e Martin Jankowiak. «Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro». In: *arXiv preprint arXiv:1912.11554* (2019).

[7] Sivaramakrishnan Krishnamoorthy Chandrasekaran et al. «Algebraic Effect Handlers go Mainstream (Dagstuhl Seminar 18172)». In: *Dagstuhl Reports* 8.4 (2018). A cura di Sivaramakrishnan Krishnamoorthy Chandrasekaran et al., pp. 104–125. ISSN: 2192-5283. DOI: 10.4230/DagRep.8.4.104.

[8] Saunders Mac Lane. «Monads and Algebras». In: *Categories for the Working Mathematician*. New York, NY: Springer New York, 1978, pp. 137–159. ISBN: 978-1-4757-4721-8. DOI: 10.1007/978-1-4757-4721-8_7.

[9] Eugenio Moggi. *An abstract view of programming languages*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1989.

[10] Mike Spivey. «A functional theory of exceptions». In: *Science of Computer Programming* 14.1 (1990), pp. 25–42. ISSN: 0167-6423. DOI: `https://doi.org/10.1016/0167-6423(90)90056-J`.

[11] Philip Wadler. «Comprehending Monads». In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP '90. Nice, France: Association for Computing Machinery, 1990, pp. 61–78. ISBN: 089791368X. DOI: `10.1145/91556.91592`.

[12] Philip Wadler. «The Essence of Functional Programming». In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '92. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1992, pp. 1–14. ISBN: 0897914538. DOI: `10.1145/143165.143169`.

[13] Bruno C.d.S. Oliveira, Adriaan Moors e Martin Odersky. «Type Classes as Objects and Implicits». In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. Reno/Tahoe, Nevada, USA: Association for Computing Machinery, 2010, pp. 341–360. ISBN: 9781450302036. DOI: `10.1145/1869459.1869489`.

[14] Simon Peyton Jones. «Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell». In: *Engineering theories of software construction*. IOS Press, gen. 2001, pp. 47–96. ISBN: ISBN 1 58603 1724.

[15] Simon Peyton Jones. «A History of Haskell: being lazy with class». In: *The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*. Giu. 2007.

[16] Rúnar Óli Bjarnason. «Stackless Scala With Free Monads». In: apr. 2012. URL: `http://days2012.scala-lang.org/sites/days2012/files/bjarnason_trampolines.pdf`.

[17] Graham Hutton e Erik Meijer. «Monadic parsing in Haskell». In: *Journal of Functional Programming* 8.4 (1998), pp. 437–444. DOI: `10.1017/S0956796898003050`.

[18] Sheng Liang, Paul Hudak e Mark Jones. «Monad Transformers and Modular Interpreters». In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 333–343. ISBN: 0897916921. DOI: `10.1145/199448.199528`.

[19] Ismael Figueroa, Paul Leger e Hiroaki Fukuda. «Which monads Haskell developers use: An exploratory study». In: *Science of Computer Programming* 201 (2021), p. 102523. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2020.102523.

[20] Paul Leger et al. «Evolution of a Haskell Repository and its Use of Monads: An Exploratory Study of Stackage». In: gen. 2022. DOI: 10.1145/3477314.3506982.

[21] Mark P. Jones. «Functional programming with overloading and higher-order polymorphism». In: *Advanced Functional Programming*. A cura di Johan Jeuring e Erik Meijer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 97–136. ISBN: 978-3-540-49270-2.

[22] Franklyn A. Turbak e David K. Gifford. *Design Concepts in Programming Languages*. The MIT Press, 2008. ISBN: 0262201755.

[23] Martin Odersky et al. «Safer Exceptions for Scala». In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala*. SCALA 2021. Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 1–11. ISBN: 9781450391139. DOI: 10.1145/3486610.3486893.

[24] Jonathan Immanuel Brachthaeuser et al. «Effects, Capabilities, and Boxes From Scope-Based Reasoning to Type-Based Reasoning and Back». In: *Proceedings Of The Acm On Programming Languages-Pacmpl* 6.OOPSLA (2022), p. 76. DOI: https://doi.org/10.1145/3527320.

[25] Chuan-kai Lin. «Programming Monads Operationally with Unimo». In: *SIGPLAN Not.* 41.9 (set. 2006), pp. 274–285. ISSN: 0362-1340. DOI: 10.1145/1160074.1159840.

[26] WOUTER SWIERSTRA. «Data types à la carte». In: *Journal of Functional Programming* 18.4 (2008), pp. 423–436. DOI: 10.1017/S0956796808006758.

[27] Oleg Kiselyov e Hiromi Ishii. «Freer Monads, More Extensible Effects». In: *SIGPLAN Not.* 50.12 (ago. 2015), pp. 94–105. ISSN: 0362-1340. DOI: 10.1145/2887747.2804319.

[28] Nicolas Wu e Tom Schrijvers. «Fusion for Free». In: *Mathematics of Program Construction*. A cura di Ralf Hinze e Janis Voigtländer. Cham: Springer International Publishing, 2015, pp. 302–322. ISBN: 978-3-319-19797-5.

[29] Tom Schrijvers et al. «Monad Transformers and Modular Algebraic Effects: What Binds Them Together». In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. Haskell 2019. Berlin, Germany: Association for Computing Machinery, 2019, pp. 98–113. ISBN: 9781450368131. DOI: 10.1145/3331545.3342595.

[30] Koen Claessen. «A Poor Man's Concurrency Monad». In: *J. Funct. Program.* 9.3 (mag. 1999), pp. 313–323. DOI: 10.1017/S0956796899003342.

[31] Gordon Plotkin e Matija Pretnar. «Handlers of Algebraic Effects». In: *Programming Languages and Systems*. A cura di Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 80–94. ISBN: 978-3-642-00590-9.

[32] Daan Leijen. «Structured Asynchrony with Algebraic Effects». In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*. TyDe 2017. Oxford, UK: Association for Computing Machinery, 2017, pp. 16–29. ISBN: 9781450351836. DOI: 10.1145/3122975.3122977.

[33] KC Sivaramakrishnan et al. «Retrofitting Effect Handlers onto OCaml». In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 206–221. ISBN: 9781450383912. DOI: 10.1145/3453483.3454039.

[34] Alex Reinking et al. «Perceus: Garbage Free Reference Counting with Reuse». In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 96–111. ISBN: 9781450383912. DOI: 10.1145/3453483.3454032.

[35] Daan Leijen. «Koka: Programming with Row Polymorphic Effect Types». In: *MSFP*. 2014.

[36] Gordon D Plotkin e Matija Pretnar. «Handling Algebraic Effects». In: *Logical Methods in Computer Science* 9.4 (dic. 2013). DOI: 10.2168/LMCS-9(4:23)2013.

[37] G Plotkin e John Power. «Computational Effects and Operations: An Overview». English. In: *Electronic Notes in Theoretical Computer Science* 73 (ott. 2004), pp. 149–163. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2004.08.008.

[38] Gordon Plotkin e John Power. «Algebraic operations and generic effects». English. In: *Applied Categorical Structures* 11.1 (2003), pp. 69–94. ISSN: 0927-2852. DOI: 10.1023/A:1023064908962.

[39] Cordelia V. Hall et al. «Type Classes in Haskell». In: *ACM Trans. Program. Lang. Syst.* 18.2 (mar. 1996), pp. 109–138. ISSN: 0164-0925. DOI: 10.1145/227699.227700.

[40] Daan Leijen. *Algebraic effects for functional programming*. Rapp. tecn. Technical Report. MSR-TR-2016-29. Microsoft Research technical report, 2016.

# Riferimenti Sitografici


[41] *New in Scala 3 - New & Shiny: The Syntax*. URL: https://docs.scala-lang.org/scala3/new-in-scala3.html#new--shiny-the-syntax (visitato il 06/03/2023).

[42] *effect: A Fully-fledged functional effect system for typescript with a rich standard library*. URL: https://github.com/Effect-TS/effect (visitato il 06/03/2023).

[43] *WasmFX - Effect Handlers for WebAssembly*. URL: https://wasmfx.dev (visitato il 06/03/2023).

[44] *ZIO - Type-safe, composable asynchronous and concurrent programming for Scala*. URL: https://zio.dev (visitato il 06/03/2023).

[45] *Scala 3 Book - Extension Methods*. URL: https://docs.scala-lang.org/scala3/book/ca-extension-methods.html (visitato il 06/03/2023).

[46] *Scala 3 Reference - Context Bounds*. URL: https://docs.scala-lang.org/scala3/reference/contextual/context-bounds.html (visitato il 06/03/2023).

[47] *Scala 3 Book - Type Classes*. URL: https://docs.scala-lang.org/scala3/book/ca-type-classes.html (visitato il 06/03/2023).

[48] *Scala 3 Reference - Type Lambdas*. URL: https://docs.scala-lang.org/scala3/reference/new-types/type-lambdas.html (visitato il 06/03/2023).

[49] *Scala 3 Reference - Wildcard Arguments in Types*. URL: https://docs.scala-lang.org/scala3/reference/changed-features/wildcards.html (visitato il 06/03/2023).

[50] *Scala 3 Migration Guide - Kind Projector Migration*. URL: https://docs.scala-lang.org/scala3/guides/migration/plugin-kind-projector.html (visitato il 06/03/2023).

[51] *Scala 3 Book - Control Structures*. URL: https://docs.scala-lang.org/scala3/book/control-structures.html#for-expressions (visitato il 06/03/2023).

[52] *http-conduit: HTTP client package with conduit interface and HTTPS support*. URL: https://hackage.haskell.org/package/http-conduit-2.3.8 (visitato il 06/03/2023).

[53] *retry: Retry combinators for monadic actions that may fail*. URL: https://hackage.haskell.org/package/retry (visitato il 06/03/2023).

[54] *Cats Effect - Stack Safety*. URL: https://typelevel.org/cats-effect/docs/2.x/datatypes/io#stack-safety (visitato il 06/03/2023).

[55] *Requests-Scala*. URL: https://github.com/com-lihaoyi/requests-scala (visitato il 06/03/2023).

[56] *mtl: Monad classes for transformers, using functional dependencies*. URL: https://hackage.haskell.org/package/mtl (visitato il 06/03/2023).

[57] *Cats MTL*. URL: https://typelevel.org/cats-mtl/index.html (visitato il 06/03/2023).

[58] *Scala 3 Reference - CanThrow*. URL: https://docs.scala-lang.org/scala3/reference/experimental/canthrow.html (visitato il 06/03/2023).

[59] *EPFL - Postdoctoral Researcher on Type Systems for Resources and Effects*. URL: https://recruiting.epfl.ch/Vacancies/2452/Description/2 (visitato il 06/03/2023).

[60] *Scala 3 Reference - Union Types*. URL: https://docs.scala-lang.org/scala3/reference/new-types/union-types.html (visitato il 06/03/2023).

[61] *ppxlet - Monadic let binding in OCaml*. URL: https://github.com/janestreet/ppx_let (visitato il 06/03/2023).

[62] *Cats Scala*. URL: https://typelevel.org/cats/ (visitato il 06/03/2023).

[63] *Arrow - Monad comprehensions*. URL: https://arrow-kt.io/docs/patterns/monad_comprehensions/ (visitato il 06/03/2023).

[64] *Scala Effekt: Extensible algebraic effects with handlers*. URL: https://b-studios.de/scala-effekt/ (visitato il 06/03/2023).

[65] *fused-effects: A fast, flexible, fused effect system*. URL: https://hackage.haskell.org/package/fused-effects (visitato il 06/03/2023).

[66] *effect-handlers: A library for writing extensible algebraic effects and handlers*. URL: https://hackage.haskell.org/package/effect-handlers (visitato il 06/03/2023).

[67] *extensible-effects: An Alternative to Monad Transformers.* URL: https://hackage.haskell.org/package/extensible-effects (visitato il 06/03/2023).

[68] *Eff.* URL: https://www.eff-lang.org (visitato il 06/03/2023).

[69] *Effekt - A research language with effect handlers and lightweight effect polymorphism.* URL: https://effekt-lang.org (visitato il 06/03/2023).

[70] *Koka - A Functional Language with Effect Types and Handlers.* URL: https://koka-lang.github.io/koka/doc/index.html (visitato il 06/03/2023).

[71] *Unison - a friendly programming language from the future.* URL: https://www.unison-lang.org (visitato il 06/03/2023).

[72] *Koka - Benchmarks.* URL: https://github.com/koka-lang/koka#benchmarks (visitato il 06/03/2023).

[73] *yield statements - C# Language Reference.* URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/yield (visitato il 06/03/2023).

[74] *function* - MDN Web Docs, JavaScript language reference.* URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function* (visitato il 06/03/2023).

[75] *yield expression - Python language reference.* URL: https://docs.python.org/3/reference/expressions.html#yield-expressions (visitato il 06/03/2023).

[76] *Iterators Technical Implementation - C# Programming Guide.* URL: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/iterators#technical-implementation (visitato il 06/03/2023).