Università degli Studi di Camerino

Scuola di Scienze e Tecnologie

Corso di Laurea in Informatica per la Comunicazione Digitale (Classe L-31)

Progettazione e implementazione di una libreria di sessioni binarie nel linguaggio Swift

Laureando
Alessio Rubicini

Relatore
Luca Padovani

A.A. 2023/2024

Abstract

Negli ultimi decenni, nel contesto della programmazione concorrente e distribuita, un grande sforzo è stato dedicato all'indagine dei cosiddetti session types, un formalismo che permette di descrivere e verificare un protocollo di comunicazione tra dei processi come un tipo dato. Nonostante ciò, sono ancora pochissimi i linguaggi di programmazione che supportano, nativamente o non, i session types. Il presente lavoro di tesi ha come obiettivo la progettazione e lo sviluppo di una libreria nel linguaggio Swift che permetta di implementare e utilizzare un sistema di verifica dei tipi nelle sessioni di comunicazione tra due processi concorrenti. La metodologia adottata comprende lo sviluppo incrementale della libreria, accompagnato da una serie di test per validare le funzionalità implementate. Inoltre, la tesi esplora e documenta le sfide affrontate durante lo sviluppo, fornendo soluzioni pratiche e contribuendo alla letteratura esistente sui session types. La libreria sviluppata fornisce le funzionalità di base attese per l'utilizzo dei session types in Swift, tra cui la completa inferenza dei tipi di sessione, branch (binari), vincoli di dualità sui tipi, controllo dinamico della linearità sui canali di comunicazione, e canali condivisi in stile client/server per l'inizializzazione delle sessioni. Se da una parte la natura multiparadigma e la sintassi e i costrutti innovativi di Swift sono stati fondamentali per lo sviluppo della libreria, d'altra parte il lavoro si è scontrato con alcune limitazioni nel linguaggio, riguardati in particolare il supporto ai tipi non copiabili, utili per fornire un corretto controllo di linearità. Tali limiti sono stati superati sfruttando un approccio ibrido con controlli di tipo a tempo di compilazione e controlli di linearità a runtime. La libreria risultante rappresenta una base di partenza completa per l'utilizzo dei session types in Swift, e le nuove funzionalità in arrivo nelle prossime versioni del linguaggio permetteranno un'implementazione ancor più sicura e affidabile. La libreria è disponibile sulla rispettiva repository GitHub [1].

[1] https://github.com/alessiorubicini/SwiftSessions

Programming is the art of telling another human what one wants the computer to do.

Donald Knuth

Indice

1 Introduzione **13**

1.1 Motivazione . . . 13

1.2 Obiettivi . . . 14

1.3 Struttura della Tesi . . . 14

2 Teoria dei session types **15**

2.1 Tipizzazione e type checking . . . 15

2.2 Session types . . . 16

2.3 Sintassi . . . 16

2.4 Esempio 1 . . . 17

2.5 Esempio 2 . . . 18

2.6 Dualità . . . 19

2.7 Linearità . . . 19

2.8 Tipologie . . . 19

2.9 Approcci di verifica . . . 20

3 Modellazione dei session types in Swift **21**

3.1 Canale di comunicazione . . . 21

3.2 Endpoint di sessione . . . 23

3.3 Controlli di linearità . . . 25

3.4 Continuazioni . . . 29

4 Primitive di sessione **31**

4.1 Stili di programmazione . . . 31

4.1.1 Continuazioni con passaggio di endpoint . . . 31

4.1.2 Continuazioni con chiusure . . . 32

4.2 Inizializzazione delle sessioni . . . 33

4.3 Invio . . . 34

4.4 Ricezione . . . 34

4.5 Ramificazione e selezione . . . 35

4.6 Terminazione . . . 37

4.7 Inizializzazione con client/server . . . 37

5 Casi d'uso **39**

5.1 Esempi basici . . . 39

5.1.1 Numero pari con chiusure 39
5.1.2 Numero pari con passaggio di canali 39
5.1.3 Numero pari con client/server 40
5.2 Esempio di branching 40
5.3 Server matematico 41
5.4 Esempio di cicli 43
5.5 Violazioni di protocollo 43
5.5.1 Esempio 1 43
5.5.2 Esempio 2 44
5.6 Violazioni di linearità 44
5.6.1 Riutilizzo di un canale 44
5.6.2 Mancato utilizzo di un canale 45

6 Conclusioni **47**
6.1 Sviluppi futuri 47

Elenco dei codici

3.1 Esempio d'uso di un AsyncChannel 22
3.2 Modellazione dell'endpoint di sessione 23
3.3 Modellazione dell'assenza di valore 24
3.4 Implementazione della ricezione dal canale 25
3.5 Implementazione dell'invio sul canale 25
3.6 Implementazione della chiusura del canale 25
3.7 Esempio di consumazione con typestate 26
3.8 Ipotesi di classe Endpoint come tipo non copiabile 27
3.9 Possibile soluzione alla limitazione delle tuple nei typestate 27
3.10 Classe Endpoint con controllo di linearità 28
3.11 Primitive classe Endpoint con controlli di linearità 28
3.12 Classe Endpoint con doppio controllo di linearità 29
4.1 Continuazioni con passaggio di endpoint 31
4.2 Continuazioni con passaggio di chiusure 32
4.3 Inizializzazione sessione con passaggio di endpoint 33
4.4 Inizializzazione sessione con chiusure, versione 1 33
4.5 Inizializzazione sessione con chiusure, versione 2 34
4.6 Primitiva di invio con passaggio di endpoint 34
4.7 Primitiva di invio con chiusure 34
4.8 Primitiva di ricezione con passaggio di endpoint 35
4.9 Primitiva di ricezione con chiusure 35
4.10 Enumerazione per ramificazione binaria 35
4.11 Primitiva di ramificazione con passaggio di endpoint 36
4.12 Primitiva di ramificazione con chiusure 36
4.13 Primitive di selezione con passaggio di endpoint 36
4.14 Primitive di selezione con chiusure 36
4.15 Primitiva di chiusura sessione 37
4.16 Classe Server per creazione sessione 37
4.17 Classe Client per utilizzo sessione 38
5.1 Esempio: controllo numero pari con chiusure 39
5.2 Esempio: controllo numero pari con passaggio di canali 39
5.3 Esempio: controllo numero pari con client/server 40
5.4 Esempio: controllo numero pari con chiusure 40
5.5 Esempio: controllo numero pari con chiusure 41

5.6 Esempio: server matematico 42
5.7 Esempio: client somma per server matematico 42
5.8 Esempio: client logaritmo per server matematico 43
5.9 Esempio: somma di numeri con ciclo 43
5.10 Esempio 2 di violazione di protocollo 44
5.11 Esempio: violazione di linearità da riutilizzo di un endpoint 45
5.12 Esempio: violazione di linearità da mancato utilizzo di un endpoint . . . 45

Elenco delle figure

2.1 Rappresentazione di un canale di comunicazione 18

3.1 Inferenza dei session types in Xcode 30

Elenco delle tabelle

1. Introduzione

Il web di oggi è sempre più orientato verso una natura distribuita, sia per quanto concerne i dati che i processi software producono ed elaborano, sia in merito ai processi stessi che vengono eseguiti. Nel tempo sono nati veri e propri approcci architetturali basati sulla suddivisione e distribuzione dei sistemi software in diversi nodi, talvolta dislocati anche geograficamente. Un esempio lampante è l'architettura a micro-servizi, basata sul concetto di sviluppare applicazioni come un insieme di servizi indipendenti e autonomi, un approccio ampiamente sviluppato e popolarizzato a partire dagli anni duemila da diversi esperti di sviluppo software e aziende tech.

1.1 Motivazione

In tale contesto, la comunicazione tra le diverse parti di un sistema è un elemento fondamentale in quanto rappresenta ciò che mantiene unito il tutto e garantisce l'implementazione dei processi previsti. È quindi chiaro che diventa cruciale assicurarsi che la comunicazione avvenga in maniera sicura, affidabile e corretta. In particolare, quando parliamo di correttezza ci riferiamo all'atto di garantire che la comunicazione avvenga secondo i requisiti e le modalità previste, in termini di che tipi di messaggi vengono scambiati tra le parti, in quale ordine, e quali sono le condizioni di termine della comunicazione. È evidente quindi che la correttezza è una qualità fondamentale in un sistema concorrente e distribuito.

Nei sistemi distribuiti, garantire che le parti comunichino correttamente è particolarmente impegnativo. La necessità di affrontare le sfide associate allo sviluppo di sistemi distribuiti robusti e affidabili ha portato nel tempo alla nascita del formalismo dei *session types*, un approccio per specificare e verificare i protocolli nelle sessioni di comunicazione, attraverso la tipizzazione degli stessi.

Sebbene pochi linguaggi di programmazione supportino nativamente i session types, come MOOL (Mini Object-Oriented Language) o ATS (Authenticated Typed Script), molti dei linguaggi più popolari e comunemente utilizzati richiedono l'uso di librerie esterne per sfruttare questo strumento. Tra questi troviamo, in crescente popolarità nell'ambito dello sviluppo per applicazioni su piattaforme Apple, il linguaggio Swift, la cui compatibilità con i session types è al momento del tutto inesplorata. La sua natura nuova e moderna, insieme alla sua adozione crescente anche nel contesto server-side attraverso iniziative come SwiftNIO e Vapor, lo rendono un candidato ideale per valutare l'applicabilità dei session types in un'ampia gamma di scenari di sviluppo software distribuito.

1.2 Obiettivi

La presente tesi si pone come principale obiettivo quello di progettare e sviluppare una libreria nel linguaggio Swift che permetta di implementare e utilizzare un sistema di verifica dei tipi nelle sessioni di comunicazione tra due processi. La libreria si propone di fornire strumenti che permettano di modellare e verificare le comunicazioni tra due processi concorrenti in maniera sicura, sfruttando le caratteristiche del linguaggio Swift e del suo type system. Il lavoro mira a colmare una lacuna nell'ecosistema di Swift, che attualmente non offre un supporto nativo per i session types, integrando concetti avanzati di tipizzazione per la gestione delle comunicazioni asincrone.

Inoltre, la tesi si prefigge di esplorare e documentare le sfide affrontate durante lo sviluppo della libreria, fornendo soluzioni e contribuendo alla letteratura esistente sui session types. Questo include la gestione delle limitazioni intrinseche del linguaggio Swift ed eventuali considerazioni sull'ottimizzazione delle performance. In questo modo, il lavoro non solo offrirà un contributo pratico attraverso la libreria stessa, ma anche un contributo teorico e metodologico per futuri sviluppatori e ricercatori interessati ai session types e alla programmazione concorrente in Swift.

1.3 Struttura della Tesi

Nel Capitolo 2 verrà dapprima fornita una panoramica generale sulle nozioni tradizionali di tipo dato e tipizzazione nei linguaggi di programmazione, per poi passare a un'introduzione del concetto di session types e della sintassi utilizzata per descriverli. Nel Capitolo 3 verrà descritto come i session types possono essere modellati nel linguaggio Swift, nonché quali costrutti e strumenti sono necessari a garantire i loro principi fondamentali. Successivamente, nel Capitolo 4 verrà descritta nel dettaglio la libreria implementata, in particolare come sono state realizzate le primitive di sessione presentate nel precedente capitolo. Il Capitolo 5 conterrà una serie di casi d'uso della libreria come esempi dimostrativi. Infine, nel Capitolo 6 verrà fornita una panoramica riassuntiva dei risultati raggiunti, insieme ad alcune considerazioni finali sui possibili sviluppi futuri della libreria implementata.

2. Teoria dei session types

Nel contesto dei linguaggi di programmazione, un concetto fondamentale è quello di tipo di dato. I tipi di dati giocano un ruolo cruciale nella programmazione, in quanto contribuiscono alla stabilità, alla sicurezza e all'efficienza del codice. Questo capitolo esplorerà dapprima la nozione di tipo di dato, la tipizzazione nei linguaggi di programmazione, e l'importanza del type checking. Successivamente entrerà nel dettaglio dei session types.

2.1 Tipizzazione e type checking

Un **tipo di dato** definisce un insieme di valori e le operazioni che possono essere eseguite su questi valori. I tipi di dati possono includere semplici tipi primitivi come numeri interi e caratteri, tipi composti come liste e tuple (coppie), e tipi definiti dall'utente come classi e interfacce. La scelta del tipo di dato appropriato per una variabile o una costante è cruciale per il corretto funzionamento del programma.

Il processo di assegnazione di un tipo a ogni variabile, costante e espressione in un programma è detto **tipizzazione**. Esistono due principali tipologie di tipizzazione:

- *Tipizzazione Statica*: i tipi delle variabili vengono determinati e verificati a tempo di compilazione, cioè prima che il programma venga eseguito. Una volta che il tipo di una variabile è stato dichiarato, non può essere cambiato durante l'esecuzione del programma, e il compilatore verifica che i tipi e le operazioni eseguite su di essi siano coerenti. Esempi di linguaggi a tipizzazione statica sono C, C++ e Swift.

- *Tipizzazione Dinamica*: i tipi delle variabili vengono determinati e verificati a runtime. Le variabili possono contenere valori di tipi diversi in momenti diversi durante l'esecuzione del programma. Esempi di linguaggi a tipizzazione dinamica sono Python e JavaScript.

Il processo di verifica della correttezza dei tipi di dati è detto **type checking**. Questo processo assicura che le operazioni siano eseguite solo su tipi di dati compatibili e previene errori di tipo che potrebbero causare malfunzionamenti del programma. Il type checking può avvenire a tempo di compilazione o a tempo di esecuzione, a seconda della tipizzazione del linguaggio.

Il type checking è un processo potente e cruciale nei linguaggi di programmazione, in quanto garantisce che tutte le operazioni nel programma siano eseguite su tipi di dati compatibili. Questo significa che le operazioni aritmetiche, di assegnamento, di

chiamata a funzioni, ecc., saranno eseguite solo se i tipi coinvolti sono coerenti e compatibili tra loro. Questo previene errori comuni come ad esempio la somma di stringhe e numeri, o l'invocazione di metodi su oggetti che non li supportano.

2.2 Session types

Come detto in precedenza, i programmi concorrenti coinvolgono più processi o componenti software che operano in parallelo o su nodi di un sistema distribuito, e la comunicazione tra questi processi è un elemento cruciale. Questo include contesti come la comunicazione tra processi in un sistema operativo o servizi web integrati tra loro.

In questo contesto, un concetto fondamentale è quello delle **sessioni**. Una sessione può essere definita come una sequenza strutturata di interazioni tra due o più parti che seguono un protocollo predefinito. Questo protocollo stabilisce le sequenze consentite di scambi di messaggi tra le parti. Le sessioni sono onnipresenti nei sistemi distribuiti e nelle comunicazioni di rete, rendendo la loro gestione corretta e sicura una sfida di primaria importanza.

Per affrontare questa sfida, dalla branca della tipizzazione dei processi e in particolare del π-calcolo, è nato un apposito formalismo chiamato ***session types*** (letteralmente, tipi di sessione), per specificare e verificare i protocolli di comunicazione tra diverse entità. In modo formale, i session types possono essere definiti come una disciplina della tipizzazione dei processi che descrive la struttura, il flusso e il comportamento delle comunicazioni tra processi. In particolare, essi specificano i tipi di messaggi scambiati durante una sessione di comunicazione, l'ordine in cui i messaggi devono essere inviati e ricevuti, nonché le condizioni di terminazione della sessione stessa.

Mentre un tipico sistema di tipi assicura che le variabili e le funzioni di un programma siano utilizzate correttamente, i session types estendono questa garanzia al corretto uso dei canali di comunicazione. Possono essere pensati come "tipi per i protocolli". L'idea alla base dei session types è che descriviamo un protocollo di comunicazione come un tipo che può essere verificato grazie al normale type checking effettuato dal compilatore.

2.3 Sintassi

La sintassi per descrivere i session types non è univoca, ed esistono numerosi modi per rappresentare formalmente il tipo di un endpoint di comunicazione utilizzato da un processo. Ai fini della presente tesi, verrà presa come riferimento una sintassi molto semplice basata sulle primitive espresse di seguito:

- `Send` (invio)

  Sintassi: $!T.S$

  Invia un valore di tipo T e procede con il protocollo descritto da S.

  Esempio: $!int.S$ significa *invia un intero e continua come specificato da S.*

- `Receive` (ricezione)

  Sintassi: $?T.S$

Riceve un valore di tipo T e procede con il protocollo descritto da S.

Esempio: $?int.S$ significa *ricevi un intero e continua come specificato da* S.

- `Branch` (ramificazione)

  Sintassi: $\&\{l_1 : S_1, l_2 : S_2, ..., l_i : S_i\}$

  Riceve un label l_i e procede con il protocollo corrispondente S_i.

  Esempio: $\&\{deposit : S_1, withdraw : S_2\}$ significa che il protocollo può continuare con S_1 o S_2, in base alla scelta del mittente tra *deposit* o *withdraw*.

- `Select` (selezione)

  Sintassi: $\oplus\{l_1 : S_1, l_2 : S_2, ..., l_i : S_i\}$

  Il mittente sceglie uno dei label offerti per continuare. Invia una label l_i e procede con il protocollo corrispondente S_i.

- `Recursion` (ricorsione/ripetizione)

  Sintassi: $\mu t.S$

  Definisce un tipo ricorsivo dove t può essere usato in S per far riferimento di nuovo al tipo dell'intera sessione.

  Esempio: $\mu t.?int.!int.t$ vuol dire *ricevi un intero, invia un intero e ripeti il protocollo.*

- `End` (terminazione)

  Sintassi: *end*

  Denota il termine della sessione. Non segue alcuna comunicazione.

Per capire meglio come descrivere i session types con una sintassi come quella sopra, procediamo con qualche esempio.

2.4 Esempio 1

Dati due processi P e Q, dove P è un processo che vuole sapere se un certo numero intero n è pari o meno, e Q è il processo che è calcola se n è pari, modelliamo la comunicazione tra i due come segue.

Il processo P utilizzerà un endpoint x di tipo S con il quale in sequenza potrà inviare un numero intero, ricevere un booleano e terminare la comunicazione; il tipo di x sarà quindi

$$S = !int.?bool.end \tag{2.1}$$

Il processo Q utilizzerà un endpoint y di tipo T con il quale in sequenza potrà ricevere un numero intero, inviare un booleano e terminare la comunicazione; il tipo di y sarà

$$T = ?int.!bool.end \tag{2.2}$$

A partire da questo esempio è già possibile intuire una delle proprietà fondamentali dei session types ovvero la **dualità**. Si dice che i due endpoint x e y sono duali l'uno dell'altro, cioè ogni input in x deve essere abbinato a un output corrispondente in y, e viceversa. Ciò garantisce che i due processi interagiscano correttamente per tutta la durata della comunicazione.

2.5 Esempio 2

Dati due processi P e Q, dove P è un processo che necessita di calcolare la sottrazione di due numeri interi n e m, e Q è un processo che fornisce varie funzioni matematiche tra cui la sottrazione, modelliamo la comunicazione tra i due processi come segue.

Il processo P utilizzerà un endpoint x di tipo S con il quale in sequenza potrà inviare la scelta dell'operazione matematica desiderata (in questo caso la sottrazione), e proseguire con il protocollo relativo all'operazione scelta. Nel caso della sottrazione, il protocollo di continuazione consiste nell'inviare un numero intero, inviare un altro numero intero, e ricevere un numero intero; il tipo di x sarà quindi descritto come

$$
\begin{aligned}
S \ = \oplus\{ & \\
& sum : !int.!int.?int.end, \\
& sub : !int.!int.?int.end, \\
& fact : !int.?int.end \\
\} &
\end{aligned} \tag{2.3}
$$

Il processo Q utilizzerà un endpoint y di tipo T, duale a x, con il quale in sequenza potrà ricevere la scelta dell'operazione matematica desiderata, e procedere con il relativo protocollo che, nel caso della sottrazione, consiste nel ricevere un numero intero, ricevere un altro numero intero, e inviare un numero intero; il tipo di y sarà descritto come

$$
\begin{aligned}
T \ = \&\{ & \\
& sum : ?int.?int.!int.end, \\
& sub : ?int.?int.!int.end, \\
& fact : ?int.!int.end \\
\} &
\end{aligned} \tag{2.4}
$$

In questo modo, P seleziona uno dei label messi a disposizione da Q, comunicandogli così la scelta dell'operazione matematica. Successivamente P procede con il protocollo associato al label scelto, che nel caso della sottrazione è $!int.!int.?int.end$.

È bene specificare che quando parliamo di endpoint x e y, stiamo parlando dei due estremi di un unico canale di comunicazione c. I processi P e Q comunicano attraverso c utilizzando i suoi due estremi x e y, ognuno avente il proprio specifico tipo di sessione, come illustrato nella figura di seguito.

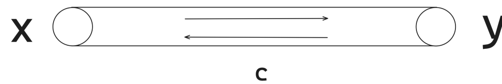

Figura 2.1: Rappresentazione di un canale di comunicazione

2.6 Dualità

Come sottolineato già nei precedenti esempi, uno dei concetti fondamentali alla base dei session types è la **dualità**. Essa stabilisce una corrispondenza tra le azioni eseguite all'interno di una sessione, garantendo che ciascuna parte aderisca allo stesso protocollo di comunicazione da una prospettiva complementare. Formalmente, la dualità impone che per ogni azione di invio nel protocollo di una parte, esista un'azione di ricezione corrispondente nel protocollo dell'altra parte. Questa dipendenza reciproca, che ovviamente non vale solo per le primitive di invio e ricezione ma anche per tutte le altre, garantisce che la comunicazione proceda in modo sincronizzato e ben definito, e che il protocollo venga seguito correttamente.

2.7 Linearità

Un altro concetto fondamentale nel contesto dei session types è la **linearità**, cruciale per garantire la corretta gestione delle risorse e dei canali di comunicazione. La linearità garantisce che venga applicato un utilizzo lineare delle risorse, cioè che ciascun endpoint di sessione venga consumato (utilizzato) esattamente una volta nel corso di una sessione. Se questo principio venisse meno, le risorse potrebbero venire utilizzate più volte o non utilizzate affatto, causando potenziali errori come deadlock.

L'utilizzo lineare delle risorse migliora l'affidabilità e la sicurezza dei programmi concorrenti prevenendo problemi comuni come deadlock, livelock o esaurimento delle risorse. Inoltre, la linearità contribuisce alla scalabilità e alle prestazioni dei programmi concorrenti garantendo che le risorse siano gestite in modo efficiente e rilasciate in modo tempestivo. Ciò è particolarmente importante nei sistemi distribuiti in cui le risorse sono condivise e devono essere gestite con attenzione per evitare conflitti e colli di bottiglia.

Dal punto di vista strettamente pratico, l'applicazione della linearità nei session types implica in genere type system in grado di verificare staticamente che le risorse e i canali vengano utilizzati in modo lineare. Questo controllo statico garantisce che i programmi aderiscano ai protocolli di comunicazione e che la sequenza di azioni prevista venga seguita senza deviazioni. Tuttavia, come vedremo nei prossimi paragrafi, esistono differenti approcci alla verifica dei tipi di sessione.

2.8 Tipologie

I tipi di sessione possono essere classificati in due principali tipologie, in base al numero di partecipanti alla sessione di comunicazione:

- **binari**: coinvolgono esattamente due partecipanti impegnati in un protocollo di comunicazione. Questi tipi vengono spesso utilizzati per modellare e verificare le interazioni tra un client e un server o tra due componenti in un sistema distribuito.

- **multiparte**: estendono il concetto a protocolli che coinvolgono più di due partecipanti. Ciò consente la specifica e la verifica di modelli di comunicazione complessi in sistemi distribuiti con più entità interagenti.

2.9 Approcci di verifica

La verifica dei tipi di sessione può essere affrontata attraverso vari approcci. Questi metodi rientrano in tre categorie: verifica statica, dinamica e ibrida [Ozu].

- La **verifica statica** dei session types implica il controllo dell'aderenza ai protocolli di comunicazione in fase di compilazione, quindi prima che il programma venga eseguito. Questo approccio garantisce che i protocolli di comunicazione e i tipi di messaggi siano corretti in base ai tipi specificati o inferiti. La verifica statica in genere comporta l'utilizzo del type system stesso per garantire che il programma segua i protocolli di comunicazione prescritti senza richiedere l'esecuzione del runtime.
- La **verifica dinamica** prevede il controllo della correttezza dei flussi, dei tipi di messaggi e della linearità in fase di runtime, cioè durante l'esecuzione del programma. Questo approccio in genere comporta controlli specifici in fase di esecuzione per garantire che i messaggi vengano inviati e ricevuti nell'ordine corretto e con i tipi corretti, e garantire quindi che le parti aderiscano ai tipi di sessione specificati.
- La **verifica ibrida** combina elementi degli approcci statici e dinamici per fornire un metodo più completo per verificare i tipi di sessione. La correttezza dell'ordine e dei tipi di messaggi vengono controllati **staticamente** in fase di compilazione, mentre la linearità e altre proprietà vengono verificate **dinamicamente** durante l'esecuzione del programma per garantire che le risorse vengano utilizzate nel modo corretto.

3. Modellazione dei session types in Swift

Il seguente capitolo è dedicato alla parte progettuale della libreria sviluppata, e discute come i session types possono essere modellati nel linguaggio Swift sfruttando i suoi particolari costrutti e funzionalità, in modo tale da implementare correttamente le sessioni di comunicazioni binarie e soddisfare le proprietà fondamentali descritte finora.

Negli esempi trattati nel precedente capitolo, abbiamo visto come i processi comunicano utilizzando due endpoint diversi e duali x e y per interfacciarsi con un unico canale di comunicazione c. I tipi S e T di questi endpoint descrivono il protocollo che deve essere seguito dai processi P e Q per comunicare l'uno con l'altro attraverso tale canale. In altre parole, S e T rappresentano il tipo della sessione di comunicazione dalle prospettive dei due processi.

Prima di trattare la modellazione di tali concetti in Swift, è necessaria una premessa riguardo come li interpretiamo a un livello di astrazione leggermente inferiore come quello di un linguaggio di programmazione. Ai fini della progettazione della libreria, individuiamo due concetti distinti tra di loro:

- la comunicazione asincrona tra i processi, a livello di sistema operativo, viene realizzata nella pratica mediante un **canale di comunicazione**, un canale di basso livello e nascosto che permette a processi e thread differenti di scambiare messaggi tra di loro in modo asincrono con un meccanismo in stile coda;
- le estremità del canale, rappresentative dei due processi durante la comunicazione, vengono identificate come **endpoint di sessione**. Un endpoint è un punto di accesso al canale di comunicazione, che specifica se e quale tipo di messaggi possono essere inviati e ricevuti dal canale attraverso l'endpoint stesso.

Procediamo dunque a descrivere come modelliamo questi due concetti sfruttando il linguaggio di programmazione Swift.

3.1 Canale di comunicazione

Nel linguaggio Swift, le funzionalità riguardanti la concorrenza vengono normalmente trattate grazie al **Grand Central Dispatch (GCD)**, un framework per la gestione della concorrenza e la programmazione asincrona su piattaforme Apple, progettato per semplificare la gestione dei thread e delle operazioni asincrone. Mentre il GCD fornisce vari meccanismi utili alla programmazione asincrona, come ad esempio code di dispatch

(Dispatch Queues) e operazioni (Operation Queues) per gestire l'esecuzione asincrona di codice e l'accesso concorrente alle risorse, non prevede di base alcun meccanismo per implementare un vero e proprio canale simile a quello descritto precedentemente.

Esternamente al framework GCD e al linguaggio stesso, Apple mette a disposizione *Swift Async Algorithms* [App23], una libreria open-source che fornisce una serie di algoritmi asincroni per lavorare con sequenze e stream di dati in Swift, sfruttando le nuove capacità di concorrenza introdotte con Swift 5.5.

Una delle tante classi messe a disposizione dalla libreria è chiamata `AsyncChannel` e implementa un canale asincrono come mezzo per trasferire dati tra thread in modo sicuro e asincrono, permettendo così di inviare e ricevere valori tra attività concorrenti senza bloccare i thread stessi. Un canale di questo tipo può banalmente essere usato come segue:

```
let channel = AsyncChannel<String>()

Task {
    while let resultOfLongCalculation = doLongCalculations() {
        await channel.send(resultOfLongCalculation)
    }
    channel.finish()
}

for await calculationResult in channel {
    print(calculationResult)
}
```

Codice 3.1: Esempio d'uso di un AsyncChannel

Nell'esempio, viene creato un oggetto `AsyncChannel` con il quale è possibile scambiare valori di tipo stringa. Tramite il costrutto `Task` viene eseguita una qualche computazione in modo asincrono, inviando le stringhe risultanti da questa computazione sul canale. Infine, vengono letti in modo asincrono i valori che sono stati inviati sul canale, e vengono stampati nello standard output.

Un `AsyncChannel<A>` rappresenta quindi un canale di comunicazione asincrono, sul quale è possibile leggere e scrivere (o meglio inserire e prelevare) valori di tipo `A`. È bene sottolineare che la classe `AsyncChannel` impone che il tipo `A` sia conforme al protocollo `Sendable`. Il protocollo `Sendable`, introdotto anch'esso in Swift 5.5, è utilizzato principalmente per definire tipi di dati che possono essere condivisi in sicurezza in un contesto concorrente tra più thread, senza correre il rischio di data race, e quindi senza la necessità di controlli e gestioni particolari sulla memoria.

È quindi evidente che la classe `AsyncChannel` è un'ottima soluzione per implementare il canale primitivo tra i processi di cui vogliamo tipizzare le sessioni di comunicazione. Procediamo dunque ad analizzare come gli endpoint di sessione possono essere implementati.

3.2 Endpoint di sessione

I session types permettono di definire in modo rigoroso la sequenza e il tipo dei messaggi che possono essere scambiati tra due processi in una sessione di comunicazione. Per modellare tale comportamento è quindi necessario che i processi non accedano al canale di comunicazione in modo diretto, ma attraverso "interfacce" secondarie che possano regolare le operazioni e i tipi di messaggi consentiti. Chiamiamo tali interfacce **endpoint di sessione**.

Come spiegato in precedenza, un endpoint è un punto di accesso che un processo può utilizzare per inviare o ricevere dati da un canale di comunicazione. Esso specifica in modo rigoroso quali operazioni il processo può effettuare sul canale, e quali tipi di dati può trattare in tali operazioni. In altre parole, un endpoint di sessione non solo facilita la comunicazione asincrona in quanto "wrapper" del canale di comunicazione, ma garantisce che essa avvenga in modo sicuro e conforme alle regole definite dal protocollo.

Date queste premesse, modelliamo un endpoint come segue:

```
1  public final class Endpoint<A, B> {
2      private let channel: AsyncChannel<Sendable>
3
4      init(with channel: AsyncChannel<Sendable>) {
5          self.channel = channel
6      }
7
8      init<C, D>(from endpoint: Endpoint<C, D>) {
9          self.channel = endpoint.channel
10     }
11
12     ...
13 }
```

Codice 3.2: Modellazione dell'endpoint di sessione

La classe `Endpoint<A, B>` rappresenta un endpoint di sessione tramite il quale un processo può:

- inviare oggetti di tipo `A`
- ricevere oggetti di tipo `B`

Ad esempio, un oggetto di tipo `Endpoint<Int, Bool>` rappresenta un endpoint con il quale è possibile inviare un numero intero e ricevere un valore booleano. Riprendendo il principio di dualità espresso prima, l'endpoint duale del precedente sarà di tipo `Endpoint<Bool, Int>`, ed è un endpoint con il quale è possibile inviare un valore booleano e ricevere un numero intero.

Come si può notare, la classe `Endpoint` incapsula un oggetto `AsyncChannel`, che funge appunto da canale di comunicazione sottostante. Su di esso è possibile inviare qualsiasi oggetto sia conforme al protocollo `Sendable`. In questo caso, non specifichiamo un particolare tipo di dato come ad esempio `Int` o `String` perché non conosciamo a priori come il canale verrà utilizzato, e dev'essere perciò il più generico possibile.

Data la classe Endpoint descritta come sopra, è logico modellare di conseguenza la comunicazione utilizzando una serie di endpoint concatenati. Prendendo come riferimento l'esempio 2.4 del processo che calcola se un numero è pari o meno, modelliamo la comunicazione come segue.

Il processo P utilizzerà un tipo di sessione $S = !int.?bool.end$, quindi saranno necessari i seguenti endpoint:

1. Endpoint<Int, Empty> per inviare il numero intero. Il protocollo prevede l'invio di un intero, perciò la ricezione non è consentita;
2. Endpoint<Empty, Bool> per ricevere il valore booleano risultante. Il protocollo prevede la ricezione di un booleano, perciò l'invio non è consentito;
3. Endpoint<Empty, Empty> per chiudere la comunicazione. Il protocollo prevede il termine della comunicazione, perciò non è consentita alcuna operazione sul canale.

Dualmente, il processo Q, il cui tipo di sessione è $T = ?int.!bool.end$, impiegherà i seguenti endpoint:

1. Endpoint<Empty, Int> per ricevere il numero intero
2. Endpoint<Bool, Empty> per inviare il valore booleano
3. Endpoint<Empty, Empty> per chiudere la comunicazione

In questo contesto, Empty è utilizzato per indicare che un'operazione di invio o ricezione non è consentita su un dato canale, in quanto un endpoint può essere utilizzato una sola volta per una sola operazione. È quindi necessario rappresentare questo valore nullo in Swift. La soluzione migliore per rappresentare tale assenza è tramite l'enumerazione vuota:

```
1 enum Empty {
2
3 }
```

Codice 3.3: Modellazione dell'assenza di valore

L'enumerazione Empty non contiene alcun elemento, e il linguaggio Swift non permette di estendere le enumerazioni, rendendola a tutti gli effetti un tipo che non può essere istanziato né esteso in alcuna maniera. L'unico modo in cui Empty può essere utilizzata è per annotare endpoint in cui un'operazione non è ammessa.

Arrivati a questo punto, possiamo procedere a modellare le operazioni fondamentali che possono essere invocate su un dato endpoint. Dal momento che la classe Endpoint agisce da "wrapper" per il canale di comunicazione sottostante, le operazioni di cui avrà bisogno saranno banalmente l'invio di un oggetto, la ricezione di un oggetto, e la chiusura dell'endpoint. Sfruttando i costrutti messi a disposizione da Swift, siamo in grado di definire queste operazioni nella classe Endpoint sulla base di determinate condizioni nei suoi parametri di tipo. In particolare, sfruttando il costrutto extension *class* where *condition*, otteniamo il seguente risultato:

```
1 extension Endpoint where A == Empty {
2     func recv() async -> Sendable {
3         return await channel.first(where: { _ in true })!
4     }
5 }
```

Codice 3.4: Implementazione della ricezione dal canale

Ciò vuol dire che il metodo `recv` sarà disponibile solo sulle istanze della classe `Endpoint` il cui parametro di tipo `A` corrisponde ad `Empty`, cioè quando l'endpoint non permette l'invio di alcun dato. Adottiamo la stessa logica anche per il metodo di invio:

```
1 extension Endpoint where B == Empty {
2     func send(_ element: Sendable) async {
3         await channel.send(element)
4     }
5 }
```

Codice 3.5: Implementazione dell'invio sul canale

Allo stesso modo, il metodo `send` sarà disponibile sulle istanze della classe `Endpoint` il cui parametro di tipo `B` corrisponde ad `Empty`, cioè quando l'endpoint non permette la ricezione di alcun dato. Di nuovo, seguiamo la stessa logica per il metodo di chiusura di un canale:

```
1 extension Endpoint where A == Empty, B == Empty {
2     func close() {
3         channel.finish()
4     }
5 }
```

Codice 3.6: Implementazione della chiusura del canale

Il vantaggio di tale implementazione risiede nella sicurezza di tipo che ne deriva. Dal momento che i metodi `send` e `recv` saranno disponibili solo quando l'endpoint permetterà rispettivamente l'invio o la ricezione, abbiamo la certezza che nessuno potrà mai invocare un metodo di invio su un endpoint di ricezione, e viceversa. Ugualmente, il metodo `close` potrà essere invocato solo su un'istanza di tipo `Endpoint<Empty, Empty>`, cioè su un endpoint che non permette più alcuna comunicazione e che rappresenta quindi il termine della sessione.

3.3 Controlli di linearità

Grazie alla classe `Endpoint` modellata fin qui abbiamo già in parte assicurato il soddisfacimento del vincolo di dualità, dal momento che ogni endpoint `Endpoint<A, B>` avrà un suo duale di tipo `Endpoint<B, A>`. Le metodologie con le quali vengono soddisfatti completamente i vincoli di dualità saranno illustrate e chiarite nel dettaglio nel Capitolo 4 riguardante le primitive di comunicazione di sessione. Analizziamo ora come possiamo verificare il soddisfacimento sui vincoli di linearità.

Come spiegato nel Capitolo 2, la linearità garantisce che ciascun endpoint di comunicazione venga consumato (utilizzato) esattamente una volta nel corso di una sessione, e l'applicazione della linearità nei session types implica type system in grado di verificare

staticamente l'utilizzo lineare delle risorse.

In Swift, è possibile implementare tale comportamento sfruttando il **typestate** [Fm]. Il typestate, introdotto recentemente con la versione 5.9 di Swift, è un design pattern emerso in linguaggi con type system avanzati e rigidi modelli di proprietà della memoria, come ad esempio Rust. Esso introduce il concetto di macchina a stati nel type system del linguaggio, in modo che lo stato di un oggetto sia codificato nel suo tipo e che le transizioni tra gli stati si riflettano nel sistema di tipi. Fondamentalmente, aiuta a individuare gravi errori logici in fase di compilazione anziché in fase di esecuzione.

Abbiamo già in parte esplorato il concetto di typestate nella sezione precedente, quando abbiamo definito le operazioni invocabili su un oggetto `Endpoint` sulla base di ciò che permette di inviare e ricevere, quindi sulla base del suo stato. Ma in questo contesto, la massima applicazione dei typestate la troviamo proprio nei controlli di linearità.

Oltre al typestate, Swift 5.9 introduce anche i **tipi non copiabili**, detti anche tipi "move-only". Una struttura o un'enumerazione può essere contrassegnata con l'annotazione `~Copyable`, informando così il compilatore che il suo valore non può essere copiato. I metodi che prendono come argomento tipi non copiabili devono esplicitamente dichiarare come trattano tali oggetti, e lo fanno attraverso apposite parole chiave come `borrowing` e `consuming`. Per i nostri fini di controllo della linearità, ciò che ci interessa è proprio la parola chiave `consuming`. Questo perché quando passiamo un valore non copiabile ad una funzione `consuming`, non è possibile utilizzare nuovamente quel valore successivamente alla chiamata di funzione.

```
1  struct File: ~Copyable {
2      consuming func close() { ... }
3  }
4
5  let file = File()
6  file.close()
7  file.close()     // ERROR: 'file' consumed more than once
```

Codice 3.7: Esempio di consumazione con typestate

Nell'esempio, dichiariamo una struttura `File` di tipo `~Copyable`. Inoltre definiamo al suo interno un metodo `close` marcato con la parola `consuming`, in modo da dire al compilatore che quel metodo deve *consumare* il valore della struttura. Se andiamo a creare un oggetto di tipo `File` e proviamo ad utilizzarlo più di una volta invocando il metodo `close`, otterremo un errore a tempo di compilazione il quale indica che l'oggetto è stato consumato più di una volta.

È chiaro che tale meccanismo sarebbe perfetto per implementare i controlli di linearità nel contesto dei session types. La soluzione consisterebbe nel dichiarare la classe `Endpoint` come `~Copyable`, e marcare i suoi metodi `send`, `recv` e `close` come `consuming`. In questo modo un endpoint può essere utilizzato nella comunicazione solamente una volta, dopodiché il suo valore viene consumato e mai più utilizzato.

```
1  public final class Endpoint<A, B>: ~Copyable {
2
3      consuming func send(_ element: Sendable) { ... }
4      consuming func recv() -> Sendable { ... }
5      consuming func close()
6
7  }
```

Codice 3.8: Ipotesi di classe Endpoint come tipo non copiabile

Tuttavia, il supporto del linguaggio Swift per tali meccanismi risulta ancora giovane, e nel tentare un'implementazione come quella appena spiegata ci siamo scontrati con due importanti limitazioni nell'utilizzo dei typestate.

La prima limitazione sta nella totale assenza del supporto di Swift all'utilizzo di tipi non copiabili all'interno di tuple, utilizzate in maniera sostanziale nella libreria per passare le coppie payload/canale di continuazione durante la comunicazione (vedi Sezione 3.4). Tale mancanza non viene motivata all'interno del documento contenente la proposta di implementazione dei typestate in Swift [Gro+23], ma, nella sezione "Direzioni Future" dello stesso documento, gli autori affermano che dovrebbe essere possibile per una tupla contenere elementi non copiabili, rendendo la tupla non copiabile se uno qualsiasi dei suoi elementi lo è. Poiché la struttura delle tuple è sempre nota, sarebbe ragionevole permettere che gli elementi all'interno di una tupla possano essere presi, mutati e consumati indipendentemente, come il linguaggio consente oggi per gli elementi di una tupla di essere mutati indipendentemente tramite `inout`.

In ogni caso, sarebbe possibile superare temporaneamente tale limitazione evitando l'utilizzo delle tuple native del linguaggio, e creando invece una nostra tupla personalizzata:

```
1  enum And<A, B>: ~Copyable {
2
3  }
```

Codice 3.9: Possibile soluzione alla limitazione delle tuple nei typestate

Tuttavia, la seconda limitazione incontrata rende futile questa soluzione alternativa.

La seconda limitazione riguarda il mancato supporto di Swift all'uso di strutture ed enumerazioni non copiabili come parametri di tipo, altra funzionalità su cui facciamo ampio affidamento nella libreria. Nello stesso documento citato precedentemente [Gro+23], gli autori affermano che la proposta di introduzione dei non copiabili in Swift comporta una restrizione severa, cioè che i tipi non copiabili non possono conformarsi ai protocolli né essere usati come argomenti di tipo per funzioni o tipi generici, inclusi tipi comuni della libreria standard come `Optional` e `Array`. Tutti i parametri generici in Swift oggi comportano un'ipotesi implicita che il tipo sia copiabile. L'integrazione completa probabilmente comporterebbe anche modifiche al runtime di Swift e alla libreria standard per accomodare i tipi non copiabili nelle API che non erano state originariamente progettate per essi, e questa integrazione potrebbe quindi avere restrizioni di retro-compatibilità.

Questa limitazione rende essenzialmente impossibile sfruttare i typestate e le funzioni

`consuming` all'interno della libreria per gestire i controlli di linearità. Per superare tale limitazione, è stato scelto di sviluppare la libreria seguendo un approccio di verifica ibrido: la verifica sulla correttezza dei protocolli e delle primitive di comunicazione viene effettuata staticamente grazie al type checking del compilatore Swift, mentre i controlli di linearità vengono effettuati dinamicamente a tempo di esecuzione attraverso appositi meccanismi nella classe `Endpoint`.

In particolare, la classe `Endpoint` è stata munita di un apposito flag `isConsumed` che indica se l'endpoint è stato consumato o meno. Questo flag è accessibile dall'esterno in sola lettura, e viene modificato solamente all'interno di un metodo privato `consume` della classe `Endpoint`.

```
1  public final class Endpoint<A, B> {
2      ...
3
4      private(set) var isConsumed: Bool = false
5
6      private func consume() {
7          guard !isConsumed else {
8              fatalError("\(self.description) was used twice")
9          }
10         isConsumed = true
11     }
12 }
```

Codice 3.10: Classe Endpoint con controllo di linearità

Le operazioni `send`, `recv` e `close`, quando eseguite, invocano il metodo `consume` il quale controlla che l'endpoint non sia stato consumato: se non è stato consumato allora abilita il flag e procede, altrimenti solleva un errore che interrompe l'esecuzione del programma.

```
1  extension Endpoint where A == Empty {
2      func recv() async -> Sendable {
3          consume()
4          return await channel.first(where: { _ in true })!
5      }
6  }
7
8  extension Endpoint where B == Empty {
9      func send(_ element: Sendable) async {
10         consume()
11         await channel.send(element)
12     }
13 }
14
15 extension Endpoint where A == Empty, B == Empty {
16     func close() {
17         consume()
18         channel.finish()
19     }
20 }
```

Codice 3.11: Primitive classe Endpoint con controlli di linearità

Il controllo implementato permette di marcare un endpoint come consumato ogni volta che viene chiamata una delle sue primitive. Al successivo tentativo di utilizzo, verrà sollevato un errore riguardo il riutilizzo dell'endpoint.

Dal momento che un endpoint di sessione deve essere utilizzato una e una sola volta, un'altra casistica di violazione della linearità è quella in cui un endpoint non viene mai utilizzato, risultando quindi in un mancato soddisfacimento del protocollo di comunicazione. Sfruttando il meccanismo di controllo di linearità appena spiegato, possiamo estendere le funzionalità della classe `Endpoint` definendo un distruttore tramite l'apposita parola chiave `deinit`:

```
1  public final class Endpoint<A, B> {
2      ...
3      deinit {
4          if !isConsumed {
5              fatalError("\(self.description) was not consumed")
6          }
7      }
8      ...
9  }
```

Codice 3.12: Classe Endpoint con doppio controllo di linearità

Proprio come il costruttore di una classe viene invocato quando inizializziamo un oggetto di quella classe, al contrario un distruttore viene invocato quando un oggetto sta per essere de-allocato dal sistema di gestione della memoria di Swift. La soluzione consiste quindi nel controllare se un endpoint, nel momento in cui sta venendo de-allocato, è stato utilizzato o meno. Se non è mai stato utilizzato, viene sollevato un errore segnalando il mancato uso dello stesso.

La soluzione appena descritta permette un corretto controllo di linearità sull'utilizzo degli endpoint, e rappresenta un giusto compromesso per superare le limitazioni imposte dalla versione attuale del linguaggio Swift. Ulteriori considerazioni su tale aspetto sono riportate nel Capitolo 6.

3.4 Continuazioni

Finora abbiamo delineato i concetti di canale di comunicazione ed endpoint di sessione, e abbiamo visto come quest'ultimo è stato modellato e strutturato per garantire i vincoli chiave dei session types. Inoltre, abbiamo visto come un processo utilizzi una serie di endpoint concatenati, uno per ogni operazione di invio o ricezione, per realizzare il protocollo della sessione di comunicazione. In questa sezione esploriamo come questi endpoint sono effettivamente legati tra di loro e come i processi passano dall'uno all'altro durante la comunicazione.

Un processo, una volta effettuata un'operazione, necessita di sapere come deve procedere secondo il protocollo definito, cioè deve sapere su quale endpoint effettuare l'operazione successiva, e così via fino al termine della comunicazione. La necessità è che i due processi comunicanti si dicano l'un l'altro su quale endpoint "ricontattarsi" dopo ogni comunicazione in accordo con il protocollo della sessione.

Esaminiamo rapidamente il problema riprendendo nuovamente l'esempio 2.4:

- il processo P inizia la comunicazione inviando un numero intero su un endpoint di tipo `Endpoint<Int, Empty>`. Una volta inviato il numero, necessita di

un nuovo endpoint di tipo `Endpoint<Empty, Bool>` per ricevere il risultato booleano da parte del processo Q.

- allo stesso modo, il processo Q inizia la comunicazione ricevendo un numero intero da un endpoint di tipo `Endpoint<Empty, Int>`. Una volta ricevuto il numero, necessita di un nuovo endpoint di tipo `Endpoint<Bool, Empty>` per inviare il risultato booleano al processo P.

La soluzione a tale problema risiede nell'adozione di uno stile di programmazione chiamato **continuation-passing**, letteralmente passaggio di continuazioni. Questo stile prevede che un processo non si limiti ad inviare semplicemente i dati, ma invii insieme ad essi anche un nuovo endpoint che l'altro processo utilizzerà per *continuare* la comunicazione, da qui il concetto di **continuazione**. Di conseguenza, un processo non si limiterà più ad inviare un semplice dato, ad esempio di tipo `String` o `Int`, ma invierà una coppia del tipo `(Payload, Continuazione)`, dove il payload rappresenta i dati effettivi che sta scambiando con l'altro processo, mentre la continuazione rappresenta l'endpoint che l'altro processo dovrà utilizzare per rispondere a sua volta e continuare il protocollo di comunicazione. Nel caso dell'esempio precedente, i due session types possono quindi essere descritti in modo completo come segue.

Il processo P utilizzerà una sessione di tipo x dove potrà inviare una coppia contenente un numero intero e un endpoint di continuazione sul quale è possibile inviare un booleano e un canale di chiusura:

$$x =$$

```
Endpoint<(Int, Endpoint<(Bool, Endpoint<Empty, Empty>), Empty>), Empty>
```

Il processo Q utilizzerà invece una sessione di tipo y dove potrà ricevere una coppia contenente un numero intero e un endpoint di continuazione sul quale è possibile inviare un booleano e un endpoint di terminazione.

$$y =$$

```
Endpoint<Empty, (Int, Endpoint<(Bool, Endpoint<Empty, Empty>), Empty>)>
```

È importante sottolineare che i session types sopra descritti non sono stati scritti dall'autore della presente tesi, ma sono stati inferiti in modo completamente autonomo direttamente dal compilatore di Swift, come mostrato di seguito:

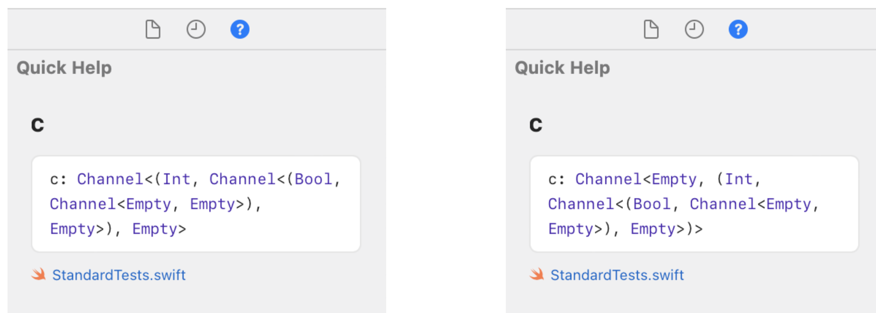

Figura 3.1: Inferenza dei session types in Xcode

L'inferenza dei tipi degli endpoint è una caratteristica estremamente utile in quanto maggior è la complessità del protocollo, maggiore è la loro difficoltà di scrittura da parte del programmatore. Ulteriori considerazioni in merito sono riportate nel Capitolo 4.

4. Primitive di sessione

Nel precedente capitolo abbiamo esplorato come i session types possono essere modellati in Swift, grazie ai particolari costrutti e funzionalità del linguaggio e all'utilizzo di endpoint appositamente progettati per implementare comunicazioni strutturate tra processi. Ora procediamo a descrivere come questi endpoint possono essere utilizzati nella pratica per realizzare tali sessioni di comunicazione. In particolare, il seguente capitolo descrive le primitive di sessione che implementano la logica necessaria e che costituiscono il collante tra i processi comunicanti e gli endpoint di comunicazione.

4.1 Stili di programmazione

Nella Sezione 3.4 abbiamo discusso la logica in base alla quale i processi passano da un endpoint all'altro durante la comunicazione, e di come essi scambiano non solo dati ma appunto anche endpoint di continuazione. Durante l'implementazione delle primitive di sessione esplorate in questo capitolo, è stato notato che si andavano delineando due differenti modi di implementare tali pattern di comunicazione nel linguaggio Swift, entrambi con i propri vantaggi e svantaggi. Per motivi di completezza, è stato deciso di sviluppare la libreria mantenendo entrambi gli stili di programmazione, liberamente utilizzabili dai programmatori e facilmente intercambiabili tra di loro. Di seguito descriviamo i due stili nel dettaglio.

4.1.1 Continuazioni con passaggio di endpoint

Seguendo la logica delle continuazioni descritta nella Sezione 3.4, l'approccio più naturale allo sviluppo delle primitive suggerisce di far in modo che le primitive stesse `send` e `recv` rispettivamente ritornino e accettino oggetti di tipo `Endpoint`, cioè gli endpoint di continuazione. Senza addentrarci ulteriormente nelle primitive, riportiamo di seguito un esempio di come la comunicazione viene strutturata seguendo questo stile:

```
1 // Inizializzazione sessione
2 let (e1, e2) = create();
3
4 // Processo P
5 let e3 = send(42, on: e2)
6 let (isEven, e4) = recv(from: e3)
7 close(e4)
```

Codice 4.1: Continuazioni con passaggio di endpoint

Nell'esempio, la primitiva `create` inizializza la sessione creando due endpoint duali `e1` e `e2`, cioè gli endpoint di partenza che i due processi dovranno utilizzare per iniziare la comunicazione. Successivamente uno dei due processi invia il numero intero `42` sul suo endpoint `e2` tramite la primitiva `send`, la quale ritorna l'endpoint di continuazione `e3`. Il processo utilizza quindi la primitiva `recv` sul nuovo endpoint `e3`, il quale ritorna a sua volta un altro endpoint di continuazione `e4`. Infine, il processo chiude e termina la comunicazione tramite l'endpoint `e4`.

È subito possibile intuire facilmente un vantaggio e uno svantaggio di tale stile di programmazione. Il vantaggio si ritrova nella semplicità e nell'intuitività del codice scritto, in quanto esso segue esattamente il flusso logico descritto precedentemente nella Sezione 3.2: ogni volta che un processo invia un messaggio, riceve in risposta un endpoint di continuazione che deve utilizzare per continuare la comunicazione. Lo svantaggio, invece, è più tecnico e deriva dalla necessità di dover dichiarare nuove variabili ogni volta che viene invocata una primitiva di sessione. Nell'esempio, dichiarare `e3` ed `e4` è necessario in quanto `e1` ed `e2` non solo sono costanti (cosa facilmente superabile) ma sono di un tipo diverso da quello ritornato dalle primitive, e Swift ne impedisce quindi la riassegnazione.

Inoltre, un altro svantaggio di questo stile è l'incapacità del linguaggio Swift di inferire autonomamente il tipo degli endpoint sulla base delle operazioni eseguite, a causa della separazione delle invocazioni in vari statement separati. Ciò obbliga l'utilizzatore della libreria a dover dichiarare esplicitamente il protocollo della sessione. Tale limitazione verrà esplorata più dettagliatamente nelle prossime sezioni.

4.1.2 Continuazioni con chiusure

In quanto linguaggio general purpose e multi-paradigma, Swift incorpora al suo interno anche elementi di programmazione funzionale tra cui le chiusure (closures). Le **chiusure** sono blocchi di codice "auto-contenuti" che possono essere passati e utilizzati all'interno di un programma. Possono catturare e memorizzare riferimenti a variabili e costanti dallo scope in cui sono definite, e possono essere utilizzate come funzioni in-line. Sono simili alle funzioni anonime o lambda-espressioni che troviamo in altri linguaggi di programmazione, e sono solitamente utili per la gestione di operazioni asincrone come il completamento di task o callback.

In questo contesto, le chiusure sono utili per gestire le continuazioni del protocollo di comunicazione senza ricorrere al passaggio diretto degli endpoint. Il concetto alla base di tale stile di programmazione è il seguente: ogni volta che un processo invoca una primitiva di sessione, oltre all'endpoint da utilizzare e agli eventuali dati da inviare, esso include anche il blocco di codice da eseguire dopo l'operazione effettuata. Tale blocco di codice prende come parametro l'endpoint di continuazione, il cui passaggio è gestito direttamente dalla primitiva. Per chiarire meglio la logica ecco di seguito un esempio:

```
1  create { e in
2      // Processo Q
3      recv(from: e) { num, e in
4          send(num % 2 == 0, on: e) { e in
5              close(e)
6          }
7      }
8  } _: { e in
9      // Processo P
10     send(42, on: e) { e in
11         recv(from: e) { isEven, e in
12             close(e)
13         }
14     }
15 }
```

Codice 4.2: Continuazioni con passaggio di chiusure

La primitiva di inizializzazione della sessione `create` prende due chiusure che contengono il codice dei due processi comunicanti. Ogni processo, quando invoca una primitiva di sessione, passa alla primitiva stessa il codice da eseguire successivamente. All'interno della chiusura può accedere all'endpoint di continuazione e agli eventuali dati ricevuti se si tratta di una primitiva di ricezione.

Il principale vantaggio di questo stile è la possibilità da parte di Swift di inferire completamente e autonomamente i tipi degli endpoint di comunicazione, senza la necessità che l'utilizzatore li dichiari esplicitamente. Questo è possibile in quanto l'intero codice dei due processi non è separato in vari singoli statement come nel caso precedente, ma è incapsulato all'interno di chiusure delle quali Swift riesce a inferire facilmente il tipo. Inoltre, grazie allo shadowing delle variabili, tale stile di programmazione elimina anche la necessità di dichiarare nuove variabili ad ogni invocazione, come accadeva invece nello stile con passaggio diretto di canali.

D'altra parte è evidente che passare le continuazioni come chiusure implica un'indentazione del codice sempre più profonda, man mano che la complessità del protocollo di comunicazione cresce, portando a un codice sintatticamente complesso e difficile da leggere e interpretare. Come evidenziato in precedenza, entrambi gli stili di programmazione hanno i propri pregi e difetti, rendendo difatti la loro implementazione combinata la scelta più ragionevole.

4.2 Inizializzazione delle sessioni

L'inizializzazione della sessione consiste nella creazione di un canale di comunicazione generico, e di due endpoint duali che condividono tale canale sottostante. Successivamente, i due processi che necessitano di comunicare possono usare i due endpoint risultanti dall'inizializzazione per iniziare la comunicazione e seguire di conseguenza il protocollo stabilito. Seguendo lo stile di continuazioni con passaggio di endpoint, implementiamo la primitiva di creazione della sessione come segue:

```
1  static func create<A, B>() -> (Endpoint<A, B>, Endpoint<B, A>) {
2      let channel: AsyncChannel<Sendable> = AsyncChannel()
3      let e1 = Endpoint<A, B>(with: channel)
4      let e2 = Endpoint<B, A>(with: channel)
5      return (e1, e2)
6  }
```

Codice 4.3: Inizializzazione sessione con passaggio di endpoint

La primitiva `create` crea un canale asincrono di tipo `AsyncChannel` sul quale è possibile scambiare qualsiasi oggetto aderente al protocollo `Sendable`. Successivamente, crea due endpoint duali che condividono tale canale, e li ritorna sottoforma di tupla.

Seguendo, invece, lo stile di programmazione delle continuazioni con chiusure, la primitiva `create` può essere implementata in due modi. Il primo prevede che la primitiva prenda come parametri due chiusure, le quali contengono tutto il codice completo che i processi devono eseguire per comunicare sul proprio endpoint, e che vengono eseguite separatamente e in modo asincrono andando a consumare i due endpoint:

```
1  static func create<A, B>(_ sideOne: @escaping (_: Endpoint<B, A>) async -> Void, _
       sideTwo: @escaping (_: Endpoint<A, B>) async -> Void) async {
2      let channel: AsyncChannel<Sendable> = AsyncChannel()
3      let e1 = Endpoint<A, B>(with: channel)
4      let e2 = Endpoint<B, A>(with: channel)
5      Task {
6          await sideOne(e2)
7      }
8      Task {
9          await sideTwo(e1)
10     }
11 }
```

Codice 4.4: Inizializzazione sessione con chiusure, versione 1

Similmente, la seconda opzione prevede che venga presa solamente una chiusura come para-

metro, la quale va a consumare uno degli endpoint creati, mentre l'altro viene ritornato dalla primitiva stessa e utilizzato dall'altro processo:

```
static func create<A, B>(_ closure: @escaping (_: Endpoint<B, A>) async -> Void) async
    -> Endpoint<A, B> {
    let channel: AsyncChannel<Sendable> = AsyncChannel()
    let e1 = Endpoint<A, B>(with: channel)
    let e2 = Endpoint<B, A>(with: channel)
    Task {
        await closure(e2)
    }
    return e1
}
```

Codice 4.5: Inizializzazione sessione con chiusure, versione 2

4.3 Invio

La primitiva di invio permette di inviare un payload di tipo `A` attraverso un endpoint di tipo `Endpoint<(A, Endpoint<B, C>), Empty>` dove `Endpoint<B, C>` rappresenta l'endpoint di continuazione per il processo destinatario. Il processo chiamante, o mittente in questo caso, continuerà la comunicazione tramite l'endpoint duale di tipo `Endpoint<C, B>`. Di seguito la sua implementazione nei due differenti stili di programmazione:

```
static func send<A, B, C>(_ payload: A, on endpoint: Endpoint<(A, Endpoint<B, C>),
    Empty>) async -> Endpoint<C, B> {
    await endpoint.send(payload)
    return Endpoint<C, B>(from: endpoint)
}
```

Codice 4.6: Primitiva di invio con passaggio di endpoint

```
static func send<A, B, C>(_ payload: A, on endpoint: Endpoint<(A, Endpoint<B, C>),
    Empty>, continuation: @escaping (Endpoint<C, B>) async -> Void) async {
    await endpoint.send(payload)
    let continuationEndpoint = Endpoint<C, B>(from: endpoint)
    await continuation(continuationEndpoint)
}
```

Codice 4.7: Primitiva di invio con chiusure

La logica del flusso di comunicazione è esattamente la stessa per entrambi gli stili di programmazione. Come spiegato precedentemente, la differenza è nel modo in cui viene gestito l'endpoint di continuazione: nel primo caso esso viene ritornato dalla primitiva, mentre nel secondo viene passato come parametro alla chiusura di continuazione fornita dal processo chiamante.

A differenza di quanto spiegato nella Sezione 3.4, si noti che l'endpoint di continuazione non viene creato e passato tramite il canale di comunicazione da un processo all'altro, ma viene creato dalle primitive di sessione stesse che lo passano poi al processo chiamante. Questa soluzione semplifica la logica di continuazione ed elimina la necessità di scambiare endpoint di continuazione tramite il canale di comunicazione.

4.4 Ricezione

La primitiva di ricezione permette di ricevere un payload di dati di tipo `A` da un endpoint di tipo `Endpoint<Empty, (A, Endpoint<B, C>)>`, dove `Endpoint<B, C>` rappresenta l'end-

point di continuazione per il processo chiamante. Di seguito la sua implementazione nei due differenti stili di programmazione:

```
1 static func recv<A, B, C>(from endpoint: Endpoint<Empty, (A, Endpoint<B, C>)>) async
      -> (A, Endpoint<B, C>) {
2     let msg = await endpoint.recv()
3     return (msg as! A, Endpoint<B, C>(from: endpoint))
4 }
```

Codice 4.8: Primitiva di ricezione con passaggio di endpoint

```
1 static func recv<A, B, C>(from endpoint: Endpoint<Empty, (A, Endpoint<B, C>)>,
      continuation: @escaping ((A, Endpoint<B, C>)) async -> Void) async {
2     let msg = await endpoint.recv()
3     let continuationEndpoint = Endpoint<B, C>(from: endpoint)
4     await continuation((msg as! A, continuationEndpoint))
5 }
```

Codice 4.9: Primitiva di ricezione con chiusure

4.5 Ramificazione e selezione

La ramificazione e la selezione sono meccanismi fondamentali per la gestione del flusso di comunicazione all'interno di interazioni complesse, perché consentono di modellare scenari di comunicazione in cui il comportamento del sistema dipende da scelte effettuate durante l'esecuzione.

La **ramificazione** consente di definire percorsi di esecuzione alternativi all'interno di una sessione. Si basa sull'idea dei "punti di ramificazione", o "branching points", dove il flusso di controllo si divide in base alla scelta effettuata. Ogni ramo rappresenta un possibile protocollo di continuazione della comunicazione. Quando un processo ramifica il protocollo mettendo a disposizione vari percorsi, l'altro sceglie uno di questi percorsi continuando con il protocollo che ne consegue.

Come illustrato nella Sezione 2.3 sulla sintassi dei session types, normalmente la ramificazione prevede che ogni branch messo a disposizione da un processo sia identificato da un certo label l_i al quale corrisponde un protocollo di continuazione S_i. Il processo che effettua la selezione sceglie uno dei label, e procede con il rispettivo protocollo. Si evince quindi che modellare e implementare ramificazioni di protocollo in un linguaggio come Swift diventa una sfida, dal momento che occorre prevedere n ramificazioni ognuna delle quali è identificata da un preciso label al quale è "staticamente" associato un preciso protocollo, quindi un preciso tipo dato.

Per garantire un minimo utilizzo della ramificazione, nella libreria è stato scelto di implementare ramificazioni binarie, ovvero branch a due scelte. La ramificazione è rappresentata da un'apposita enumerazione:

```
1 enum Or<A, B> {
2     case left(A)
3     case right(B)
4 }
```

Codice 4.10: Enumerazione per ramificazione binaria

Un oggetto di tipo `Or<A, B>` rappresenta quindi una ramificazione che mette a disposizione due sole scelte: il protocollo `A` o il protocollo `B`. La primitiva di ramificazione, chiamata per comodità `offer` in quanto il processo "offre" due scelte, è implementata come segue:

```swift
static func offer<A, B, C, D>(_ endpoint: Endpoint<Empty, Or<Endpoint<A, B>,
    Endpoint<C, D>>>) async -> Or<Endpoint<A, B>, Endpoint<C, D>> {
    let bool = await endpoint.recv() as! Bool
    if bool {
        return Or.left(Endpoint<A, B>(from: endpoint))
    } else {
        return Or.right(Endpoint<C, D>(from: endpoint))
    }
}
```

Codice 4.11: Primitiva di ramificazione con passaggio di endpoint

```swift
static func offer<A, B, C, D>(on endpoint: Endpoint<Empty, Or<Endpoint<A, B>,
    Endpoint<C, D>>>, _ side1: @escaping (Endpoint<A, B>) async -> Void, or side2:
    @escaping (Endpoint<C, D>) async -> Void) async {
    let bool = await endpoint.recv() as! Bool
    if bool {
        let continuationEndpoint = Endpoint<A, B>(from: endpoint)
        await side1(continuationEndpoint)
    } else {
        let continuationEndpoint = Endpoint<C, D>(from: endpoint)
        await side2(continuationEndpoint)
    }
}
```

Codice 4.12: Primitiva di ramificazione con chiusure

Dal momento che la ramificazione è binaria, possiamo utilizzare un valore booleano (vero/-falso) come "label" per rappresentare la scelta effettuata dal processo. Se il label ricevuto è `true`, allora vuol dire che il processo ha scelto il primo branch. Altrimenti, vuol dire che ha scelto il secondo. Difatti, le primitive di selezione `left` e `right` che un processo utilizza per selezionare uno dei due branch sono implementate come segue:

```swift
static func left<A, B, C, D>(_ endpoint: Endpoint<Or<Endpoint<A, B>, Endpoint<C, D>>,
    Empty>) -> Endpoint<B, A> {
    await endpoint.send(true)
    return Endpoint<B, A>(from: endpoint)
}

static func right<A, B, C, D>(_ endpoint: Endpoint<Or<Endpoint<A, B>, Endpoint<C, D>>,
    Empty>) -> Endpoint<D, C> {
    await endpoint.send(false)
    return Endpoint<D, C>(from: endpoint)
}
```

Codice 4.13: Primitive di selezione con passaggio di endpoint

```swift
static func left<A, B, C, D>(_ endpoint: Endpoint<Or<Endpoint<A, B>, Endpoint<C, D>>,
    Empty>, continuation: @escaping (Endpoint<B, A>) async -> Void) async {
    await endpoint.send(true)
    let continuationEndpoint = Endpoint<B, A>(from: endpoint)
    await continuation(continuationEndpoint)
}

static func right<A, B, C, D>(_ endpoint: Endpoint<Or<Endpoint<A, B>, Endpoint<C, D>>,
    Empty>, continuation: @escaping (Endpoint<D, C>) async -> Void) async {
    await endpoint.send(false)
    let continuationEndpoint = Endpoint<D, C>(from: endpoint)
    await continuation(continuationEndpoint)
}
```

Codice 4.14: Primitive di selezione con chiusure

Se il processo vuole scegliere il primo branch, allora invoca la primitiva `left` che invia un valore booleano `true` sul canale, comunicando all'altro processo la scelta del primo branch. Altrimenti, se vuole scegliere il secondo branch, invoca la primitiva `right` che allo stesso modo invia un valore `false` sul canale, comunicando la scelta del secondo branch.

4.6 Terminazione

L'ultima primitiva è quella di terminazione, ovvero quella invocata alla fine del protocollo per terminare la sessione di comunicazione.

```
1  static func close(_ endpoint: Endpoint<Empty, Empty>) {
2      endpoint.close()
3  }
```

Codice 4.15: Primitiva di chiusura sessione

Tutto ciò che fa è consumare l'ultimo endpoint di tipo `Endpoint<Empty, Empty>` e chiudere il canale di comunicazione, il quale invoca a sua volta l'apposito metodo `finish` dell'oggetto `AsyncChannel`. Quest'ultimo è semplicemente un'azione di sicurezza che va a riprendere immediatamente tutte le operazioni sospese sul canale.

4.7 Inizializzazione con client/server

È chiaro che un protocollo di comunicazione può diventare estremamente complesso, soprattutto nel momento in cui implica ramificazioni (4.5). Se la comunicazione cresce in complessità, è necessario di conseguenza aumentare la modularità e la riusabilità delle sessioni di comunicazione in modo tale da facilitare il più possibile l'utilizzo dei session types.

A tal scopo, la libreria è stata fornita di un meccanismo di inizializzazione delle sessioni in stile client/server: un **server** è responsabile della creazione e gestione di sessioni con un protocollo specifico. Più **client** possono essere creati e utilizzati con lo stesso server per interagire dualmente in base al protocollo definito. Ciò consente di definire il lato di un protocollo una sola volta e di utilizzarlo quante volte si desidera.

```
1  public class Server<A, B> {
2      private let publicChannel: AsyncChannel<Sendable>
3
4      init(_ closure: @escaping (_: Endpoint<A, B>) async -> Void) async {
5          publicChannel = AsyncChannel()
6          Task {
7              while true {
8                  for await request in publicChannel {
9                      let asyncChannel = request as! AsyncChannel<Sendable>
10                     let endpoint = Endpoint<A, B>(with: asyncChannel)
11                     Task {
12                         await closure(endpoint)
13                     }
14                 }
15             }
16         }
17     }
18
19     public func connect(with channel: AsyncChannel<Sendable>) async {
20         await publicChannel.send(channel)
21     }
22 }
```

Codice 4.16: Classe Server per creazione sessione

Quando viene inizializzato, un server prende come parametro una chiusura contenente il protocollo di comunicazione che deve implementare. Ciò vuol dire che i client che si connetteranno al server, dovranno implementare a loro volta tale protocollo per comunicare con esso. Il server utilizza quindi un canale asincrono esattamente uguale a quello usato nelle sessioni per rimanere in ascolto di richieste di connessione da parte di client. Quando riceve una richiesta, sottoforma anch'essa di canale asincrono da utilizzare per la nuova sessione, crea un nuovo endpoint ed esegue il suo protocollo su di esso.

```
1  public class Client {
2
3      init<A, B>(for server: Server<A, B>, _ closure: @escaping (_: Endpoint<B, A>)
       async -> Void) async {
4          let channel: AsyncChannel<Sendable> = AsyncChannel()
5          await server.connect(with: channel)
6          let c = Endpoint<B, A>(with: channel)
7          Task {
8              await closure(c)
9          }
10     }
11 }
```

Codice 4.17: Classe Client per utilizzo sessione

D'altra parte, un client viene creato connettendosi direttamente a un server esistente, e prende anch'esso una chiusura contenente il suo lato della comunicazione. Così crea un nuovo canale di comunicazione asincrono, quello da utilizzare per la sessione, e lo invia al server come richiesta di connessione. Dopodiché, crea un nuovo endpoint (duale a quello creato dal server) sullo stesso canale ed esegue la sua chiusura su tale endpoint consumandolo.

Questo meccanismo permette così di creare un singolo server che implementa uno specifico protocollo di comunicazione, e di creare un numero indefinito di client che utilizzano tale server, e che quindi comunicano usando il suo protocollo senza implementarlo interamente ogni volta.

L'inizializzazione di sessioni client/server non è solo utile a garantire modularità e riutilizzo, ma aiuta anche nell'implementazione di protocolli che prevedono cicli. Nel caso in cui sia necessario ripetere n volte una parte del protocollo di comunicazione, è sufficiente implementare tale parte come un server e creare n client all'interno di un ciclo. Un esempio pratico è esplorato nella Sezione 5.4.

5. Casi d’uso

In questo capitolo verranno mostrati esempi di utilizzo della libreria implementata. Gli esempi presentati coprono vari casi d’uso standard, oltre a violazioni volute dei principi fondamentali dei session types per dimostrare la sicurezza derivata dal loro utilizzo.

5.1 Esempi basici

Riprendiamo l’esempio 2.4 del processo che calcola se un numero è pari o meno, e vediamo la sua implementazione completa nei due differenti stili di programmazione messi a disposizione dalla libreria.

5.1.1 Numero pari con chiusure

```
1  await Session.create { e in
2      await Session.recv(from: e) { num, e in
3          await Session.send(num % 2 == 0, on: e) { e in
4              Session.close(e)
5          }
6      }
7  } _: { e in
8      await Session.send(42, on: e) { e in
9          await Session.recv(from: e) { isEven, e in
10             // isEven è true
11             Session.close(e)
12         }
13     }
14 }
```

Codice 5.1: Esempio: controllo numero pari con chiusure

5.1.2 Numero pari con passaggio di canali

```
1  typealias Protocol = Endpoint<Empty, (Int, Endpoint<(Bool, Endpoint<Empty,
       Empty>), Empty>)>
2
3  let c = await Session.create { (c: Protocol) in
4      let (num, e1) = await Session.recv(from: e)
5      let e2 = await Session.send(num % 2 == 0, on: e1)
6      Session.close(e2)
7  }
8
9  let e1 = await Session.send(42, on: e)
10 let (isEven, e2) = await Session.recv(from: e1)
11 Session.close(e2)
```

Codice 5.2: Esempio: controllo numero pari con passaggio di canali

5.1.3 Numero pari con client/server

In aggiunta, vediamo l'implementazione dello stesso esempio ma sfruttando l'inizializzazione della sessione in stile client/server:

```
// Server
let s = await Server { e in
    await Session.recv(from: e) { num, e in
        await Session.send(num % 2 == 0, on: e) { e in
            Session.close(e)
        }
    }
}

// Client
let client1 = await Client(for: s) { e in
    await Session.send(42, on: e) { e in
        await Session.recv(from: e) { isEven, e in
            Session.close(e)
        }
    }
}
```

Codice 5.3: Esempio: controllo numero pari con client/server

5.2 Esempio di branching

Dati due processi P e Q, dove P offre una scelta di protocollo tra la somma di due numeri e il calcolo del fattoriale, implementiamo il processo P come segue:

```
let e = await Session.create { e in
    await Session.offer(on: e) { e in
        await Session.recv(from: e) { num1, e in
            await Session.recv(from: e) { num2, e in
                let sum: Int = num1 + num2
                await Session.send(sum, on: e) { e in
                    Session.close(e)
                }
            }
        }
    } or: { e in
        await Session.recv(from: e) { num, e in
            var result = 1
            for i in 1...num {
                result *= i
            }
            await Session.send(result, on: e) { e in
                Session.close(e)
            }
        }
    }
}
```

Codice 5.4: Esempio: controllo numero pari con chiusure

Il processo Q, che seleziona uno dei due protocolli messi a disposizione da P (in questo caso la somma di numeri), è implementato come segue:

```
1  await Session.left(e) { e in
2      await Session.send(2, on: e) { e in
3          await Session.send(3, on: e) { e in
4              await Session.recv(from: e) { result, e in
5                  Session.close(e)
6              }
7          }
8      }
9  }
```

Codice 5.5: Esempio: controllo numero pari con chiusure

Utilizzando il pannello di aiuto di Xcode, possiamo vedere nel dettaglio i tipi dei due endpoint inferiti autonomamente da Swift. In particolare, il tipo dell'endpoint utilizzato dal processo P è il seguente:

```
Endpoint<
  Empty,
  Or<
    Endpoint<
      Empty,
      (Int, Endpoint<Empty, (Int, Endpoint<Empty, (Int, Endpoint<(Int,
    Endpoint<Empty, Empty>), Empty>)>)>)>
    >,
    Endpoint<
      Empty,
      (Int, Endpoint<(Int, Endpoint<Empty, Empty>), Empty>)
    >
  >
>
```

Mentre il tipo dell'endpoint utilizzato dal processo Q è descritto come segue:

```
Endpoint<
  (
    Int,
    Endpoint<
      Empty,
      (
        Int,
        Endpoint<
          (Int, Endpoint<Empty, Empty>),
          Empty
        >
      )
    >
  ),
  Empty
>
```

5.3 Server matematico

Esploriamo l'utilizzo dei branch con un caso più complesso. In questo esempio, un processo P agisce da server fornendo varie operazioni matematiche ai client che si connettono ad esso. In particolare, P fornisce innanzitutto una scelta tra operazioni aritmetiche base e operazioni logaritmiche. Nel caso delle operazioni base mette a disposizione somma e sottrazione, mentre come operazioni logaritmiche fornisce logaritmo naturale o logaritmo in base 10.

Il processo P è quindi implementato come segue:

```
let s = await Server { e in
    await Session.offer(on: e) { e in
        // Prima scelta: operazioni base
        await Session.offer(on: e) { e in
            // Somma
            await Session.recv(from: e) { num1, e in
                await Session.recv(from: e) { num2, e in
                    let result: Int = num1 + num2
                    await Session.send(result, on: e) { e in
                        Session.close(e)
                    }
                }
            }
        } or: { e in
            // Sottrazione
            await Session.recv(from: e) { num1, e in
                await Session.recv(from: e) { num2, e in
                    let result: Int = num1 - num2
                    await Session.send(result, on: e) { e in
                        Session.close(e)
                    }
                }
            }
        }
    } or: { e in
         // Seconda scelta: operazioni logaritmiche
        await Session.offer(on: e) { e in
            // Logaritmo naturale
            await Session.recv(from: e) { (number: Double, e) in
                let commonLogarithm = log(number)
                await Session.send(commonLogarithm, on: e) { e in
                    Session.close(e)
                }
            }
        } or: { e in
            // Logaritmo in base 10
            await Session.recv(from: e) { (number: Double, e) in
                let commonLogarithm = log10(number)
                await Session.send(commonLogarithm, on: e) { e in
                    Session.close(e)
                }
            }
        }
    }
}
```

Codice 5.6: Esempio: server matematico

Implementiamo quindi due client che usufruiscono delle sue funzioni:

```
let _ = await Client(for: s) { e in
    // Sceglie operazioni aritmetiche base
    await Session.left(e) { e in
        await Session.left(e) { e in
            await Session.send(5, on: e) { e in
                await Session.send(5, on: e) { e in
                    await Session.recv(from: e) { result, e in
                        Session.close(e)
                        // Il risultato è 10
                    }
                }
            }
        }
    }
}
```

Codice 5.7: Esempio: client somma per server matematico

```
let _ = await Client(for: s) { e in
    // Sceglie le operazioni logaritmiche
    await Session.right(e) { e in
        // Sceglie logaritmo comune (in base 10)
        await Session.right(e) { e in
            let number: Double = 100.0 // Approssimazione di e
            await Session.send(number, on: e) { e in
                await Session.recv(from: e) { result, e in
                    Session.close(e)
                    // Il risultato è 2.0
                }
            }
        }
    }
}
```

Codice 5.8: Esempio: client logaritmo per server matematico

5.4 Esempio di cicli

Come menzionato nella Sezione 4.7, l'inizializzazione delle sessioni seguendo un'architettura client/server può essere utile per implementare protocolli che prevedono iterazioni. Nel seguente esempio, vediamo che una sessione del genere può essere implementata.

```
var sum = 0
let numbers = [1, 5, 22, 42, 90]

let s = await Server { e in
    await Session.recv(from: e) { num, e in
        sum += num
        Session.close(e)
    }
}

for number in numbers {
    let _ = await Client(for: e) { e in
        await Session.send(number, on: e) { e in
            Session.close(e)
        }
    }
}
```

Codice 5.9: Esempio: somma di numeri con ciclo

Nell'esempio viene implementata una semplice somma di numeri. Dapprima viene creato un server il cui protocollo consiste nel ricevere un numero ed aggiungerlo alla somma globale. Poi, per ogni numero da sommare viene creato un client che invia al server il rispettivo numero.

5.5 Violazioni di protocollo

Nei seguenti esempi, dimostriamo come il type checking del compilatore Swift aiuta a prevenire errori di tipo nelle sessioni di comunicazione grazie alla libreria implementata.

5.5.1 Esempio 1

In questo esempio, violiamo volutamente il protocollo di comunicazione inviando una stringa invece di un numero intero sul canale di comunicazione.

```
1  await Session.create { e in
2      await Session.recv(from: e) { num, e in
3          await Session.send(num % 2 == 0, on: e) { e in
4              Session.close(e)
5          }
6      }
7  } _: { e in
8      await Session.send("Hello, World", on: e) { e in // ERRORE
9          await Session.recv(from: e) { isEven, e in
10             Session.close(e)
11         }
12     }
13 }
```

Xcode restituisce immediatamente un errore segnalando che l'endpoint `e` prevede l'invio di un elemento di tipo `Int`, invece passiamo un elemento di tipo `String`. L'errore riporta infatti:

```
Cannot convert value of type 'String' to expected argument type 'Int'
```

5.5.2 Esempio 2

Ora violiamo il protocollo di comunicazione invertendo le operazioni di invio e ricezione nella seconda chiusura:

```
1  await Session.create { e in
2      await Session.recv(from: e) { num, e in
3          await Session.send(num % 2 == 0, on: e) { e in
4              Session.close(e)
5          }
6      }
7  } _: { e in
8      await Session.recv(from: e) { isEven, e in  // ERRORE
9          await Session.send(42, on: e) { e in
10             Session.close(e)
11         }
12     }
13 }
```

Codice 5.10: Esempio 2 di violazione di protocollo

Anche in questo caso, Xcode restituisce un errore a tempo di compilazione il quale riporta:

```
Cannot convert value of type 'Endpoint<(Int, Endpoint<(Bool, Endpoint<Empty,
    Empty>), Empty>), Empty>' to expected argument type 'Endpoint<Empty, (A,
    Endpoint<(Int, Endpoint<Empty, Empty>), Empty>)>'
```

5.6 Violazioni di linearità

In quest'ultimi esempi, dimostriamo l'efficacia dei controlli di linearità dinamici implementati nella libreria attraverso alcuni classici esempi di violazione della linearità.

5.6.1 Riutilizzo di un canale

Nel seguente esempio andiamo a riutilizzare volutamente l'endpoint `e1` nella prima chiusura, il che rappresenta una piena violazione dell'utilizzo lineare degli endpoint:

```
await Session.create { e in
    await Session.recv(from: e) { num, e1 in
        await Session.send(num % 2 == 0, on: e1) { e2 in
            Session.close(e2)
            // Utilizziamo nuovamente il canale c1
            await Session.send(false, on: e1) { e3 in
                Session.close(e3)
            }
        }
    }
} _: { e in
    await Session.send(42, on: e) { e1 in
        await Session.recv(from: e1) { (isEven: Bool, e2) in
            Session.close(e2)
        }
    }
}
```

Codice 5.11: Esempio: violazione di linearità da riutilizzo di un endpoint

Eseguendo il test riportato sopra, Xcode interromperà l'esecuzione del programma e restituirà il seguente errore:

```
Endpoint<(Bool, Endpoint<Empty, Empty>), Empty> was consumed twice.
```

5.6.2 Mancato utilizzo di un canale

Un'altra casistica di violazione della linearità è quella in cui un endpoint non viene utilizzato, risultando in una mancata implementazione di una parte del protocollo, come di seguito:

```
await Session.create { e in
    await Session.recv(from: e) { num, e1 in
        await Session.send(num % 2 == 0, on: e1) { e2 in
            Session.close(e2)
        }
    }
} _: { e in
    await Session.send(42, on: e) { e1 in
        // Mancato utilizzo del canale e1
    }
}
```

Codice 5.12: Esempio: violazione di linearità da mancato utilizzo di un endpoint

Eseguendo il test riportato sopra, Xcode interromperà l'esecuzione del programma e restituirà il seguente errore:

```
Endpoint<Empty, (Bool, Endpoint<Empty, Empty>)> was not consumed.
```

6. Conclusioni

L'obiettivo del presente lavoro di tesi era progettare e sviluppare una libreria nel linguaggio Swift, volta a implementare un sistema di verifica e inferenza dei tipi nelle comunicazioni tra due processi, oltre a esplorare e documentare le sfide affrontate durante lo sviluppo della stessa. La libreria risultante offre le funzionalità di base necessarie per l'uso dei session types in Swift, tra cui: completa inferenza dei tipi di sessione, ramificazioni binarie, vincoli di dualità sui tipi, controllo dinamico della linearità sugli endpoint, e inizializzazione delle sessioni di comunicazione in stile client/server. Sebbene la natura multiparadigma di Swift e i suoi costrutti innovativi abbiano facilitato lo sviluppo della libreria, sono state riscontrate alcune limitazioni nel linguaggio, in particolare riguardo al supporto dei tipi non copiabili, cruciali per un corretto controllo della linearità. Questi limiti sono stati superati adottando un approccio ibrido che combina controlli di tipo a tempo di compilazione e controlli di linearità a tempo di esecuzione. In conclusione, l'obiettivo è stato pienamente raggiunto, fornendo una libreria che rappresenta una solida base per l'uso dei session types in Swift.

6.1 Sviluppi futuri

Il progetto di tesi è stato realizzato in un periodo di grande fermento per la programmazione concorrente in Swift, che ha visto un significativo sviluppo negli ultimi due anni grazie all'introduzione di nuove funzionalità e miglioramenti che hanno reso il linguaggio più sicuro ed efficiente per lo sviluppo di applicazioni concorrenti. Ad esempio, la versione 5.5 di Swift ha introdotto un modello di concorrenza strutturata e gli attori, strumenti che offrono un modo sicuro per definire task concorrenti e proteggere i dati da data race e altri problemi comuni.

Come descritto nella Sezione 3.3, la versione 5.9 di Swift, rilasciata nel settembre 2023, ha introdotto i tipi non copiabili, utili in questo contesto per garantire un corretto controllo di linearità nelle sessioni di comunicazione, ma ancora inutilizzabili a causa di alcune importanti limitazioni del linguaggio. Recentemente, Apple ha tenuto la WWDC24, la conferenza annuale per gli sviluppatori, durante la quale ha presentato varie novità riguardanti le sue tecnologie, tra cui la nuova major version di Swift 6.0. Questa versione, attualmente in beta e in arrivo nei prossimi mesi insieme alle nuove major dei sistemi operativi Apple, introduce diversi miglioramenti proprio per i tipi non copiabili [App24], superando di fatto le limitazioni riscontrate. In particolare, Swift 6.0 introduce il supporto ai generici per i tipi non copiabili, funzionalità che da sola sarebbe sufficiente a colmare le lacune incontrate e passare dall'attuale approccio di verifica ibrido ad un approccio statico e più sicuro come quello pensato inizialmente.

Queste novità aprono nuove prospettive per migliorare ulteriormente la libreria sviluppata, potenziandone l'affidabilità e l'efficienza. In conclusione, il lavoro svolto non solo ha raggiunto gli obiettivi prefissati, ma ha anche gettato le basi per sviluppi futuri significativi. La libreria sviluppata rappresenta un contributo importante per la programmazione concorrente in Swift, e le nuove opportunità offerte dalle prossime versioni del linguaggio promettono di rendere questo strumento ancora più potente e versatile. La libreria è disponibile sulla rispettiva repository GitHub [1].

[1] https://github.com/alessiorubicini/SwiftSessions

Bibliografia

[App23] Apple. *swift-async-algorithms*. https://github.com/apple/swift-async-algorithms. 2023.

[App24] Apple. *Consume noncopyable types in Swift*. https://developer.apple.com/wwdc24/10170. 2024.

[Fm] Simon Fowler e ABCD members. *Session Types in Programming Languages: A Collection of Implementations*. URL: https://groups.inf.ed.ac.uk/abcd/session-implementations.html.

[Gro+23] Joe Groff et al. *Noncopyable structs and enums*. https://github.com/swiftlang/swift-evolution/blob/main/proposals/0390-noncopyable-structs-and-enums.md. 2023.

[Ozu] Alex Ozun. *Typestate - the new Design Pattern in Swift 5.9*. URL: https://swiftology.io/articles/typestate/.

Ringraziamenti

Vorrei riservare questo spazio finale della mia tesi di laurea ai ringraziamenti verso tutti coloro che hanno contribuito in qualsiasi modo a questo mio percorso di crescita personale e professionale. In primo luogo ringrazio la mia famiglia, in particolare i miei genitori e mia sorella, per avermi sempre sostenuto nel mio percorso di studi e per avermi permesso di arrivare fin qui. Ringrazio tutti gli amici che mi sono sempre stati vicini, tra i quali tengo particolarmente a ringraziare Francesca, Nicola, Claudia e Daniela. Ringrazio la mia amica e instancabile compagna di studi Vittoria, insieme alla quale ho affrontato molte delle sfide e delle difficoltà incontrate durante questo importante percorso. Ringrazio Manuela e tutti i dipendenti di AppLoad per avermi accolto durante i mesi del tirocinio e per avermi formato, permettendomi non solo di terminare questo percorso ma soprattutto di svolgere un lavoro che amo. Infine ringrazio il mio relatore, il prof. Luca Padovani, non solo per aver accettato di supervisionare questo mio lavoro di tesi, ma soprattutto per la disponibilità, la professionalità e soprattutto la passione che ha messo in gioco sin dal primo momento. Grazie di cuore a tutti.