ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica — Scienza e Ingegneria
Corso di Laurea Triennale in Informatica per il Management

GESTIONE DELL'ALLOCAZIONE DI FUNZIONI SERVERLESS IN GO TRAMITE LO SVILUPPO DI UN SISTEMA DI ANNOTAZIONE AUTOMATICA

Relatore:
Chiar.mo Prof.
SAVERIO GIALLORENZO

Correlatore:
Dott. MATTEO TRENTIN

Presentata da:
LORENZO MARETTI

Sessione Luglio 2026
Anno Accademico 2025/2026

Indice

Capitolo 1

Introduzione

1.1 Serverless computing

Il settore informatico è stato protagonista negli ultimi ventanni di un enorme cambiamento, dettato da un ambiente costantemente in crescita e all'avanguardia. Negli anni '90 e anche '00, creare e avviare un'attività su Internet richiedeva avere a disposizione o affittare un computer virtuale e effettuare la configurazione manuale del sistema operativo e dell'ambiente di esecuzione da parte dello sviluppatore. Ciò comportava un notevole spreco in termini di energia e denaro, dovuto al mantenimento del server 24 ore su 24, anche nei periodi di inattività. Dalla metà dei '10, arrivarono sul mercato piattaforme serverless come AWS Lambda e Google Cloud Functions che misero a disposizione dei programmatori i propri server, rimuovendo la necessità di gestire tali risorse. Questo cambio di paradigma dà la possibilità agli sviluppatori di focalizzare l'attenzione sulla scrittura del codice e della logica di business.

Questo nuovo approccio, chiamato serverless computing, è un modello di sviluppo ed esecuzione di applicazioni che consente agli sviluppatori di creare ed eseguire codice applicativo senza dover gestire infrastrutture back-end. A differenza di quanto possa far intendere il nome, però, "serverless" non significa "senza server", in quanto questa componente è presente ed è fornita dai cloud service providers. Va inteso dal punto di vista del consumatore, per cui il server è una parte invisibile con cui non interagisce in alcun modo. Questo modello offre alcuni importanti vantaggi in termini di efficienza e flessibilità: è un modello pay-as-you-go che deve essere sostenuto economicamente solo nei momenti in cui l'applicazione sfrutta effettivamente il server; è un modello poliglotta, indipendente dal linguaggio di programmazione o dal framework scelto dallo sviluppatore, e velocizza lo sviluppo di applicazioni, permettendo ai programmatori di concentrarsi solo sulla logica di back-end; infine, è un modello scalabile che permette di gestire grandi quantità di connessioni simultanee al server, creando molteplici istanze temporanee dei processi che eseguono il codice necessario alla gestione delle richieste.

Il serverless computing porta con sè anche alcuni aspetti negativi, tra cui: un avvio a rilento, detto "cold start", se il codice chiamato viene eseguito per la prima volta o dopo un certo periodo di inattività, impiegando qualche secondo in più rispetto al normale tempo di esercizio; scrivere un'applicazione utilizzando le capacità e le funzionalità di un service provider rende l'applicazione incompatibile con gli strumenti di altri service provider, che sono tipicamente specifici per quella piattaforma; inoltre, i modelli di esecuzione serverless non sono progettati per eseguire codice per periodi limitati e i processi di lunga durata possono costare di più rispetto ai tradizionali ambienti server dedicati.

1.2 FaaS

All'interno della definizione di serverless enunciata prima, definiamo una sotto-categoria di questo modello, che rappresenta il contesto nel quale è stato sviluppato questo progetto, chiamato con l'acronimo FaaS (Functions as a Service). FaaS è un servizio serverless che permette allo sviluppatore di caricare singoli frammenti di codice, le functions, nel cloud. Il modello FaaS opera secondo una logica orientata agli eventi: le funzioni caricate nel cloud vengono eseguite solo quando necessario, in risposta a un determinato evento, scalando automaticamente in base alla domanda. Questo sistema offre gli stessi vantaggi elencati nella sezione precedente: l'utente paga solo per l'uso che viene fatto di questo codice e può sviluppare molto più velocemente queste funzioni. Alcuni esempi di piattaforme FaaS sono AWS Lambda e Google Cloud Functions. Facendo un esempio pratico, per la creazione di un sito e-commerce si può delegare la gestione del backend al fornitore di servizi e dedicarsi all'implementazione di funzioni che si attivano in risposta a determinati eventi, come un utente che decide di comprare un prodotto.

Un'altra caratteristica del paradigma FaaS è la natura stateless delle funzioni, il cui ciclo di vita inizia e termina con il compito per cui sono state programmate e non conservano dati in memoria tra un'esecuzione e l'altra. Tale approccio pay-as-you-go elimina gli sprechi economici derivanti dai periodi di inattività.

1.3 Obiettivo del Progetto di Tesi

Il progetto presentato in questa tesi si inserisce nel lavoro di ricerca presentato nel paper [5], riguardante le politiche di scheduling delle piattaforme serverless. Il lavoro svolto punta a integrare un'analisi statica di un codice di partenza per ottenere il linguaggio "essenziale" miniSL (mini Serverless Language), il cui obiettivo è estrarre le sole chiamate a servizi esterni (chiamate "call") per ricavarne funzioni di costo e modellare il comportamento delle funzioni serverless. L'obiettivo a lungo termine è quello di realizzare cAPP (cost-aware APP), un'estensione del linguaggio dichiarativo APP che utilizza

espressioni di costo per selezionare i nodi su cui eseguire queste funzioni, che minimizzano la latenza.

L'argomento trattato in questa tesi è l'implementazione di un sistema automatico (pipeline) di traduzione che, a partire da codice di input scritto in Go, ricava il corrispondente linguaggio miniSL. La pipeline è composta da due fasi principali che si susseguono in quest'ordine:

- Annotazione: vengono aggiunte al codice Go di partenza delle annotazioni per isolare if, for e invocazioni a funzioni interne o a servizi esterni.
- Estrazione: trasforma il codice annotato nel linguaggio miniSL.

Riassumendo, l'obiettivo di questo progetto è fornire uno strumento che semplifichi il processo di creazione del modello miniSL, estraendo le sole componenti necessarie affinchè i sistemi di scheduling possano operare scelte efficienti basate sulla latenza prevista.

Nei capitoli successivi, e in particolare nel capitolo relativo all'implementazione, viene descritto interamente come è stato strutturato il lavoro e le scelte implementative che sono state fatte, allegando esempi di esecuzione della pipeline e dell'output in miniSL.

Capitolo 2

Background: Go e miniSL

Prima di proseguire, definiamo nel alcuni concetti alla base della tesi, necessari per comprendere al meglio l'implementazione.

2.1 Linguaggio Go

Come linguaggio con cui sviluppare il progetto è stato scelto Go, un linguaggio ideato per il cloud computing e le infrastrutture di rete, molto efficace nell'analisi statica del codice. Nel contesto del nostro progetto, Go fornisce pacchetti come `go/parser`, `go/token` e `go/ast` che permettono di scomporre il codice in un Abstract Syntax Tree (AST) in modo semplice. Inoltre, il linguaggio permette di implementare facilmente controlli complessi, come il rilevamento di chiamate ricorsive dirette o circolari (tramite algoritmi DFS) e la validazione di funzioni annidate.

2.2 Mini Serverless Language

Il miniSL è un linguaggio semplice ed essenziale, pensato per analizzare il comportamento delle funzioni nelle applicazioni serverless. Il linguaggio è stato creato con l'idea di ignorare le operazioni che non riguardano le chiamate ai servizi esterni e non hanno un impatto significativo sulla latenza. I costrutti che non vengono scartati dal processo di traduzione sono i cicli if, for e le invocazioni di funzioni "call".

La definizione del linguaggio è mostrata di seguito con la Figura 2.1:

```
F ::= (p̄) => { S }
S ::= ε | call h(Ē) S | if (G) { S } else { S } | for (i in range(0,E)){ S }
G ::= E | call h(Ē)
E ::= n | i | p | E ♯ E
♯ ::= + | - | > | == | >= | && | * | /
```

Figura 2.1: Linguaggio miniSL

Il significato dei simboli è riportato nella tabella qui sotto:

Tabella 2.1: Sintassi e semantica dei simboli nel linguaggio miniSL.

Simbolo	**Descrizione**	**Riferimento**
F	**Funzione**: associa una sequenza di parametri $\overline{p}$ a uno statement S eseguito all'attivazione dell'evento.	`F ::= (`$\overline{p}$`) => {S}`
S	**Statement (Istruzione)**: rappresenta il flusso di controllo, incluse chiamate a servizi, condizionali e iterazioni.	`S`
ϵ	**Istruzione Vuota**: indica la terminazione di un blocco di istruzioni.	ϵ
`call h(E)`	**Invocazione di Servizio**: chiamata a un servizio esterno h con parametri E. Può essere usata come istruzione o come guardia.	`call h(E)`
`if (G)`	**Condizionale**: esegue rami diversi in base alla guardia G (espressione booleana o chiamata a servizio).	`if...else`
`for (i...)`	**Iterazione**: ciclo che utilizza una variabile contatore i in un range da 0 al valore dell'espressione E.	`for...range`
G	**Guardia**: espressione o chiamata di servizio che deve restituire un valore booleano per determinare il flusso del condizionale.	`G`
E	**Espressione**: termini aritmetici o logici che possono includere costanti, contatori o parametri.	`E`
n, i, p	**Atomi**: rispettivamente numeri interi (n), indici di iterazione (i) e parametri di invocazione (p).	`n, i, p`
$\sharp$	**Operatori Binari**: set di operatori aritmetici ($+, -, *, /$), di confronto ($>, ==, >=$) e logici (&&).	$\sharp$

2.3 Sintassi delle Annotazioni

Tabella 2.2 mostra un resoconto generale su quale annotazione viene associata ai costrutti considerati nell'analisi, nella fase intermedia che precede la fase di estrazione:

Tabella 2.2: Corrispondenza tra costrutti Go e rispettive annotazioni miniSL.

Costrutto	**Annotazione (Commento Go)**
Punto di Ingresso	`// miniSL: main(parametri)`
Definizione Funzione Interna	`// miniSL: function nome(parametri)`
Mappatura Servizio Esterno	`// miniSL: defCall nome_esterno(parametri)`
Invocazione Servizio Esterno	`// miniSL: call nome(parametri)`
Invocazione Funzione Interna	`// miniSL: invoke nome(parametri)`
Blocco Condizionale	`// miniSL: if(condizione)`
Ramo Alternativo	`// miniSL: else`
Ciclo Iterativo	`// miniSL: for(contatore, limite)`
Chiusura Blocco	`// miniSL: end`

Una volta generate le annotazioni, il codice commentato viene analizzato per generare il codice miniSL. Le annotazioni vengono tradotte secondo la sintassi della tabella riassuntiva qui presentata:

Tabella 2.3: Corrispondenza tra Annotazioni e Output MiniSL

Annotazione	**Output MiniSL**
`// miniSL: main(p1, p2)`	`(p1, p2) => {`
`// miniSL: function f(p1)`	`function f(p1) {`
`// miniSL: call s(arg)`	`call s(arg)`
`// miniSL: invoke f(arg)`	`inline f(arg)`
`// miniSL: if(cond)`	`if (cond) {`
`// miniSL: else`	`} else {`
`// miniSL: for(i, N)`	`for (i in range(0, N)) {`
`// miniSL: end`	`}`

Capitolo 3

Implementazione

Dopo aver approfondito la teoria alla base del progetto, è il momento di trattare lo sviluppo e l'implementazione dell'idea dietro al programma. Nell'ottica di descrivere al meglio la struttura del progetto, le prossime sezioni del capitolo seguiranno un approccio dal generale al particolare, esponendo l'argomento nel suo complesso, per scendere nel particolare man mano che si procede.

3.1 Architettura del programma ed esecuzione

L'obiettivo generale è calcolare il costo dell'esecuzione di una funzione serverless, che effettua chiamate a servizi esterni (il concetto delle call nel progetto) in un sistema FaaS. L'idea del progetto è creare una catena di lavoro che inizi a lavorare su un codice Go "pulito" e restituisca come risultato finale una versione dello stesso codice nel linguaggio "giocattolo" miniSL. Il file in esame verrà analizzato da un annotatore, uno script che si occuperà di inserire dei commenti all'interno di questo file secondo le specifiche, e successivamente compilato da un secondo script, detto estrattore, che trasformerà questi commenti nel codice miniSL. Il sistema adottato utilizza un sistema di automazione scritto in Go, chiamato Task, ed è organizzato in forma di pipeline (Figura 3.1): un sistema composto da due fasi, l'annotazione e l'estrazione, che operano in sequenza e la cui invocazione è raccolta interamente nel file Taskfile.yml.

Possiamo considerare quest'ultimo script come l'intera pipeline e l'inserimento del comando da terminale come lo stage 1. L'annotazione e l'estrazione, sono rispettivamente fase 2 e 3 di Figura 3.1. Inizialmente, Taskfile.yml chiede all'utente di fornire da terminale alcuni argomenti in input insieme al comando principale, quali un entrypoint (nome di una funzione), il nome del file Go da annotare e il nome del file Go di output. Dopodiché queste variabili vengono processate dallo script annotatore.go che genera in output un file temporaneo annotato con la sintassi MiniSL, appoggiandosi a un file di configurazione JSON. Il codice commentato viene passato al file estrattore.go che, sfruttando anch'esso

un file JSON esterno, produrrà in output lo script finale in linguaggio MiniSL, che verrà salvato nella directory desiderata, o un errore dovuto a un codice non previsto dalle specifiche del linguaggio.

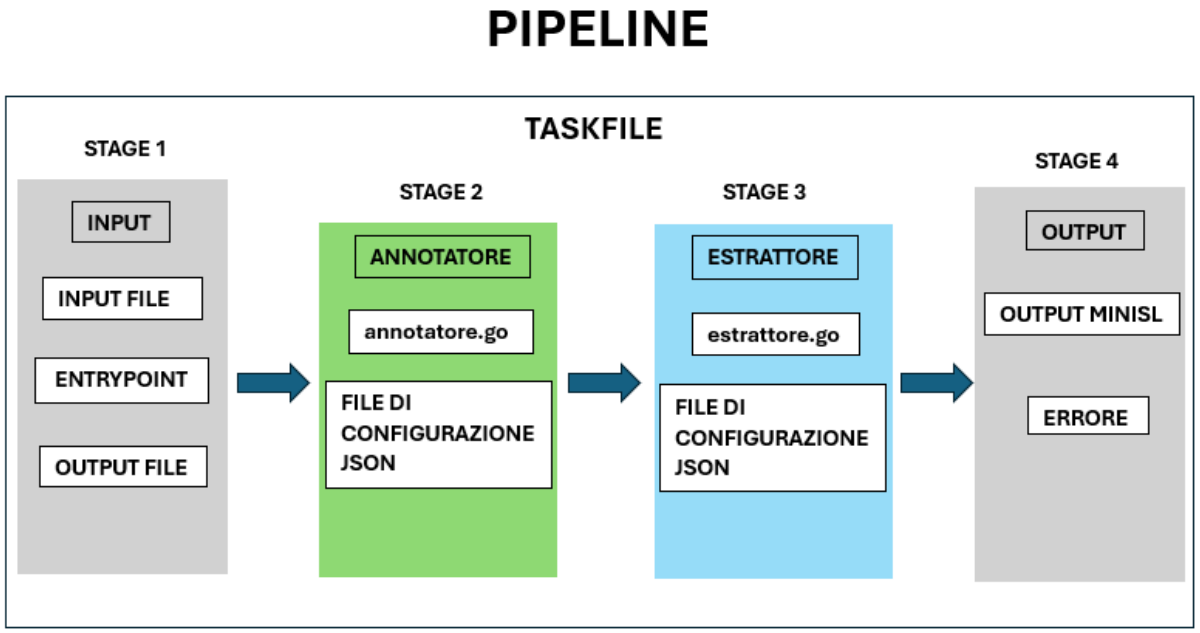

Figura 3.1: Architettura generale dell'applicazione

Nei prossimi sotto-capitoli introdurremo la logica del file Taskfile.yml e i comandi per avviare l'esecuzione vera e propria.

3.1.1 Task e Taskfile

Task è un esecutore di attività, scritto in Go, che lavora su file in formato YAML. Le attività permettono di implementare l'esecuzione di diversi comandi in un unico flusso automatizzato, così da non dover inserire manualmente ogni comando per avviare le diverse fasi del progetto.

L'esigenza di utilizzare Task nasce dalla necessità di avere un progetto versatile, in grado di essere gestito da più sistemi operativi, da Windows a Linux a macOS. Prima di Task il punto di riferimento storico per l'automazione di build era Make, sviluppato negli anni '70 in ambiente Unix e progettato con una sintassi basata su target e dipendenze. Rispetto a Make, Task non richiede adattamenti specifici per poter eseguire operazioni su Windows. Inoltre, con gli script Makefile, il cui linguaggio non prevede l'utilizzo di spazi bianchi ma del carattere `TAB` ed è facile cadere in errori di formattazione e riscontrare problemi difficili da debuggare. Task invece utilizza il formato YAML, una

sintassi più chiara e leggibile. Un'altra limitazione di Make è la dichiarazione `.PHONY` che ha il compito di comunicare quali target sono da considerare comandi da eseguire; altrimenti, in caso di presenza di un omonimo file, verrebbero ignorati. Di seguito ecco un breve esempio di script Makefile e del suo equivalente Taskfile:

```
APP_NAME = myapp
GO_FILES = $(wildcard *.go)
GO_CMD = go
GO_BUILD = $(GO_CMD) build
GO_TEST = $(GO_CMD) test

all: build

build: $(APP_NAME)

$(APP_NAME): $(GO_FILES)
   $(GO_BUILD) -o $(APP_NAME) $(GO_FILES)

test:
   $(GO_TEST) -v ./... -cover

.PHONY: all build test
```

Listato 3.1: Esempio di Makefile

Da Listato 3.1, è presente una prima sezione dedicata alla dichiarazione delle variabili e una seconda dedicata ai target, ossia `all`, `build` e `test`, i comandi veri e propri che il programma mette a disposizione. Le dipendenze sono indicate con `:` e definiscono la dipendenza dell'elemento a sinistra verso l'elemento a destra (in questo caso, ad esempio, il target build dipende dalla variabile `APP_NAME`). Sotto ad ogni target, l'indentazione con `TAB` identifica il comando o la serie di comandi associati a quel target, come `$( GO_BUILD ) -o $( APP_NAME ) $( GO_FILES )` per `build` o `$(GO_TEST) -v ./... -cover` per `test`. Il target `all` è una convenzione comune nei Makefile e specifica tramite dipendenza l'ordine delle operazioni da eseguire nel caso in cui oltre al comando `make` non venga specificato un target. Infine, l'ultima riga dello script, come già accennato in precedenza, assicura che tutti i target vengano etichettati come eseguibili.

```
version: "3"

vars:
  APP_NAME: myapp

tasks:
```

```
7   default:
8     cmds:
9       - task: build
10
11  build:
12    cmds:
13      - go build -o {{.APP_NAME}} *.go
14    sources:
15      - "*.go"
16    generates:
17      - "{{.APP_NAME}}"
18
19  test:
20    cmds:
21      - go test -v ./... -cover
```

Listato 3.2: Esempio di Taskfile.yml

YAML permette invece una migliore leggibilità e comprensione del codice. Nel Listato 3.2 la prima riga specifica la versione del formato Taskfile, così che sia possibile usufruire delle nuove funzionalità semplicemente cambiando il valore della versione, un aspetto importante di cui Make non dispone. La sezione `vars` contiene le variabili utilizzate dallo script e si presenta molto più pulita e chiara rispetto all'equivalente Make. Ciò avviene perché Task non ha bisogno di dichiarare esplicitamente i comandi Go come variabili. Gli equivalenti dei target sono i task, ovvero `default`, `build` e `test`, definiti nel Taskfile sotto la sezione `tasks`. Il task `default` ha la stessa funzione del target `all` del Makefile ed esegue l'attività indicata nel caso l'utente non specificasse quale task compiere. L'azione che ogni task, quando invocato, dovrà eseguire è stabilita nella sottosezione `cmds`, abbreviazione di commands. La formattazione YAML agevola la scrittura di codice evitando di dover prestare attenzione all'indentazione con TAB e facilita la manutenibilità del codice.

Anche la gestione delle dipendenze mostra alcune differenze rispetto alla versione Make; per dichiararle ci sono diversi modi: i più espliciti consistono nell'inserire, per il task desiderato, una sezione `deps` in cui scrivere la sequenza di operazioni da cui dipende quel task prima di essere avviato, oppure, come nel Listato 3.2, far dipendere il task `default` dal task `build` tramite la sezione `cmds`; mentre una via più implicita si fonda su dipendenze basate sui file tramite `sources` e `generates`, il primo dice a Task di controllare tutti i file con estensione .go mentre il secondo specifica il nome del file da produrre in output. Il comportamento è identico in entrambi gli script, ma Task richiede maggiore chiarezza su quali file monitorare e quali produrre.

Oltre alle funzionalità appena elencate, Task fornisce altri due comandi predefiniti: `task --list` e `task --parallel`. Il primo mostra l'elenco dei task definiti nel Taskfile compresa la loro descrizione testuale se presente, mentre il secondo esegue i task spe-

cificati da terminale contemporaneamente ed è la soluzione migliore per velocizzare il processo con azioni indipendenti tra loro (`task --parallel build test` dallo script di esempio).

3.1.2 Il file Taskfile.yml

Come descritto nelle sezioni precedenti, il progetto sfrutta Task per automatizzare l'esecuzione del programma. In questa sezione analizziamo la logica del file Taskfile.yml e come eseguirlo da terminale. Il file contiene una prima parte dedicata alla definizione delle variabili globali `TIMESTAMP` e `TEMP_DIR`, come da Listato 3.3.

```
version: "3"

vars:
  TIMESTAMP:
    sh: |
      if command -v date >/dev/null 2>&1; then
        date +%s 2>/dev/null || echo $RANDOM$RANDOM
      else
        echo $RANDOM$RANDOM
      fi
  TEMP_DIR:
    sh: |
      if [ "$OS" = "Windows_NT" ] || [ -n "$WINDIR" ]; then
        if [ -n "$TEMP" ]; then
          echo "$TEMP"
        elif [ -n "$TMP" ]; then
          echo "$TMP"
        else
          echo "./temp"
        fi
      else
        if [ -d "/tmp" ]; then
          echo "/tmp"
        else
          echo "./temp"
        fi
      fi
```

Listato 3.3: Variabili globali di Taskfile.yml

Entrambe le variabili garantiscono l'unicità di ogni file temporaneo creato e il suo salvataggio in una directory predefinita, facilmente recuperabile dal sistema. `TIMESTAMP` ottiene il valore prodotto in output dallo script identificato dall'etichetta `sh`. Queste

istruzioni vengono eseguite quindi come script di shell e prevedono l'installazione di Git Bash se si sta operando su Windows come sistema operativo, in modo da simulare un ambiente Unix-like. La condizione del primo `if` verifica se il comando `date` è disponibile nel sistema e `command -v` restituisce il percorso del comando. `date` restituisce, per sua definizione, il numero di secondi passati dal 1 Gennaio 1970, in questo modo avrà un valore diverso ad ogni invocazione e assicura che `TIMESTAMP` abbia un valore univoco. Se `date +%s` fallisce oppure il percorso non esiste, viene generato un numero pseudo-casuale attraverso `$RANDOM$RANDOM`.

`TEMP_DIR` definisce, invece, in quale cartella creare i file temporanei. Per farlo determina dinamicamente la directory corretta, a seconda che ci si trovi su Windows o su Linux/macOS. Se la prima condizione è soddisfatta, ci troviamo in ambiente Windows; successivamente lo script controlla se nel sistema sono presenti le variabili `$TEMP` o `$TMP`, e ripiega in caso contrario sulla cartella locale `./temp`. La stessa logica viene applicata, per il ramo else, a un ambiente Unix-like e la variabile `TEMP_DIR` conterrà il percorso della cartella locale dei file temporanei.

Il fulcro del codice è costituito infine dalla task principale `pipeline`. La task definisce al suo interno delle proprie variabili locali, come mostrato nel Listato 3.4.

```
vars:
      ENTRYPOINT: "{{.ENTRYPOINT | default \"main\"}}"
      INPUT: "{{.INPUT}}"
      OUTPUT: "{{.OUTPUT}}"
      TEMP_FILE: "{{.TEMP_DIR}}/pipeline_{{.TIMESTAMP}}.tmp"
```

Listato 3.4: Variabili locali della pipeline

I primi tre elementi sono, inoltre, anche i parametri da passare in input al terminale al momento dell'esecuzione dell'istruzione. Durante l'inserimento del comando, l'unica limitazione riguarda il numero di argomenti di input, dei quali devono fare necessariamente parte le variabili `INPUT` e `OUTPUT` per il corretto prosieguo dell'operazione. Il valore della variabile `ENTRYPOINT`, se non specificato, assume il valore 'main' di default.

Seguente alla sezione `vars` sono presenti alcuni controlli sulla validità dei dati di input e degli script di annotazione ed estrazione, elencati nel costrutto delle `preconditions` (Listato 3.5). Le precondizioni sono una lista di comandi shell, utili per verificare che l'ambiente sia pronto e per prevenire i principali errori; ogni controllo deve essere soddisfatto prima di eseguire il task.

```
preconditions:
      - sh: test -n "{{.INPUT}}"
        msg: "ERRORE: Specificare INPUT=<file_input>"
      - sh: test -n "{{.OUTPUT}}"
        msg: "ERRORE: Specificare OUTPUT=<file_output>"
      - sh: test -f "{{.INPUT}}"
        msg: "ERRORE: File {{.INPUT}} non trovato"
```

```
8         - sh: test -f "annotatore.go"
9           msg: "ERRORE: annotatore.go non trovato"
10        - sh: test -f "estrattore.go"
11          msg: "ERRORE: estrattore.go non trovato"
```

Listato 3.5: preconditions

Un'altra funzionalità introdotta rispetto a Make è la possibilità di associare alle `preconditions` un messaggio di errore che chiarisca la ragione del problema riscontrato. Nel contesto del progetto, le istruzioni `test -f` e `test -n` si accertano della presenza degli script nel filesystem e che le variabili di input non siano vuote.

Infine, l'ultima sezione della task `pipeline` (Listato 3.6) include i comandi attorno ai quali si sviluppa l'intero programma, eseguiti nell'ordine predisposto.

```
1  cmds:
2         - echo "Esecuzione annotatore..."
3         - go run annotatore.go {{.ENTRYPOINT}} {{.INPUT}} > "{{.
             TEMP_DIR}}/pipeline_{{.TIMESTAMP}}.tmp"
4         - echo "Esecuzione estrattore..."
5         - go run estrattore.go {{.ENTRYPOINT}} "{{.TEMP_DIR}}/
             pipeline_{{.TIMESTAMP}}.tmp" > {{.OUTPUT}}
6         - echo "Completato! Output salvato in {{.OUTPUT}}"
7         - defer: rm -f "{{.TEMP_DIR}}/pipeline_{{.TIMESTAMP}}.tmp"
             2>/dev/null || del "{{.TEMP_DIR}}\pipeline_{{.TIMESTAMP
             }}.tmp" 2>nul || true
```

Listato 3.6: comandi eseguiti dalla task

Le istruzioni `echo` non producono alcun effetto concreto se non quello di informare l'utente su cosa sta succedendo, stampando l'informazione a terminale. Dopodichè, con `go run` vengono compilati sul momento ed eseguiti gli script 'annotatore.go' e 'estrattore.go'. L'output generato da entrambi i codici viene salvato nel file dichiarato dopo il carattere `>`.

L'ultimo comando della lista, associato alla parola chiave `defer`, viene eseguito a prescindere dal fatto che le operazioni sopra siano state completate con successo e solo alla fine del task. `defer`, abbreviazione di "deferred command", è una direttiva specifica del tool Task che consente di definire comandi che verranno avviati alla fine dell'intero task. In questo caso particolare è utile per rimuovere il file temporaneo creato dall'annotatore ed è compatibile sia in ambiente Unix grazie al comando `rm` sia in ambiente Windows grazie al comando `del`. Il file temporaneo, se presente, viene dunque eliminato per mantenere la directory pulita e non avere file passati in eccesso. Prima di concludere il sottocapitolo, vediamo come applicare la teoria accumulata fino ad ora con qualche esempio.

3.1.3 Esempio di esecuzione del programma

Avendo integrato il progetto nel proprio sistema, è possibile eseguire il programma aprendo il terminale, o prompt dei comandi, locale e cambiando, tramite l'apposito comando, directory inserendo il percorso alla cartella contenente l'eseguibile Taskfile.yml (in questo caso specifico il percorso ha al suo interno anche il resto degli script di codice del progetto). Inserire il comando del Listato 3.7 se si sta lavorando su Windows, altrimenti eseguire l'input del Listato 3.8.

```
C:\Users\Lenovo\Desktop\Tesi_miniSL> task pipeline ENTRYPOINT=
    main INPUT=testInput.go OUTPUT=risultato.go
```

Listato 3.7: Esecuzione su Windows

```
$ task pipeline ENTRYPOINT=main INPUT=testInput.go OUTPUT=
    risultato.go
```

Listato 3.8: Esecuzione su Unix

Se tutto procede correttamente, il terminale mostrerà la seguente stampa (Listato 3.9):

```
task: [pipeline] echo "Esecuzione annotatore..."
Esecuzione annotatore...
task: [pipeline] go run annotatore.go main testInput.go > "C:\
    Users\Lenovo\AppData\Local\Temp/pipeline_.tmp"
task: [pipeline] echo "Esecuzione estrattore..."
Esecuzione estrattore...
task: [pipeline] go run estrattore.go main "C:\Users\Lenovo\
    AppData\Local\Temp/pipeline_.tmp" > risultato.go
task: [pipeline] echo "Completato! Output salvato in risultato.go
    "
Completato! Output salvato in risultato.go
task: [pipeline] rm -f "C:\Users\Lenovo\AppData\Local\Temp/
    pipeline_.tmp" 2>/dev/null || del "C:\Users\Lenovo\AppData\
    Local\Temp\pipeline_.tmp" 2>nul || true
```

Listato 3.9: Esecuzione su Windows

Il file prodotto in output viene salvato nella directory corrente, insieme ai file di progetto. La stampa di errori, che non riguardano le preconditions, nel codice in input è, invece, esterna al Taskfile ed è da ricercare nello script dell'annotatore: durante i controlli di validazione, nel caso di insuccesso, il codice stampa il messaggio di errore appropriato utilizzando `stderr`, uno dei tre canali di input/output di default di ogni sistema operativo. `stderr` offre un'alternativa, destinata interamente agli errori, al canale di output

`stdout`, dedicato ai risultati di un programma. Il messaggio è mostrato a terminale ed interrompe l'esecuzione della pipeline prima dell'avvio dell'estrattore.

Nel prossimo sottocapitolo, introduciamo la fase di annotazione di uno script Go con la sintassi miniSL.

3.2 1° fase: Annotazione

L'annotatore rappresenta la componente principale e più corposa della pipeline, in quanto gestisce i maggiori errori derivabili dalle logiche dell'input non annotato. È la fase responsabile della corretta identificazione dei costrutti e delle operazioni che vogliamo tradurre nel linguaggio miniSL. La logica di annotazione ruota attorno al principio delle call, generando i commenti con la sintassi miniSL solo per le parti di codice relative alle chiamate a servizi esterni.

Durante l'annotazione, vengono generati i commenti sopra gli elementi che hanno una corrispondenza in miniSL, secondo la sintassi spiegata nei capitoli d'introduzione. Ad esempio, l'annotazione di un'invocazione di una call come nel Listato 3.10:

```
1 func main() {
2     service()
3 }
```

Listato 3.10: Call in Go

Corrisponderà all'output del Listato 3.11:

```
1 func main() {
2     //miniSL: call service()
3     service()
4 }
```

Listato 3.11: Esempio annotazione in output

Il codice viene analizzato seguendo l'AST (Abstract Syntax Tree) statement-by-statement e attraversando il grafo delle chiamate di funzione. Una volta ricevuto in input il nome della funzione fungente da entrypoint e controllata la sua validità, l'annotazione procede circoscritta al contenuto della funzione, annotando in ordine i nodi dell'albero sintattico che ne deriva. In caso di presenza di chiamate di funzione, l'analisi interromperà temporaneamente il suo flusso regolare per procedere a commentare la funzione invocata e il suo contenuto, comportandosi esattamente come con la funzione entrypoint. I nodi processati dall'annotatore sono definiti dagli statement accettati dalla sintassi miniSL.

Un approccio di questo tipo rappresenta una soluzione migliore rispetto a un'analisi sequenziale del codice sorgente, che non permetterebbe la corretta annotazione delle chiamate di funzione, creando una forte dipendenza dalla posizione all'interno dello script.

Inoltre, tale strategia implicherebbe uno spreco energetico inutile in termini di costi computazionali: alcune funzioni potrebbero venire analizzate anche qualora non vengano mai invocate, oppure venire lette più volte dall'annotatore. Altro punto a sfavore di questa scelta di lavoro è l'impossibilità di tenere traccia dello stato delle variabili, quindi il loro valore, durante l'annotazione. Al momento una caratteristica importante del codice è, infatti, la capacità di tracciare il cambiamento del valore che le variabili assumono nello script, per poi poterlo sostituire nelle annotazioni. Un'implementazione sequenziale creerebbe valori errati e una confusione generale nel risalire all'origine del problema.

Per gestire la struttura delle annotazioni, inoltre, è stato creato un file JSON di supporto, che servisse da documentazione dei formati utilizzati per ogni costrutto. La sintassi miniSL è così raccolta interamente nel file annotatore.json e la logica dell'annotatore è basata sulla comunicazione con il codice JSON al momento della produzione di ogni nota.

Prima di entrare nel dettaglio delle scelte tecniche adottate, nelle prossime sezioni descriviamo l'organizzazione dello script JSON e alcune preferenze concettuali dell'annotatore.

3.2.1 JSON e la sua applicazione

La creazione dei commenti avviene tramite un file di configurazione JSON che include i pattern di riconoscimento delle annotazioni e i template per la generazione di essi. Questa scelta progettuale garantisce maggiore flessibilità al codice in ottica di eventuali sviluppi futuri, e semplifica l'annotazione evitando l'hardcoding dei template miniSL nello script. La manutenibilità del codice sorgente è così salvaguardata dalla possibilità di apportare modifiche solamente al file JSON in caso di bisogno.

Il file di configurazione presenta una struttura di questo tipo:

```
{
  "start_annotation": "// miniSL:",
  "function": "{{.start}} function {{.name}}({{.params}})",
  "main": "{{.start}} main({{.params}})",
  "call": "{{.start}} call {{.name}}({{.params}})",
  "invoke": "{{.start}} invoke {{.name}}({{.params}})",
  "if": "{{.start}} if({{.cond}})",
  "for": "{{.start}} for({{.var}},{{.limit}})",
  "else": "{{.start}} else",
  "end": "{{.start}} end"
}
```

Listato 3.12: File di configurazione JSON

A sinistra di ogni riga è dichiarato un identificatore che mappa gli elementi che verranno considerati durante l'annotazione, mentre a destra sono definiti i template formati

temporaneamente da placeholder racchiusi tra doppie parentesi, che verranno sostituiti al momento dell'analisi. La parola chiave `start_annotation` indica il prefisso che differenzia le note inserite dall'utente da quelle da tradurre in miniSL tramite l'estrattore, mentre le definizioni dei termini `call` e `invoke` distinguono le chiamate a servizi esterni dalle funzioni standard. La dichiarazione `end`, invece, determina la fine con parentesi graffa di ogni if, for o funzione.

Come mostrato nel Listato 3.12, molti template presentano una forma ben delineata, con placeholder che si ripetono in quasi tutte le strutture sintattiche: `.start` è il prefisso `start_annotation`, `.name` identifica il nome della funzione o call invocata o definita, `.params` verrà scambiato con la lista dei parametri passati in input e `.cond` specifica esclusivamente la condizione di un if. Per i cicli for, come previsto dalla sintassi miniSL, l'annotazione sarà invece composta dalla coppia di variabili (`.var`,`.limit`) di cui il secondo assume un valore numerico.

All'interno del codice dell'annotatore, i due file comunicano attraverso la funzione `applyTemplate` (Listato 3.13), l'unico punto di accesso per l'applicazione dei template JSON del sistema.

```
1  func applyTemplate(key string, data map[string]string) string {
2    tmplStr, ok := annotatorConfig[key]
3    if !ok {
4      return ""
5    }
6    tmpl, err := template.New("annotation").Parse(tmplStr)
7    if err != nil {
8      return ""
9    }
10   var buf bytes.Buffer
11   tmpl.Execute(&buf, data)
12   return buf.String()
13 }
```

Listato 3.13: funzione applyTemplate

`applyTemplate` è responsabile della creazione delle annotazioni sotto forma di stringhe e prende in input due parametri: una chiave (di tipo stringa) e una mappa contenente i dati (sempre di tipo stringa) da sostituire al posto dei placeholder. Una volta invocata, la funzione utilizza la chiave per trovare il template di riferimento (main, call, if, for, ...) e, se quest'ultima esiste e il parsing della sintassi template non ritorna errore, i dati della mappa vengono applicati tramite `tmpl.Execute(&buf, data)`. In output viene restituita l'annotazione completa. In caso di errori, la funzione non interrompe completamente il processo di annotazione, ma restituisce una stringa vuota, non mostrando l'annotazione per quel caso specifico.

Una struttura come la mappa dei dati aggiunge ulteriore flessibilità al progetto e garantisce che tutti i dati siano stringhe, evitando errori di tipo durante l'esecuzione dei template. Inoltre, il JSON e la funzione sono indipendenti tra loro, ciò significa che è possibile aggiungere template o placeholder ad annotatore.json senza modificare `applyTemplate`. Di seguito ecco un esempio del funzionamento con la funzione `main()` senza parametri.

```
annotations[startLine] = append(annotations[startLine],
   applyTemplate("main", map[string]string{
      "start":  annotatorConfig["start_annotation"],
      "params": strings.Join(params, ","),
    }))
```

Listato 3.14: esempio funzionamento di applyTemplate

Quando la funzione viene chiamata con la chiave `main`, i placeholder vengono dinamicamente sostituiti con i valori richiesti. In questa situazione `params` verrà aggiornato previa analisi e registrazione di eventuali parametri nell'omonima variabile all'interno dell'annotatore. L'output atteso sarà questo:

```
// miniSL: main()
```

Listato 3.15: esempio output funzione main

Le annotazioni generate vengono successivamente salvate in una mappa globale chiamata `annotations`, dichiarata all'inizio del codice. La coppia chiave-valore della mappa è costituita da un intero, per indicare il numero della riga dell'elemento annotato, e una stringa contenente l'annotazione. Ogni nuovo commento viene aggiunto in coda tramite il metodo built-in `append` dell'oggetto map.

3.2.2 Scelte implementative: annotatore

Architettura generale e strutture dati

L'architettura su cui si basa il sistema si compone di più processi, di analisi, validazione e generazione di annotazioni, agenti contemporaneamente durante l'esecuzione del programma, ma separati a livello di codice all'interno del file. Le specifiche del progetto impongono la necessità di tenere traccia dello stato del programma durante l'esecuzione, in particolare delle variabili. La struttura dati fondamentale in questo senso è `VariableState`, responsabile di tracciare attraverso quattro mappe le variabili aventi un valore, quelle che potrebbero causare errore e i parametri. Come anticipato nell'introduzione della sezione, l'annotazione deve ritornare errore soltanto se l'analisi del codice sorgente contiene problemi di sintassi in un contesto in cui vengono effettuate delle chiamate a servizi esterni; altrimenti si possono ignorare.

```
type VariableState struct {
  variables       map[string]string
  notEvaluable    map[string]bool
  parameters      map[string]bool
  problematicVars map[string]bool
}
```

Listato 3.16: Struttura dati VariableState

A completare l'architettura si aggiungono due mappe globali, `internalFuncs` e `funcToCallName`, per tenere traccia di tutte le funzioni definite nel file di input, distinguendole tra funzioni interne e chiamate a servizi esterni. Come anticipato spiegando il JSON, tra le variabili globali è implementata anche la mappa `annotations`, che accumula progressivamente le annotazioni create.

Gestione dello stato e analisi statica del codice

La funzione cruciale dell'implementazione è `analyzeStatementWithState`, l'annotatore vero e proprio, ovvero il metodo che raccoglie al suo interno la maggior parte delle altre funzioni di validazione e analisi e si occupa di riempire la lista di annotazioni vista nella sezione precedente. Durante lo svolgimento dello script, l'elaborazione del codice sorgente avviene attraverso un'analisi statica del file di input, mantenendo traccia dei valori delle variabili, quando possibile, e contrassegnando come problematiche le altre. Una variabile è problematica quando, secondo le regole del linguaggio miniSL, non è possibile prevedere il suo valore staticamente, oppure essa viene assegnata tramite chiamate di funzione. La strategia attuata è necessaria affinchè si possano tracciare variabili all'interno di blocchi if o for; al contrario, sarebbe impossibile valutare assegnazioni inserite in contesti dinamici.

Il metodo `analyzeStatementWithState` processa ogni statement singolarmente, aggiorna lo stato e decide l'annotazione per tale elemento. La sua logica è composta interamente da un blocco switch che gestisce ogni caso verificabile.

```
func analyzeStatementWithState(stmt ast.Stmt, state *
   VariableState) {
    switch s := stmt.(type) {
     case *ast.AssignStmt:
      // Processa l'assegnamento
     case *ast.IfStmt:
    // Simula i rami dell'if e verifica i vincoli di sicurezza
     case *ast.ForStmt:
      // Simula le iterazioni e annota il blocco
     case *ast.ExprStmt:
      // Decide quali funzioni annotare con invoke o con call
     case *ast.IncDecStmt:
```

```
12          // Aggiorna il valore del contatore: i++ e i--
13        case *ast.BlockStmt:
14          analyzeStatementsSequentially(s.List, state)
15        case *ast.DeclStmt:
16          // Assegnazioni di dichiarazioni, come per il primo caso
17      }
18  }
```

Listato 3.17: Struttura funzione analyzeStatementWithState

Il Listato 3.17 mostra lo scheletro generale della funzione, senza i dettagli implementativi. Il metodo prende come parametri in input uno statement e un puntatore alla struttura dati `VariableState`, per controllare e, eventualmente, aggiornare lo stato delle variabili conosciute fino a quel momento. A seconda del tipo di statement, il blocco si dirama nei seguenti casi più importanti: se si tratta di un'assegnazione, la variabile viene catalogata come costante, nota tramite espressione aritmetica o non valutabile; se ricade nei casi `*ast.IfStmt` o `*ast.ForStmt`, ispeziona staticamente il flusso dei relativi blocchi; nel quarto caso, invece, controlla le chiamate di funzione e decide se annotarle con la sintassi destinata alle call o alle funzioni locali. L'eventuale `*ast.BlockStmt` è altrettanto importante per processare il contenuto dei blocchi if, for e funzione, e il parametro `s.List` è una lista ordinata degli statement contenuti.

Sistema di pre annotazione

Riguardo al caso `*ast.ExprStmt`, è stata presa la decisione di specificare manualmente quali funzioni sarebbero state call e quali no, prima dell'esecuzione dell'annotatore. Ciò è stato implementato applicando una sintassi differente ai due elementi: le call, sia che siano definite nel codice sia che siano chiamate a servizi esterni, vengono marcate con il template `// miniSL: defCall()`, mentre le funzioni locali, se viene trovata la definizione corrispondente nello script, vengono marcate con `invoke` durante l'annotazione. Oltre a `defCall`, il codice reagisce nella stessa maniera con qualunque commento impostato dall'utente che inizi con il prefisso `// miniSL:`, considerandolo come valido di default. Così facendo è possibile personalizzare lo script secondo le proprie esigenze. Questa scelta cede parte della responsabilità della buona riuscita del programma all'utente, ma rende più funzionale il codice senza la necessità di appesantirlo con ulteriori controlli per determinare la natura delle funzioni.

Gestione del flusso di analisi

Le annotazioni associate al placeholder `invoke` implicano l'inlining, ossia la completa sostituzione di una funzione con il suo corpo, di quei metodi. La scelta concettuale alla base dell'inlining impone che questa tecnica venga propagata anche agli `invoke` annidati in altre funzioni, senza limite di profondità, fino alla completa espansione

del codice. A livello di implementazione, ciò viene gestito da una funzione apposita, `annotateFunctionBodyWithState`, di cui parleremo prossimamente.

La funzione `analyzeStatementWithState` agisce cooperando con un'altra funzione "regista" (Listato 3.18), la quale è composta semplicemente da un ciclo for che scorre i nodi dell'albero sintattico in ordine e invoca, per ogni statement, il metodo sopra citato.

```
1 func analyzeStatementsSequentially(stmts []ast.Stmt, state *
     VariableState) {
2   for _, stmt := range stmts {
3     analyzeStatementWithState(stmt, state)
4   }
5 }
```

Listato 3.18: Struttura funzione analyzeStatementSequentially

L'obiettivo di `analyzeStatementSequentially` è garantire la coerenza sequenziale nell'analisi: quando un'istruzione aggiorna lo stato, la modifica è visibile alle istruzioni successive. Non interpreta i nodi e non emette regole proprie; assicura soltanto che lo stato venga propagato correttamente. In assenza di un approccio di questo tipo, la responsabilità ricadrebbe interamente su `analyzeStatementWithState` e richiederebbe di scrivere un ciclo for interno per ogni blocco annidato (if, for o invoke), duplicando il codice e appesantendo inutilmente lo script. Inoltre, con la disposizione attuale, è più facile inserire regole sull'analisi sequenziale, aggiungendo codice solo nella funzione `analyzeStatementSequentially`.

Dopo aver avviato l'esecuzione del programma da terminale, se non ci sono stati problemi, il flusso inizia dalla funzione entrypoint. Prima di analizzare la funzione invocando il metodo `analyzeFunctionBodyWithState`, che a cascata chiamerebbe gli altri metodi elencati sopra, viene caricata la configurazione delle stringhe di annotazione dal file JSON e viene eseguito il parsing Go con `parser.ParseFile` per costruire l'AST.

3.2.3 Sistema di tracciamento delle variabili

Uno degli aspetti più complessi dell'annotatore è rappresentato dal tracciamento delle variabili, fondamentale per garantire il massimo determinismo nel progetto. Il sistema di tracciamento si basa sulla struttura `VariableState` vista precedentemente (Listato 3.16), che mantiene le variabili classificate in quattro mappe distinte:

- `variables`: contiene le variabili con valori noti o calcolabili staticamente, attraverso espressioni aritmetiche o simboliche, ossia quando dipendono da parametri di funzione o costanti.

- `notEvaluable`: questa categorizzazione è fondamentale per prevenire errori semantici in futuro, contrassegna le variabili il cui valore non può essere determi-

nato staticamente a causa di dipendenze da chiamate di funzione o da variabili sconosciute.

- `parameters`: identifica le variabili che rappresentano parametri formali delle funzioni e, per questo motivo, il cui valore può essere non noto in quanto derivante dall'invocazione della funzione a cui appartengono.

- `problematicVars`: traccia le variabili assegnate in contesti non deterministici, come i rami di un if con guardia non valutabile o cicli for la cui terminazione non è determinata.

Nel codice sono presenti anche funzioni che gestiscono le operazioni di gestione dello stato. La funzione `newVariableState` crea una nuova struttura di stato per le variabili ed è chiamata all'inizio della funzione main. Oltre all'inizializzazione della struttura, vengono gestite anche le mappature delle variabili, implementate nei metodi `setVariable`, `setNoEvaluable`, `setParameter` e `setProblematicVar`. Ogni variabile analizzata viene inserita in una di queste liste a seconda della situazione attraverso queste funzioni.

Durante il processo di condizioni if o cicli for, una caratteristica importante del codice è la clonazione dello stato delle variabili per procedere con l'esecuzione senza alterare lo stato originale. La funzione `clone()`, esposta di seguito, crea una copia indipendente della struttura `VariableState`:

```
1  func (vs *VariableState) clone() *VariableState {
2    newState := newVariableState()
3    for k, v := range vs.variables {
4      newState.variables[k] = v
5    }
6    for k, v := range vs.notEvaluable {
7      newState.notEvaluable[k] = v
8    }
9    for k, v := range vs.parameters {
10     newState.parameters[k] = v
11   }
12   for k, v := range vs.problematicVars {
13     newState.problematicVars[k] = v
14   }
15   return newState
16 }
```

Listato 3.19: Funzione clone

Viene usata in if con condizione determinabile per percorrere il ramo falso e annotarlo correttamente, senza sporcare lo stato ufficiale con valori errati, oppure in cicli for per salvare lo stato delle annotazioni e conservare solo quello dell'ultima iterazione. È utilizzata anche quando viene effettuato l'inlining di una funzione: avendo scope diversi, viene

creato uno stato locale inizializzato a partire da quello attuale, in modo che, analizzando la funzione chiamata, non venga modificato lo stato del chiamante.

I vantaggi di questa scelta sono diversi: senza clone, le assegnazioni fatte in un ramo non preso inquinerebbero lo stato globale ed è possibile esplorare più scenari generando annotazioni esatte, mantenendo integro allo stesso tempo lo stato principale.

Assegnamenti e dichiarazioni

Una logica più basilare appartiene, invece, agli assegnamenti e alla dichiarazione di variabili, dove avviene il tracciamento. Questi statement sono gestiti prevalentemente nella funzione `analyzeStatementWithState`, in cui vengono valutati, tramite più controlli di validazione, passando per più funzioni dedicate esclusivamente a questo. Il sistema tenta prima una valutazione numerica attraverso la funzione `evalExprWithState`, spiegata di seguito:

```
1  func evalExprWithState(expr ast.Expr, state *VariableState) (int,
       bool) {
2    switch e := expr.(type) {
3    case *ast.BasicLit:
4      ...
5    case *ast.Ident:
6      ...
7    case *ast.BinaryExpr:
8      ...
9    default:
10     return 0, false
11   }
12 }
```

Listato 3.20: Funzione evalExprWithState

E' una funzione che cerca di valutare aritmeticamente un espressione dall'albero sintattico (`ast.Expr`), recuperando i dati dalla struttura globale che gestisce lo stato. Funziona gestendo tre casi definiti all'interno di un blocco switch: se l'espressione è un numero scritto e visibile nel codice sorgente, viene ricevuta dal primo caso e convertita in un intero; altrimenti, se è una variabile (ad esempio "x"), cerca il valore di quest'ultima in `VariableState` e lo applica. Il caso `*ast.BinaryExpr` è inserito per controllare le espressioni aritmetiche binarie (x+3, x*y, x/5) e chiama ricorsivamente `evalExprWithState` per gli elementi a sinistra e a destra dell'operatore aritmetico.

Questo metodo è importante soprattutto per valutare le condizioni degli if o le guardie dei cicli for e serve a informare l'analisi degli statement successivi sul contesto nel quale sta operando. Un espressione potrebbe, tuttavia, contenere parametri della funzione nella quale è definita. In questo caso, il codice fa affidamento su un ulteriore

metodo che avvisa il sistema del problema specifico della formula, in modo che non restituisca errore al sistema. La circostanza impedisce, infatti, di ridurre l'espressione a un numero concreto, avendo elementi non noti. Secondo le specifiche miniSL, però, la situazione è sufficientemente deterministica da poter essere annotata anche in presenza di valori simbolici. Questi saranno sostituiti dai parametri passati in input al momento dell'invocazione della funzione. Durante l'annotazione degli invoke, l'annotatore richiama `analyzeStatementWithState` che a sua volta invoca la nuova funzione `isExpressionEvaluableWithParameters`.

La struttura di `isExpressionEvaluableWithParameters` è identica alla sua funzione complementare, presentata nel Listato 3.20. Di seguito è riportato un esempio del contesto appena descritto e accettato dalle specifiche del programma:

```
1  func somma(a int) int {
2    y := a + 1 // non causa errore
3  }
4
5  func main(){
6      x := 5
7      somma(5)
8  }
```

Listato 3.21: Esempio di un assegnazione valida con elementi non noti

Motivo di errore e di interruzione è, tuttavia, qualsiasi assegnazione, pur sintatticamente e semanticamente corretta, compresa in un blocco di codice avente una guardia o condizione non calcolabile (con variabili non conosciute o chiamate di funzioni/call). Le variabili assegnate in questi contesti vengono automaticamente salvate nella mappa `problematicVars` della struttura `VariableState`. Nel Listato 3.22 sono elencati i contesti non deterministici che portano all'errore sopra spiegato.

```
1  func main(){
2      z := test()
3      if(test()){ // Non si puo' verificare quale ramo scegliere
4          x := 5
5      }else{
6          x := 10
7      }
8
9      for i := 0; i < z; i++ {
10     x := 1  //  La guardia del for non e' determinabile
11   }
12 }
```

Listato 3.22: Esempio di contesti non deterministici per assegnazioni

Nelle prossime sezioni iniziamo a descrivere le logiche di annotazione per ogni costrutto miniSL.

3.2.4 Funzioni: annotazione

La fase di annotazione delle funzioni costituisce il cuore del programma, poichè è la base, a partire dall'entrypoint, dalla quale inizia l'analisi. Prima di invocare il metodo `analyzeFunctionBodyWithState`, viene effettuata l'individuazione e la classificazione delle funzioni del codice sorgente, scansionando tutte le dichiarazioni di funzione grazie alla struttura `ast.FuncDecl`, definita nel pacchetto `go/ast` della standard library. I dati risultanti dalla computazione vengono memorizzati, a seconda della loro natura, nella mappa globale `funcToCallName`, se la riga immediatamente precedente alla funzione ha un commento `// miniSL: defCall nomeCall()`, oppure nella mappa globale `internalFunc`, per i metodi non marcati come servizi esterni. Se nel codice sono presenti chiamate di funzione senza definizione, ma con pre commento `// miniSL: call nomeCall()`, rientrano nell'ambito della prima mappa.

E' importante sottolineare come, nei casi delle call, siano le annotazioni ad avere la precedenza per le chiamate ai servizi esterni: se, ad esempio, la funzione `test()` è annotata dall'utente come `// miniSL: call service()`, il nome di riferimento per l'annotatore, e in particolare l'estrattore, sarà quest'ultimo.

Validazione e costruzione del grafo delle chiamate

Prima di procedere con l'annotazione vera e propria, l'annotatore esegue delle verifiche sulla sicurezza delle invocazioni, per eludere eventuali loop. Inizialmente, costruisce il grafo delle chiamate per registrare quali funzioni richiamano quali altre, salvando i vincoli in un'altra mappa globale dichiarata all'inizio del codice, chiamata `functionCalls`.

```
functionCalls = {
  "main": ["X", "Y"],    // chiama X e Y (Y indirettamente)
  "X": ["Y"],            // chiama Y
  "Y": ["Z"],            // chiama Z
  "Z": []                // non chiama funzioni interne
}
```

Listato 3.23: Esempio del grafo delle chiamate

Il Listato 3.23 mostra un esempio, con dati fittizi, della struttura di `functionCalls`. A ogni funzione è associata una lista di stringhe contenente i nomi delle funzioni invocate internamente, sia direttamente che indirettamente.

In un secondo momento, l'annotatore verifica altri due aspetti importanti per eludere possibili loop che bloccherebbero l'esecuzione. Avvalendosi del grafo delle chiamate, controlla che non siano presenti ricorsioni dirette, ossia che una funzione invochi se

stessa, o ricorsioni circolari, ovvero cicli di chiamate tra più funzioni. Una situazione di questo tipo, facilmente manipolabile in un contesto normale, entrerebbe in contrasto con la politica di espansione delle invoke del progetto, creando una serie di inlining infiniti che arresterebbero il prosieguo della ricerca. Sotto è mostrato un esempio di errore causato da una ricorsione diretta:

```
func test() {
  test()
}

func main() {
  test()
}
```

Listato 3.24: Esempio di chiamata ricorsiva diretta

e il relativo messaggio di errore:

```
task: [pipeline] echo "Esecuzione annotatore..."
Esecuzione annotatore...
task: [pipeline] go run annotatore.go main test.go > "C:\Users\
   Lenovo\AppData\Local\Temp/pipeline_.tmp"
Errore: la funzione 'test' invoca se stessa (ricorsione diretta
   non ammessa).
exit status 1
task: Failed to run task "pipeline": exit status 1
```

Listato 3.25: Output causato da una ricorsione diretta

Allo stesso modo viene gestita una ricorsione circolare (Listato 3.26), supponendo che l'entrypoint sia il main, dove è mostrato all'utente il messaggio di errore inscritto nel Listato 3.27.

```
func test() {
  test2()
}

func test2() {
  test()
}

func main() {
  test()
}
```

Listato 3.26: Esempio di chiamata ricorsiva circolare

```
task: [pipeline] echo "Esecuzione annotatore..."
Esecuzione annotatore...
task: [pipeline] go run annotatore.go main test.go > "C:\Users\
   Lenovo\AppData\Local\Temp/pipeline_.tmp"
Errore: rilevata chiamata circolare che coinvolge la funzione '
   main'.
exit status 1
task: Failed to run task "pipeline": exit status 1
```

Listato 3.27: Output causato da una chiamata circolare

L'ultimo test di sicurezza è effettuato sulle chiamate di funzione annidate. Secondo il modello miniSL, non è previsto avere funzioni/call passate come parametri di altre funzioni/call. La verifica di una situazione del genere, ad esempio `test( test2() )` all'interno del codice è gestita restituendo il seguente messaggio di errore: "Errore: non è possibile passare funzioni/call come parametri a funzioni/call." Questa scelta è stata presa per l'impossibilità di sviluppare l'inlining su un'invocazione annidata e la difficoltà a garantire la coerenza del codice, annidando ulteriormente la dipendenza dei dati.

Annotazione del corpo delle funzioni

La parte centrale del codice, come detto precedentemente, è gestita dal metodo `analyzeStatementWithState`. Una volta ricevuto il nome della funzione entrypoint, viene creata l'annotazione relativa all'intestazione della funzione, annotata come `// miniSL: main()` esclusivamente se si tratta del metodo main; altrimenti, è commentata come `// miniSL: function nome(parametri)`.

Dopodichè, per l'analisi del corpo della funzione, `analyzeStatementsSequentially` percorre ogni statement aggiornando lo stato globale e generando l'annotazione appropriata. Quando incontra una chiamata a una funzione o a un servizio esterno, verifica a quale mappa globale appartiene il nome del metodo, se a alla lista delle funzioni interne o delle call. Se la funzione è una call, è prodotta l'annotazione `// miniSL: call nome(parametri)`. Proseguendo, il programma deve generare un commento con i parametri sostituiti dal loro rispettivo valore, recuperato sempre dalla struttura globale. L'annotatore esegue, quindi, dei controlli di sicurezza per verificare che alla call non vengano passati argomenti non valutabili. Se così fosse, per il seguente input:

```
// miniSL: defCall test(x)
func test(x int){}

func main() {
    x := errore() // errore() e' una funzione locale non call
  test(x)
}
```

Listato 3.28: Esempio di parametri non validi passati a call

la pipeline si bloccherebbe, stampando a terminale il seguente messaggio:

```
task: [pipeline] echo "Esecuzione annotatore..."
Esecuzione annotatore...
task: [pipeline] go run annotatore.go main test.go > "C:\Users\
   Lenovo\AppData\Local\Temp/pipeline_.tmp"
Errore: variabile non calcolabile passata a call.
exit status 1
task: Failed to run task "pipeline": exit status 1
```

Listato 3.29: Errore ritornato per una call con parametri non noti

Nel secondo caso, invece, se l'istruzione processata è una funzione, è necessario eseguire un controllo aggiuntivo, in linea con la logica generale del progetto. Se lo statement sotto analisi è una chiamata di funzione, primariamente viene effettuato lo stesso test di validità svolto per le call, integrando una verifica sull'invocazione annidata di un servizio esterno nel corpo della funzione. Lo scenario descritto è visibile nel listato successivo e rappresenta una fonte di errore se la call annidata prende in input variabili non note.

```
// miniSL: defCall test(x)
func test(x int){}

func somma(z int){
    test(z)
}

func main() {
    x := errore() // errore() e' una funzione locale non call
  somma(x)
}
```

Listato 3.30: Esempio di chiamata annidata di una call in una function

La variabile `x`, non calcolabile, è passata come argomento alla funzione interna `somma(z)`, che a sua volta esegue una chiamata a un servizio esterno propagando l'utilizzo di `x`. In questo caso, l'output è identico a quello mostrato nel Listato 3.29. La variante di questo contesto prevede l'impiego di una variabile non valutabile come parametro di una funzione che non passa quella variabile come argomento alle sue call interne. In quel caso, per il sistema non è un errore in quanto non sono coinvolte call e l'annotazione può proseguire.

Superate tutte le prove di validazione, per i metodi interni viene realizzata l'annotazione `// miniSL: invoke nome(parametri)`. Dopo aver commentato lo statement,

l'esecuzione si sposta sulla definizione della function per annotare il corpo della funzione, accertando nuovamente che tutti i vincoli di validità vengano rispettati.

Il template di call e invoke è, infine, completato dalla sostituzione dei valori reali dei parametri. In questa fase, viene fatto un confronto con le mappe della struttura `VariableState` per recuperare il valore delle variabili presenti.

Esempio di output corretto

In circostanze ottimali, dato un input del genere:

```
// miniSL: defCall test(x)
func test(x int){}

func somma(z int){
    test(z)
}

func main() {
    x := 5
  somma(x)
}
```

Listato 3.31: Esempio di input corretto

Questo di seguito è l'output annotato relativo alle funzioni o call del codice sorgente:

```
// miniSL: defCall test(x)
func test(x int){}
// miniSL: end

// miniSL: function somma(z)
func somma(z int){
    // miniSL: call test(5)
    test(z)
}
// miniSL: end

// miniSL: main()
func main() {
    x := 5
    // miniSL: invoke somma(5)
  somma(x)
}
// miniSL: end
```

Listato 3.32: Esempio di output corretto

3.2.5 If: annotazione

L'annotazione dei costrutti if rappresenta una delle logiche più corpose e importanti dell'annotatore. Come introdotto nel capitolo sulla sintassi miniSL, lo studio di questo elemento richiede una maggiore verifica della sicurezza del codice, dovuta ai tanti fattori in gioco. L'approfondimento degli if avviene totalmente nel relativo caso della funzione `analyzeStatementWithState`. Inizialmente, il sistema esegue tutti i controlli di validazione sulla guardia per procedere a processare il ramo then o else a seconda del risultato ottenuto.

Controlli sulla guardia

I casi di errore che si possono verificare analizzando la condizione sono legati soprattutto alla presenza di invocazioni di funzioni locali o di variabili non conosciute. Prima di tutto, viene controllato che l'if non sia stato preannotato dall'utente; in quel caso, il comportamento è lo stesso applicato a una chiamata a un servizio esterno nella condizione.

Durante l'ispezione della guardia, la condizione che richiede più attenzione è imposta dall'invocazione di una call. Primariamente, l'annotatore controlla che la call non contenga variabili memorizzate nella mappa globale `notEvaluable` o `problematicVars`. L'errore che ne deriva è lo stesso che viene gestito quando si eseguono controlli sulle singole call. Se ciò non avviene, la sola presenza della call è sufficiente a rendere il blocco di codice rilevante per le specifiche miniSL, e il determinismo è garantito in qualunque caso, eccetto per le assegnazioni non valutabili. La logica di validazione procede, infine, ad analizzare i tre casi principali di una guardia if: chiamata a funzioni standard, condizione composta da variabili non conosciute e presenza di una call nella guardia. Il controllo sui parametri, posto anteriormente, ha il compito di verificare l'errore principale di questo contesto, dimostrando la validità degli argomenti, cosi che il sistema possa proseguire classificando i casi successivi.

Funzioni o variabili non note nella guardia

Prima di emettere il commento, l'annotatore sostituisce le variabili con i loro valori attuali, se possibile, e valuta la condizione. Se la guardia invoca una funzione interna o fa uso di variabili non stimabili, il codice controlla che nessuno dei due rami contenga chiamate a servizi esterni, nè direttamente nè indirettamente per mezzo di altre funzioni. L'esempio di codice errato mostrato sotto riflette il problema appena enunciato:

```
1 // miniSL: defCall prodotto(x)
2 func prodotto(x int) int {
3   return 1
4 }
5
```

```
6  func main() {
7    x := errore()  // errore() e' una function
8    if x > 2 {  // oppure "if function()"
9      prodotto(5)
10   } else {
11     fmt.Println("ciao")
12   }
13 }
```

Listato 3.33: Esempio di input con guardia non determinabile

Come visibile nel Listato 3.33, la condizione if contiene `x`, salvata nella mappa `notEvaluable` dopo l'assegnazione. In questo contesto, viene analizzato il nodo percorrendo i due rami del blocco if e se viene rilevata la presenza di una call o di una function che nel suo corpo invoca una call, ritorna questo errore:

```
1 task: [pipeline] echo "Esecuzione annotatore..."
2 Esecuzione annotatore...
3 task: [pipeline] go run annotatore.go main test.go > "C:\Users\
    Lenovo\AppData\Local\Temp/pipeline_.tmp"
4 Errore: guardia if contiene variabile non valutabile e blocco
    then contiene call o funzioni con call.
5 exit status 1
6 task: Failed to run task "pipeline": exit status 1
```

Listato 3.34: Messaggio di errore per una condizione non determinabile

Se nè il ramo then nè il ramo else includono call nel proprio blocco, il nodo non rappresenta un problema per il programma e non causa errore. L'unica incertezza dipende dalle assegnazioni definite all'interno dei due rami che rendono le variabili coinvolte salvate nella mappa `problematicVars`. L'annotatore procede comunque a creare l'annotazione per tutto il blocco if, avvalendosi della funzione `clone()`, presentata nel Listato 3.19. Il metodo viene chiamato sullo stato originale e crea quattro mappe identiche, iterando su quelle già presenti e copiando i dati al loro interno. I rami dell'if vengono elaborati singolarmente, ognuno con la propria struttura clonata, per garantire l'isolamento di ogni blocco.

Questa scelta implementativa è stata mantenuta per non creare confusione con i compiti assegnati all'estrattore, specializzando il più possibile i due interpreti della pipeline in ruoli ben separati e non in conflitto tra loro. Sarà l'estrattore ad analizzare i commenti e a decidere quali tradurre in miniSL e quali no.

Condizione determinabile nella guardia

Quando la condizione della guardia è valutabile, poichè costituita da elementi conosciuti e operatori aritmetici o booleani, il sistema calcola quale ramo percorrere. Il ramo scelto

viene annotato secondo lo stato globale che viene aggiornato qualora gli statement del blocco lo richiedano. Per il ramo non preso, vengono prodotte comunque le annotazioni regolari attraverso un clone di `VariableState`, seguendo il principio espresso nella sezione soprastante.

Per concludere, ecco un esempio di input valido, la cui annotazione non è bloccata dai check di sicurezza del programma:

```
// miniSL: defCall prodotto(x)
func prodotto(x int) int {
  return 1
}

func test(z int){
    return z
}

func main() {
  x := 5
  if x > 2 {
        test(x)
    x := x + 100
  } else {
        x := 1
    prodotto(20)
  }
    test(x)
}
```

Listato 3.35: Esempio di input con guardia valutabile

All'esecuzione del file annotatore.go, alla variabile `x` è assegnato il valore "5" e, siccome la condizione `(x > 2)` è determinabile e vera, viene seguito il ramo then. In questo blocco, la variabile viene riassegnata con un'espressione aritmetica calcolabile e, dunque, viene aggiornato lo stato. L'ultima invocazione della funzione `test(x)` dimostra come l'analisi sia stata attuata correttamente secondo le specifiche del progetto. L'output annotato è il seguente:

```
// miniSL: defCall prodotto(x)
func prodotto(x int) int {
        return 1
}

// miniSL: function test(z)
func test(z int) {
        return z
```

```
}
// miniSL: end

// miniSL: main()
func main() {
        x := 5
// miniSL: if(5 > 2)
        if x > 2 {
// miniSL: invoke test(5)
                test(x)
                x := x + 100
// miniSL: else
        } else {
                x := 1
// miniSL: call prodotto(20)
                prodotto(20)
        }
// miniSL: end
// miniSL: invoke test(105)
        test(x)
}
// miniSL: end
```

Listato 3.36: Esempio di output ideale

3.2.6 For: annotazione

Annotare un ciclo for richiede due fasi principali ed è trattato similmente all'annotazione del corpo di una funzione. Secondo il template miniSL, l'annotazione generata in output ha una struttura di questo tipo: `// miniSL: for(init, limit)`. La sicurezza del costrutto dipende dall'unica variabile `limit`, che indica il numero di iterazioni del ciclo, mentre `init` è il contatore, ma viene mantenuto allo stato simbolico.

Controlli sulla guardia

Come per i blocchi if, anche per i cicli for l'annotatore, inzialmente, genera la stringa destinata a diventare l'annotazione finale, sostituendo i parametri, e successivamente procede a eseguire i controlli di validazione. Se la guardia contiene, nella sua condizione, una variabile salvata nella mappe delle variabili senza valore, considera ogni assegnazione interna non precisabile e la variabile non valida per le successive espressioni. Inoltre, verifica che nel blocco non siano presenti chiamate a servizi esterni.

```
// miniSL: defCall prodotto(x)
```

```
func prodotto(x int) int {
  return 1
}

func main() {
  x := test()  // test() e' una funzione interna del codice
  for i := 0; i < x; i++ {
    prodotto(z)
  }
}
```

Listato 3.37: Esempio di input errato per un ciclo for

Seguendo lo stesso approccio visto nella sezione precedente, se il blocco for invoca una call, l'annotazione si interrompe e restituisce questo errore:

```
...
Errore: call 'prodotto' trovata in un for loop con condizione non
    valutabile.
exit status 1
task: Failed to run task "pipeline": exit status 1
```

Listato 3.38: Messaggio di errore per una condizione non determinabile

Il sistema gestisce correttamente anche il caso in cui una call fosse invocata indirettamente tramite una function, ad esempio `test()`. L'output sarebbe il seguente:

```
Errore: funzione 'test' che contiene call trovata in un for loop
   con condizione non valutabile.
exit status 1
```

Listato 3.39: Messaggio di errore per una invocazione di call indiretta

La sola fase di controllo è rappresentata dall'analisi della condizione del for e, qualora non riportasse errore, l'esecuzione inzierebbe ad annotare il corpo del for.

Annotazione del corpo del ciclo

L'annotazione di un blocco for segue lo stesso modello applicato agli if e alle funzioni: prima di entrare nel ciclo viene clonato la struttura contenente le mappe che tracciano lo stato delle variabili. Dopodichè, una differenza fondamentale rispetto agli altri costrutti risiede nella logica di analisi: non viene visitato l'albero sintattico del corpo del ciclo, ma vengono simulati soltanto gli effetti delle assegnazioni, aggiornando il clone dello stato con i nuovi valori. La simulazione corrisponde al ciclo del for sotto analisi, con una guardia del tipo: `i = start; i < limit; i++`, dove `limit` è l'argomento di cui si verifica la validità e `start` assume il valore di `init`.

Quando la simulazione del ciclo for giunge all'ultima iterazione (`limit - 1`), allora l'AST viene realmente percorso tramite la funzione `analyzeStatementsSequentially`, per produrre le annotazioni degli statement. La soluzione di considerare solo le assegnazioni fino alla penultima iterazione e aggiornare il clone evita che lo stato originale venga "sporcato" da dati non deterministici e che vengano generate annotazioni duplicate (per N iterazioni si creerebbero N commenti per statement).

Esempio di output corretto

Durante la simulazione del ciclo, se il corpo del for non modifica lo stato al punto da rendere qualche variabile non deterministica, all'ultima ripetizione del blocco le istruzioni vengono annotate correttamente. Prendendo come esempio l'input sottostante:

```
// miniSL: defCall prodotto()
func prodotto(x) int {
  return 1
}

func test(a){
    fmt.Println("Funzione di test")
}

func main() {
  x := 5
    y := x + 5
  for i := 0; i < x; i++ {
    prodotto(x)
        y := y * 2
  }
    test(y)
}
```

Listato 3.40: Esempio di input valido per un ciclo for

il rispettivo output è mostrato di seguito:

```
// miniSL: defCall prodotto(x)
func prodotto(x int) int {
        return 1
}

// miniSL: function test()
func test(a int) {
        fmt.Println("Funzione di test")
}
```

```
// miniSL: end

// miniSL: main()
func main() {
        x := 10
        z := 5
// miniSL: for(i,5)
        for i := 0; i < z; i++ {
// miniSL: invoke test(5)
                test(z)
                x = x * 2
        }
// miniSL: end
// miniSL: invoke test(320)
        test(x)
}
// miniSL: end
```

Listato 3.41: Esempio di output con input corretto

3.3 2° fase: Estrazione

La seconda fase della pipeline è rappresentata dall'estrattore, il reale traduttore da codice annotato a linguaggio miniSL. Superato ogni ostacolo sintattico, il file di input può dunque essere trasposto nella sua corrispondente parafrasi miniSL, rispettando le regole di traduzione della Tabella 2.3 presentata nei capitoli precedenti.

La stessa estrazione delle annotazioni miniSL segue più fasi interne: inizialmente, il sistema legge e isola dal file annotato tutti i commenti riguardanti il linguaggio miniSL e caratterizzati dal prefisso definito nei paragrafi sopra; dopodichè produce una lista di sole annotazioni, prive di tutto il codice Go circostante e del prefisso `// miniSL:`. Uno script in input di questo tipo:

```
// miniSL: main()
func main() {
        z := 5
        // miniSL: call serviceA()
        serviceA()
// miniSL: if(x > 0)
        if(x > 0){
            // miniSL: call serviceB()
            serviceB()
        // miniSL: else
```

```
11          }else{
12              // miniSL: call serviceC()
13              serviceC()
14          }
15  // miniSL: end
16  }
17  // miniSL: end
```

Listato 3.42: Esempio di input per l'estrattore

Verrebbe estratto dall'estrattore per creare la seguente lista:

```
1  main()
2  call serviceA()
3  if(x > 0)
4  call serviceB()
5  else
6  call serviceC()
7  end
8  end
```

Listato 3.43: Esempio di struttura dopo l'estrazione delle annotazioni

Data la sequenza di commenti, il sistema elaborerà solamente la lista ottenuta dal codice completo. Il problema principale risiede nella gestione dell'annidamento dei blocchi una volta ricevuto in input il nome della funzione entrypoint, ossia distinguere la chiusura di un blocco interno (if, for) dalla chiusura della funzione stessa. La soluzione trovata risiede in un contatore di profondità di annidamento, la cui implementazione è spiegata nelle sezioni successive. Dopodichè, tramite questa strategia, viene creato un catalogo delle funzioni e salvato in un'apposita mappa. A questo punto, il compito di convertire la lista di commenti in sintassi miniSL è delegato alla funzione principale `generateMiniSLBody`.

L'estrattore si distingue nella pipeline per l'applicazione di una regola che definisce il filtro da impiegare durante la generazione del codice. Il principio di base è, come spiegato nei capitoli precedenti, che solo i costrutti che contengono chiamate a servizi esterni sono importanti per l'obiettivo del progetto. I blocchi if e for vengono tradotti in miniSL solo se la guardia, i rami then e else, o i blocchi iterativi contengono call. In questo modo, si riduce significativamente la complessità del codice, focalizzando l'analisi solo sulle porzioni di codice rilevanti.

Un altro aspetto importante è l'espansione delle funzioni interne tramite inlining. Durante l'analisi dello script annotato, quando l'estrattore incontra un'annotazione `invoke`, la riga in cui viene chiamata la funzione viene sostituita con il corpo della funzione invocata, applicando la sostituzione dei parametri formali con gli argomenti effettivi. L'inli-

ning può avvenire anche all'interno di un altro inlining, fino a quando ogni chiamata di funzione, se contenente call, non è stata esaminata.

Come succede per l'annotatore, anche l'estrattore separa la logica di parsing dalla definizione della sintassi e lo fa poggiandosi su un file di configurazione JSON che definisce i template finali da applicare per generare il codice miniSL.

3.3.1 L'applicazione del file JSON all'estrattore

Dati i vantaggi di estensione e manutenibilità già visti per la fase di annotazione, si è scelto di applicare una configurazione simile anche per l'estrazione delle annotazioni. Il file estrattore.json è composto da due parti: il campo `annotationPrefix`, che contiene il prefisso stabilito congiuntamente con l'annotatore e funge da delimitatore, e un campo `commands`, una mappa che definisce tutte le traduzioni sintattiche supportate dal progetto.

```
{
    "annotationPrefix": "// miniSL:",
    "commands": {
        "for": {
            "pattern": "for\\((\\w+),(\\w+)\\)",
            "template": "for (%s in range(0, %s)) {"
        },
        "if": {
            "pattern": "if\\((.+)\\)",
            "template": "if (%s) {"
        },
        "call": {
            "pattern": "call (\\w+)\\((.*?)\\)",
            "template": "call %s(%s)"
        },
        "invoke": {
            "pattern": "invoke (\\w+)\\((.*?)\\)",
            "template": "inline %s(%s)"
        },
        "else": {
            "pattern": "else",
            "template": "} else {"
        },
        "end": {
            "pattern": "end",
            "template": "}"
        },
        "main": {
```

```
            "pattern": "main\\((.*?)\\)",
            "template": "(%s) => {"
        },
        "function": {
            "pattern": "function (\\w+)\\((.*?)\\)",
            "template": "function %s(%s)\n{"
        }
    }
}
```

Listato 3.44: File estrattore.json

Ogni comando segue lo stesso modello: un pattern per l'estrattore da ricercare per sapere quale annotazione è stata estratta e un template di output per la generazione del codice miniSL. I placeholder dei template verranno sostituiti con i valori catturati al momento della lettura dell'annotazione. Il file estrattore.go definisce due strutture dati utili per raccogliere i pattern e i template contenuti nel json e sfruttarli per riconoscere e tradurre la lista di annotazioni.

```
type CommandRule struct {
  Pattern  string `json:"pattern"`
  Template string `json:"template"`
}

type MiniSLConfig struct {
  AnnotationPrefix string                 `json:"annotationPrefix
     "`
  Commandsmap[string]CommandRule `json:"commands"`
}
```

Listato 3.45: Strutture dati collegate al JSON

La struttura `CommandRule` definisce come riconoscere e trasformare un tipo di annotazione: i campi `Pattern` e `Template` della struct Go corrispondono alle chiavi `"pattern"` e `"template"` del JSON. Questo scheletro viene richiamato nella definizione della struttura `MiniSLConfig` sottostante, che contiene la configurazione per il parsing delle annotazioni e associa ogni parola chiave della mappa `"commands` del JSON al rispettivo pattern e template di `CommandRule`.

Una volta create queste strutture dati, viene caricata la configurazione esterna in una variabile globale `var config MiniSLConfig`, accessibile da tutte le funzioni dell'estrattore. L'applicazione degli schemi presi da estrattore.json avviene tramite più funzioni. La funzione primaria è `matchesAnnotation`, spiegata di seguito:

```
func matchesAnnotation(key, line string) bool {
    rule, exists := config.Commands[key]
```

```
    if !exists {
        return false
    }

    re := regexp.MustCompile("^" + rule.Pattern + "$")
    return re.MatchString(line)
}
```

Listato 3.46: Funzione matchesAnnotation

La funzione prende in input una chiave `key` che può assumere il valore "for", "if", ecc., a seconda del tipo di comando da verificare, e una stringa `line`, corrispondente alla riga di annotazione da analizzare. Quando viene invocata, la funzione esegue una ricerca nella mappa `config.Commands` per trovare una corrispondenza di `key` e il rispettivo pattern e restituisce `true` se lo trova. Assunto che la riga elaborata è traducibile in miniSL, l'applicazione dei template vera e propria avviene con la funzione `transformLine`, mostrata al Listato 3.47.

```
func transformLine(line string) string {
  // prova tutti i pattern di configurazione
  for key, rule := range config.Commands {
    re := regexp.MustCompile("^" + rule.Pattern + "$")
    matches := re.FindStringSubmatch(line)
    if len(matches) > 0 {
      args := matches[1:]
      switch key {
      case "call":
        // Memorizza il servizio chiamato
        calledServices[matches[1]] = true
        return fmt.Sprintf(rule.Template, anySlice(args)...)
      case "if":
        // Gestione speciale per le condizioni if
        cond := matches[1]
        if regexp.MustCompile(`^\s*\w+\(.*\)\s*$`).MatchString(
           cond) {
          return fmt.Sprintf("if(call %s)", cond)
        }
        return fmt.Sprintf("if(%s)", cond)
      case "for":
        return fmt.Sprintf(rule.Template, anySlice(args)...)
      }
    }
  }
  return ""
```

```
26 }
```

Listato 3.47: Funzione transformLine

Questa funzione rappresenta il ponte tra le annotazioni e il codice miniSL, e definisce tre casi specifici per la traduzione: il caso di una "call", di un "for" e di un "if". La sintassi finale, infatti, prevede che solo i tre costrutti sopra elencati vengano trasformati nell'output finale poichè la gestione delle invoke tramite inlining non necessita di una traduzione. Attenzione particolare merita il caso che un'annotazione corrisponda a un if: mentre l'annotatore tiene traccia solo del significato semantico della guardia di un if e non distingue tra chiamate a servizi esterni e function locali, l'estrattore implementa un'operazione di confronto per verificare che la condizione contenga una call, e generare l'apposito output mostrato nel listato. Con `return`, per ogni caso dello switch, sostituisce i placeholder con il valore contenuto nelle variabili `cond` o `args`.

3.3.2 Scelte implementative: estrattore

Mappe e strutture globali

La complessità del codice dell'estrattore è inferiore rispetto a quella della sua controparte, soprattutto a causa della minore quantità di compiti assegnati. Rispetto all'annotatore, il flusso di esecuzione segue un andamento più lineare: l'estrazione delle annotazioni non è più fornita dal processo dell'AST, ma da operazioni su una lista piatta di stringhe. Il motivo di questa implementazione risiede nella maggiore semplicità di parsing e sostituzione dei parametri: non è necessario costruire una gerarchia composta da nodi, dal momento che la lista sfrutta i marcatori `end` per indicare i livelli di annidamento e l'inlining è completato con una ricerca e successiva sostituzione su stringhe.

```
1 type MiniSLFunction struct {
2   Params []string
3   Body   []string
4 }
```

Listato 3.48: Struttura MiniSLFunction

Per questo motivo lo script estrattore.go definisce una struttura `MiniSLFunction`, una rappresentazione temporanea di una singola funzione prima dell'output finale. La mappa `Params` contiene la lista dei parametri della funzione, mentre `Body` racchiude, in ordine, le stringhe di istruzioni, compresi i marcatori di apertura e chiusura dei blocchi, della funzione in oggetto.

Il sistema, inoltre, definisce altre due mappe, mostrate al Listato 3.49,che offrono un punto di accesso veloce per ottenere rapidamente i dati durante l'esecuzione del progetto.

```
1 var (
2   functionDefs    = map[string]MiniSLFunction{}
```

```
3     calledServices = map[string]bool{}
4 )
```

Listato 3.49: Variabili functionDefs e calledServices

La lista `functionDefs` funge da glossario delle funzioni definite nel codice e fornisce una scorciatoia al sistema durante l'espansione delle invoke, il quale non deve preoccuparsi di scansionare nuovamente l'intero file, ma può recuperare il corpo della funzione richiesta da questa mappa.

La mappa `calledServices`, invece, rappresenta il set di servizi coinvolti nella fase di estrazione e serve per redigere un elenco delle call utilizzate, da inserire all'inizio dell'output per avere una panoramica sui servizi esterni. La lista viene aggiornata a run-time durante la lettura delle annotazioni.

Estrazione delle funzioni

Data la lista di annotazioni isolate, il primo passo dell'elaborazione è svolto dalla funzione `extractFunctions`, il cui obiettivo è costruire la mappa `functionDefs`. Come accennato precedentemente, la logica della funzione ruota attorno al principio di un contatore di annidamento, e per implementarlo si serve di due variabili in particolare: `inFunction`, un valore booleano che comunica al sistema se si è all'interno di una funzione, e `nested`, il vero e proprio contatore.

```
1 func extractFunctions(annotations []string) {
2     var currentName string
3     var currentParams []string
4     var currentBody []string
5     inFunction := false
6     nested := 0
7 
8       //Resto della funzione ...
9 }
```

Listato 3.50: Funzione extractFunctions

Oltre alle due variabili appena descritte, la funzione definisce altre tre variabili: `currentName` per tenere traccia del nome della funzione corrente, e `currentParams` e `currentBody` per tracciare i parametri e il corpo della funzione corrente, in previsione di utilizzare i seguenti dati per creare la struttura `MiniSLFunction` della funzione analizzata e riempire `functionDefs`. Dopodichè implementa un ciclo for per scorrere la lista di annotazioni, così strutturato:

```
1 func extractFunctions(annotations []string) {
2     //Sezione listata sopra ...
3       for _, line := range annotations {
```

```
    if matchesAnnotation("function", line) {
      // Inizia una nuova funzione ...
      inFunction = true
      nested = 0
    } else if inFunction && matchesAnnotation("end", line) {
      if nested > 0 {
        // Decrementa il nesting ...
      } else {
        // Questo end chiude la funzione
        functionDefs[currentName] = MiniSLFunction{Params:
           currentParams, Body: currentBody}
        inFunction = false
      }
    } else if inFunction && matchesAnnotation("if", line) {
      // Incrementa il nesting ...
    } else if inFunction && matchesAnnotation("for", line) {
      // Incrementa il nesting ...
    } else if inFunction {
      // Qualsiasi altra linea dentro la funzione ...
    }
  }
}
```

Listato 3.51: Funzione extractFunctions

Ogni volta che il sistema incontra il marcatore "function", imposta la variabile `inFunction = true` e inizializza `nested = 0`. Durante l'esecuzione del corpo della funzione, alla conferma di un costrutto if o for, il contatore viene incrementato di 1, mentre ad ogni corrispondenza di un delimitatore "end" si deve decidere: se `nested > 0`, significa che ci troviamo a un livello di annidamento superiore o uguale al primo e il marcatore indica la chiusura di un blocco; altrimenti, il marcatore determina la chiusura della funzione stessa, la variabile `inFunction` viene impostata su `false` e la funzione completa viene salvata in `functionDefs`. Prima di chiudere definitivamente la funzione, ogni istruzione viene aggiunta alla lista `currentBody` tramite il metodo `currentBody = append(currentBody, line)`.

Filtering semantico

Il principio di filtering semantico è incarnato in due funzioni: `functionHasCall` e `functionHasRecursiveCall`. La prima funzione esegue una ricerca ricorsiva che, partendo dalla funzione in esame, controlla se questa contiene chiamate a call dirette o indirette, tramite l'istruzione `invoke`.

```
func functionHasCall(lines []string) bool {
```

```
  visited := make(map[string]bool)
  return functionHasCallRecursive(lines, visited)
}
```

Listato 3.52: Funzione functionHasCall

Come mostrato nel listato sovrastante, `functionHasCall` prende in input la lista di stringhe, corrispondenti alle istruzioni di un blocco, e inizializza una mappa `visited` di funzioni già visitate. Questa variabile locale previene il blocco dell'analisi di raggiungibilità di call, evitando di rimanere sospesi in un loop di invoke. Al ritorno dallo studio del blocco di codice, il flag torna a essere `false` per non falsare future esplorazioni. La verifica della presenza di servizi esterni è delegata interamente alla funzione `functionHasCallRecursive`, che prende come parametri la lista di annotazioni e la mappa delle funzioni visitate. Da questa funzione dipende la correttezza dell'output finale, poichè è l'unica a implementare la logica di filtering richiesta e un errore in questo blocco di codice, a fronte di un'implementazione restante corretta, comprometterebbe l'esito dell'estrazione.

```
func functionHasCallRecursive(lines []string, visited map[string]
    bool) bool {
  for _, line := range lines {
    if matchesAnnotation("call", line) {
      return true
    }
    if matchesAnnotation("invoke", line) {
      fnName := extractInvokeName(line)
      if fnName != "" && !visited[fnName] {
        visited[fnName] = true
        if fn, ok := functionDefs[fnName]; ok {
          if functionHasCallRecursive(fn.Body, visited) {
            return true
          }
        }
        delete(visited, fnName)
      }
    }
  }
  return false
}
```

Listato 3.53: Funzione functionHasCallRecursive

La struttura di `functionHasCallRecursive` distingue due casi che si possono presentare scorrendo la variabile `lines`: tramite la funzione `matchesAnnotation` , vista in precedenza, se l'istruzione analizzata è una call, il ciclo for si interrompe e la funzione ritorna

il valore `true`; in caso contrario, si estrae il nome della funzione con `extractInvokeName` e, se non è ancora stata visitata, si entra nel corpo della funzione invocando ricorsivamente `functionHasCallRecursive`. Quando il ciclo ricorsivo restituisce `true`, la funzione propaga il valore verso l'alto e si esce definitivamente dal metodo.

La combinazione di queste funzioni svolge una parte cruciale nell'espansione degli invoke, nella generazione di blocchi interni o inlining e nella generazione del corpo dell'entrypoint. Nella prossima sezione esploriamo le funzioni responsabili dell'applicazione delle funzioni esposte fino ad ora e di come viene costruito l'ouptut finale.

Espansione inline

La sostituzione delle istruzioni invoke tramite inlining è il meccanismo più complesso spiegato nello script dell'estrattore. La funzione principale per l'espansione inline è `expandInvoke`, una funzione ricorsiva che ritorna una lista di stringhe, rappresentante il codice miniSL espanso, come risultato dell'espansione. Un aspetto importante di questa funzione è la gestione dell'indentazione: per avere un codice finale il più chiaro e comprensibile possibile accetta un parametro `indent` che specifica il grado di indentazione corrente. Ad ogni riga generata viene anteposto il corretto numero di spazi per mezzo della apposita funzione `indentLine`, mostrata di seguito:

```
1 func indentLine(line string, indent int) string {
2   return strings.Repeat("    ", indent) + line
3 }
```

Listato 3.54: Funzione indentLine

In questo modo si ottiene una sintassi miniSL più vicina allo stile degli altri linguaggi di programmazione, oltre a essere più facilmente leggibile per l'utente.

```
1  func expandInvoke(line string, indent int) []string {
2  //...
3      fn, ok := functionDefs[fnName]
4       if !ok || len(fn.Params) != len(argValues) {
5        return nil // Funzione non trovata o numero parametri
             errato
6       }
7
8    paramMap := map[string]string{}
9    for i, param := range fn.Params {
10     paramMap[param] = argValues[i]
11   }
12
13     // Resto della funzione...
14 }
```

Listato 3.55: Verifica della definizione e costruzione della mappa di sostituzione

Inizialmente, `expandInvoke` prende in input una stringa, ossia la riga dell'istruzione invoke, e la variabile intera `indent`. Dopodichè estrae dall'annotazione il nome della funzione e gli argomenti passati in input. A questo punto, controlla che la funzione esista, ricercandola nella lista `functionDefs`, e crea la mappa di sostituzione `paramMaps`, mostrata al Listato 3.56, che associa ogni parametro al suo valore reale. Se, ad esempio, la function `calc(a, b)` viene chiamata con `invoke calc(1, 10)`, `paramMap` sarà popolata come segue: {"a": "1", "b": "10"}.

La parte centrale della funzione è un ciclo for che itera su tutte le istruzioni del corpo della funzione da espandere. Il ciclo for deve gestire diversi casi di istruzioni ed è strutturato in questo modo:

```
var result []string
  for i := 0; i < len(fn.Body); i++ {
    l := fn.Body[i]

    if matchesAnnotation("call", l) {

      for k, v := range paramMap {
        l = strings.ReplaceAll(l, k, v)
      }
      result = append(result, indentLine(transformLine(l), indent
         ))

    } else if matchesAnnotation("invoke", l) {

      sub := expandInvoke(l, indent)
      if len(sub) > 0 {
        result = append(result, sub...)
      }

    } else if matchesAnnotation("if", l) {
      // Sostituisce i parametri nei blocchi then ed else
      // Include il blocco solo se contiene chiamate
    } else if matchesAnnotation("for", l) {
        // Sostituisce i parametri nel blocco
        // Verifica se esiste una call nel blocco
    }
  }
  return result
```

Listato 3.56: Ciclo principale di expandInvoke

Inizializza una variabile `result` che conterrà il codice miniSL del corpo della funzione. Quando il sistema incontra una call, sostituisce tutti i parametri con i loro valori usando il metodo `ReplaceAll` e aggiunge la stringa tradotta con `transformLine` alla variabile `result`. Nel caso in cui la riga analizzata sia un'invoke, la funzione chiama ricorsivamente se stessa per espandere quella chiamata.

La gestione dei casi if è la parte più complessa da gestire a causa della molteplicità dei controlli da effettuare sui due rami del costrutto. Prima di tutto, vengono sostituiti i parametri nella guardia dell'if; poi vengono estratti i rami then e else attraverso una funzione `extractIfElseBlock`, il cui obiettivo è solamente quello di raccogliere separatamente le istruioni dei due blocchi, e si applica lo stesso procedimento. A questo punto, vengono effettuate, in ordine, verifiche sulla presenza di chiamate a servizi tramite la funzione `functionHasCall` esterni, partendo dalla guardia per poi spostarsi ai rami dell'if. La logica di questo caso prevede che il blocco if venga incluso e aggiunto a `result` solo se la guardia o uno dei due blocchi contiene una call. Se il corpo dell'if viene incluso, la traduzione dei blocchi then/else è affidata alla funzione `generateMiniSLBodyFromLines` che vedremo successivamente.

Il caso in cui `matchesAnnotation` ritorna `true` quando l'annotazione è un comando `for` è gestito in maniera simile al caso if. Vengono rimpiazzati i parametri con il loro valore concreto e, dopo aver estratto il corpo del ciclo con la funzione `extractForBlock`, si controlla se il loop include call invocando `functionHasCall`. In caso positivo, le istruzioni vengono processate dalla funzione `generateMiniSLBodyFromLines` e aggiunte a `result`.

Generazione dell'output

A partire dalla funzione entrypoint, la generazione del codice miniSL è dettata dalla funzione `generateMiniSL`, che si occupa di generare l'header formato dalle call coinvolte dall'entrypoint e di creare la scrittura "`(params) => {`" inserendo i parametri della funzione di input. La funzione prosegue delegando a `generateMiniSLBody` la generazione ricorsiva del corpo della funzione entrypoint e aggiunge la parentesi di chiusura al termine della ricorsione.

Il vero motore dell'estrazione è la funzione `generateMiniSLBody`, che implementa una struttura simile a quella già vista per la funzione `expandInvoke` nella sezione precedente e scorre direttamente la variabile `annotations`, ossia il corpo principale della funzione entrypoint.

```
1 func generateMiniSLBody(annotations []string, indent int) []
     string {
2   var output []string
3
4   for i := 0; i < len(annotations); i++ {
5     line := annotations[i]
```

```
6
7       if matchesAnnotation("call", line) {
8         output = append(output, indentLine(transformLine(line),
              indent))
9       } else if matchesAnnotation("invoke", line) {
10        body := expandInvoke(line, indent)
11        if len(body) > 0 {
12          output = append(output, body...)
13        }
14      } else if matchesAnnotation("if", line) {
15        // Controlla se la guardia contiene una call
16              // Controlla se uno dei due rami contiene una call
17              // Invoca riccorsivamente se stessa
18        i = endIdx
19      } else if matchesAnnotation("for", line) {
20        // Controlla se contiene call
21              // Invoca ricorsivamente se stessa
22        }
23        i = endIdx
24      }
25    }
26
27    return output
28  }
```

Listato 3.57: Struttura della funzione generateMiniSLBody

La funzione, visibile al Listato 3.57, implementa la stessa logica per l'espansione inline, utilizzando una variabile `output` per il codice miniSL finale, una variabile `indent` per l'indentazione e un ciclo for per gestire le varie istruzioni. La gestione dei casi è analoga alla funzione `expandInvoke`: fatta eccezione per la sostituzione dei parametri, non richiesta in questo caso, ogni ramo dell'if estrae l'annotazione, verifica la sua rilevanza e genera la sintassi miniSL invocando ricorsivamente `generateMiniSLBody` per le annotazioni if e for. Proprio per salvaguardare la linearità dell'elaborazione, è stata inserita la dichiarazione "`i = endIdx`" come promemoria per il loop for principale, per evitare di processare righe già analizzate. Questa riga è fondamentale per aggiornare la posizione corrente dell'estrattore nella lista `annotations`. Proponiamo un esempio concreto per comprendere l'efficacia di questa dichiarazione, considerando questa lista di annotazioni:

```
[0]  call serviceA()
[1]  if(x > 0)
[2]      call serviceB()
[3]      call serviceC()
[4]  end
```

```
[5] call serviceD()
```

Listato 3.58: Esempio lista di annotazioni

Quando l'estrattore raggiunge l'istruzione in posizione `[1]`, entra nel blocco if tramite la funzione `extractIfElseBlock`, che processa le righe da `[2]` a `[4]` e restituisce il valore della posizione del marcatore `end`. A questo punto, usciti dal blocco dell'if, il ciclo for principale può continuare il loop incrementando il valore di `i`, a cui è stato assegnato il valore 4, di 1. In modo analogo viene applicata la stessa strategia nel caso delle annotazioni contenenti un for.

Come accennato durante l'illustrazione dello script dell'inlining, parte della generazione del codice miniSL è delegata anche alla funzione `generateMiniSLBodyFromLines`. Questa funzione è una copia quasi identica di `generateMiniSLBody`, poichè applica la stessa logica interna e la medesima architettura. La differenza sostanziale è dove viene invocata: `generateMiniSLBody` è chiamata a processare le annotazioni originali dell'entrypoint e lavora sulla mappa `annotations`, mentre `generateBodyFromLines` è invocata da `expandInvoke` quando il sistema incontra un ciclo if o for annidato nella funzione da espandere. Questa funzione lavora con le righe già estratte e con i parametri già sostituiti.

La ragione per la quale si è deciso di tenere questa duplicazione, aggravando la ridondanza del codice, è legata all'indipendenza e al disaccoppiamento che questa scelta permette. Un eventuale futuro cambiamento alle specifiche del progetto riguardo alla gestione delle espansioni inline, ad esempio, verrebbe trattato con maggiore sicurezza, dovendo apportare modifiche solo alla funzione che si occupa di generare codice miniSL dalle funzioni espanse. Inoltre, lavorando su contesti diversi, non è necessario aggiungere flag o parametri distintivi per ogni situazione, modifica che un'unica funzione centralizzata richiederebbe.

Nei prossimi capitoli esponiamo brevemente il risultato finale che ogni costrutto si aspetta di ricevere dato un input di esempio.

3.3.3 Funzioni: estrazione

L'estrazione delle function interne e delle call è un processo composto dagli elementi analizzati finora. Dato il seguente file di input:

```
// miniSL: defCall prodotto(x)
func prodotto(x int) int {
  return 1
}

func test(z int) {
  if 10 > 5 {
    prodotto(z)
  }
```

```
}

func main() {
  x := 10
  z := 5
  prodotto(x)

  test(z)

}
```

Listato 3.59: Esempio di input per function e call

il risultato prodotto in output è mostrato nel listato sottostante:

```
service prodotto : (void) -> void;

() => {
    call prodotto(10)
    if(10 > 5) {
        call prodotto(5)
    }
}
```

Listato 3.60: Sintassi miniSL per function e call

Ponendo come entrypoint la funzione main, il risultato è quello atteso: all'inizio del file è inserito un header contenente l'elenco delle call chiamate direttamente o indirettamente dall'entrypoint, secondo la formula `service nomeServizio : (void) -> void;`. Per il momento, il pattern richiede di listare soltanto il nome del servizio invocato, senza precisare i parametri in input presi dal servizio e il tipo dell'output. Come da specifiche, la funzione `test` viene espansa essendo rilevante per il sistema. Se `test` non contenesse nessuna chiamata a un servizio esterno, la sintassi miniSL sarebbe la seguente:

```
service prodotto : (void) -> void;

() => {
    call prodotto(10)
}
```

Listato 3.61: Output per invoke senza call

Come ultima casistica, ipotizziamo che l'entrypoint sia la funzione `test`.

```
(z) => {
    if(10 > 5) {
```

```
3           call prodotto(z)
4       }
5   }
```

Listato 3.62: Output quando l'entrypoint è diverso dal main

Da Listato 3.62 si nota come la traduzione della funzione entrypoint accetti i parametri nella dicitura `() =>` { e, non conoscendo il valore concreto di `z`, non ritorna errore e non viene sostituita nelle sue ricorrenze nel file.

3.3.4 If: estrazione

Per semplicità, d'ora in avanti, considereremo sempre come entrypoint la funzione main. La traduzione del costrutto if segue la logica descritta nelle sezioni precedenti, in questo paragrafo vedremo i vari risultati possibili. Prendiamo come file di esempio lo script presentato di seguito:

```
1  // miniSL: defCall prodotto(x)
2  func prodotto(x int) int {
3    return 1
4  }
5
6  func test(z int) {
7    calc()
8  }
9
10 func calc() {
11   return 0
12 }
13
14 func main() {
15
16   z := 1
17
18   if prodotto(z) {
19     test(z)
20   } else {
21     test(z)
22   }
23
24 }
```

Listato 3.63: Input di esempio per estrazione degli if

La fase di estrazione manterrà entrambi i rami dell'if, avendo una call nella guardia, ma non espanderà l'invoke non essendo rilevante per l'analisi. La traduzione produrrà questo codice finale:

```
service prodotto : (void) -> void;

() => {
    if(call prodotto(1)) {
    }
    else {
    }
}
```

Listato 3.64: Input di esempio per estrazione degli if

Ora esploriamo le due situazioni possibili quando almeno uno dei due rami contiene una chiamata a un servizio esterno. A prescindere dal fatto che la guardia includa una call o una condizione valutabile, il comportamento del programma è lo stesso in entrambi i casi. Dato un file di input strutturato in questo modo:

```
// miniSL: defCall prodotto(x)
func prodotto(x int) int {
  return 1
}

// miniSL: defCall somma(z)
func test(z int) {
  return 1
}

func main() {

  z := 1

  if prodotto(z) {
    test(z)
  } else {
    Smt.Println("ramo else")
  }

}
```

Listato 3.65: Input di esempio per estrazione dei rami dell'if

Il codice atteso è il seguente:

```
service prodotto : (void) -> void;
service somma : (void) -> void;

() => {
    if(call prodotto(1)) {
        call somma(1)
    }
}
```

Listato 3.66: Output di esempio per estrazione del ramo then dell'if

Nel caso in cui solo il ramo then dell'if invochi una call al suo interno, il ramo else non viene estratto in miniSL. Nel caso opposto, invece, se invertiamo il contenuto dei due rami visti sopra, il ramo then viene generato, ma il corpo rimarrà vuoto. Il Listato 3.67 presenta quest'ultimo tipo di situazione.

```
service prodotto : (void) -> void;
service somma : (void) -> void;

() => {
    if(call prodotto(1)) {
    }
    else {
        call somma(1)
    }
}
```

Listato 3.67: Output di esempio per estrazione del ramo then dell'if

3.3.5 For: estrazione

La logica di estrazione, che ha il suo punto di inizio nella funzione `generateMiniSL`, analizza come ultimo caso la possibilità che l'annotazione risponda al costrutto for. Consideriamo che lo script da elaborare comprenda entrambe le uniche situazioni da gestire per i cicli for, ovvero che il loop chiami o meno un servizio esterno. Dato un input di questo tipo:

```
// miniSL: defCall prodotto(x)
func prodotto(x int) int {
  return 1
}

func test(z int) {
  return 1
```

```
}

func main() {

  z := 5

  for i := 0; i < z; i++ {
    prodotto(z)
  }

  for i := 0; i < z; i++ {
    test(i)
  }

}
```

Listato 3.68: Input di esempio per estrazione di un for

Il codice finale avrà generato solo il primo ciclo for, in quanto il secondo viene marcato come non importante durante l'esecuzione della pipeline. L'output corrispondente all'input appena visto è il seguente:

```
service prodotto : (void) -> void;

() => {
    for (i in range(0, 5)) {
        call prodotto(5)
    }
}
```

Listato 3.69: Output miniSL di un costrutto for

Capitolo 4

Conclusioni

Il lavoro presentato ha contribuito a creare un sistema automatizzato che può integrare e migliorare le politiche di scheduling delle funzioni in ambienti FaaS. In particolare, mi sono concentrato su un linguaggio di partenza specifico (Go), ottimo per le sue caratteristiche nel costituire la base per un'analisi statica, al fine di ottenere una versione semplificata della logica delle funzioni in linguaggio miniSL. La pipeline e il processo di implementazione mostrati nel Capitolo 3 possono essere riadattati anche per produrre codice miniSL partendo da altri linguaggi di programmazione.

I singoli componenti della pipeline, inoltre, agiscono in modo indipendente tra loro e basano la loro funzione su due file JSON, rendendo l'annotatore e l'estrattore due strumenti riconfigurabili e riutilizzabili come strumenti a sè stanti.

4.1 Limitazioni Tecniche

Il lavoro svolto ha raggiunto l'obiettivo preposto ma presenta alcune limitazioni. La principale riguarda la dipendenza dallo sviluppatore: la fase di annotazione, infatti, ha bisogno che le chiamate ai servizi esterni vengano pre annotate dall'utente nella fase di stesura del codice, non potendo riconoscere in autonomia le call presenti nel file da annotare.

Altre limitazioni più tecniche riguardano la gestione delle variabili: le riassegnazioni, come mostrato nella Sezione 3.2.3, devono rispettare vincoli specifici e non possono avvenire in blocchi di codice non deterministici; altrimenti, restituiscono un errore. Il modello in questione è, quindi, fortemente dipendente dal determinismo delle guardie delle funzioni e dei blocchi if e for ed è compito del programmatore scrivere un codice Go il più preciso ed esplicito possibile. Le criticità relative all'estrattore si trovano, invece, nell'impossibilità, al momento, di gestire riassegnazioni tramite espressioni aritmetiche complesse durante l'espansione inline di un invoke.

Infine, è bene tenere a mente che la fase di annotazione considera solo i costrutti più semplici e basilari gestiti anche in miniSL, e non è possibile gestire casi troppo complessi nel codice sorgente.

4.2 Sviluppi Futuri

La base gettata dal lavoro di questa tesi tiene aperte molteplici vie per migliorare e astrarre maggiormente la pipeline. In primis, si potrebbero migliorare le riassegnazioni delle variabili, includendo nelle variabili conosciute anche quelle riassegnate tramite funzioni o con operatori che non siano quelli base o semplici booleani.

Un'ulteriore evoluzione riguarda il potenziamento dell'estrattore per risolvere le criticità legate alle espressioni aritmetiche, gestendo il caso in cui una funzione viene invocata più volte con valori diversi nei parametri e l'estensione del supporto ad altri linguaggi di programmazione. L'architettura del sistema progettato, basata sul processo dell'albero sintattico (AST) e sulla creazione di una pipeline, è concettualmente portabile.

In conclusione, uno dei principali miglioramenti riguarda l'integrazione dell'annotatore con funzioni per renderlo capace di riconoscere automaticamente le chiamate ai servizi esterni, sollevando il programmatore da questo compito. Questa implementazione richiederebbe un modello di analisi semantica, più attento alla logica di business che alla forma del codice, diverso da quello attuale, che si limita a un'analisi sintattica.

Bibliografia

[1] Lorenzo Maretti. *Tesi_MiniSL*. https://github.com/maretz1/Tesi_MiniSL, 2025. Consultato il 21 febbraio 2026.

[2] IBM. *What is Function-as-a-Service (FaaS)?* https://www.ibm.com/think/topics/faas, 2024. Consultato il 21 febbraio 2026.

[3] IBM. *What is serverless computing?* https://www.ibm.com/think/topics/serverless, 2024. Consultato il 21 febbraio 2026.

[4] The Go Authors. *The Go Programming Language.* https://go.dev, 2026. Consultato il 21 febbraio 2026.

[5] Giuseppe De Palma, Saverio Giallorenzo, Cosimo Laneve, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. *Serverless scheduling policies based on cost analysis.*

Ringraziamenti

Vorrei ringraziare il Prof. Saverio Giallorenzo e il Dr. Matteo Trentin per la disponibilità, il costante supporto e i consigli datimi durante la stesura della tesi, accompagnandomi senza farmi mai mancare la serenità necessaria per portare avanti questo elaborato.

Infine, un ringraziamento va a tutta la mia famiglia per avermi supportato in questo percorso di studi, e ai miei compagni di corso e amici per aver alleggerito questi anni.