Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica per il Management

SVILUPPO DI UN SISTEMA DI ANNOTAZIONE AUTOMATICA PER GOVERNARE L'ALLOCAZIONE DI FUNZIONI TYPESCRIPT SERVERLESS

Relatore:
Chiar.mo Prof.
SAVERIO GIALLORENZO

Presentata da:
MAURIZIO AMADORI

Correlatore:
Dott. MATTEO TRENTIN

I sessione
Anno Accademico 2024/25

Al bambino che sognava, e sogna ancora, dentro di me,
di conoscere, diventare grande e crescere.

Indice

1 Introduzione **9**
- 1.1 Cloud Computing . . . 9
- 1.2 Problemi e Limiti . . . 9
- 1.3 Contributi . . . 10

2 Background **13**
- 2.1 Serverless e FaaS . . . 13
  - 2.1.1 Serverless Scheduling Policies based on Cost Analysis . . . 14
- 2.2 Tool Utilizzati . . . 15
  - 2.2.1 TypeScript . . . 15
  - 2.2.2 Mini Serverless Language . . . 15
  - 2.2.3 Tree-sitter . . . 17
  - 2.2.4 Antlr4 . . . 17
  - 2.2.5 Acorn . . . 18

3 Implementazione **19**
- 3.1 Overview . . . 19
  - 3.1.1 Compiler example . . . 20
- 3.2 Annotazioni . . . 21
  - 3.2.1 Configurazioni . . . 23
- 3.3 Annotator . . . 23
  - 3.3.1 Annotazione . . . 24
  - 3.3.2 Gestione Variabili . . . 27
- 3.4 Extractor . . . 29
  - 3.4.1 Estrazione . . . 29
  - 3.4.2 Inlining . . . 31
- 3.5 Checker . . . 33
- 3.6 Come Usare il Compilatore . . . 35
  - 3.6.1 Architettura del Sistema . . . 35
  - 3.6.2 Installazione e Configurazione . . . 36
  - 3.6.3 Modalità di Utilizzo . . . 36
  - 3.6.4 Configurazione e Personalizzazione Componenti . . . 38
  - 3.6.5 Comandi Disponibili . . . 39

4 Conclusioni **41**
- 4.1 Impieghi e Limitazioni . . . 41
- 4.2 Possibili Sviluppi Futuri . . . 42

Bibliografia **45**

Elenco delle figure

1.1 Process Pipeline 11

2.1 Typical Serverless Platform Architecture 14
2.2 Mini Serverless Language Definition 17

3.1 Compiler Pipeline 19
3.2 Flow Chart Annotatore 24
3.3 Flow Chart Estrattore 29
3.4 Architettura della Pipeline del Compilatore 35

Elenco delle tabelle

3.1 Elenco delle Annotazioni 22
3.2 Comandi Make Disponibili nel Compilatore miniSL 39

Capitolo 1

Introduzione

1.1 Cloud Computing

Negli ultimi due decenni, il panorama dell'informatica ha subito una trasformazione radicale grazie all'avvento del cloud computing, che ha rivoluzionato il modo in cui le risorse informatiche vengono erogate, gestite e scalate. Questo paradigma ha permesso alle organizzazioni di accedere a potenza computazionale, storage e servizi software in modalità on-demand, riducendo drasticamente i costi di infrastruttura e aumentando l'agilità operativa, grazie alla velocità di trasmissione con cui è possibile trasmettere i dati su Internet oggi. Il cloud computing si è affermato come pilastro fondamentale dell'innovazione digitale, abilitando modelli di business dinamici, distribuiti e altamente scalabili.

All'interno di questo ecosistema, ha preso piede un approccio ancora più flessibile e orientato all'efficienza: il serverless Computing. Nonostante il nome possa suggerire l'assenza di server, il modello serverless implica semplicemente che la gestione dell'infrastruttura sottostante viene completamente astratta dal punto di vista dello sviluppatore. Questo consente di focalizzarsi esclusivamente sulla business logic, demandando al provider cloud la responsabilità di provisioning, scaling, bilanciamento del carico e fault tolerance. Una specializzazione del paradigma serverless è Function-as-a-Service (FaaS), una tipologia di servizio cloud che consente di eseguire singole funzioni in risposta a determinati eventi, senza la necessità di gestire server.

Il modello FaaS si caratterizza per una forte aderenza ai principi dell'event-driven architecture e dell'elasticità automatica. Le funzioni sono tipicamente stateless, a breve durata e scalano automaticamente in base al numero di richieste. Questo lo rende ideale per scenari come elaborazione di dati in tempo reale, automazione, microservizi e API gateway. Tra i provider più diffusi figurano AWS Lambda, Google Cloud Functions, Azure Functions e altri.

1.2 Problemi e Limiti

Nonostante i benefici evidenti in termini di costi, scalabilità e velocità di sviluppo, le architetture serverless e in particolare FaaS presentano anche una serie di problemi e limitazioni, che devono essere attentamente valutati durante la progettazione di sistemi complessi.

1. **Cold Start:** uno dei principali problemi di prestazioni che FaaS presenta è il cold start, ossia il tempo di latenza iniziale necessario per avviare un'istanza della funzione. Questo può influire negativamente sulle performance, soprattutto in contesti real-time o con requisiti di bassa latenza.

2. **Gestione dello stato:** le funzioni serverless sono, per loro natura, stateless. Ciò impone la necessità di utilizzare servizi esterni (come database o cache distribuite) per la persistenza dello stato, complicando l'architettura applicativa.

3. **Debugging e monitoraggio:** il debugging di applicazioni serverless può risultare più complesso rispetto ai modelli tradizionali, poiché l'esecuzione è distribuita, asincrona e astratta dal controllo diretto dello sviluppatore. Anche il tracciamento degli errori e il monitoraggio delle performance richiedono strumenti specifici.

4. **Vendor lock-in:** l'elevata dipendenza da specifiche implementazioni dei provider cloud può generare situazioni di vendor lock-in, rendendo difficile la migrazione o l'interoperabilità tra ambienti differenti.

5. **Limiti di esecuzione:** i provider impongono restrizioni sulle risorse disponibili per ogni funzione (tempo massimo di esecuzione, memoria, dimensione del pacchetto, ecc.), che possono rendere FaaS inadatto a certi tipi di carico computazionale o a task di lunga esecuzione.

6. **Costi imprevedibili:** sebbene il modello pay-per-use possa risultare economico per carichi variabili, l'adozione indiscriminata di FaaS può portare a costi alti in scenari ad alta intensità di invocazioni. In generale può essere difficile determinare i volumi dei picchi di richieste.

Nel contenuto di questa tesi verrà affrontato il problema di semplificare il processo di produzione di politiche di scheduling personalizzate, al fine di rendere più accessibile l'utilizzo di quest'ultime. Attraverso il loro utilizzo è infatti possibile migliorare le prestazioni dei sistemi FaaS, ad esempio, mitigando il problema della latenza. Pertanto, con il progetto sviluppato in questa tesi, cercheremo di rendere più facile ed accessibile l'utilizzo di politiche di scheduling personalizzate, visto il loro impatto sulle prestazioni di queste architetture.

1.3 Contributi

Questa tesi parte dal lavoro svolto all'interno dei paper [4] e [5], nei quali si utilizza il linguaggio miniSL e un analizzatore per migliorare le prestazioni delle funzioni serverless, tramite l'indirizzamento delle funzioni su nodi di calcolo specifici. In questa tesi propongo un traduttore, utile a semplificare e automatizzare il processo con cui si ottiene codice miniSL, e ad applicare i risultati dei paper citati. Definito all'interno dei paper [4] e [5], miniSL (mini Serverless Language) è un semplice linguaggio che implementa alcuni costrutti di base della programmazione di alto livello, e viene utilizzato nei paper [4] e [5] per ricavare funzioni di costo attraverso degli analizzatori standard. Per questo motivo, all'interno della tesi, esploreremo un sistema per tradurre il linguaggio TypeScript in linguaggio miniSL (sezione 2.2.2). Infatti è proprio grazie ai dati estratti

da miniSL che è stato possibile governare le politiche di scheduling serverless, personalizzandole per funzione, come fatto e dimostrato nel paper [5]. L'implementazione qui proposta rende più accessibile e meccanizza il processo con cui si ottiene miniSL, permettendo un'ulteriore automatizzazione dell'approccio proposto nei paper [5] e [4]. La soluzione implementata usa TypeScript come (esempio di) linguaggio con cui vengono scritte le funzioni serverless e, partendo da quest'ultimo, genera codice miniSL che può essere utilizzato per calcolare i costi associati alle funzioni. Il processo di compilazione avviene con delle assunzioni che vanno a circoscrivere il perimetro dei casi trattati e che, successivamente, vedremo nel dettaglio. Per iniziare, esploriamo la pipeline nelle sue tre fasi principali:

Figura 1.1: Process Pipeline

- **Annotazione:** in questa fase l'annotatore (sezione 3.3) arricchisce il codice TypeScript con delle annotazioni. Questo è uno step fondamentale in quanto le annotazioni garantiscono indipendenza rispetto al linguaggio di partenza (TypeScript).
- **Estrazione:** in questa fase, l'estrattore (sezione 3.4) si occupa di generare codice miniSL partendo dalle annotazioni costruite nella fase precedente.
- **Checking:** in questa fase finale, il checker (sezione 3.5) si occupa di controllare il codice miniSL prodotto, verificandone la correttezza.

Nel panorama FaaS, il progetto descritto sotto in dettaglio, è stato sviluppato per semplificare la pipeline che porta alla creazione di politiche di scheduling personalizzate, aiutando a calcolare funzioni di costo partendo da codice scritto in TypeScript. L'implementazione è comunque utilizzabile con qualsiasi sistema che produce codice contenente le annotazioni nella forma descritta e definita in seguito (sezione 3.2). In particolare, l'applicazione del progetto si posiziona nel controller, come mostrato in fig. 2.1.

Le tre fasi appena accennate vengono descritte approfonditamente all'interno della tesi nelle sezioni presenti nel capitolo 3. In particolare la sezione 3.2 descrive le annotazioni utilizzate nell'implementazione che accomunano annotatore ed estrattore. Esse sono fondamentali perchè definiscono un livello di astrazione, tale da consentire di generalizzare facilmente il processo di produzione di codice miniSL, partendo da potenzialmente un qualsiasi linguaggio di programmazione. Annotatore ed estrattore vengono approfonditi successivamente, nella sezione 3.3, dove si spiega com'è stato implementato l'annotatore e quali sono le sue funzionalità e i suoi vincoli. Vengono inoltre discusse le problematiche e i vantaggi che questo strumento porta con sé, oltre alle tecnologie utilizzate per implementarlo. Viene poi trattato come si è scelto di analizzare il codice sorgente per arricchirlo con le annotazioni, quali casi speciali si è deciso di gestire durante l'annotazione e quali no, con annesse relative motivazioni. Per esempio, uno fra i casi gestiti è la riassegnazione di una variabile, mentre le funzioni di ordine superiore si è scelto di non gestirle, se non sollevando un'eccezione, per semplicità. All'interno della sezione 3.4 si

tratta dell'estrattore e delle relative funzionalità, entrando nel dettaglio di cosa questa componente può fare. In particolare, viene spiegato perchè si è scelto di non gestire alcune casistiche, come la ricorsione e si è deciso invece di gestirne altre, come la chiamata a catena di più funzioni. La sezione 3.5 descrive com'è stato controllato il codice miniSL generato e accenna ai tool utilizzati per effettuare questo controllo sintattico. Infine, nel capitolo 4 si va ad analizzare il sistema implementato nel complesso riassumendo vincoli, problematiche, vantaggi e possibili sviluppi futuri.

Capitolo 2

Background

Per poter comprendere al meglio il contributo della tesi è necessario dettagliare alcuni concetti. Abbiamo accennato, nell'introduzione, a cosa intendiamo per serverless. Di seguito, oltre a ricapitolare, definiamo meglio i concetti.

2.1 Serverless e FaaS

Con il termine serverless intendiamo un modello di esecuzione che fa parte della branca del cloud computing, dove il provider del servizio alloca le risorse della macchina appena queste vengono richieste. In particolare, il FaaS (Function as a Service) è una versione del serverless che permette agli utenti di costruire architetture distribuite componendo più funzioni singole, senza doversi preoccupare di gestire le risorse che queste funzioni utilizzano, eseguendo. I vantaggi del FaaS sono quindi quelli di produrre soluzioni scalabili, permettendo all'utente di adattare i costi rispetto all'effettivo utilizzo dell'architettura. Al tempo stesso, FaaS ha anche il pregio di mantenere un elevato grado di:

- **elasticità:** proprietà che rende il sistema adattabile e capace di rispondere bene al cambio di esigenze di cui l'utente necessita.
- **disponibilità:** capacità di un sistema di essere sempre disponibile e tollerante ai guasti.
- **interoperabilità:** proprietà che permette di integrare diverse funzionalità implementate con tecnologie differenti in un unico sistema, tramite la loro composizione.

Tutto ciò permette all'utente di potersi concentrare meglio sulla logica dell'architettura, delegando la responsabilità della gestione delle risorse ai service provider. Tuttavia, bisogna sottolineare che contare sulla gestione da parte di terzi può causare problemi di lock-in. Questo problema è legato al fatto che può essere costoso cambiare provider una volta che il sistema è stato sviluppato, in quanto lo stack tecnologico di un altro provider potrebbe essere differente. Quindi c'è il rischio che il cambiamento risulti impossibile, per i troppi costi di transizione o per l'impossibilità di far cooperare tecnologie differenti. Inoltre abbiamo problemi quali:

- **debugging complesso:** l'aumento di complessità nel debugging dovuto all'esecuzione remota del codice.

- **cold start:** la reiterata istanziazione delle risorse a ogni esecuzione di una funzione genera un overhead che può causare un delay significativo.
- **inadatto a servizi long-running:** è difficile o impossibile gestire programmi di grandi dimensioni o appllicazioni stateful e di lunga durata.

Alla fig. 2.1, trovate una rappresentazione di una classica architettura serverless FaaS (presa dal paper [5]), nella quale possiamo vedere le diverse componenti dell'architettura di una piattaforma serverless e come queste dialogano fra loro. In particolare, nel modello di computazione FaaS, il codice viene eseguito sotto forma di funzioni isolate, attivate da eventi esterni, senza la necessità per lo sviluppatore di gestire direttamente server o infrastruttura. Quando si verifica un evento, questo viene intercettato da un controller che, a sua volta, inoltra la richiesta di esecuzione al worker designato. Il worker prende in carico l'evento ed avvia l'ambiente necessario ed esegue la funzione associata. Al termine dell'esecuzione, l'istanza che ha gestito la funzione viene chiusa, liberando automaticamente le risorse allocate.

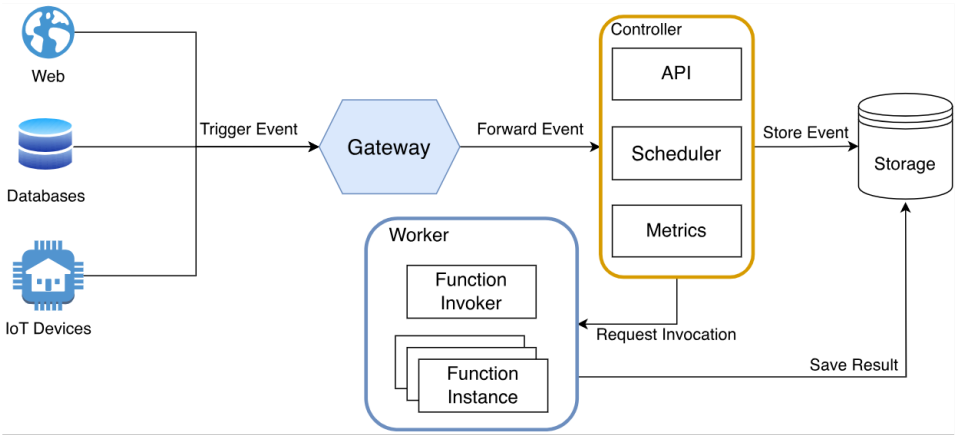

Figura 2.1: Typical Serverless Platform Architecture

2.1.1 Serverless Scheduling Policies based on Cost Analysis

Questa tesi parte dalle basi messe dai paper [4] e [5], cercando poi di automatizzare il processo che porta dallo sviluppo delle funzioni all'applicazione delle politiche di scheduling personalizzate. Nei paper [4], [5] si inizia a discutere di come integrare le informazioni ottenute dall'analisi statica delle funzioni serverless per informare le politiche di scheduling con il fine di migliorare le performance di quest'ultime, aiutando l'utente a selezionare il nodo migliore su cui allocare la funzione. Nello specifico viene definita una pipeline per estrarre equazioni di costo da un semplice linguaggio che rappresenta funzioni serverless, nella quale si utilizzano dei risolutori standard per determinare le equazioni di costo. I dati ottenuti dalla pipeline vengono poi integrati nelle politiche di scheduling della piattaforma. Per quanto riguarda il mainstream, infatti, vengono utilizzate politiche di scheduling rigide che allocano le funzioni in base ai worker disponibili. Quest'ultime possono però avere problemi di località, ad esempio causati dalla posizione geografica delle risorse richieste dalla funzione eseguita. Il problema in questione viene attenuato

dalla soluzione proposta nei paper [4] e [5]. La pipeline appena descritta, oltre ad impiegare il linguaggio miniSL, usa un analizzatore e dei risolutori standard, come PUBS [1] o CoFloCo [7], per estrarre le equazioni di costo da miniSL. Il linguaggio utilizzato per rappresentare le funzioni serverless, miniSL (mini Serverless Language), supporta le funzionalità minimali per descrivere il comportamento delle funzioni in ambiente serverless, da cui è possibile estrarre equazioni di costo tramite il procedimento descritto sopra. MiniSL si concentra sui costrutti essenziali necessari per descrivere le operazioni di accesso ai servizi, i comportamenti condizionali tramite semplici guardie e le iterazioni. L'esecuzione delle funzioni in ambiente serverless è innescata da eventi. Al momento della loro attivazione, le funzioni ricevono una sequenza di parametri di invocazione, che rappresentano le informazioni di input su cui esse operano. Nel contesto di miniSL, vi è un'unica funzione che contiene tutto il codice eseguibile, alla quale vengono passati i parametri e da cui viene estrapolata un'equazione di costo.

2.2 Tool Utilizzati

All'interno della tesi, si utilizzano diversi tool e tecnologie. A seguire, trovate una breve descrizione e introduzione ai principali tool utilizzati all'interno della tesi.

2.2.1 TypeScript

Per quanto riguarda il linguaggio in cui le funzioni serverless sono scritte, è stato utilizzato TypeScript [9], un linguaggio di programmazione di alto livello che estende JavaScript aggiungendo la tipizzazione statica opzionale. Progettato per migliorare la scalabilità e la manutenibilità del codice, TypeScript è largamente utilizzato nel mainstream e può essere impiegato sia lato server che lato client. Noi lo abbiamo utilizzato come linguaggio di partenza in cui il codice serverless è scritto, ma con un adeguato annotatore può essere utilizzato qualsiasi altro linguaggio eseguibile su piattaforma serverless.

2.2.2 Mini Serverless Language

Mini Serverless Language, abbreviato in miniSL, è un linguaggio minimale che utilizziamo per analizzare il comportamento delle funzioni nel contesto del serverless computing. In particolare, miniSL si concentra esclusivamente su costrutti essenziali, utili a definire: operazioni per l'accesso ai servizi, comportamenti condizionali tramite semplici guardie, iterazioni con contatori.

La sintassi del linguaggio miniSL è definita attraverso una grammatica nel paper [4] e ripresa nella fig. 2.2. Di seguito viene fornita una legenda per facilitare la comprensione dei simboli e delle notazioni utilizzate:

- `F` indica una funzione. Una funzione è definita come una sequenza di parametri `p` seguita da una freccia `=>` e da un blocco di istruzioni `S`. Ogni funzione viene eseguita al verificarsi di un evento di trigger.

- `S` indica uno statement (istruzione). Rappresenta le istruzioni che compongono il corpo della funzione. Può essere:

  - ε – Istruzione vuota (spesso omessa);
  - `call h(E)` – Chiamata a un servizio esterno `h` con parametri `E`;
  - `if (G) { S } else { S }` – Istruzione condizionale con due rami alternativi;
  - `for (i in range(0, E)) { S }` – Iterazione da 0 a `E` (escluso), usando il contatore `i`.

- `G` è una guardia (guard expression). Una guardia è un'espressione booleana che controlla il flusso condizionale. Può essere:
  - una normale espressione `E` che restituisce un booleano;
  - una chiamata a servizio `call h(E)` che si assume restituisca un booleano.

- `E` è un'espressione. Un'espressione rappresenta una combinazione di valori e operatori. Le espressioni possono includere:
  - un numero intero `n`;
  - un contatore `i`;
  - un parametro `p`;
  - un'espressione binaria $E \sharp E$, dove $\sharp$ è un operatore.

- $\sharp$ rappresenta un'operatore binario. Insieme di operatori aritmetici e logici supportati:

$$+, -, *, /, >, >=, ==, \&\&$$

- `p`, `p'` sono parametri di invocazione. Variabili che vengono passate come input alla funzione al momento del trigger. Provengono da un insieme numerabile.

- **`i, j`** sono contatori. Variabili usate come indici nei costrutti di iterazione, anch'esse provenienti da un insieme numerabile.

- `n` è un numero intero, rappresenta costanti numeriche.

- **`h, g, ...`** sono nomi di servizi. Rappresentano identificatori di servizi esterni che possono essere invocati tramite `call`.

- **Sovralinee** ($\bar{\mathtt{x}}$) indicano sequenze. Ad esempio, $\bar{\mathtt{p}}$ rappresenta una sequenza di parametri come $\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_n$.

```
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 | !E
♯ ::= + | - | * | / | > | < |
      >= | == | <= | && | ||
```

Figura 2.2: Mini Serverless Language Definition

All'interno della tesi miniSL è per noi il risultato finale, in quanto rappresenta la base su cui si fonda il sistema definito nei paper [4] e [5], infatti è da quest'ultimo che vengono estrapolate le informazioni per creare politiche di scheduling personalizzate.

2.2.3 Tree-sitter

Tree-sitter [3] è un tool che permette di generare parser e una libreria per il parsing. Tree-sitter permette di costruire un concrete syntax tree (CST) per un file sorgente ed efficientemente aggiornarlo quando il file viene modificato. I principali obiettivi di Tree-sitter sono quelli di essere:

- **Generale** abbastanza da poter fare il parsing di qualsiasi linguaggio di programmazione
- **Veloce** abbastanza da catturare ogni tasto premuto su una tastiera in un'editor di testo.
- **Robusto** abbastanza da permettere di ottenere utili risultati anche se ci sono errori sintattici
- **Indipendente** così che la runtime library (che è scritta in C11) possa essere inserita in qualsiasi applicazione.

In particolare, all'interno della tesi, Tree-sitter viene utilizzato dall'annotatore (sezione 3.3) per fare il parsing del codice TypeScript, e ottenere un CST che possa essere utilizzato per analizzare il codice e inserirvi ulteriori annotazioni sotto forma di commenti.

2.2.4 Antlr4

ANTLR4 (ANother Tool for Language Recognition) [8] è un generatore di parser potente e flessibile, utilizzato per costruire linguaggi, interpreti, compilatori e strumenti di analisi del codice. Sviluppato in Java, ANTLR consente di definire grammatiche formali da cui genera automaticamente lexer e parser in diversi linguaggi di programmazione. Grazie alla sua sintassi chiara e all'integrazione con strumenti moderni, è ampiamente adottato

nella progettazione di DSL (Domain-Specific Languages) e nell'analisi di linguaggi esistenti. In particolare, è stato utilizzato all'interno di questa implementazione per definire la grammatica di miniSL e generare un parser che è stato utilizzato per implementare il checker (sezione 3.5).

2.2.5 Acorn

Acorn [6] è un piccolo e veloce parser per JavaScript scritto in JavaScript, può essere utilizzato tramite NPM (Node Package Manager) e viene rilasciato come open source sotto licenza MIT. All'interno della tesi viene utilizzato dall'estrattore (sezione 3.4) per capire se alcune espressioni sono corrette e valide, in modo tale da tradurle in miniSL.

Capitolo 3

Implementazione

Questo capitolo mostra come la soluzione proposta, composta da annotatore, estrattore e checker sia stata implementata, e analizza nel dettaglio le scelte fatte. Inoltre sarà possibile comprendere come questi componenti funzionino singolarmente e nel complesso, tramite la visione di esempi pratici ed esaustivi che mostrano il funzionamento dell'intero processo di compilazione.

3.1 Overview

L' implementazione ha l'obiettivo di definire delle annotazioni usate sotto forma di commento inline per calcolare il costo dell'esecuzione di una funzione serverless, che effettua chiamate a servizi esterni in un sistema serverless (FaaS). Un annotatore si preoccuperà di scrivere queste annotazioni in codice TypeScript che poi verrà compilato in miniSL da un estrattore. Questo ci è utile per estrarre le funzioni di costo e definire le politiche di scheduling tramite una piattaforma serverless esterna.

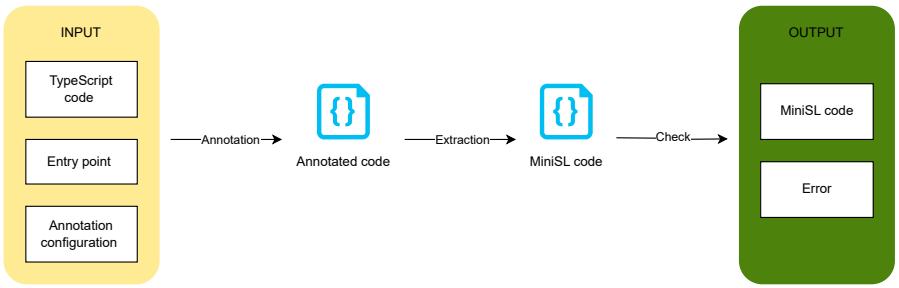

Figura 3.1: Compiler Pipeline

Il workflow descritto nell' immagine 3.1 comincia col prendere in input un file TypeScript con già le annotazioni dei servizi esterni, un'entry point corrispondente al nome di una funzione da cui incomincerà l'annotazione e un file di configurazione per le annotazioni. Si inizierà quindi il processo di annotazione durante il quale verrà generato un CST tramite Tree-Sitter [3], per determinare in quali rami sono presenti chiamate a servizi esterni in modo tale da aggiungerci gli opportuni commenti. Il risultante codice arricchito dalle annotazioni viene salvato in un file denominato Annotated code, che servirà da input per l'estrattore. A questo punto Annotated code insieme a un entry point,

corrispondente al nome di una funzione, e a un file di configurazione per le annotazioni, vengono passati in input all'estrattore. Durante il processo di estrazione verrà verificato che non ci siano chiamate ricorsive nel codice tramite l'utilizzo di un'AST, e verrà generato il codice miniSL basato sulle annotazioni presenti in Annotated code. Il codice miniSL sarà quindi controllato in fase di checking tramite un parser scritto con l'ausilio di ANTLR4 [8]. In caso di errori o di situazioni non gestite, il processo terminerà con la segnalazione di quest'ultime.

3.1.1 Compiler example

Si prenda come esempio introduttivo quello ai listati 3.1 e 3.2. Il listato 3.1 che contiene un esempio di input, presenta delle annotazioni che descrivono il codice. In particolare call indica la chiamata a un servizio esterno e defCall indica la definizione di quel servizio. Le annotazioni presenti sono però incomplete perché insufficienti per descrivere pienamente il codice e arrivare al risultato sperato presentato nel listato 3.2. L'annotatore (sezione 3.3) si occuperà di completarle in modo tale che l'estrattore (sezione 3.4) possa produrre il codice miniSL visibile al listato 3.2.

```
function main(deta: string[], length: number): void {
    for (let i = 0; i < length; i++) {
        // miniSL: call printMessage(deta, i)
        postMessage(deta, i);
    }

    printMessage(deta);
}

function printMessage(messages: string[]): void {
    for (let i = 0; i < messages.length; i++) {
        console.log("New " + i + " message: " + messages[i]);
    }
}

// miniSL: defCall postMessage(messages, position)
function postMessage(messages: string[], position: number): void
{
    fetch("https://example.com/api/messages", {
        method: "POST",
        headers: {
            "Content-Type": "application/json"
        },
        body: JSON.stringify({ message: messages[position] })
    })
}
```

Listing 3.1: Input Code

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

(deta, length) => {
```

```
4     for(i in range(0,  length)) {
5       call postMessage(deta, i)
6     }
7   }
```

Listing 3.2: miniSL Code

Il sistema di annotazioni viene utilizzato per descrivere il codice in modo tale da creare indipendenza tra il linguaggio di partenza e miniSL. Questa compilazione ci permette di passare quindi da un linguaggio target a miniSL, linguaggio dal quale siamo in grado di estrarre le funzioni di costo, in modo tale da poter ricavare il costo dell'esecuzione della funzione e del servizio esterno. Nelle prossime sezioni vedremo come l'annotatore, l'estrattore e il checker funzionano, analizzeremo i loro pregi e i loro limiti, considerando esempi più dettagliati di quest'ultimo che mira a essere solo un'introduzione a come queste componenti funzionano.

3.2 Annotazioni

Le annotazioni sono dei commenti utilizzati per descrivere il codice che le segue, in questo modo siamo potenzialmente indipendenti dal linguaggio in cui il codice è scritto quando lo andiamo a tradurre in linguaggio miniSL. La sintassi delle annotazioni è:

annotazione = startAnnotation, miniSLID, ":", controlStatement, [functionName, "(", [guard], ")"], [endAnnotation]

La lista completa delle annotazioni è visibile sotto forma di esempio nell'elenco 3.1. Qui le annotazioni non presentano **endAnnotation** perché sono scritte in TypeScript, per cui un commento di riga termina con la fine della riga e quindi **endAnnotation** può essere omesso.

Tabella 3.1: Elenco delle Annotazioni

// miniSL: for(i,E)
annotazione di un for con un contatore [`i`] e un'espressione [`E`] che ne determita l'ultimo indice

// miniSL: if(G)
annotazione di un'if con la sua guardia [`G`]

// miniSL: else
annotazione di un else

// miniSL: end
annotazione di chiusura di un blocco

// miniSL: call service(param, param)
annotazione di una chiamata a un servizio esterno con dei parametri [`param`]

// miniSL: invoke internalFunction()
annotazione di una chiamata a una funzione interna senza parametri

// miniSL: function internalFunction()
definizione di una funzione interna senza parametri

// miniSL: defCall service(param ,param)
definizione di un servizio esterno con dei parametri [`param`]

Come descritto nell'elenco 3.1 l'annotazione di un for ha come parametro una variabile che fa da contatore [`i`] e un'espressione aritmetica [`E`] che definisce il numero di iterazioni. Abbiamo poi la descrizione dell'if la cui guardia [`G`] può essere o un'espressione booleana o la chiamata a un servizio esterno call. Dopodiché else è utilizzato per separare i due blocchi di un if ed end è usato per definire la chiusura di un blocco di codice. Questi due costrutti per un if completo a due blocchi non vanno usati entrambi sull'else, ma basta utilizzare l'annotazione else per separare le alternative e end per chiudere il secondo blocco. L'annotazione call definisce la chiamata a un servizio esterno che sia esso implementato da una funzione scritta dall'utente o da una funzione di una libreria. call è seguito dal nome del servizio che si sta richiamando e dai suoi eventuali parametri attuali. Analogamente, il costrutto invoke definisce la chiamata a una funzione interna ed è seguito dal nome della funzione richiamata e dai suoi parametri attuali se presenti. Abbiamo poi dei costrutti dichiarativi come function, il quale permette all'utente di definire una funzione interna che sarà poi richiamata da invoke, esso è seguito dal nome della funzione interna definita e dai suoi parametri formali se ce ne sono. Infine c'è defCall che è il duale di function ma ci permette di definire servizi esterni siano questi dichiarati da un utente o appartenenti a una libreria, ed è seguito dal nome del servizio definito e dai suoi parametri formali se ce ne sono. I parametri delle annotazioni if, for e call sono quelli descritti nel paper [4] e riportati nella sezione 2.2.2.

3.2.1 Configurazioni

Le caratteristiche delle annotazioni nel linguaggio scelto possono essere modificate e vanno specificate all'interno del file di configurazione `config.json`. Nel listato 3.3 troviamo:

- **startAnnotation**
- **endAnnotation**
- **miniSLID**
- **controlStatements**

I primi due parametri, **startAnnotation** e **endAnnotation**, esprimono ciò che identifica l'inizio e la fine del commento di riga. Se la fine del commento di riga coincide con la fine della riga, **endAnnotation** può essere lasciato vuoto. **miniSLID** è l'identificatore di un'annotazione, ovvero ciò che fa di un normale commento all'interno del linguaggio un'annotazione. I **controlStatements** sono le keywords utilizzate per interpretare le annotazioni e generare il codice miniSL. È importante ridefinire questi parametri a seconda del linguaggio di partenza, in particolar modo se si utilizza singolarmente l'estrattore che è più versatile (sezione 3.4).

```
{
    "startAnnotation" : "//",
    "endAnnotation" : "",
    "miniSLID" : "miniSL",
    "controlStatements" : {
        "for": "for",
        "call": "call",
        "if": "if",
        "else": "else",
        "end": "end",
        "function": "function",
        "invoke": "invoke",
        "defCall": "defCall"
    }
}
```

Listing 3.3: Configuration File

3.3 Annotator

L'annotatore è una componente che, partendo da un file contenente codice TypeScript parzialmente annotato, produce in output lo stesso codice arricchito da ulteriori annotazioni necessarie all'estrattore per generare codice miniSL. Il codice parzialmente annotato, in particolare, è codice che contiene già al suo interno tutte le annotazioni per quanto riguarda le call a servizi esterni e la loro definizione tramite defCall. Questo prerequisito serve per semplificare il lavoro dell'annotatore, che altrimenti sarebbe dovuto essere in grado di riconoscere almeno un buon numero di invocazioni a servizi esterni. Tutto ciò

non riguarda solo il codice prodotto dall'utente ma anche eventuali funzioni di librerie o framework. Considerando quindi che analizzare tutte le chiamate a servizi esterni avrebbe aumentato notevolmente la sua complessità, dati i molteplici modi con cui può essere fatta una chiamata HTTP oggi, abbiamo deciso di imporre questo vincolo. Se si vuole usare l'annotatore in coppia con l'estrattore descritto nella sezione 3.4 è importante che i loro file di configurazione `config.json` abbiano lo stesso contenuto.

3.3.1 Annotazione

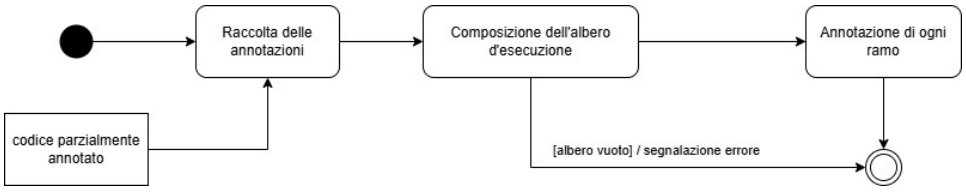

Figura 3.2: Flow Chart Annotatore

L'annotatore all'avvio prende in input: il percorso del file di input, il percorso del file di output e il nome della funzione da cui incomincia l'esecuzione che chiameremo entry point e per noi sarà sempre la funzione main. L'annotatore compone un primo concrete syntax tree utilizzando un parser generato con Tree-Sitter [3], un tool compatibile coi principali linguaggi di programmazione. Utilizzando il CST, l'annotatore individua tutti i commenti che sono stati inseriti in precedenza manualmente dall'utente. A partire dall'albero viene generato un grafo per individuare i percorsi di esecuzione che coinvolgono le annotazioni call. Individuati tutti i percorsi di esecuzione utili attraversando il grafo dall'entry point a tutte le annotazioni call, si compone un'execution tree che verrà utilizzato per arricchire ulteriormente il codice con altre annotazioni. L'ET ha ora solo i rami che contengono le annotazioni fatte dall'utente. In questo modo le annotazioni verranno aggiunte solo per i rami che ci interessano, ovvero quelli utili a calcolare il costo dell'esecuzione di una funzione serverless, che fa chiamate a servizi esterni. L'annotatore incomincia ora a inserire i commenti percorrendo tutte le rotte all'interno dell'ET e inserendo le annotazioni dove e come opportuno. In particolare posizionandole subito prima del codice che ci interessa e a cui sono rivolte, si veda la sezione 3.2. In questo momento avviene anche una semplice gestione delle variabili e dei parametri delle funzioni. In particolare, se prima della condizione di un if o di un for viene riassegnata una variabile o un parametro (all'interno della stessa funzione), l'annotatore con alcune limitazioni cattura questa nuova valorizzazione. Una volta ottenuto il nuovo valore, l'annotatore, procederà a sostituirlo direttamente all'interno della condizione. Se, invece, la riassegnazione non viene valutata come corretta o gestibile viene sollevata un'eccezione.

```
1 function main(x: number, y: number, type: boolean): void {
2     if (type) {
3         evaluateX(x);
4         evaluateY(y);
5     } else {
6         evaluateX(x);
7     }
```

```
}

function evaluateX(x: number): number {
    if (x > 0) {
        return 0;
    }
    return 5;
}

function evaluateY(y: number): number {
    if (y > 0) {
        return generateNumber();
    } else {
        return 5;
    }
}

function generateNumber(): number {
    // miniSL: call random()
    return random();
}

// miniSL: defCall random()
function random(): number {
    console.log("Random external service called");
    return Math.random();
}
```

Listing 3.4: Input Code

```
// miniSL: function main(x, y, type)
function main(x: number, y: number, type: boolean): void {
    // miniSL: if(type)
    if (type) {
        evaluateX(x);
        // miniSL: invoke evaluateY(y)
        evaluateY(y);
    } // miniSL: else
    else {
        evaluateX(x);
        // miniSL: end
    }
    // miniSL: end
}

function evaluateX(x: number): number {
    if (x > 0) {
        return 0;
    }
    return 5;
```

```
21 }
22
23 // miniSL: function evaluateY(y)
24 function evaluateY(y: number): number {
25     // miniSL: if(y > 0)
26     if (y > 0) {
27         return // miniSL: invoke generateNumber()
28         generateNumber();
29     } // miniSL: else
30     else {
31         return 5;
32         // miniSL: end
33     }
34     // miniSL: end
35 }
36
37 // miniSL: function generateNumber()
38 function generateNumber(): number {
39     // miniSL: call random()
40     return random();
41     // miniSL: end
42 }
43
44 // miniSL: defCall random()
45 function random(): number {
46     console.log("Random external service called");
47     return Math.random();
48 }
```

Listing 3.5: Output Annotated Code

Si prenda come esempio di input il listato 3.4. Partendo da quest'ultimo possiamo vedere le annotazioni già presenti nell'input, necessarie all'annotatore per poter completare l'annotazione. Quest'ultimo attraversando il CST generato da Tree-Sitter [3] compone un grafo delle chiamate fatte dalle funzioni. L'annotatore cerca poi, nel CST, le funzioni annotate come call eseguendo infine un DFS sul grafo per determinare tutti i percorsi dall'entry point alle annotazioni call, generando così un'execution tree. L'ET, che ha come root l'entry point, permette di annotare tutto il codice in maniera corretta, evitando l'annotazione di rami inutili o superflui. A questo punto l'estrattore visita tutto l'ET inserendo le annotazioni subito prima dei nodi di interesse come:

- if
- else
- for
- dichiarazioni di funzione
- chiamte di funzione
- chiusura di blocchi

Producendo in output il codice annotato presente nel elenco 3.5. Si noti come entrambi gli if presenti nell'esempio vengano sempre annotati completamente anche se uno dei due blocchi è vuoto, questo avviene per semplicità. In particolare il blocco dell'else potrebbe essere omesso e saltato in quanto non contiene annotazioni utili, spostando l'end alla fine dell'if in sostituzione dell'else.

3.3.2 Gestione Variabili

Durante l'annotazione di blocchi if o for, l'annotatore verifica se le variabili utilizzate nelle rispettive condizioni sono state riassegnate all'interno della stessa funzione, prima dell'esecuzione dell'if o del for. Se questo è il caso, una volta individuate le riassegnazioni, quest'ultime vengono parsate e valutate una a una per determinare se possono essere staticamente risolte. Nel caso la risposta sia affermativa si procede a svolgere la riassegnazione inserendo il nuovo valore nella condizione, in caso contrario verrà sollevata un'eccezione. In particolare risolviamo la riassegnazione solo nei seguenti casi:

- espressioni numeriche che possono contenere solamente la variabile che si sta riassegnando oltre a operatori e numeri
- espressioni booleane che possono contenere solamente la variabile che si sta riassegnando oltre a operatori e valori booleani

Durante questo processo, si considera solo l'ultima riassegnazione prima della condizione come valida. Se c'è una sequenza di riassegnazioni, queste non vengono innestate. Inoltre è importante notare come questo aumenti notevolmente la complessità, motivo per cui non viene gestito il caso in cui la riassegnazione di una variabile è condizionata.

```
1  function evaluateY(y: number): number {
2      y = 1 + y;
3
4      if (y > 0) {
5          y = 1 + y;
6          if (y > 0) {
7              return generateNumber();
8          }
9      } else {
10         return 5;
11     }
12 }
```

Listing 3.6: Input Code

```
1  // miniSL: function evaluateY(y)
2  function evaluateY(y: number): number {
3      y = 1 + y;
4
5      // miniSL: if(1 + y > 0)
6      if (y > 0) {
7          y = 1 + y;
8          // miniSL: if(1 + y > 0)
```

```
9           if (y > 0) {
10              return // miniSL: invoke generateNumber()
11              generateNumber();
12              // miniSL: end
13          }
14      } // miniSL: else
15      else {
16          return 5;
17          // miniSL: end
18      }
19      // miniSL: end
20  }
```

Listing 3.7: Output Annotated Code

Si prenda come esempio quello al listato 3.6, che mostra una piccola modifica del listato 3.4. Durante la fase di annotazione la funzione **evaluateY** invoca funzioni che contengono delle call per cui viene annotata. Partendo dall'alto si incomincia ad annotare la funzione e una volta arrivati al primo if [riga 4] viene controllato se ci sono variabili nella condizione. In questo caso si trova **y**, per cui viene cercata la sua riassegnazione tra l'if e la firma della funzione. Se la riassegnazione, come in questo caso, viene trovata il suo valore viene sostituito nella condizione e si compone l'annotazione finale. L'annotatore incontra poi il secondo if [riga 6] e lo annota come in precedenza, ma in questo caso, siccome incontra più riassegnazioni di **y**, prende come buona l'ultima, non facendo per semplicità la composizione statica delle riassegnazioni. L'annotazione viene poi conclusa con l'invocazione di **generateNumber** e la chiusura di tutti i blocchi, quindi il risultato finale sarà quello mostarto nel listato 3.7.

```
1   function evaluateY(y: number): number {
2       if (y > 0) {
3           y = 1 + y; // sollevo un'eccezione
4       }
5
6       if (y > 0) {
7           return generateNumber();
8       } else {
9           return 5;
10      }
11  }
```

Listing 3.8: Input Code

Analizziamo più nel dettaglio il caso di assegnazione condizionata, prendendo come esempio quello nel listato 3.8, che mostra una piccola modifica del listato 3.4. Quando un if prova a riassegnare una variabile in maniera condizionata [riga 3] crea un problema, in quanto bisogna valutare la condizione del primo if per determinare come la variabile viene riassegnata, e quindi poter innestare il nuovo valore nella condizione del secondo if [riga 6]. Questo discorso è chiaramente valido per qualsiasi blocco condizionato all'interno del codice. Di conseguenza, nel caso in cui ci sia un riassegnamento di una variabile utilizzata nella condizione di un if o di un for, ma questa assegnazione avvenga all'interno di un precedente if (o qualsiasi altro blocco condizionato), viene sollevata un'eccezione

e l'annotazione termina. Il primo if [riga 2], in sé e per sé, non crea quindi problemi se non quando si sta annotando il secondo [riga 6]. Infatti per determinare il valore di **y** nel secondo if bisognerebbe fare una tabella di verità, assegnando un valore diverso a **y** a seconda della condizione e dei suoi parametri. Precisamente, in questo caso, non è possibile determinare in maniera univoca il valore di **y**, questo crea un problema che non abbiamo risolto e che viene gestito sollevando un'eccezione.

Ricapitolando, l'annotatore partendo da un entry point e dal codice di input contenente già le annotazioni di call e defCall, produce in output il codice annotato. Durante questo processo l'annotatore si preoccupa di annotare solamente i rami dell'ET che portano all'esecuzione delle call, in quanto sono quelli che ci interessano. Le variabili presenti nelle condizioni dell'if o del for verranno quindi opportunamente riassegnate, se ridefinite tra l'esecuzione di quest'ultimi e la firma della funzione. In questa fase vengono considerati i casi speciali in cui questa riassegnazione non è valida, perché non univoca o perché non viene fatta con espressioni booleane o aritmetiche. In tali occasioni viene sollevata un'eccezione. Questo risultato ci permette di prendere l'output dell'annotatore e passarlo in input all'estrattore [sezione 3.4] per produrre codice miniSL.

3.4 Extractor

L'estrattore è una componente che, dato un codice annotato in un linguaggio scelto dall'utente e un nome di una funzione che chiameremo entry point, ci permette di ottenere il codice miniSL associato alle annotazioni presenti nel codice del file di input. Le annotazioni possono essere aggiunte al codice di input in maniera semi-automatica tramite l'annotatore o anche manualmente vista la loro non elevata complessità. Se si utilizza l'annotatore descritto nella sezione 3.3 è importante che il file di configurazione `config.json` di quest'ultimo abbia il medesimo contenuto di quello dell'estrattore.

3.4.1 Estrazione

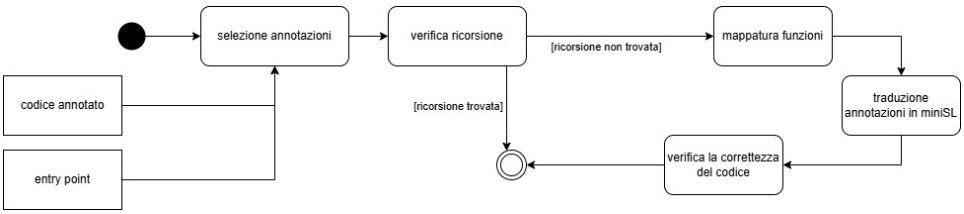

Figura 3.3: Flow Chart Estrattore

L'estrattore all'avvio legge tutto il file di input e ne estrae le annotazioni selezionando quelle che presentano un commento con **miniSLID**. Si prenda come esempio di input file il listato [3.9]. Una volta letto il codice, viene generato un albero delle chiamate basandosi sulle annotazioni function e invoke, tramite il quale l'annotatore controlla se avviene una chiamata ricorsiva all'interno dell'albero. Se questo è il caso, viene sollevata un'eccezione, mentre se non c'è ricorsione l'estrattore mappa tutte le funzioni function

alle relative annotazioni associate. Il processo prosegue traducendo le funzioni in linguaggio miniSL a partire dall'entry point, percorrendo poi tutto l'albero delle chiamate finché non ha generato il codice di tutte le funzioni. Quando si trovano invocazioni ad altre funzioni, queste vengono tradotte immediatamente (se non è già stato fatto in precedenza) per poi essere innestate direttamente nella funzione chiamante tramite inlining. Una volta che tutto il codice è stato tradotto, il blocco di miniSL generato, visibile nel listato [3.10], viene stampato a console.

Durante la traduzione delle annotazioni in miniSL vengono controllate le regole definite nel paper [4] e riportate nella sezione 2.2.2 per assicurarsi che il codice possa essere tradotto e i vincoli di miniSL non siano violati. Se una delle regole non viene rispettata, verrà sollevata un'eccezione e il processo terminerà.

```
1  // miniSL: defCall map(jobs,index)
2  function map(jobs: Job[], index: number): void {
3      console.log(`Mapping jobs for index ${index} by calling an
       external service`);
4  }
5
6  // miniSL: defCall reduce(jobs,mapIndex,reduceIndex)
7  function reduce(jobs: Job[], mapIndex: number, reduceIndex:
   number): void {
8      console.log(`Reducing jobs for mapIndex ${mapIndex} and
       reduceIndex ${reduceIndex} by calling an external service`);
9  }
10
11 // miniSL: function main(jobs,m,r)
12 function main(jobs: Job[], m: number, r: number): void {
13     // miniSL: for(i,m)
14     for (let i = 0; i < m; i++) {
15         // miniSL: call map(jobs,i)
16         map(jobs, i);
17         // miniSL: for(j,r)
18         for (let j = 0; j < r; j++) {
19             // miniSL: call reduce(jobs,i,j)
20             reduce(jobs, i, j);
21             // miniSL: end
22         }
23         // miniSL: end
24     }
25     // miniSL: end
26 }
```

Listing 3.9: Input Code

```
1  service map : (void) -> void;
2  service reduce : (void) -> void;
3
4  (jobs, m, r) => {
5    for(i in range(0, m)) {
```

```
6       call map(jobs,i)
7       for(j in range(0, r)) {
8         call reduce(jobs,i,j)
9       }
10    }
11  }
```

Listing 3.10: miniSL Code

Nell'esempio sopra riportato possiamo vedere come il codice TypeScript del listato 3.9 venga tradotto in codice miniSL, ottenendo così il risultato presente nel listato 3.10. L'estrattore, preso in input il codice TypeScript, verificherà che non ci siano ricorsioni, inizierà quindi ad analizzare la function main che considereremo come l'entry point. Dopodiché andrà a tradurre quest'ultima in miniSL verificando che all'invocazione di una call corrisponda il suo defCall. Infine il risultato presente al listato 3.10 viene stampato in output e verificato dal Checker. L'estrattore non ha la percezione di profondità dell'AST, per cui non possiede il concetto di scope, ma analizza il codice di input e prende come valida qualsiasi funzione dichiarata opportunamente con lo stesso nome di quella richiamata. La scelta di avere di fatto un AST sostanzialmente piatto dal punto di vista dello scope semplifica il lavoro dell'estrattore, ma rimane un vincolo importante che potrà essere risolto in sviluppi futuri. La scelta di utilizzare delle annotazioni configurabili, invece, consente all'estrattore di produrre codice miniSL indipendentemente dal linguaggio in cui le annotazioni sono scritte. Rendendolo versatile e permettendoci di utilizzarlo per tradurre codice partendo potenzialmente da qualsiasi linguaggio.

3.4.2 Inlining

Durante la mappatura delle funzioni, esse sono associate al loro blocco di annotazioni. Successivamente, la prima volta che una function viene richiamata essa viene tradotta e salvata separatamente per poter essere riutilizzata in caso di invocazioni future. In questo momento il codice tradotto, prima di essere innestato al posto dell'annotazione invoke viene ulteriormente elaborato sostituendo i parametri attuali, passati durante l'invocazione, coi parametri formali presenti in fase dichiarativa. In caso di mismatch dei parametri viene sollevata un'eccezione e il programma termina. Questo procedimento avviene solamente per l'invocazione di funzioni interne tramite invoke di function, come mostra il listato 3.11, che produce in output lo stesso risultato visto sopra nel listato 3.10. Questa scelta è stata fatta per poter analizzare e gestire codice serverless un po' più strutturato, ma per semplicità non si sono gestiti i seguenti casi :

- ricorsione
- funzioni di ordine superiore

sollevando in tutti e due i casi un'eccezione e terminando l'esecuzione. I controlli fatti per verificare la correttezza e la congruenza dei parametri di invoke e function sono svolti utilizzando in buona parte delle semplici RegEX, come anche la sostituzione dei parametri formali con quelli attuali durante l'inlining.

```
1  // miniSL: function makeWork(jobs,m, r)
2  function makeWork(jobs: Job[], m: number, r: number): void {
```

```
3       // miniSL: for(i,m)
4       for (let i = 0; i < m; i++) {
5           // miniSL: call map(jobs,i)
6           map(jobs, i);
7           // miniSL: for(j,r)
8           for (let j = 0; j < r; j++) {
9               // miniSL: call reduce(jobs,i,j)
10              reduce(jobs, i, j);
11              // miniSL: end
12          }
13          // miniSL: end
14      }
15      // miniSL: end
16  }
17
18  // miniSL: function main(jobs,m,r)
19  function main(jobs: Job[], m: number, r: number): void {
20      // miniSL: invoke makeWork(jobs,m,r)
21      makeWork(jobs, m, r);
22      // miniSL: end
23  }
```

Listing 3.11: Input Code

Per quanto riguarda le call a servizi esterni, prendiamo come esempio il listato 3.12 [riga 20]. L'estrattore inizialmente verifica che effettivamente la firma del metodo combaci con quella definita da defCall, ma non si preoccupa di analizzare il contenuto del metodo definito tramite questo costrutto. DefCall, come nel listato 3.12 [riga 1], ci permette di verificare che un servizio esterno sia stato definito, in questo modo possiamo segnare come chiamate esterne anche librerie o altro codice non scritto da noi, senza doverci preoccupare di quello che è il suo contenuto. Questo approccio risolve il problema di non sapere se effettivamente una funzione è o meno una call, permettendoci di sollevare un'eccezione quando questo accade. Se per esempio, come nel listato 3.12, ci ritroviamo una funzione nella guardia di un if, anziché assumere a prescindere che questa sia una call, tramite defCall possiamo verificare che questa sia stata etichettata come tale dall'utente. Nel caso la call non sia definita solleviamo un errore e terminiamo il processo, in quanto in miniSL un'if non può avere una funzione interna nella sua guardia. In questo caso infatti non siamo in grado di determinare di che tipo di funzione si tratti, se di una call o di un'invoke. Si veda il paper [4] o la sezione 2.2.2.

```
1   // miniSL: defCall PremiumService()
2   function PremiumService(): void {
3       console.log(‘An external premium service called‘);
4   }
5
6   // miniSL: defCall BasicService()
7   function BasicService(): void {
8       console.log(‘An external basic service called‘);
9   }
10
```

```
11 // miniSL: defCall IsPremiumUser(username)
12 function IsPremiumUser(username: string): boolean {
13     return username === 'premiumUser';
14 }
15
16 // miniSL: function main(username)
17 function main(username: string): void {
18     // miniSL: if(IsPremiumUser(username))
19     if (IsPremiumUser(username)) {
20         // miniSL: call PremiumService()
21         PremiumService();
22         // miniSL: else
23     } else {
24         // miniSL: call BasicService()
25         BasicService();
26         // miniSL: end
27     }
28     // miniSL: end
29 }
```

Listing 3.12: Input Code

Ricapitolando, durante la traduzione di queste annotazioni viene controllato che la guardia dell'if sia corretta, la stessa cosa avviene per le guardie delle funzioni e dei for. Per quanto riguarda l'esempio del listato 3.12 si verifica che la guardia dell'if sia un'espressione booleana o una chiamata a un servizio esterno. Se stessimo trattando un for, l'estrattore controllerebbe che nella guardia il suo secondo termine sia un'espressione aritmetica, mentre se trattassimo una call a un servizio esterno verrebbe verificato che i suoi parametri siano variabili o espressioni. Le verifiche riguardanti le espressioni vengono fatte utilizzando **acorn** [6] un piccolo parser scritto in JavaScript per JavaScript che produce un'AST. Navigando l'albero verifico se l'espressione è corretta e ne inferisco la tipologia (booleana o aritmetica), per poi comportarmi in maniera differente a seconda delle casistiche sopra riportate.

3.5 Checker

Il checker utilizza un parser scritto con l'ausilio del tool ANTLR4 [8]; questo ci è utile per verificare che il codice miniSL prodotto rispetti la grammatica mostrata nel listato 3.13. La grammatica utilizzata è stata scritta seguendo le linee guida fornite nel paper [4], le cui regole sono state riportate nella sezione 2.2.2. A partire dalla grammatica è stato poi generato un parser per miniSL che ho integrato con l'estrattore per verificare che il codice generato da quest'ultimo fosse formalmente corretto. Si prenda come esempio quello presente nel listato 3.10 e la grammatica presente nel listato 3.13. Nel caso in cui ci siano incongruenze tra il codice miniSL generato e la grammatica, il checker visualizzerà gli eventuali errori che il parser ha riscontrato, permettendo un rapido confronto col codice miniSL che invece viene sempre visualizzato.

```
1 grammar miniSLGrammar;
2
```

```
3  /*
4   * Parser Rules
5   */
6
7  prg: serviceDecl* init;
8
9  init: '(' formalParams? ')' '=>' '{' stm '}';
10
11 serviceDecl: 'service' ID ':' '(' 'void' ')' '->' 'void' ';';
12
13 stm: serviceCall
14    | 'if' '(' expOrCall ')' '{' stm '}' ('else' '{' stm '}')? stm
15    | 'for' '(' ID 'in' 'range' '(' NUMBER ',' exp ')' ')' '{' stm
       '}' stm
16    | ;
17
18 serviceCall: 'call' ID '(' (exp (',' exp)*)? ')' stm;
19
20 expOrCall: exp | serviceCall;
21
22 exp: exp ('+' | '-' | '*' | '/' | '&&' | '||' | '==' | '!=' | '<'
    | '<=' | '>' | '>=') exp # binExp
23    | '(' exp ')' # parenExp
24    | ID '.' ID # callFun
25    | STRING # stringExp
26    | NUMBER # valExp
27    | ID # derExp;
28
29 formalParams: ID (',' ID)*;
30
31 /*
32  * Lexer Rules
33  */
34
35 fragment CHAR: [a-zA-Z];
36 ID: [a-zA-Z_][a-zA-Z_0-9]*;
37
38 fragment DIGIT: [0-9];
39 NUMBER: DIGIT+;
40
41 STRING: '"' (~["\r\n])* '"';
42
43 // Comments and Whitespace
44 NEWLINE: ('\r\n' | '\r' | '\n') -> skip;
45 LINECOMMENTS: '//' ~('\n'|'\r')* -> skip;
46 WS: [ \t]+ -> skip;
47 BLOCKCOMMENTS: '/*' .*? '*/' -> skip;
```

Listing 3.13: miniSL ANTLR4 Grammar

3.6 Come Usare il Compilatore

Il progetto da me sviluppato è consultabile su GitHub [2]. Vediamo brevemente com'è possibile utilizzare il compilatore in aggregato e, in seguito, componente per componente. Per una guida più approfondita, il file `README.md` su GitHub riporta maggiori dettagli.

3.6.1 Architettura del Sistema

Il sistema segue un'architettura modulare basata su tre componenti principali che operano in sequenza:

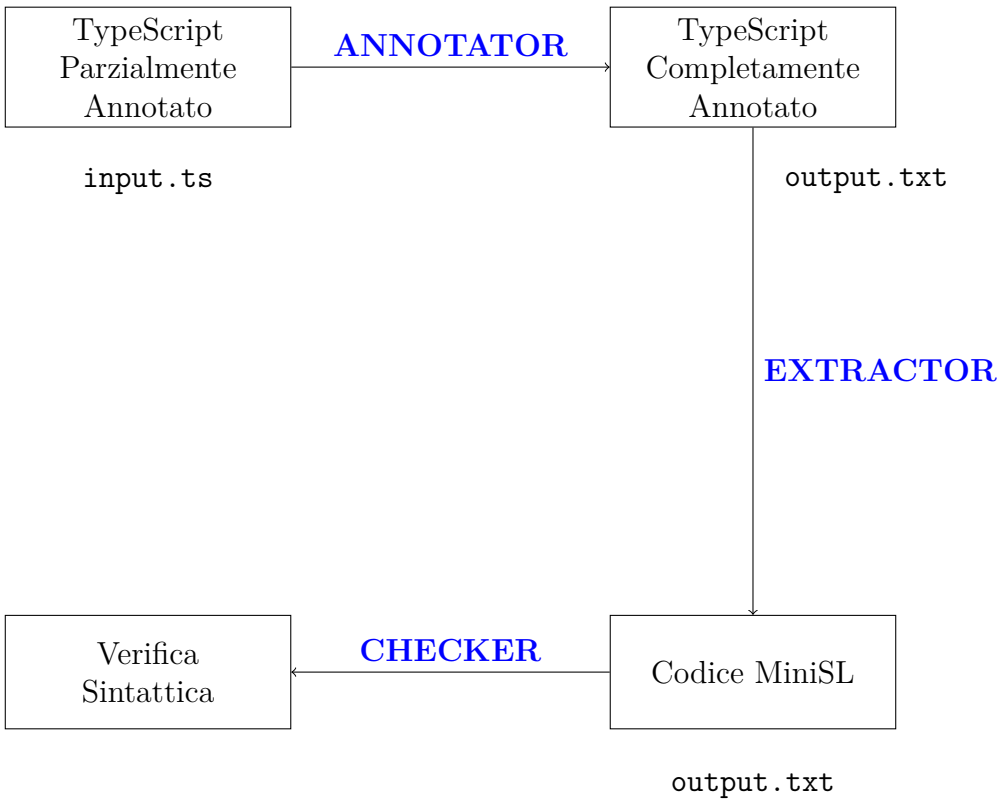

Figura 3.4: Architettura della Pipeline del Compilatore

Componenti del Sistema

Annotator
Trasforma file TypeScript parzialmente annotati (contenenti solo annotazioni **call** e **defCall**) in file completamente annotati con tutte le strutture di controllo necessarie per l'estrazione di miniSL.

Extractor
Converte il codice TypeScript completamente annotato in codice miniSL, implementando la logica di traduzione delle annotazioni in costrutti del linguaggio target.

Checker
Verifica la correttezza sintattica del codice miniSL generato utilizzando una grammatica ANTLR, garantendo la conformità alle specifiche del linguaggio.

3.6.2 Installazione e Configurazione

Il sistema richiede un ambiente di esecuzione con i seguenti prerequisiti:

- Node.js (versione 16 o superiore)
- NPM (Node Package Manager)
- Make (per l'automazione dei processi)

I comandi Make definiti vanno lanciati tutti dalla locazione del progetto (e non da dentro le singole componenti). L'installazione e configurazione del sistema avviene attraverso il comando:

```
1 make setup
```

Listing 3.14: Setup iniziale del sistema

Questo comando esegue automaticamente:

1. Installazione delle dipendenze NPM per tutti i componenti
2. Compilazione del codice TypeScript in JavaScript eseguibile
3. Preparazione dell'ambiente per l'esecuzione della pipeline

3.6.3 Modalità di Utilizzo

Esecuzione della Pipeline Completa

L'utilizzo principale del tool avviene attraverso l'esecuzione della pipeline completa:

```
1 make pipeline
```

Listing 3.15: Esecuzione pipeline completa

Il processo automatico eseguito comprende tre fasi sequenziali:

1. **Fase di Annotazione**:
   Trasformazione di `annotator/inputCode/input.ts` in `annotator/output.txt`
2. **Fase di Estrazione**:
   Conversione di `annotator/output.txt` in codice miniSL salvato in `extractor/output.txt`
3. **Fase di Verifica**:
   Validazione sintattica del codice miniSL da `extractor/output.txt` con output su console

Come appena descritto, i file prodotti da ogni componente vengono usati come input per la successiva, a meno che non si specifichi il percorso di un file all'interno del comando, che sia questo il comando per eseguire la pipeline completa o un solo componente.

Utilizzo con Parametri Personalizzati

Il sistema può essere utilizzato senza specificare parametri, ma supporta anche la personalizzazione attraverso parametri specifici:

```
make pipeline ENTRY_POINT=myMainFunction
```

Listing 3.16: Pipeline con entry point personalizzato

```
make annotate INPUT_FILE=inputCode/myCustomFile.ts ENTRY_POINT=
myFunction
```

Listing 3.17: Annotazione di file specifico

Esecuzione di Componenti Singoli

Per analisi specifiche o attività di debugging, ogni componente può essere eseguito indipendentemente:

Annotator

```
make build-annotator
make annotate INPUT_FILE=inputCode/input.ts ENTRY_POINT=main
```

Listing 3.18: Esecuzione isolata dell'Annotatore

Input: Percorso del file TypeScript contenente il codice con annotazioni parziali (**call** e **defCall**), e il nome della funzione da cui iniziare l'annotazione. Entrambi i parametri sono opzionali e possono essere omessi.

Output: File `output.txt` con annotazioni complete

Extractor

```
make build-extractor
make extract INPUT_FILE=../annotator/output.txt ENTRY_POINT=main
```

Listing 3.19: Esecuzione isolata dell'Estrattore

Input: Percorso del file TypeScript completamente annotato e nome della funzione da cui iniziare l'estrazione. Entrambi i parametri sono opzionali e possono essere omessi.

Output: Codice miniSL in `output.txt`

Checker

```
make build-checker
make check INPUT_FILE=extractor/output.txt
```

Listing 3.20: Esecuzione isolata del Checker

Input: Percorso del file contente il codice miniSL da validare

Output: Report di validazione sintattica stampato a console

Modalità di Sviluppo Rapido

Per iterazioni di sviluppo e testing, il tool offre comandi di sviluppo rapido:

```
make dev-annotator     # Build + esecuzione rapida per annotator
make dev-extractor     # Build + esecuzione rapida per extractor
make dev-checker       # Build + esecuzione rapida per checker
```

Listing 3.21: Comandi di sviluppo rapido

Esempio di Sessione Completa

Un esempio tipico di utilizzo del sistema comprende i seguenti passaggi:

```
# 1. Setup iniziale (eseguito una volta)
make setup

# 2. Preparazione file di input
# Editare annotator/inputCode/input.ts con codice TypeScript
# parzialmente annotato

# 3. Esecuzione pipeline completa
make pipeline

# 4. Analisi dei risultati
# - Verificare annotator/output.txt per annotazioni complete
# - Analizzare extractor/output.txt per codice miniSL
# - Esaminare output console del checker per validazione

# 5. Iterazioni e rifinimento
make dev-annotator  # Modifiche all'algoritmo di annotazione
make dev-extractor  # Modifiche alla logica di estrazione
make dev-checker    # Aggiornamenti alla grammatica
```

Listing 3.22: Sessione completa di utilizzo

3.6.4 Configurazione e Personalizzazione Componenti

File di Configurazione

Ogni componente del sistema utilizza file di configurazione specifici:

`annotator/config.json`
: Configurazione per la generazione delle annotazioni, inclusi pattern di riconoscimento e regole di trasformazione

`extractor/config.json`
: Configurazione per l'estrazione di miniSL, inclusi pattern di riconoscimento e regole di trasformazione

`miniSLChecker/miniSLGrammar/miniSLGrammar.g4`
: Grammatica ANTLR che definisce la sintassi formale del linguaggio miniSL

3.6.5 Comandi Disponibili

La Tabella 3.2 riporta l'elenco completo dei comandi Make disponibili nel sistema.

Tabella 3.2: Comandi Make Disponibili nel Compilatore miniSL

Comando	**Descrizione**
`make help`	Mostra l'aiuto con tutti i comandi disponibili
`make install`	Installa tutte le dipendenze dei componenti
`make build`	Compila tutti i componenti del sistema
`make clean`	Pulisce i file di build generati
`make setup`	Esegue installazione e build completo
`make pipeline`	Esegue la pipeline completa (annotate → extract → check)
`make annotate`	Esegue l'annotator con input specificato se definito, altrimenti utilizza i valori di default
`make extract`	Esegue l'extractor con input specificato se definito, altrimenti utilizza l'output dell'annotatore e i valori di default
`make check`	Esegue il checker sul file specificato se definito, altrimenti utilizza l'output dell'extractor
`make check-standalone`	Esegue il checker su file predefinito interno al componente
`make dev-annotator`	Build ed esecuzione rapida annotator
`make dev-extractor`	Build ed esecuzione rapida extractor
`make dev-checker`	Build ed esecuzione rapida checker
`make test`	Esegue test su tutti i componenti

Questo insieme di comandi fornisce un framework completo e flessibile per la sperimentazione, validazione e analisi, permettendo sia l'uso automatizzato della pipeline completa che l'impiego di componenti singoli con parametri personalizzati.

Capitolo 4

Conclusioni

All'interno della tesi abbiamo analizzato com'è possibile semplificare e automatizzare il processo di produzione di politiche di scheduling personalizzate in ambito FaaS, per potere migliorare le prestazioni di questo tipo di architetture. In particolare modo, il nostro studio si è interessato a tradurre TypeScript in miniSL, linguaggio da cui è possibile estrarre le informazioni necessarie per formulare migliori politiche di scheduling come mostrato nei paper [4] e [5]. Nel capitolo 3, abbiamo visto tutte le fasi di questo processo di traduzione, e come possa essere spezzettato in parti differenti e ben distinte che possono risultare utili anche singolarmente. Infatti, nella sezione 3.3 si mostra come tramite l'annotatore è stato possibile produrre delle annotazioni (sezione 3.2) per permettere poi nella sezione 3.4 all'estrattore di generare codice miniSL. Mostrando come le annotazioni siano fondamentali per creare uno strato di generalizzazione che crea indipendenza tra miniSL e qualsiasi linguaggio si voglia tradurre. L'estrattore sezione 3.4, grazie a questa scelta implementativa, ha un carattere generale, in tale modo è possibile riutilizzarlo in futuro. L'estrattore, a sua volta, integra al suo interno un checker (sezione 3.5) che abbiamo utilizzato per verificare i risultati ottenuti e che può essere utilizzato separatamente. La grammatica utilizzata per il checker è stata creata basandosi sulle specifiche riportate nella fig. 2.2.

4.1 Impieghi e Limitazioni

La soluzione proposta può quindi essere utilizzata, in parte o completamente, per produrre codice miniSL; semplificando il processo con cui si costruiscono le politiche di scheduling personalizzate, nelle modalità descritte dal paper [5]. Bisogna, al tempo stesso, tenere presente che, per quanto riguarda l'annotatore (sezione 3.3), non è possibile riassegnare a piacimento le variabili utilizzate dalle annotazioni all'interno del codice che si vuole annotare. Va infatti considerato che, in caso di riassegnazione condizionata, verrà sollevata un'eccezione. Inoltre, viene ugualmente sollevata un'eccezione se la variabile non è riassegnata con:

- espressioni numeriche che possono contenere solamente la variabile che si sta riassegnando oltre a operatori e numeri
- espressioni booleane che possono contenere solamente la variabile che si sta riassegnando oltre a operatori e valori booleani

Tenete inoltre presente che l'annotatore non annota tutto il codice disponibile, ma solo alcuni costrutti semplici e basilari presenti anche all'interno di miniSL. Infine, è importante ricordare che l'annotatore ha comunque bisogno dell'utente, perché non è in grado di determinare da solo le chiamate esterne fatte da una funzione, ma necessita che qualcuno le annoti in partenza.

Per quanto concerne l'estrattore (sezione 3.4), bisogna innanzitutto riportare che non gestisce i casi in cui ci siano funzioni ricorsive ma, come l'annotatore, è in grado di gestire funzioni annidate e chiamate ad altre funzioni interne, anche se con uno scope sostanzialmente "piatto". Non sono inoltre gestite funzioni di ordine superiore, in nessun caso. L'estrattore verifica però che i vincoli imposti da miniSL siano validi. Il checker (sezione 3.5) controlla che il codice miniSL prodotto dall'estrattore sia sintatticamente corretto, segnalando eventuali errori.

4.2 Possibili Sviluppi Futuri

A partire da quanto realizzato in questa tesi, si aprono diverse possibilità di estensione e miglioramento. Qui di seguito ne trovate alcune:

1. **Riconoscimento delle chiamate a servizi esterni**
   Uno dei primi sviluppi potenziali riguarda il raffinamento dell'annotatore, in particolare, nella capacità di identificare chiamate a servizi esterni. Questo non è un compito affatto banale, visti i molteplici modi con cui oggi è possibile fare una chiamata HTTP: tramite librerie diverse, pattern asincroni, wrapper, o persino DSL interni. L'obiettivo sarà quindi quello di definire un modello più astratto per intercettare queste chiamate, anche attraverso pattern generici basati sull'analisi semantica e non solo sintattica.

2. **Gestione avanzata delle variabili**
   Un altro ambito di miglioramento riguarda la gestione delle variabili. Attualmente, l'annotatore fa delle ipotesi che possono risultare limitanti rispetto alla riassegnazione delle variabili. In futuro, si potrebbe sviluppare un approccio più robusto e formale, che tenga conto della riassegnazione condizionata in funzione del flusso di controllo. Una possibile direzione è la costruzione di una tabella di verità, utile per mantenere una rappresentazione sequenziale coerente dello stato delle variabili lungo il programma.

3. **Espansione multi-linguaggio dell'annotatore**
   Infine, lo sviluppo più significativo è l'espansione dell'annotatore ad altri linguaggi ritenuti utili. Le annotazioni definite (sezione 3.2) sono infatti di carattere generale. È possibile quindi sviluppare un'annotatore più avanzato riutilizzando estrattore e checker per la traduzione in miniSL. In prima istanza, aumentare la pletora di linguaggi gestiti sarebbe già un buon risultato. Per uniformare il tutto e garantire un'interfaccia semplice tra annotatore e Tree-sitter sarà necessario:

   - modificare e ampliare l'annotatore per supportare più linguaggi, mantenendo però la stessa interfaccia verso i CST generati da Tree-sitter;

- utilizzare Tree-sitter come generatore di CST multi-linguaggio, grazie al suo ampio supporto e all'uniformità strutturale che offre;
- introdurre uno strato di astrazione intermedio, che agisca da normalizzatore tra i diversi CST prodotti da Tree-sitter.

Questo livello di astrazione avrà il compito di:

- mappare costrutti comuni come funzioni, blocchi condizionali, cicli, chiamate a funzione;
- presentare all'annotatore un'interfaccia omogenea, indipendentemente dal linguaggio sorgente;
- gestire le idiosincrasie sintattiche e semantiche dei diversi linguaggi in modo separato.

Questo approccio modulare e agnostico rispetto al linguaggio di programmazione permetterà di scalare facilmente il sistema ad altri linguaggi, semplicemente aggiungendo nuove definizioni di mapping e regole specifiche nel layer di astrazione, mantenendo stabile e coerente il nucleo dell'annotatore.

Bibliografia

[1] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Proceedings of the Static Analysis Symposium (SAS)*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 2008.

[2] Maurizio Amadori. prgTesi. https://github.com/mmaurii/prgTesi.git, 2025. Accesso: 19 giugno 2025.

[3] Max Brunsfeld. Tree-sitter. https://tree-sitter.github.io/, 2018. Accessed: 2025-05-24.

[4] Giuseppe De Palma, Saverio Giallorenzo, Cosimo Laneve, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. Serverless scheduling policies based on cost analysis. *arXiv preprint arXiv:2310.20391*, 2023.

[5] Giuseppe De Palma, Saverio Giallorenzo, Cosimo Laneve, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. Leveraging static analysis for cost-aware serverless scheduling policies. *International Journal on Software Tools for Technology Transfer*, 26(6):781–796, 2024.

[6] Marijn Haverbeke and contributors. Acorn - a small, fast javascript parser written in javascript. https://www.npmjs.com/package/acorn, 2012. MIT License.

[7] Antonio Flores Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In *Proceedings of the 12th Asian Symposium on Programming Languages and Systems (APLAS)*, volume 8858 of *Lecture Notes in Computer Science*, pages 275–295. Springer, 2014.

[8] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[9] TypeScript Team. TypeScript: Javascript with syntax for types. https://www.typescriptlang.org/, 2012. Accessed: 2025-06-24.

Ringraziamenti

Desidero innanzitutto ringraziare il Prof. Saverio Giallorenzo e il Dr. Matteo Trentin. per il prezioso supporto e l'aiuto nello sviluppo di questo progetto. La loro competenza e professionalità mi hanno sempre fatto sentire seguito e sostenuto, pur lasciandomi lo spazio necessario per portare avanti questa tesi con serenità e autonomia.

Ringrazio anche tutto il corpo docente che mi ha guidato nel corso di questi anni universitari, fornendomi gli strumenti e le conoscenze necessari per arrivare fino a questo traguardo.

Un ringraziamento speciale va alla mia famiglia, che mi ha sempre incoraggiato, lasciandomi libero di scegliere il mio percorso, responsabilizzandomi e permettendomi di affrontare questi anni di studi con tranquillità e fiducia.

Grazie di cuore anche ai miei amici e alla mia ragazza, che hanno saputo comprendere i miei momenti di assenza dovuti agli impegni universitari e che sono stati sempre pronti a sostenermi quando ne ho avuto bisogno.

Grazie a tutti voi per avermi accompagnato in questo percorso e per essermi stati vicini.