Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Un prototipo per lo scheduling di funzioni basato su analisi di costo in piattaforme serverless

Relatore:
Chiar.mo Prof.
Cosimo Laneve

Correlatore:
Dott.
Matteo Trentin

Presentata da:
Simone Boldrini

Sessione III
Anno Accademico 2022-23

Abstract

La presente tesi si concentra sull'analisi delle funzioni scritte in un linguaggio specifico per la generazione di equazioni di costo essenziali per ottimizzare l'esecuzione delle funzioni serverless. Questo obiettivo viene perseguito attraverso un percorso di ricerca articolato in diverse fasi. In primo luogo, si definisce una grammatica specifica per il linguaggio HLCostLan, fornendo le basi per l'analisi semantica delle funzioni. Successivamente, si sviluppa un interprete in grado di analizzare programmi scritti in HLCostLan e di restituire le equazioni di costo associate a ciascuna funzione. Inoltre, si procede con la generazione del corrispondente codice WebAssembly, che costituisce l'ambiente di esecuzione per le funzioni analizzate.
Il WebAssembly svolge un ruolo cruciale nell'implementazione delle soluzioni proposte, garantendo portabilità e interoperabilità tra diversi ambienti di runtime. Una volta ottenute le equazioni di costo, si utilizzano strumenti specifici come PUBS (Practical Upper Bounds Solver) e CoFloCo (Cost Flow Complexity Analysis) per condurre un'analisi dettagliata delle prestazioni delle funzioni.
Il nostro progetto verrà poi integrato con un codice dichiarativo cAPP che ci permette di definire politiche di schedulazione in base ai costi associati alle funzioni, al fine di ottimizzare l'allocazione delle risorse e lo scheduling delle esecuzioni delle funzioni, consentendo una maggiore efficienza e scalabilità nell'ambito delle applicazioni basate su architetture serverless.

Indice

1 Introduzione **6**
- 1.1 Cloud Computing . . . 6
  - 1.1.1 Infrastructure as a Service . . . 6
  - 1.1.2 Platform as a Service . . . 7
  - 1.1.3 Software as a Service . . . 7
  - 1.1.4 Function As a Service . . . 8
- 1.2 Allocation Priority Policies . . . 9
  - 1.2.1 cAPP . . . 10
- 1.3 Obiettivo della tesi . . . 12

2 Linguaggio e semantica **14**
- 2.1 Grammatica . . . 15
  - 2.1.1 Grammatica del linguaggio . . . 15
- 2.2 Another Tool for Language Recognition . . . 16
  - 2.2.1 EBNF: Simboli e notazioni in ANTLR . . . 17
  - 2.2.2 Differenza tra Lexer e Parser in ANTLR . . . 17
- 2.3 Semantica del Linguaggio . . . 20
- 2.4 Compilatore o Interprete? . . . 20

3 CostCompiler **22**
- 3.1 Implementazione . . . 24
  - 3.1.1 Struttura del codice . . . 29
- 3.2 Regole di Inferenza . . . 31
- 3.3 Generazione delle Equazioni di costo . . . 33
- 3.4 PUBS . . . 35
  - 3.4.1 Analisi di costo . . . 35
  - 3.4.2 Relazione di Costo . . . 37
  - 3.4.3 Stima del costo per Nodo . . . 39
  - 3.4.4 PUBS in pratica . . . 40

4 Generazione di WebAssembly **44**
4.1 Introduzione WebAssembly 45
4.2 WebAssembly Text Format 45
4.2.1 WebAssembly System Interface 48
4.3 Entità di WebAssembly 48
4.4 CostCompiler to WAT 50
4.5 Esecuzione del modulo WebAssembly 53

5 Conclusioni **55**
5.1 Sviluppi futuri 56

Bibliografia **57**

Elenco delle figure

1.1 Esempio di sistema serverless basato su APP 9
1.2 From Deploy to Scheduling for Listing 1.1 e 1.2 12

2.1 Funzione di parsing e lexing 18
2.2 Esempio di albero di AST 19

3.1 Relazioni di costo del programma 3.11 35
3.2 Esempio di Analisi di Costo 36
3.3 Esempio di output PUBS su Listing 1 41
3.4 Esempio di output PUBS su Listing 6 43

4.1 Stack Machine WASM 47
4.2 Schema Entità WebAssembly 49
4.3 Debug del file output.wasm 54

Indice di Codice

1.1 La guardia della condizione è un espressione 9
1.2 La guardia della condizione è un'invocazione a un servizio esterno 10
1.3 Funzione con logica Map-Reduce 10
1.4 cAPP for Listing 1.1 e 1.2 11
1.5 cAPP for Listing 1.3 11
2.1 Grammatica del linguaggio HLCostLan 15
2.2 Esempio di codice HLCostLan: example/Listing6 19
3.1 Listing8 23
3.2 Equazioni di costo per Listing8 23
3.3 Interfaccia Node 24
3.4 Implementazione del Visitor AST 25
3.5 Inizio Controllo semantico 27
3.6 Controllo semantico di ForNode 27
3.7 Controllo di tipo di CallNode 28
3.8 Metodo isSubtype 29
3.9 Esempio Testing 30
3.10 toEquation() del ProgramNode 34
3.11 Esempio di Analisi di Costo 35
3.12 Listing 1 40
3.13 Grammatica PUBS 42
3.14 Listing 6 42
3.15 Equazione di costo PUBS per Listing6 43
4.1 Esempio di funzione in wat 46
4.2 Esempio di funzione in wat 46
4.3 codeGeneration() per l'if Node 51
4.4 codeGeneration() per il for Node 52
4.5 Esecuzione del modulo WebAssembly 53

Capitolo 1

Introduzione

Il cloud computing è un modello di distribuzione dei servizi informatici che consente di accedere a risorse e applicazioni tramite Internet, senza doverle mantenere localmente sul proprio computer o server. Attraverso il cloud computing, le risorse come l'archiviazione dei dati, la potenza di calcolo e il software vengono forniti come servizi virtuali da parte di fornitori specializzati, noti come provider di cloud(Amazon,Google ecc.).
La peculiarità sta nel fatto che le risorse vengono istanziate su richiesta, generalmente sotto forma di macchine virtuali. I provider gestiscono l'infrastruttura fisica e mettono a disposizione degli utenti le risorse necessarie in base alle loro esigenze. In questo modo, le aziende possono evitare di investire in costosi hardware e software, riducendo i costi operativi e aumentando la flessibilità.

1.1 Cloud Computing

Possiamo distinguere diverse tipologie di servizi cloud in base al tipo di risorse che vengono fornite:

1.1.1 Infrastructure as a Service

L'Infrastructure as a Service (IaaS) è un modello di cloud computing che fornisce risorse informatiche virtualizzate tramite Internet. Con l'IaaS, i provider di cloud mettono a disposizione degli utenti l'infrastruttura fisica necessaria, inclusi server virtuali, storage, reti e altre risorse, consentendo loro di creare e gestire l'ambiente informatico in modo flessibile e scalabile. Attraverso l'IaaS, gli utenti possono evitare di dover investire in hardware e

infrastrutture costose, riducendo i costi di gestione e manutenzione. I provider di cloud si occupano dell'acquisizione, dell'installazione e della gestione dell'hardware, nonché della fornitura delle risorse virtuali agli utenti. Questo modello consente alle aziende di concentrarsi sullo sviluppo delle proprie applicazioni e servizi, piuttosto che preoccuparsi dell'infrastruttura sottostante. La peculiarità sta nel fatto che le risorse vengono istanziate su richiesta o domanda di una piattaforma che ne ha bisogno.

1.1.2 Platform as a Service

Il Platform as a Service (PaaS) offre un ambiente di sviluppo e di esecuzione completo per le applicazioni. I provider di cloud mettono a disposizione degli sviluppatori un insieme di strumenti, framework e servizi che semplificano il processo di sviluppo, test e distribuzione delle applicazioni. PaaS offre un'ampia gamma di strumenti di sviluppo, come linguaggi di programmazione, framework e ambienti di sviluppo integrati (IDE), che semplificano il processo di sviluppo delle applicazioni. Gli sviluppatori possono scrivere il codice, testare e distribuire le applicazioni direttamente nell'ambiente fornito dal PaaS, senza dover configurare manualmente l'infrastruttura. Alcuni esempi di questo modelli li troviamo in Google App Engine, Microsoft Azure App Service e AWS Elastic Beanstalk.

1.1.3 Software as a Service

Il Software as a Service(SaaS) è modello di cloud computing che fornisce agli utenti applicazioni basate su cloud attraverso Internet. Con il SaaS, i provider di cloud ospitano e gestiscono l'infrastruttura e i software applicativi, consentendo agli utenti di accedere e utilizzare le applicazioni tramite Internet, senza dover installare il software sul proprio computer. Consiste nell'utilizzo di programmi installati su un server remoto, cioè fuori del computer fisico o dalla LAN locale, spesso attraverso un server web; l'utente può accedere al programma tramite un browser web, come se fosse un programma installato localmente. I modelli di pagamento del SaaS sono spesso basati sul consumo effettivo delle risorse, consentendo agli utenti di pagare solo per ciò che effettivamente utilizzano. Questo modello di pricing basato su abbonamento o utilizzo può essere vantaggioso per le aziende, in quanto consentono di evitare costi iniziali elevati e di prevedere meglio i costi operativi. Alcuni esempi più comuni di SaaS li troviamo in Google Workspace, Microsoft Office 365, Dropbox.

1.1.4 Function As a Service

Function as a Service(FaaS) è un modello di cloud computing che consente agli sviluppatori di eseguire e gestire le proprie funzioni senza dover gestire l'infrastruttura sottostante. In FaaS, gli sviluppatori suddividono le loro applicazioni in funzioni modulari e indipendenti. Ogni funzione esegue un'attività specifica e può essere attivata in risposta a eventi o richieste specifiche. Ad esempio, una funzione può essere scatenata da un evento di caricamento di un file su un sistema di archiviazione cloud, da una richiesta HTTP o da un timer programmato. Quando una funzione viene attivata, il fornitore di servizi cloud gestisce automaticamente la sua esecuzione, inclusa la gestione delle risorse necessarie. Le funzioni vengono eseguite in ambienti isolati e scalati automaticamente in base alle richieste di carico. Una volta completata l'esecuzione della funzione, le risorse vengono deallocate per massimizzare l'efficienza e minimizzare i costi. FaaS è diventato un'opzione popolare per lo sviluppo di microservizi, serverless application e scenari di elaborazione event-driven, fornendo un modo flessibile ed efficiente per eseguire singole funzioni di codice. L'utente sostanzialmente deve solo fornire la logica aziendale insieme alle sue dipendenze e il fornitore del cloud è responsabile dell'allocazione di risorse, della scalabilità e della disponibilità del servizio. Queste piattaforme vengono generalmente fatturate in base al consumo, consentendo agli sviluppatori di ridurre i costi aumentando al tempo stesso l'agilità.[1] Alcuni esempi di FaaS li troviamo in AWS Lambda, Google Cloud Functions, Azure Functions.

Serverless Computing

Serverless computing rappresenta un paradigma che va oltre il semplice Function-as-a-Service (FaaS), in cui il fornitore di servizi cloud si occupa completamente della gestione dell'infrastruttura sottostante. Il FaaS è un sottoinsieme del Serverless computing che si concentra specificamente sulla gestione delle singole funzioni come entità indipendenti. In questo contesto, le funzioni vengono eseguite in modo scalabile e senza che l'utente debba preoccuparsi della gestione diretta dell'infrastruttura sottostante.[2]
Nel serverless computing, il provider cloud si occuperà di allocare le risorse necessarie per eseguire il codice e di deallocarle una volta terminata l'esecuzione. Questo facilita la distribuzione e la scalabilità del sistema, cercando di automatizzare quest'ultima fase. Focalizzandoci in questa gestione delle risorse, viene introdotto un linguaggio cAPP che ci permetterà di sincronizzare le risorse attraverso determinate politiche.[3]

1.2 Allocation Priority Policies

Partiamo da un linguaggio dichiarativo APP(*Allocation Priority Policies*)[4] che ci permette di definire politiche di allocazione delle risorse per le funzioni serverless, derivato da un estensione dello scheduler di OpenWhisk. Vediamo qui di seguito un esempio del funzionamento:

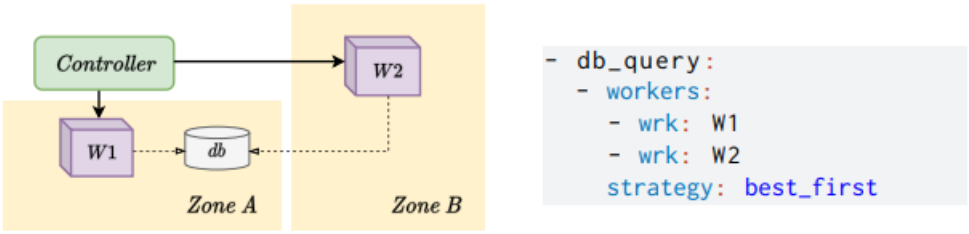

Figura 1.1: Esempio di sistema serverless basato su APP

A destra troviamo il codice dichiarativo APP che fornisce le regole per allocare/invocare le funzioni: abbiamo due **Workers** che sono disponibili, e una strategia che regola come allocare queste risorse nei rispettivi worker; inoltre una **Strategy** che determina come allocare le risorse e secondo quale priorità. Attualmente però la strategy ci permette di allocare secondo poche strategie generiche gia consolidate. L'obiettivo diventa ora automatizzare questo processo, definendo le politiche di scheduling utilizzando le informazioni derivate dall'analisi statica delle funzioni che dovranno essere eseguite, per definire le politiche di allocazione delle risorse. Osserviamo adesso un'altro esempio andando ancor di più in profondità:
Data una grammatica definita nel paper [5], osserviamo alcuni esempi di funzioni:

```
1    (isPremiumUser, par) => {
2        if (isPremiumUser) {
3            call PremiumService(par)
4        } else {
5            call BasicService(par)
6        }
7    }
```

Listing 1.1: La guardia della condizione è un espressione

Il parametro *isPremiumUser* è un valore che indica se l'utente richiede il servizio premium o meno, e in base a questo valore viene eseguita una chiamata

o l'altra. Di conseguenza assumiamo che il ramo condizionale prende il massimo della latenza tra le due chiamate, non sappiamo staticamente se verrà eseguito il ramo *then* o *else* e quindi non possiamo preventivare il tempo di esecuzione della funzione.

```
1    (username, par) => {
2        if (call isPremiumUser(username)) {
3            call PremiumService(par)
4        } else {
5            call BasicService(par)
6        }
7    }
```

Listing 1.2: La guardia della condizione è un'invocazione a un servizio esterno

Nel caso sopra riportato invece, se l'utente che scatena l'evento è un membro premium, il tempo di esecuzione previsto dalla funzione è la somma delle latenze delle invocazioni dei servizi *isPremiumUser* e *PremiumService*. Quindi possiamo preventivare il ritardo atteso(come tempo di esecuzione peggiore), cioè la somma della latenza del servizio *isPremiumUser* più il massimo tra la latenza dei servizi *PremiumService* e *BasicService*.
Osserviamo ora un esempio di funzione con logica Map-Reduce:

```
1    (jobs, m, r)=>{
2        for(i in range(0,m)){
3            call Map(jobs, i)
4            for(j in range(0,r)){
5                call Reduce(jobs, i, j)
6            }
7        }
8    }
```

Listing 1.3: Funzione con logica Map-Reduce

Il parametro jobs descrive una sequenza di lavori map-reduce, dove il numero dei jobs è indicato dal parametro m. La fase di *Map* che genera m sottotask "ridotti" è implementata da un servizio esterno *Map* che riceve jobs e un indice i che indica il job mappato. Per ogni i, ci sono j sottotask. In questo caso ci aspettiamo che la latenza dell'intera funzione è data dalla somma di m volte la latenza di *Map* e m $\times$ r volte la latenza di *Reduce*.[6]

1.2.1 cAPP

cAPP è un estensione di APP, dove le politiche di schedulazioni delle funzioni dipendono dai costi associati alle possibili esecuzioni delle funzioni sui

worker disponibili[5]. cAPP è stato modificato al fine di implementare nuovi costrutti quali *min_latency* e *max_latency* che ci permettono di definire un upper bound e un lower bound per la latenza di una funzione.

```
-premUser:
    -workers:
        -wrk: W1
        -wrk: W2
    strategy: min_latency

```

Listing 1.4: cAPP for Listing 1.1 e 1.2

```
-mapReduce:
    -workers:
        -wrk: W1
        -wrk: W2
    strategy: random
    invalidate:
        strategy: max_latency

```

Listing 1.5: cAPP for Listing 1.3

Nel Listing 1.4, osserviamo come diamo la possibilità di allocare la funzione *premUser* su due workers, e la strategia di allocazione è quella di minimizzare la latenza, ovvero di dare priorità al worker su cui la soluzione dell'espressione di costo è minima. Illustriamo però meglio le fasi della tecnica di *min_latency*: Quando viene creato lo script cAPP, viene creata l'associazione tra il codice delle funzioni e il loro script etichettando le funzioni con *//tag:premUser*. Successivamente abbiamo bisogno del calcolo delle funzioni di costo, il codice delle funzioni viene utilizzato per dedurre i programmi di costo corrispondenti. Quando le funzioni vengono invocate, possiamo calcolare la soluzione del programma di costo data la conoscenza dei parametri di invocazione. Quando riceviamo una richiesta ad esempio per il Listing 1.1 prendiamo il suo programma di costo (rappresentato dal punto di intersezione sinistra) e la corrispondente politica cAPP per implementare la politica di schedulazione prevista. Questa politica può essere ottenuta in due fasi: Calcolando i programmi di costo dal Solver, eseguendoli in ogni worker, e scegliendo il worker che ha latenza minima per contattare il *PremiumService*. Invece nel caso del Listing 1.5, abbiamo una strategia di invalidazione *max_latency*, dopo aver selezionato un worker con una determinata strategia, andiamo a verificare se il worker è in grado di eseguire la funzione andando a risolvere l'espressione di costo corrispondente sostituendo i parametri m e r con la latenza dei servizi *Map* e *Reduce* dal worker selezionato e verifichiamo che la latenza sia inferiore a quella definita nello script.[5]

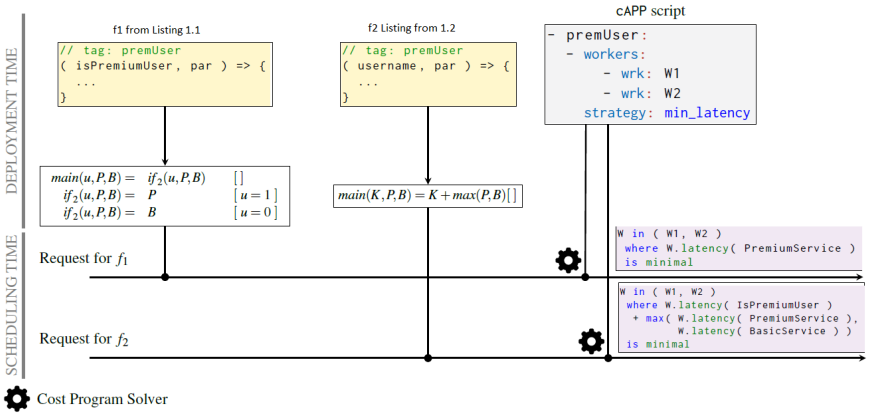

Figura 1.2: From Deploy to Scheduling for Listing 1.1 e 1.2

1.3 Obiettivo della tesi

L'obiettivo principale della tesi è approfondire le tecniche utilizzate per analizzare la semantica delle funzioni scritte in un linguaggio specifico, il quale sarà dettagliatamente delineato nei capitoli successivi. Tale analisi semantica è cruciale per generare le equazioni di costo associate a ciascuna funzione. Queste equazioni di costo forniscono dati fondamentali, come la complessità, che rappresenta un limite superiore delle prestazioni della funzione stessa. Queste informazioni sono essenziali per ottimizzare l'esecuzione delle funzioni serverless una volta richieste, consentendo un'allocazione efficiente delle risorse e una riduzione dei tempi di esecuzione.

Il percorso di ricerca si articola in diverse fasi. Innanzitutto, sarà definita una grammatica specifica per il linguaggio di programmazione in questione. Successivamente, verrà sviluppato un interprete per tale linguaggio, in grado di analizzare un programma scritto in un linguaggio ad alto livello e di restituire le relative equazioni di costo per ciascuna funzione. Inoltre, verrà generato il corrispondente codice WebAssembly, pronto per essere eseguito in un ambiente serverless. WebAssembly svolge un ruolo cruciale come ambiente di esecuzione per il codice generato dalle nostre analisi di costo. Utilizzando il WebAssembly come target per le nostre ottimizzazioni, possiamo garantire una maggiore portabilità e interoperabilità delle nostre soluzioni, consentendo l'esecuzione efficiente delle funzioni serverless in una varietà di ambienti di runtime. Una volta ottenute le equazioni di costo, ci concentreremo sull'analisi dettagliata di ciascuna funzione, impiegando strumenti specifici come PUBS (Pratical Upper Bounds Solver) e CoFloCo (Cost Flow Complexity

Analysis). Tali strumenti, menzionati rispettivamente nella letteratura [7] e [8], sono progettati per calcolare il limite superiore delle prestazioni delle funzioni, date le rispettive equazioni di costo. Quest'analisi approfondita ci fornirà una comprensione più dettagliata delle prestazioni delle funzioni, permettendoci di ottimizzare ulteriormente l'allocazione delle risorse e il processo di scheduling delle esecuzioni.

Capitolo 2

Linguaggio e semantica

Un linguaggio di programmazione è un linguaggio formale, ovvero un insieme di simboli e regole che definiscono la struttura e il significato delle istruzioni che compongono un programma. In questo capitolo definisco il linguaggio che andremo ad utilizzare per descrivere i programmi e la semantica che gli daremo. Il linguaggio è stato pensato per essere comprensibile e allo stesso tempo espressivo, in modo da poter descrivere programmi complessi. La semantica è stata pensata per essere semplice da implementare e allo stesso tempo potente, in modo da poter descrivere programmi complessi.
Un linguaggio viene definito da una grammatica $G = (N, T, \rightarrow, S)$ dove:

- N è l'insieme dei **non terminali**
- T è l'insieme dei **terminali**
- $\rightarrow$ è l'insieme delle **produzioni**
- S è il **simbolo iniziale**

Mentre il linguaggio generato da una grammatica G è definito come:

$$\mathcal{L}(G) = \{\gamma | \gamma \in T^* \wedge \Rightarrow^+ w\} \tag{2.1}$$

e T^* è la chiusura di Kleene.
La grammatica (che descriveremo nel dettaglio nel capitolo successivo), è riconosciuta dal plugin ANTLR che ci permette di generare un parser per il linguaggio. Il parser generato da ANTLR è un parser LL(*), ovvero un parser che riconosce linguaggi non ambigui e che non richiede backtracking, utilizzando un numero arbitrario di lookahead symbols, questo ci permette di avere un parser efficiente ottimizzando il riconoscimento del linguaggio.

2.1 Grammatica

2.1.1 Grammatica del linguaggio

Di seguito espongo la grammatica del linguaggio **HLCostLan** che descrive il linguaggio che utilizzerò per descrivere i programmi.
Riporto di seguito le produzioni della grammatica che descrivono il linguaggio, mentre per visualizzare il file g4 nella sua totalità si rimanda alla Repository Git del progetto.

```
grammar HLCostLan;
prg : complexType* serviceDecl* functionDecl* init;

init: '('formalParams? ')''=''>' '{' stm '}';

serviceDecl:
    'service'ID':''('(type(','type)*)?')''→''type';

functionDecl:
    'fn'ID'('formalParams?')''→'(type)'{'stm'}' ;

stm :
    | serviceCall
    |'if''('expOrCall')''{'stm'}''else''{'stm'}'
    |'for''('ID'in''('NUMBER','exp ')'')''{'stm'}'
    | letIn
    | functionCall
    | 'return' expPlus ;

serviceCall:'call'ID'('(exp(','exp)*)?')'(';'stm)?;

functionCall : ID'('( exp (','exp)* )? ')';

letIn: 'let' (ID '=' expPlus)+ 'in' stm;
```

Listing 2.1: Grammatica del linguaggio HLCostLan

Il non terminale **prg** è il terminale iniziale della grammatica, e descrive un programma. Un programma è composto da una sequenza di dichiarazioni di tipi complessi, dichiarazioni di servizi(che possono avere un overhead in termini di invocazioni che influiscono sul costo), dichiarazioni di funzioni, e infine l'init. L'init è la funzione che viene invocata all'avvio del programma.

Il non terminale **init** descrive la funzione init, che deve essere dichiarata una sola volta ed è composta da una sequenza di parametri formali, e da una istruzione.
La **serviceDecl** descrive la dichiarazione di un servizio, che deve essere dichiarato una sola volta. Un servizio è composto da un nome, una sequenza di parametri formali, e un tipo di ritorno.
La **functionDecl** descrive la dichiarazione di una funzione, la quale è composta da un nome (univoco), una sequenza di parametri formali, e un tipo di ritorno. Nel checking semantico controlliamo che non vengano dichiarate due volte funzioni o servizi con lo stesso nome.
Inoltre il Type checking controlla che i tipi di ritorno delle funzioni e dei servizi siano coerenti con i tipi di ritorno delle chiamate.

2.2 Another Tool for Language Recognition

ANTLR (Another Tool for Language Recognition) è uno strumento potente e flessibile per l'analisi dei linguaggi di programmazione, linguaggi di markup e dati strutturati. È ampiamente utilizzato per generare parser e lexer per vari linguaggi di programmazione e per costruire compilatori, interpreti, traduttori di linguaggi e altre applicazioni che necessitano di analisi di linguaggi. Ecco alcuni usi comuni di ANTLR:

1. Generazione di parser e lexer: ANTLR può generare parser e lexer per molti linguaggi di programmazione, rendendo più semplice l'analisi sintattica e lessicale.

2. Costruzione di compilatori e interpreti: ANTLR è spesso utilizzato nella costruzione di compilatori e interpreti, poiché fornisce gli strumenti per analizzare il codice sorgente e costruire l'albero di sintassi astratta.

3. Traduzione di linguaggi: ANTLR può essere utilizzato per tradurre codice da un linguaggio di programmazione ad un altro. Questo è utile per la migrazione del codice, il refactoring e altre attività di manutenzione del software.

4. Analisi di dati strutturati: ANTLR può essere utilizzato per analizzare dati strutturati, come file XML o JSON, rendendo più semplice l'estrazione e la manipolazione dei dati.

ANTLR data una grammatica in input, con estensione *.g4*, genera una cartella *gen* contenente i file necessari per generare il parser.

2.2.1 EBNF: Simboli e notazioni in ANTLR

I simboli utilizzati da ANTLR, rispettano la notazione EBNF(Extended Backus-Naur Form), che è una notazione formale per descrivere la sintassi di un linguaggio, definita come standard internazionale da ISO-14977.

- () - Le parentesi vengono utizzate per raggruppare elementi diversi, e per definire l'ordine di valutazione.
- **?** - i Token seguiti da un punto interrogativo sono opzionali, e possono presentarsi 0 o 1 volta.
- * - Chiusura di Kleene, i token seguiti da un asterisco possono presentarsi 0 o più volte.
- + - i token seguiti da un segno più possono presentarsi 1 o più volte.
- | - il simbolo pipe viene utilizzato per definire una scelta, ovvero un token o un'altro.
- **.** - Il punto definisce che un token può apparire una singola volta.
- $\sim$ - Il simbolo not definisce che un simbolo non può apparire.
- **..** - Il simbolo Range definisce un intervallo di token.

L'uso di lettere maiuscole e minuscole è importante, in quanto ANTLR distingue tra token maiuscoli e minuscoli.[9]

2.2.2 Differenza tra Lexer e Parser in ANTLR

I linguaggi di programmazione sono costituiti da parole chiavi(keywords) e costrutti definiti in modo preciso. Un file sorgente viene inviato come flusso ad un lexer, carattere per carattere da una qualche interfaccia di input. Il lexer si occuppa di raggruppare i caratteri in token, ovvero sequenze di caratteri che rappresentano parole chiavi, identificatori, numeri, stringhe, e altri costrutti del linguaggio. Analogamente al lexer, il parser si occupa di riconoscere la grammatica del linguaggio, dandogli un qualche significato semantico. La cartella *gen* contiene il codice sorgente del parser e lexel, generati automaticamente; è presente anche un file *.tokens* che contiene i token riconosciuti dal lexer, e un file *.interp* che contiene la tabella di interpretazione del parser.

Il lexer si occupa di riconoscere i token, in ANTLR viene generato un lexer DFA, ovvero un lexer che riconosce linguaggi non ambigui e che non richiede backtracking. Questo ci permette di avere un lexer efficiente ottimizzando il riconoscimento del linguaggio. Il parser, invece, data una sequenza di token, riconosce la grammatica, e genera un albero di parsing(noto anche come albero di derivazione), che rappresenta la derivazione secondo le regole della grammatica libera da contesto. Nell'albero di parsing ogni nodo interno corrisponde a una regola della grammatica e ogni foglia corrisponde a un token del linguaggio.[10]

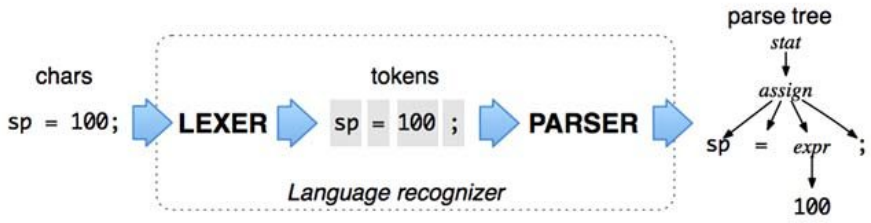

Figura 2.1: Funzione di parsing e lexing

Una volta ottenuto l'albero di parsing possiamo andare a visitarlo, e generare un albero di sintassi astratta, che rappresenta il significato del programma. L'albero di sintassi astratta(AST) è un albero che rappresenta il significato del programma, e viene utilizzato per eseguire il programma. A questo punto si estende l'interfaccia *BaseListener* per costruire un albero di sintassi astratta per implementare i metodi necessari per visitare l'albero di parsing e generare l'albero di sintassi astratta.[11]

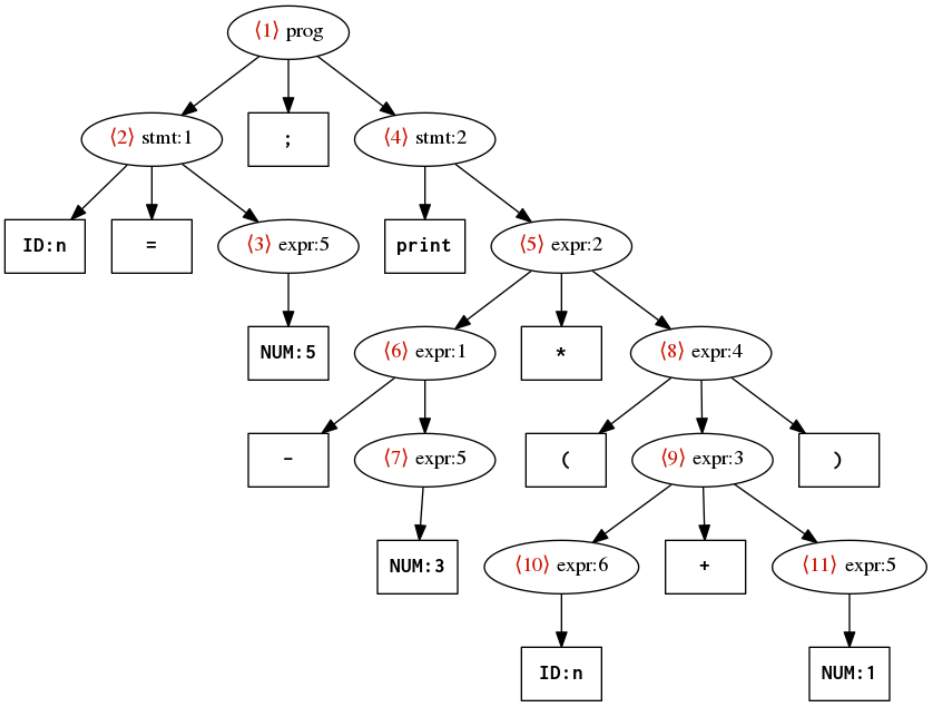

Figura 2.2: Esempio di albero di AST

Facciamo un esempio di un frammento di codice realmente generato da ANTLR, per la grammatica *HLCostLan*:

```
service BasicService: (int) -> void;
fn svc(i: int) -> void{
    for(m in (0,10)){
        call BasicService(i)
    }

}
(len : int) => {
  svc(len)
}
```

Listing 2.2: Esempio di codice HLCostLan: example/Listing6

Come abbiamo visto nella grammatica 2.1, il non terminale **prg** e' il terminale iniziale della grammatica. In questo caso prg, prende un parametro len di tipo intero, ed avrà un CallNode ad una funzione *svc* che prende un parametro di tipo intero, e ritorna void.
La funzione *svc* invece, contiene un for, che itera da 0 a 10, e ad ogni itera-

zione effettuata una chiamata al servizio *BasicService* con parametro *i*.

2.3 Semantica del Linguaggio

La semantica di un linguaggio di programmazione è l'insieme delle regole che definiscono il significato delle istruzioni, delle espressioni e delle strutture di controllo del flusso nel linguaggio. In altre parole, la semantica definisce "cosa fa" un programma scritto in quel linguaggio.

Ad esempio, se definiamo un linguaggio che permette l'assegnamento(non è il nostro caso perché la grammatica HLCostLan non lo permette) dovremmo andare a controllare che la variabile assegnata sia stata dichiarata altrimenti dovrà generare un errore.
I controlli semantici del compilatore in oggetto sono di altra natura, quali:

- Controllare che non vengano dichiarate due volte funzioni o servizi con lo stesso nome.
- Controllare che i parametri utilizzati nelle chiamate di funzioni o servizi siano corretti e quindi già stati precedentemente dichiarati
- Controllare che i tipi di ritorno delle funzioni e dei servizi siano corretti.

Inoltre effettuiamo il type checking, che è un sottoinsieme del controllo semantico, che controlla che i tipi delle espressioni siano corretti.

- I tipi delle chiamate devono essere corretti, ovvero i parametri attuali devono essere dello stesso tipo dei parametri formali.
- In un costrutto *let* le espressioni devono essere dello stesso tipo della variabile a cui vengono assegnate.
- In un costrutto *if* l'espressione deve essere di tipo booleano, mentre i valori di ritorno degli statemente *then* e *else* devono essere dello stesso tipo.
- La funzione deve tornare il tipo dichiarato.

2.4 Compilatore o Interprete?

Nel processo di traduzione del codice di un determinato linguaggio in istruzioni eseguibili, emergono due approcci predominanti: il compilatore e l'in-

terprete. Mentre entrambi condividono l'obiettivo di rendere il codice sorgente eseguibile, le loro metodologie differiscono significativamente, portando a implicazioni distinte per le prestazioni, la portabilità e lo sviluppo delle applicazioni software.
Il compilatore adotta un approccio "one-shot", traducendo l'intero codice sorgente in un'unica fase e producendo un file eseguibile indipendente dal codice originale. Questo file può essere eseguito più volte senza la necessità di ulteriori traduzioni, garantendo una maggiore efficienza nell'esecuzione del programma. Tuttavia, il processo di compilazione può richiedere più tempo e risorse iniziali rispetto all'approccio interpretato, poiché l'intero codice deve essere tradotto prima dell'esecuzione.
D'altra parte, l'interprete adotta un approccio "line-by-line", traducendo e eseguendo istruzioni del codice sorgente una alla volta. Questo significa che il codice viene tradotto e eseguito in tempo reale durante l'esecuzione del programma. Sebbene questo approccio possa ridurre il tempo di avvio e la memoria richiesta per l'esecuzione, può comportare una minore efficienza durante l'esecuzione effettiva, poiché le istruzioni devono essere interpretate ad ogni esecuzione.
Nel nostro progetto, grazie anche al lexer e il parser di Antlr, abbiamo scelto di adottare un approccio ibrido, che combina le caratteristiche di entrambi i metodi. In particolare, utilizziamo un approccio "one-shot" per tradurre il codice per le cost equation, ma generiamo anche un codice intermedio come WAT, come vedremo nel paragrafo 4.2. Questo approccio ci consente di ottenere i vantaggi di entrambi i metodi, garantendo al contempo un'efficienza e una flessibilità ottimali durante il processo di traduzione e esecuzione del codice, considerando anche i due file di output a cui si deve far fronte. [12]

Capitolo 3

CostCompiler

CostCompiler è un interprete per il linguaggio di programmazione definito nel capitolo precedente Grammatica 2.1. Una volta ricevuto il programma, CostCompiler procede alla verifica e correttezza sintattica e semantica del programma, successivamente si occupa della generazione dell'albero di sintassi astratta. Questo albero rappresenta una versione astratta del programma, che astrae i dettagli sintattici del codice e si concentra sulla sua struttura logica, associando ad ogni costrutto (eg.*if-then-else* un unico nodo, *IfNode* presente nel omonimo file in src/ast) i rispettivi sottonodi(nel caso di *IfNode* conterrà la guardia condizionale e i due statement). Dopo aver generato l'albero di sintassi astratta, CostCompiler si occupa della verifica Semantica 2.3 del linguaggio andando ad effettuare i controlli semantici e di tipo sul programma in input, andando a garantire alcune invarianti (eg. le chiamate di funzioni devono rispettare i tipi di ritorno).
Una volta effettuata la verifica semantica, CostCompiler procede con la generazione delle equazioni di costo, andando a visitare l'AST secondo determinati criteri, ad ogni nodo figlio verrà passata una Mappa che contiene la mappatura di ogni variabile in una stringa che sarà la stessa stringa che compare nelle equazioni di costo.
Ogni nodo figlio invocato attraverso la funzione *toEquation()* ritorna una stringa, rappresentante l'equazione di costo del nodo figlio, e il padre va a concatenare le stringhe dei figli(anche in base al tipo di figlio da cui ricevere l'equazione), ad esempio la *return* $<EXP>$ sarà diverso dal *return* $<\text{function}(Par)>$.
Il risultato finale di questo processo di concatenazione attraverso determinati nodi dell'AST è la generazione delle equazioni di costo. Una volta generate le equazioni di costo, CostCompiler le stampa in un file *equation.ces*, inoltre

lancia il risolutore PUBS 3.4 che va a calcolare gli upper bound del programma da stampare a video. Riportiamo un esempio di equazione di costo generata da CostCompiler dato un programma scritto in HLCostLang:

```
struct Params {
    address: array[int],
    payload: any,
    sender: string
}
service PremiumService : (string) -> void;
service BasicService : (any) -> void;
(isPremiumUser: bool, par: any) => {
    if ( isPremiumUser ) {
        call PremiumService("test");
    } else {
        call BasicService( par);
    }
}
```

Listing 3.1: Listing8

Una volta preso in input Listing8, CostCompiler genera le seguenti equazioni di costo:

```
eq(main(P,ISPREMIUMUSER0,B),0,[if9(ISPREMIUMUSER0,P,B)],[]).
eq(if9(ISPREMIUMUSER0,P,B),nat(P),[],[ISPREMIUMUSER0=1]).
eq(if9(ISPREMIUMUSER0,P,B),nat(B),[],[ISPREMIUMUSER0=0]).
```

Listing 3.2: Equazioni di costo per Listing8

Andando a descriverle ci troveremo ad avere una equazione per la regola *init*, dove vediamo che *main* viene chiamata con costo 0 e verrà chiamata *if9* con parametri *ISPREMIUMUSER0,P,B*.
P e B sono il costo costante delle chiamate ai servizi *isPremiumUser* e *BasicService*, mentre *ISPREMIUMUSER0* sarà la valutazione del parametro *isPremiumUser* che sarà 1 se vero, 0 altrimenti; in altri termini *ISPREMIUMUSER0* sarà la valutazione della guardia del costrutto *if-then-else* e verrà eseguita la chiamata al servizio *PremiumService* se *ISPREMIUMUSER0* sarà 1 con costo $nat(P)$, altrimenti verrà eseguita la chiamata al servizio *BasicService* con costo $nat(B)$. Una volta avere generato l'equazioni di costo dal programma, lo stampiamo in un file *equation.ces*, cosi da poter eseguire PUBS(A Practical Upper Bounds Solver), per determinarci l'Upper Bound del programma. L'obiettivo di PUBS(come vedremo in seguito 3.4) è quello di ottenere automaticamente upper bound in forma chiusa per i sistemi di equazioni di costo, di conseguenza calcola i limiti superiori per la relazione di costo indicata come "Entry", oltre che per tutte le altre relazioni di costo di cui tale "Entry" dipende.

3.1 Implementazione

Il codice è strutturato in modo gerarchico, partendo dall'interfaccia *Node*, che viene estesa dai nodi che compongono l'AST e che contiene la firma dei metodi che ogni nodo deve implementare.

```
1 package ast;
2
3 public interface Node {
4     EnvVar checkVarEQ(EnvVar e);
5     Node typeCheck(Environment e) throws TypeErrorException;
6     ArrayList<String> checkSemantics(Environment env);
7     String toEquation(EnvVar e);
8     String codeGeneration(HashMap<Node, Integer> offset_idx);
9 }
```

Listing 3.3: Interfaccia Node

Andando a descrivere i metodi dell'interfaccia Node:

- *checkVarEQ(EnvVar e)*: Questo metodo prende in input un oggetto di tipo *EnvVar* che è un HashMap che mappa un oggetto di tipo *Node* con una stringa, che rappresenta la variabile che mappa quel determinato nodo. Questo metodo ritorna un oggetto di tipo *EnvVar* che rappresenta l'ambiente aggiornato con le variabili mappate con le stringhe corrispondenti.
- *typeCheck(Environment e)*: Questo metodo prende in input un oggetto di tipo *Environment* che rappresenta l'ambiente in cui si trova il nodo, e ritorna un oggetto di tipo *Node* che rappresenta il nodo tipato, dopo aver effettuato il controllo di tipo.
- *checkSemantics(Environment env)*: Questo metodo prende in input un oggetto di tipo *Environment* e rappresenta il controllo semantico del nodo, ritorna un oggetto di tipo *ArrayList<String>* che rappresenta una lista di stringhe che rappresentano eventuali errori semantici.
- *toEquation(EnvVar e)*: Questo metodo prende in input un oggetto di tipo *EnvVar*, che mappa determinati nodi secondo quell'oggetto, e rappresenta la generazione delle equazioni di costo del nodo, ritorna un oggetto di tipo *String* che rappresenta l'equazione di costo del nodo.
- *codeGeneration()*: Prende in input un oggetto di tipo *HashMap<Node, Integer>* che mappa un nodo con un intero, utilizzato per le strutture dati complesse. *offset_idx* contiene l'indice delle strutture dati com-

plesse rispetto alla memoria lineare. Inoltre il metodo *codeGeneration* ritorna un oggetto di tipo *String* che rappresenta il codice generato del nodo.

Ogni nodo dell'AST estende l'interfaccia *Node* e implementa i metodi definiti in essa. Dopo aver definito l'interfaccia *Node*, andiamo a definire le fasi di sviluppo del nostro AST. In prima fase grazie al plugin ANTLR, dopo aver definito il file *HLCostLang.g4* che rappresenta la grammatica del nostro linguaggio, ANTLR ci genera i file *HLCostLangLexer.java* e *HLCostLangParser.java* che rappresentano rispettivamente il Lexer e il Parser del nostro linguaggio. L'interfaccia *HLCostLangVisitor.java* e *HLCostLanListener.java* vengono generate da ANTLR e rappresentano rispettivamente le interfacce per la generazione di un albero di parsing. Tutti i file generati automaticamente da ANTLR li troviamo nella cartella *src/gen*. All'interno di questa cartella troviamo anche il file *HLCostLangBaseVisitor.java* che rappresenta la classe base per la generazione dell'AST, che andiamo a implementare per generare il nostro albero di parsing nel file *HLCostLangBaseVisitorImpl.java*.

```
1  public class HLCostLanBaseVisitorImpl extends
       HLCostLanBaseVisitor<Node> {
2  @Override
3  public Node visitPrg(PrgContext ctx) {
4      ArrayList<Node> complexType = new ArrayList<>();
5      ArrayList<Node> decServices = new ArrayList<>();
6      ArrayList<Node> funDec = new ArrayList<>();
7
8      for (ServiceDeclContext decService : ctx.serviceDecl()){
9          decServices.add(visitServiceDecl(decService));
10     }
11
12     for (FunctionDeclContext fund : ctx.functionDecl()) {
13         funDec.add(visitFunctionDecl(fund));
14     }
15
16     for(ComplexTypeContext complexTypeContext : ctx.
       complexType()){
17         complexType.add(visitComplexType(complexTypeContext))
       ;
18     }
19     return new ProgramNode(complexType, decServices, funDec,
       visitInit(ctx.init()));
20 }
21
22     ...
```

Listing 3.4: Implementazione del Visitor AST

Qui vediamo l'implementazione del metodo *visitPrg* che rappresenta la creazione del nodo *ProgramNode* che rappresenta il nodo principale del nostro AST.
Come vediamo il metodo *visitPrg* prende in input un oggetto di tipo *PrgContext* che rappresenta il contesto riconosciuto rispetto alla grammatica di riferimento, e ritorna un oggetto di tipo *Node* che rappresenta il nodo principale del nostro AST.
Come vediamo *complexType, decServices, funDec* sono rispettivamente le liste di nodi che rappresentano i tipi complessi, le dichiarazioni dei servizi e le dichiarazioni delle funzioni, sono definiti come Arraylist<Node>, perchè andremo ad iterare sui contesti specificati per fare in modo che ogni contesto venga visitato e ritorni un oggetto di tipo *Node* che rappresenta il nodo del nostro AST.
Una volta visitati tutti i contesti, andiamo a creare il nodo *ProgramNode* che rappresenta il nodo principale del nostro AST, che conterrà la lista dei nodi dei tipi complessi, delle dichiarazioni dei servizi e delle dichiarazioni delle funzioni, e la chiamata iniziale *init*.
Una volta generato il nodo principale del nostro AST, andiamo a visitare tutti i nodi figli ricorsivamente, andando a creare i rispettivi nodi del nostro AST, che rappresentano il programma in input.

Controllo Semantico

Il controllo semantico è un processo che verifica la correttezza semantica del programma, ovvero verifica che il programma sia corretto rispetto alle regole del linguaggio.
Il controllo semantico viene effettuato attraverso il metodo *checkSemantics(Environment env)* che prende in input un oggetto di tipo *Environment* e ritorna un oggetto di tipo *ArrayList<String>* che rappresenta una lista di stringhe che rappresentano eventuali errori semantici.
Il controllo semantico viene effettuato in modo ricorsivo, partendo dal nodo principale del nostro AST, andando a visitare tutti i nodi figli, e andando a verificare che il programma sia corretto rispetto alle regole del linguaggio.
Il controllo semantico viene specificato nel *main.java* presente all'interno della cartella *src/com/company* e rappresenta il punto di partenza del nostro programma.

```
    HLCostLanBaseVisitorImpl visitor =
                        new HLCostLanBaseVisitorImpl();
    Node ast = visitor.visit(parser.prg());
    Environment env = new Environment();
    ArrayList<String> errorSemantics = ast.checkSemantics(env
   );
    if (!errorSemantics.isEmpty()) {
        System.err.println(errorSemantics);
        return Results.SEMANTIC_ERROR;
    }else{
        //continua l'esecuzione del programma
    }
```

Listing 3.5: Inizio Controllo semantico

Come vediamo, viene istanziato un visitor di tipo *HLCostLanBaseVisitorImpl* che rappresenta il visitor del nostro AST, e viene visitato il contesto *prg()* che rappresenta il contesto principale del nostro programma.
Una volta visitato il contesto, andiamo a creare l'ambiente *env* che rappresenta l'ambiente in cui si trova il nostro programma, andiamo a chiamare il metodo *checkSemantics(Environment env)* che rappresenta il controllo semantico del nostro programma, e ritorna una lista di stringhe che rappresentano eventuali errori semantici. Per controllare che il programma sia corretto rispetto alle regole del linguaggio basta controllare che la lista ritornata sia vuota, in caso contrario stampiamo gli errori semantici e terminiamo l'esecuzione del programma.

```
@Override
public ArrayList<String> checkSemantics(Environment env) {
    ArrayList<String> errors = new ArrayList<>();
    if(!env.checkHeadDeclaration(id))
        env.addDeclaration(id,new IntType());
    else
        errors.add("Error: Variable "+id+" already declared")
   ;

    errors.addAll(exp.checkSemantics(env));
    errors.addAll(stm.checkSemantics(env));
    return errors;
}
```

Listing 3.6: Controllo semantico di ForNode

Come vediamo, il metodo *checkSemantics(Environment env)* prende in input un oggetto di tipo *Environment* e ritorna un oggetto di tipo *ArrayList<String>* che rappresenta una lista di stringhe che rappresentano eventuali errori se-

mantici. Il metodo *checkSemantics(Environment env)* viene implementato in modo ricorsivo, partendo dal nodo principale del nostro AST, andando a visitare tutti i nodi figli, e andando a verificare che il programma sia corretto rispetto alle regole del linguaggio. Come vediamo nel *ForNode* andiamo a verificare che la variabile *id* sia dichiarata all'interno dell'ambiente *env*, e in caso contrario la aggiungiamo all'ambiente, altrimenti andiamo a stampare un errore semantico, e successivamente andiamo a visitare i nodi figli *exp* e *stm* e andiamo a concatenare gli errori semantici ritornati dai nodi figli.

Controllo dei tipi

Il controllo dei tipi è un processo che verifica che il programma sia corretto rispetto ai tipi del linguaggio, ovvero verifica che le espressioni siano corrette rispetto ai tipi del linguaggio. Il controllo dei tipi viene effettuato attraverso il metodo *typeCheck(Environment e)* che prende in input un oggetto di tipo *Environment* e ritorna un oggetto di tipo *Node* che rappresenta il nodo tipato, dopo aver effettuato il controllo di tipo, inoltre il metodo *typeCheck(Environment e)* può lanciare un'eccezione di tipo *TypeErrorException* in caso di errore di tipo, che in caso viene catturata dal main e stampata a video.
Osserviamo un esempio di controllo di tipo:

```
1  public Node typeCheck(Environment e) throws
       TypeErrorException {
2  try {
3      FunDeclarationNode fun = (FunDeclarationNode) e.
       getDeclaration(id.getId());
4      FormalParams fp = fun.getFormalParams();
5      if (fp.size() != listCount.size()) {
6          throw new TypeErrorException("Wrong number of
       parameters in call " + id.getId());
7      }
8      for (int i = 0; i < listCount.size(); i++) {
9          if (!Utils.isSubtype(fp.get(i).b.typeCheck(e),
       listCount.get(i).typeCheck(e))) {
10             throw new TypeErrorException("Incompatible type
       in call node");
11         }
12     }
13     return fun.getReturnNode().typeCheck(e);
14 }catch (ClassCastException ex){
15     throw new TypeErrorException("Error in call node");
16 }
17 }
```

Listing 3.7: Controllo di tipo di CallNode

Come vediamo, il metodo *typeCheck(Environment e)* prende in input un oggetto di tipo *Environment*, da li controlla che la funzione chiamata (sappiamo che è gia definita perché prima effettuiamo il controllo semantico) abbia lo stesso numero di parametri che stiamo passando, e che i tipi dei parametri siano compatibili con i tipi dei parametri della funzione chiamata. Viene inoltre, ritornato il tipo di ritorno della funzione chiamata.
Il metodo *Utils.isSubtype(fp.get(i).b.typeCheck(e), listCount.get(i).typeCheck(e))* è un metodo di utilità che controlla se il tipo del parametro passato è sottotipo del tipo del parametro della funzione chiamata.
Qui di seguito invece osserviamo la dichiarazione del metodo *isSubtype*:

```
1     public static boolean isSubtype(Node a, Node b) {
2         return a.getClass().isAssignableFrom(b.getClass());
3     }
```

Listing 3.8: Metodo isSubtype

Ritorna true se il tipo di a è sottotipo del tipo di b, altrimenti false.
Questo è possibile grazie al fatto che ogni nodo di tipo, viene esteso da uno specifico nodo supertipo. In questo caso abbiamo deciso che il tipo *AnyType* estende il tipo *IntType*, e tutti gli altri sottonodi che implementano l'interfaccia *NodeType*, che rappresenta il tipo di un nodo.

In seguito verrà specificato come vengono generati le equazioni di costo3.3 attraverso il metodo *toEquation(EnvVar e)* e come viene generato il WASM corrispondente andando a visitare l'AST con il metodo *codeGeneration*4.4.

3.1.1 Struttura del codice

Andiamo ad illustrare ora come abbiamo strutturato il codice, all'interno di *src* troviamo le seguenti cartelle:

- *ast*: Contiene tutti i nodi che compongono l'AST, ogni nodo estende l'interfaccia *Node* e implementa i metodi definiti in essa.
- *com/company*: Contiene il file *main.java* che rappresenta il punto di partenza del nostro programma, e contiene la logica principale del nostro programma.
- *gen*: Contiene i file generati automaticamente da ANTLR, che rappresentano il Lexer e il Parser del nostro linguaggio, e le interfacce per la generazione dell'AST.

- *utilities*: Contiene i file di utilità, come *Environment.java*, *TypeErrorException.java* che rappresenta l'eccezione lanciata in caso di errore di tipo, *Utils.java* che contiene metodi di utilità generica.
- *test*: Contiene i file di test, che descrivono i test del nostro programma, su un sottoinsieme di programmi scritti in HLCostLang.

All'interno di *ast* troviamo tutti nodi che compongono l'ast suddivisi in base al tipo di nodo, ad esempio *CallNode.java* che rappresenta il nodo di una chiamata a funzione, *IfNode.java* che rappresenta il nodo di un costrutto *if-then-else* si trovano all'interno della cartella *ast/stm*. Nodi come *BinExpNode.java* che rappresenta il nodo di un'espressione binaria, *DerExpNode.java* che rappresenta un identificativo, si trovano all'interno della cartella *ast/exp*. Nodi che servono per il controllo dei tipi come *IntType.java* che rappresenta il tipo intero, *BoolType.java* che rappresenta il tipo booleano, si trovano all'interno della cartella *ast/typeNode*. La cartella *utilities* contiene i file di utilità, come *Environment.java* che rappresenta l'ambiente in cui si trova il nostro programma, *TypeErrorException.java* che rappresenta l'eccezione lanciata in caso di errore di tipo, *Utils.java* che contiene metodi di utilità come *isSubtype*.
La cartella *test* contiene i file di test, che rappresentano i test del nostro programma, su un sottoinsieme di programmi scritti in HLCostLang.
I test sono stati scritti in modo da testare il corretto funzionamento del nostro programma, nello specifico, la correttezza del controllo semantico, del controllo dei tipi, della generazione delle equazioni di costo e del codice WASM corrispondente.
Il test ritorna un enumerato con il codice di errore specificato, in caso di errore, altrimenti ritorna *Results.PASS*. Alla fine di ogni sviluppo di una nuova funzionalità, andiamo a testare il corretto funzionamento del nostro programma, andando a eseguire i test, e verificando che il programma sia corretto rispetto alle regole del linguaggio.

```
1 public class TestCostCompiler {
2
3     @Test
4     public void test1() throws IOException {
5             System.out.println("Test Listing 1");
6             assertEquals(CostCompiler("example/Listing1"),
    Results.PASS);
7         }
8     ...
```

Listing 3.9: Esempio Testing

Il test è molto semplice, prende in input un programma in HLCostLang e verifica che la compilazione termini correttamente, in caso contrario stampa un errore a video, questo è un esempio banale di testing, ma ci permette di verificare che il programma sia corretto rispetto alle regole del linguaggio.
Inoltre ci sono altri test che verificano che il programmi ritorni degli errori specifici, definiti in base al tipo di errore, ad esempio se il programma non termina correttamente, se il programma ritorna un errore semantico, se il programma ritorna un errore di tipo, oppure se il programma ritorna un errore di compilazione, e cosí via.

3.2 Regole di Inferenza

I programmi di costo sono elenchi di equazioni che hanno termini:

$$f(\overline{x}) = e + \sum_{i \in 0..n} f_i(\overline{e_i}) \qquad [\varphi]$$

Dove le variabili si presentano nel lato destro e in φ sono un sottoinsieme di $\overline{x}$; mentre f e f_i sono i simboli delle funzioni. Ogni funzione ha un right-hand-side che è un'espressione aritmetica che può contenere:

- Un'espressione in Presburger aritmetica (PA)[13]:

  $$e ::= x \quad | \quad q \quad | \quad e + e \quad | \quad e - e \quad | \quad q \cdot e \quad | max(e_1, \ldots, e_n)$$

  Dove x è una variabile, q è una costante intera, $e_1, \ldots, e_n$ sono espressioni aritmetiche e max è un operatore che restituisce il massimo valore tra le sue espressioni.

- Un numero di invocazioni di funzioni di costo: $f_i(\overline{e_i})$.

- La guardia φ è un vincolo congiuntivo lineare nella forma: $e_1 \geq e_2$ dove e_1 e e_2 sono espressioni aritmetiche di Presburger.

La soluzione di un equazione di costo è il calcolo dei limiti di un particolare simbolo di una funziona(generalmente la prima equazione) e i limiti sono parametrici nei parametri formali dei simboli della funzione. Definiamo un insieme di regole di inferenza che raccolgano frammenti di programmi di costo che vengono poi combinati in modo diretto dalla sintassi. Usiamo una variabile di ambiente Γ come dizionari:

- Γ prende un servizio o un parametro e ritorna un espressione aritmetica di Presburger che di solito è una variabile.

- Quando scriviamo $\Gamma + i : \quad Nat$, assumiamo che i non appartenga al dominio di Γ.

I giudizi hanno forma:

- $\Gamma \vdash E : e$, che significa che il valore dell'espressione intera E in Γ è rappresentato (dall'espressione nell'aritmetica di Presburger) e
- $\Gamma \vdash E : \varphi$, significa che l'espressione booleana E in Γ è rappresentata da φ
- $\Gamma \vdash S : e; C; Q$, significa che il costo di S nell'ambiente Γ è $e + C$ dato un insieme di equazioni Q
- $\Gamma \vdash F : Q$, significa che il costo di F nell'ambiente Γ è rappresentato da un insieme di equazioni Q

[MAIN]
$$\frac{\Gamma \vdash S : e; C; Q \qquad \overline{w} = \mathrm{Var}(\overline{p}, e) \cup \mathrm{Var(C)}}{\Gamma \vdash \overline{p} \to \{S\} : 0; \emptyset; Q'; C} \tag{3.1}$$

[CALL]
$$\frac{\Gamma \vdash S : e; C; Q}{\Gamma \vdash \text{call h}(\overline{E})S : e + e'; C; Q} \tag{3.2}$$

[IF]
$$\frac{\begin{array}{l}\Gamma \vdash E : \varphi \qquad \Gamma \vdash S : e'; C; Q \quad \Gamma \vdash S : e"; C'; Q' \\ W = Var(e, e', e") \cup Var(C) \qquad Q" = \left[\begin{array}{l}\mathrm{if}_l(\overline{w}) = e' + c[\varphi] \\ \mathrm{if}_l(\overline{w}) = e" + c[\neg\varphi]\end{array}\right]\end{array}}{\Gamma \vdash \text{if } E \text{ then } S_1 \text{ else } S_2 : 0; if_l(\overline{w}); Q; Q'; Q"} \tag{3.3}$$

[LET]
$$\frac{\Gamma + i : Not \qquad \overline{w} = Var(E) \cup Var(c)}{\Gamma \vdash \text{let } i = e \text{ in } S : S; e; C; Q} \tag{3.4}$$

[FOR]
$$\frac{\begin{array}{l}\Gamma \vdash M : e \qquad \Gamma \vdash i : Not \qquad \Gamma \vdash S : e'; C; Q \\ \overline{w} = Var(e, e') \cup Var(C)\ i \qquad Q' = \left[\begin{array}{ll}\mathrm{for}_l(i, \overline{w}) = e + c & [i < e] \\ \mathrm{for}_l(i, \overline{w}) = 0 & [i \geq e]\end{array}\right]\end{array}}{\Gamma \vdash \text{for i in (0,M)} \quad \text{S} \quad : 0; for_l(0, \overline{w}); Q; Q'} \tag{3.5}$$

Riassumiamo le regole descritte in precedenza:

- Regola[call] gestisce l'invocazione di un servizio; il costo della call sarà il costo di S più il costo per l'accesso al servizio h
- Regola[if] gestisce il costrutto condizionale;quando la guardi è un'espressione definita in aritmetica di Presburger e il costo verrà rappresentato da entrambi i rami con i due condizionali φ e $\neg\varphi$.
  Rappresentiamo a livello di equazione if_l dove l è la linea di codice dovè inizia il costrutto.
- Regola [for] descritto all'interno del rispettivo frammento di codice come for_l per lo stesso motivo citato in precedenza; fefiniamo i come Nat e verifichiamo che non sia presente nell'ambiente Γ e scriviamo il rispettivo S come caso base in cui $e \geq i$ oppure $i \geq e + 1$
- Regola [LetIn] Dove viene definita E nell'ambiente Γ con costo e(il costo per eseguire l'espressione e). Andremo a valutare se in Γ è presente E e andiamo a valutare $\Gamma \vdash S$ che ritornerà un'equazione Q' con costo C.

3.3 Generazione delle Equazioni di costo

La generazione delle equazioni di costo viene eseguita andando a implementare le regole di inferenza viste in precedenza. Ogni nodo all'interno del nostro AST contiene il metodo *toEquation()* che prende come argomento la variabile del nostro ambiente Γ e sarà appunto un dizionario. Questo dizionario di tipo $EnvVar$ è un HashMap che contiene come chiave l'oggetto Nodo della variabile e come valore la stringa rappresentante. Abbiamo deciso di utilizzare questo approccio per focalizzarci sull'efficienza del farci restituire la variabile che mappa quel determinato Nodo, senza dover andare a cercare all'interno dell'HashMap la chiave che mappa quel valore, cosa che viene fatta all'inserimento di un Nodo. L'inserimento del nodo però non sempre è un operazione onerosa per il fatto che abbiamo gia il controllo semantico che ci garantisce che non ci saranno variabili non dichiarate oppure variabili non dichiarate prima del loro utilizzo.
Andiamo ad analizzare un esempio semplice, all'interno del Nodo di tipo *CallService.java* che rappresenta l'invocazione di un servizio: abbiamo il metodo *toEquation()* che prende come argomento l'ambiente Γ e restituisce una stringa che rappresenta l'equazione di costo del nodo. Questa sottostringa sarà poi riportata all'interno dell'equazione di costo del nodo padre.

```
1     @Override
2     public String toEquation(EnvVar e) {
3         return "nat("+e.get(this)+")" + (stm!= null ? "+"+stm
      .toEquation(e) : "");
4
5     }
```

La funzione $e.get(this)$ ritorna la variabile mappata per quel determinato nodo, ritorna quindi una stringa che rappresenta la variabile all'interno dell'equazione di costo. La funzione $stm.toEquation(e)$ è la chiamata sul metodo *toEquation()* del nodo figlio, che restituirà la stringa rappresentante l'equazione di costo del nodo, andando a richiamare il medesimo metodo sui sottonodi contenuti all'interno del nodo figlio, e così via.
Per avere una panoramica completa del processo di generazione delle equazioni di costo, riportiamo il frammento di codice della funzione *toEquation*() del *programNode*, che rappresenta il nodo principale del nostro AST, che andrà a richiamare il metodo *toEquation*() su tutti i nodi figli e andrà a concatenare le stringhe risultanti al fine di generare l'equazione finale.

```
1     public String toEquation(EnvVar e){
2         for (Node n : decServices){
3             n.checkVarEQ(e);
4         }
5         StringBuilder equ = new StringBuilder();
6
7         for(Node n : funDec){
8             equ.append(n.toEquation(e));
9         }
10        return main.toEquation(e) + equ;
11    }
```

Listing 3.10: toEquation() del ProgramNode

Come possiamo vedere, prima di generare le equazioni di costo del programma, andiamo a controllare che le variabili dichiarate all'interno dei servizi siano presenti all'interno dell'ambiente Γ e le mappiamo con determinate stringhe che appariranno nelle equazioni. Successivamente andiamo a iterativamente all'interno delle singole funzioni le generiamo e le concateniamo alla stringa che rappresenta le equazioni di costo del programma.
Infine ci occupiamo di generare le equazioni di costo della funzione main, che saranno concatenate anch'esse con la stringa che rappresenta le equazioni di costo del programma.

3.4 PUBS

PUBS (Pratical Upper Bounds Solver) ha l'obiettivo di ottenere automaticamente un'upper bound in forma chiusa per i sistemi di equazioni di costo, di conseguenza calcola i limiti superiori per la relazione di costo indicata come "Entry".

3.4.1 Analisi di costo

Come analisi statica dei costi, miriamo a ottenere risultati analitici per un dato programma P, i quali consentano di vincolare il costo dell'esecuzione di P su qualsiasi input x, senza dover effettivamente eseguire $P(x)$.[14]
Partiamo da un esempio, che ci aiuterà a capire meglio il concetto di analisi di costo, partendo da un programma java:

```
1    public void m(int[] v) {
2        int i=0;
3        for (i=0; i<v.length; i++)
4        if (v[i]%2==0) m1();
5        else m2();
6    }
```

Listing 3.11: Esempio di Analisi di Costo

Le seguenti relazioni catturano il costo di esecuzione di questo programma:

$$
\begin{array}{lll}
(a)\ C_m(v) & = k_1 + C_{for}(v, 0) & \{v{\geq}0\} \\
(b)\ C_{for}(v, i) & = k_2 & \{i{\geq}v,\ v{\geq}0\} \\
(c)\ C_{for}(v, i) & = k_3 + C_{m1}() + C_{for}(v, i{+}1) & \{i{<}v,\ v{\geq}0\} \\
(d)\ C_{for}(v, i) & = k_4 + C_{m2}() + C_{for}(v, i{+}1) & \{i{<}v,\ v{\geq}0\}
\end{array}
$$

Figura 3.1: Relazioni di costo del programma 3.11

Dove v indica la lunghezza dell'array v e i è la variabile di iterazione del ciclo, mentre C_m, C_{m1}, C_{m2} approssimano, rispettivamente il costo di esecuzione di m, m_1 e m_2.
I vincoli collegati ad ogni equazioni determinano le loro condizioni di applicabilità.
Ad esempio, l'equazione (a) corrisponde al costo di esecuzione del metodo m con un array di lunghezza maggiore di 0(indicato nella condizione $\{v > 0\}$), dove un costo k_1 è accumulato dal costo dell'esecuzione del ciclo, dato da C_{for}. Le costanti $k_1, \ldots, k_4$ sono valori differenti in base al modello di costo che viene selezionato. Ovvero, se il modello di costo è basato sul numero di

istruzioni eseguite, allora k_1 è 1 che corrisponde al costo dell'esecuzione di una istruzione java come *int i= 0*. Se il modello di costo si riferisce all'occupazione dell'heap, allora k_1 sarà 0, poichè l'istruzione precedente non alloca memoria.
Le equazioni c e d catturano rispettivamente il costo di esecuzione del branch *then* e *else*. Si noti che, anche se il programma è deterministico, si tratta di equazioni non deterministiche che contengono le stesse condizioni di applicabilità. Ciò è dovuto al fatto che l'array v è astratto rispetto alla sua lunghezza e quindi i valori dei suoi elementi sono staticamente sconosciuti.
Alcuni aspetti interessanti dell'equazioni di costo:

- Sono indipendenti dal linguaggio di programmazione
- Possono rappresentare diverse classi di complessità: lineare, quadratica, logaritmica, ecc.
- Possono essere utilizzate come abbiamo visto, per catturare una varietà di nozioni non banali di risorse: come il numero di chiamate a funzioni, il numero di allocazioni di memoria, il numero di accessi a memoria, ecc.

Un esempio già più completo che troviamo nel paper *Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis* [7]:

```
void del(List l, int p, int a[], int la, int b[], int lb){
  while (l!=null) {
    if (l.data<p) {
      la=rm_vec(l.data, a, la);
    } else {
      lb=rm_vec(l.data, b, lb);
    }
    l=l.next;
  }
}
int rm_vec(int e, int a[], int la){
  int i=0;
  while (i<la && a[i]<e)  i++;
  for (int j=i; j<la−1; j++) a[j]=a[j+1];
  return la−1;
}
```

(1) $Del(l, a, la, b, lb)=1+C(l, a, la, b, lb)$
$\{b\geq lb, lb\geq 0, a\geq la, la\geq 0, l\geq 0\}$
(2) $C(l, a, la, b, lb)=2$ $\{a\geq la, b\geq lb, b\geq 0, a\geq 0, l=0\}$
(3) $C(l, a, la, b, lb)=$
$25+D(a, la, 0)+E(la, j)+C(l', a, la-1, b, lb)$
$\{a\geq 0, a\geq la, b\geq lb, j\geq 0, b\geq 0, l>l', l>0\}$
(4) $C(l, a, la, b, lb)=$
$24+D(b, lb, 0)+E(lb, j)+C(l', a, la, b, lb-1)$
$\{b\geq 0, b\geq lb, a\geq la, j\geq 0, a\geq 0, l>l', l>0\}$
(5) $D(a, la, i)=3$ $\{i\geq la, a\geq la, i\geq 0\}$
(6) $D(a, la, i)=8$ $\{i<la, a\geq la, i\geq 0\}$
(7) $D(a, la, i)=10+D(a, la, i+1)$ $\{i<la, a\geq la, i\geq 0\}$
(8) $E(la, j)=5$ $\{j\geq la-1, j\geq 0\}$
(9) $E(la, j)=15+E(la, j+1)$ $\{j<la-1, j\geq 0\}$

Figura 3.2: Esempio di Analisi di Costo

Come possiamo vedere dall'immagine sopra, abbiamo a sinistra un programma scritto in java mentre a destra abbiamo l'analisi di costo del programma. Il metodo *del* prende in input una lista l, un pivot p, due array ordinati di interi a e b e la e lb che indicano rispettivamente le posizione occupate in a e b. Inoltre, si prevede che l'array a contiene gli elementi inferiori al pivot p, mentre b rispettivamente ne conterrà i valori maggiori o uguali. Partiamo

dal presupposto che tutti i valori in l siano contenuti in a o b, e il metodo *del* rimuove tutti i valori in l da a o b rispettivamente. Il metodo rm_{vec} rimuove un dato valore e da un array a di lunghezza la e ne ritorna la nuova lunghezza. Gli autori del paper [7],applicano l'analisi dei costi a questo programma, approssimando il costo del metodo *del* in termini di istruzione bytecode eseguite.
La figura 3.2 presenta i risultati dell'analisi, dopo aver effettuato una valutazione parziale. Nei risultati dell'analisi le strutture dati vengono astratte in base alle loro dimensioni:l rappresenta la lunghezza massima del percorso della corrispondente struttura dinamica, a e b sono le lunghezze degli array corrispondenti, mentre la e lb sono i valori interi delle variabili. Ci sono nove equazioni che definiscono la relazione *del* che corrisponde al costo del metodo *del* e 3 relazioni ricorsive ausiliarie C, D, E; ciascuna delle quale corrisponde a un ciclo:(C: Ciclo while in *del*, D: ciclo while in rm_{vec} e E: Ciclo for in rm_{vec}). Ogni equazioni è definita con una serie di vincoli che catturano le relazioni dimensionali tra i valori delle variabili della parte sinistra(lhs) e della parte destra(rhs). Prendiamo in esempio le equazioni per D Eq5. e Eq.6 rappresentano casi base per l'uscita dal ciclo ovvero quando $i \geq la$ e $a[i] \geq e$. Per le nostre misurazioi di costo, vengono contati 3 istruzioni bytecode in Eq.5 e 8 in Eq.6. Il costo per eseguire un iterazione del ciclo è rappresentato da Eq.7, dove la condizione $i < la$ deve essere soddisfatta e la variabili i è incrementata di uno ad ogni chiamata ricorsiva.

3.4.2 Relazione di Costo

Un'espressione di costo di base è un'espressione simbolica che indica le risorse accumulate e i blocchi fondamentali non ricorsivi per la definizione delle *relazioni di costo.*

Definizione 1. *(Espressione di costo di base)*
Le espressioni di costo sono della forma

$$exp ::= a|nat(l)|exp + exp|exp * exp|exp^a|log_a(exp)|max(s)|\frac{exp}{a}|exp - a$$

dove $a \geq 1$, l *è un'espressione lineare,* S *è un insieme non vuoto di espressioni di costo,* $nat : \mathbb{Z} \to \mathbb{Q}^+$ *è definita come* $nat(v) = max(v, 0)$ *e exp soddisfa per qualsiasi assegnamento di* $\overline{v}$ *per vars(exp) si ha* $exp[vars(\frac{exp}{\overline{v}})]$

Le espressioni di costo di base godono di due proprietà:

- Sono sempre valutate per valori non negativi
- Rimpiazzando una sottoespressione $nat(l)$ con $nat(l')$ tale che $l \geq l'$, il risultato è un upper bound per l'espressione originale.

L'analisi dei costi di un programma produce multiple relazioni interconnesse, generando un *sistema di relazioni di costo*(CRS)

Definizione 2. *(Sistema di relazioni di costo)*
Un sistema di relazioni di costo S è un set di equazioni della forma $\langle C(\overline{x}) = exp + \sum_{i=0}^{n} D_i(\overline{y_i}), \varphi \rangle$ *dove* C *e* $D_{0,\ldots,i}$ *sono relazioni di costo; tutte le variabili in* $\overline{x}$ *e* $\overline{y_i}$ *sono variabili distinte, e* φ *è una relazione di dimensione tra* $\overline{x} \cup vars(exp) \cup \overline{y_i}$.

Dato S sistema di relazioni di costo, $rel(S)$ indica l'insieme delle relazioni di costo definite in S, $def(S, C)$ indica il sottoinsieme di equazioni in S il cui lato sinistro è della forma $C(\overline{x})$. Possiamo supporre che tutte le equazioni definite in $def(S, C)$ abbiano variabili con lo stesso nome nella parte sinistra. Inoltre si suppone che ogni relazione di costo che appare nella parte sinistra dell'equazione S deve essere in $rel(S)$.

Semantica per CRS

Data una CRS S, una call è della forma $C(\overline{v})$ dove $C \in rel(S)$ e $\overline{v}$ sono valori interi. Le *call* sono valutate in due fasi, in cui la prima permette la costruzione un albero di evoluzione, mentre la seconda ottiene un valore di $\mathbb{R}^+$ da aggiungere alla costante che appare nell'albero di valutazione. Gli alberi di evoluzione sono costruiti espandendo iterativamente i nodi che includono chiamate alle relazioni. Ogni espansione avviene rispetto a un'istanza appropriata della parte destra di un'equazione applicabile. Se tutte le foglie dell'albero contengono un'espressione di costo di base, allora non ci sono più nodi da espandere e il processo termina. Questi alberi sono rappresentati utilizzando termini annidati, del tipo node(Call; Local_Cost; Children), in cui Local_Cost è una costante in $\mathbb{R}^+$ e Children è una sequenza di alberi di evoluzione.

Definizione 3. *(**Albero di evoluzione**) Data una CRS S, una call* $C(\overline{v})$, *un albero node* $(C(\overline{v}); e; \overline{t})$ *è un albero di evoluzione per* $C(\overline{v})$ *in S, indicato con* $Tree(C(\overline{v}, S))$, *se:*

1. *c'è una denominazione parziale dell'equazione* $\langle C(\overline{x}) = exp + \sum_{i=0}^{k} D_i(\overline{y_i}), \varphi \rangle$
2. *esiste un assegnamento di valori interi* $\overline{w}$ *a* $\overline{v_i}$ *per* $var(exp), \overline{y_i}$ *rispettivamente tali che* $\varphi[vars(exp)/\overline{w}, \overline{y_i}/\overline{v_i}]$ *è soddisfacibile in* $\mathbb{Z}$
3. $e = exp[vars(exp)/\overline{w}]$, T_i *è un albero di evoluzione* $tree(D_i(\overline{v_i}, S))$ *with* $i \in 0, \ldots, k$

Nel processo di risoluzione di $C(\overline{v})$ ci possono essere diverse equazioni applicabili, e quindi diversi alberi di evoluzione. Al passo 2 si cerca un assegnamento per le variabili nella parte destra di ϵ. Al passo 3 gli assegnamenti sono applicati ad exp e si continua ricorsivamente valutando le call. $Tree(C(\overline{v}, S))$ viene usato per denotare l'insieme di tutti gli alberi di evoluzione per $C(\overline{v})$.

3.4.3 Stima del costo per Nodo

Tutte le espressioni nei nodi sono istanze delle espressioni che compaiono nelle equazioni corrispondenti. Pertanto, il calcolo di $costr^{+}(\overline{x})$ e $costnr^{+}(\overline{x})$ può essere effettuato trovando innanzitutto un limite superiore di tali espressioni e quindi combinandoli attraverso un operatore di massimo. Prima calcoliamo gli invarianti per i valori che le variabili delle espressioni possono assumere rispetto ai valori iniziali e li utilizziamo per derivare limiti superiori per tali espressioni.
Calcolare le invarianti(in termini di vincoli lineari), in modo da raggruppare tutte le chiamate ai contesti di una relazione C, tra gli argomenti di una chiamata iniziale e ogni chiamata durante la valutazione che può essere fatta usando $Loops(C)$.
Ovvero, se è presente un vincolo lineare ψ tra gli argomenti di una chiamata iniziale $C(\overline{x_0})$, quelli di una chiamata ricorsiva $C(\overline{x})$, indicato con $\langle C(\overline{x_0}) \rightsquigarrow (\overline{x}, \psi)\rangle$, e se esiste il ciclo $C(\overline{x_0}) \rightsquigarrow (\overline{y}, \varphi) \in Loops(C)$ allora è possibile applicare il ciclo a uno o più step e prende un nuovo calling context $\langle C(\overline{x_0}) \rightsquigarrow (\overline{y}), \overline{\exists} \cup \overline{y_i}\dot{\psi} \wedge \varphi\rangle$.
Una volta che le invarianti sono state stabilite, è possibile determinare il limite superiore delle equazioni di costo massimizzando la loro parte nat indipendentemente. Questo approccio è reso possibile grazie alla proprietà di monotonia delle espressioni di costo. Considerando un'equazione di costo nella forma $\langle C(\overline{x}) = exp + \sum_{i=0}^{k} C(\overline{y_i}), \varphi\rangle$ e un invariante $C(\overline{x_0}) \rightsquigarrow C(\overline{x}, \Psi)$,una funzione può calcolare un limite superiore f' per ogni f che compare nell'operatore nat. Tale funzione sostituisce f con un limite superiore nelle espressioni exp in cui non è possibile determinare un limite superiore e la funzione tornerà ∞. Se questa funzione è completa, ovvero se i Ψ e φimplicano che esiste un limite superiore per un dato $nat(f)$, allora possiamo trovare un limite superiore su Ψ'.[15]

3.4.4 PUBS in pratica

Prendiamo in considerazione il seguente esempio di programma dato in input a CostCompiler:

```
struct Params {
  address: array[int],
  payload: any,
  sender: string
}
service PremiumService : (string) -> void;
service BasicService : (any) -> void;
(isPremiumUser: bool, par: any) => {
  if ( isPremiumUser ) {
    call PremiumService("pippo");
  } else {
    call BasicService( par);
  }
}
```

Listing 3.12: Listing 1

PUBS (Pratical Upper Bounds Solver) ha l'obiettivo di ottenere automaticamente un'upper bound in forma chiusa per i sistemi di equazioni di costo, calcola i limiti superiori per la relazione di costo indicata come "Entry", oltre che per tutte le altre relazioni di costo di cui tale "Entry" dipende.
Nell'output di PUBS vengono mostrati anche i passaggi intermedi eseguiti che coinvolgono il calcolo delle funzioni di classificazione e degli invarianti di ciclo.

```
CRS $pubs_aux_entry$(A,B,C) -- THE MAIN ENTRY

  * Non Asymptotic Upper Bound: max([nat(A),nat(B)])

  * LOOPS $pubs_aux_entry$(D,E,F) -> $pubs_aux_entry$(G,H,I)

  * Ranking function: N/A

  * Invariants $pubs_aux_entry$(A,B,C) -> $pubs_aux_entry$(D,E,F)

     entry  : []
     non-rec: [A=D,B=E,C=F]
     rec    : [0=1]
     inv    : [1*A+ -1*D=0,1*B+ -1*E=0,1*C+ -1*F=0]

CRS main(A,B,C)

  * Non Asymptotic Upper Bound: max([nat(A),nat(B)])

  * LOOPS main(D,E,F) -> main(G,H,I)

  * Ranking function: N/A

  * Invariants main(A,B,C) -> main(D,E,F)

     entry  : []
     non-rec: [A=D,B=E,C=F]
     rec    : [0=1]
     inv    : [1*A+ -1*D=0,1*B+ -1*E=0,1*C+ -1*F=0]
```

Figura 3.3: Esempio di output PUBS su Listing 1

Come vediamo nell'immagine sopra, PUBS restituisce un analisi dell'intera equazione (*pub_aux_entry*) e delle singole funzioni da cui essa dipende, in questo caso *pubs_aux_if*9 e *pubs_aux_main*.
In questo caso con "Listing1" abbiamo un Upper Bound non Asintotico di $max(Nat(A), Nat(B))$ che ci determina che il costo del programma dipende dalle variabili A e B, e che il costo del programma sarà il massimo tra i due.
PUBS ha una grammatica che definisce l'equazione di costo che deve essere rispettata da ogni equazione di costo generata da CostCompiler, che è la seguente:

```
<equation> ::=
    eq(Head,costExpression,[listOfCall],[
   ListOfSizeRelation]).

<Head> ::= Name | Name(Par).

<costExpression> ::= nat(<variable>)
             | <costExpression> + <costExpression>
             | <costExpression> - <costExpression>
             | <costExpression> * <costExpression>
             | max(<costExpression>,<costExpression>)
   .

<listOfCall> ::= [] | <call> <listOfCall>.
<call> ::= <function>(<listOfParameters>).
<listOfParameters> ::= [] | <variable> <
   listOfParameters>.
```

Listing 3.13: Grammatica PUBS

Dove <Head> è il nome della entry che andremo ad analizzare insieme ai suoi parametri. CostExpression è l'espressione di costo che rappresenta il costo della entry e rispetta la grammatica della aritmetica di Presburger. ListOfCall è la lista delle chiamate alle altre entry, che sono rappresentate come <call> e <listOfCall>, la lista di queste chiamate; in questo modo PUBS riesce a costruire un grafo delle dipendenze tra le entry. Infine abbiamo <listOfSizeRelation> che sarà la lista delle relazioni di costo che dipendono dalla entry che stiamo analizzando, e che PUBS andrà a calcolare.
Riportiamo un'altro esempio di equazione di costo generata da CostCompiler, questa volta per il programma scritto in Listing6:

```
    service BasicService: (int) -> void;
    fn svc(i: int) -> void{
        for(m in (0,10)){
            call BasicService(i)
        }
    }
    (len : int) => {
        svc(len)
    }
```

Listing 3.14: Listing 6

Come vediamo, la funzione *init* chiamerà la funzione *svc* con parametro len, che a sua volta chiamerà la funzione BasicService per 10 volte, quindi il

costo del programma sarà l'invocazione della funzione $svc + 10 \cdot nat(B)$, dove $nat(B)$ è l'invocazione del servizio *BasicService*.
L'equazione di costo risultante sarà la seguente:

```
1    eq(main(B),1,[svc(B)],[]).
2    eq(svc(B),0,[for3(0, B)],[] ).
3    eq(for3(M, B) ,nat(B),[for3(M+1, B)], [10>= M]).
4    eq(for3(M, B) ,0,[],[M >= 10+ 1]).
```

Listing 3.15: Equazione di costo PUBS per Listing6

Nella prima riga troviamo l'entry *main* che prende in input B, con costo 1, chiama la funzione *svc*. Quest'ultima andrà a chiamare la funzione $for3$ inserendo un'ulteriore parametro che sarà il counter del ciclo con parametro 0 e B, che avrà costo 0 in caso $M >= 10 + 1$ altrimenti avrà costo $nat(B)$.
E come controprova mostriamo ora il risultato di PUBS su Listing6:

```
CRS $pubs_aux_entry$(A) -- THE MAIN ENTRY

  * Non Asymptotic Upper Bound: 1+11*nat(A)

  * LOOPS $pubs_aux_entry$(B) -> $pubs_aux_entry$(C)

  * Ranking function: N/A

  * Invariants $pubs_aux_entry$(A) -> $pubs_aux_entry$(B)

     entry  : []
     non-rec: [A=B]
     rec    : [0=1]
     inv    : [1*A+ -1*B=0]
```

Figura 3.4: Esempio di output PUBS su Listing 6

Capitolo 4

Generazione di WebAssembly

La ricerca di soluzioni efficienti e portabili per eseguire codice in ambienti diversi è diventata una priorità fondamentale nell'informatica moderna. In questo contesto, WebAssembly (Wasm) emerge come una tecnologia chiave, fornendo un formato binario sicuro, veloce e indipendente dalla piattaforma. Nel corso di questo capitolo, esploriamo il processo di generazione di codice WebAssembly attraverso un interprete dedicato a un linguaggio personalizzato. Il nostro linguaggio, creato per soddisfare specifiche esigenze o paradigmi di programmazione unici, si propone di offrire una flessibilità senza precedenti agli sviluppatori. Attraverso un interprete appositamente progettato, saremo in grado di tradurre il codice sorgente del nostro linguaggio in istruzioni Wasm, consentendo così l'esecuzione di programmi in un ambiente virtuale altamente performante e sicuro. Nel corso di questo capitolo, esamineremo in dettaglio il processo di compilazione, passando attraverso le fasi cruciali che trasformano il nostro codice sorgente in un modulo WebAssembly. Dalla rappresentazione intermedia alla gestione delle dipendenze, esploreremo come l'interprete si adatta alle specificità del nostro linguaggio per garantire una corretta esecuzione e ottimizzazione delle risorse. Il capitolo si propone inoltre di approfondire le sfide e le opportunità che emergono durante il processo di generazione di WebAssembly. Analizzeremo le scelte di progettazione dell'interprete, l'ottimizzazione del codice e la gestione delle risorse, fornendo un quadro completo delle considerazioni che guidano il nostro approccio alla generazione di codice Wasm. Forniremo inoltre le istruzione per eseguire il modulo WebAssembly generato, e analizzeremo il processo di esecuzione e di debug dello stesso attraverso un motore di runtime.[16]

4.1 Introduzione WebAssembly

WebAssembly è un formato di istruzioni altamente performante, portabile e sicuro, progettato per essere eseguito in ambienti virtuali. Originariamente concepito da un gruppo di lavoro congiunto tra Google, Mozilla, Microsoft e Apple, il principale obiettivo di WebAssembly è fornire un formato binario indipendente dalla piattaforma, garantendo al contempo sicurezza e velocità. Sebbene sia stato ideato principalmente per essere eseguito all'interno dei browser web, Wasm trova applicazioni anche in contesti diversi come server, dispositivi IoT e applicazioni desktop.
Le istruzioni Wasm si distinguono dalle istruzioni di un processore reale in quanto sono progettate per l'esecuzione in un ambiente virtuale. Ciò implica che le istruzioni Wasm non possono essere eseguite direttamente da un processore fisico; richiedono invece una traduzione preliminare in istruzioni native. Questo processo di traduzione è gestito da un motore di runtime, il quale interpreta le istruzioni Wasm e le traduce in istruzioni native del sistema ospite. Oltre alla traduzione delle istruzioni, il motore di runtime si occupa anche della gestione della memoria e delle risorse del sistema, fornendo un'astrazione sicura e indipendente dalla piattaforma.
In sintesi, WebAssembly rappresenta una tecnologia versatile e potente che offre un'alternativa efficace alle tradizionali soluzioni di esecuzione di codice, poiché il formato binario di WebAssembly è progettato per essere facilmente scaricabile e decodificabile dai browser web. Consente dunque l'esecuzione di codice web a velocità prossime a quelle delle applicazioni native. Questa caratteristica rende Wasm un'opzione attraente per una vasta gamma di applicazioni, inclusi giochi, strumenti di produttività, applicazioni di elaborazione multimediale e molto altro ancora.[17]

4.2 WebAssembly Text Format

Il formato di testo WebAssembly (WAT) [**WebAssemblyTextFormat**] è un formato di testo leggibile dall'uomo per la rappresentazione di moduli WebAssembly. Il formato è stato progettato per essere utilizzato come rappresentazione intermedia durante il processo di compilazione, fornendo un'astrazione leggibile dall'uomo per il codice Wasm. Consiste in un insieme di istruzioni mnemoniche e direttive che corrispondono direttamente alle istruzioni binarie di WebAssembly. Il compilatore che abbiamo sviluppato genera un file di testo .wat ed in seguito il tool wat2wasm [18] genera il file binario .wasm, che a sua volta potrà essere eseguito da un motore di runtime.

I tipi di dati principali che troviamo in wat sono:

- **i32** 32-bit integer
- **i64** 64-bit integer
- **f32** 32-bit float
- **f64** 64-bit float

Un singolo parametro *(param i32)* e il tipo di ritorno *(result i32)*.

```
(func (param i32) (param i32) (result f64) ...)
```

Listing 4.1: Esempio di funzione in wat

I parametri locali possono essere dichiarati all'interno di una funzione, e sono accessibili solo all'interno della funzione stessa. I comandi *local.get* e *local.set* vengono utilizzati per accedere agli indici dei parametri locali. Possiamo usare anche l'operatore $ per accedere ai parametri locali, in maniera più human-readable.

```
(func $fun (param i32) (param i32) (result f64)
    (local $par1 i32)
    (local $par2 i32)
    (local $par3 f64)
    ...
    (local.get $par1)
    (local.get $par2)
    (local.set $par3)
    ...
)
```

Listing 4.2: Esempio di funzione in wat

Come vediamo in questo esempio la funzione $fun prende in input due parametri di tipo intero e ritorna un valore di tipo float. Inoltre all'interno della funzione vengono dichiarati tre parametri locali, due di tipo intero e uno di tipo float.

Stack Machine

L'esecuzione del WebAssembly è definita in termini di Stack-Machine, dove l'idea generale è che ogni tipo di istruzione esegue operazioni di tipo *push/-pop* dallo stack.
Quando viene chiamata una funzione, inizia con uno stack vuoto che viene gradualmente riempito e svuotato man mano che le istruzioni del corpo vengono eseguite. Quindi, ad esempio, dopo aver eseguito la seguente funzione:

```
(func $somma(param $p1 i32)(param $p2 i32)
    (result i32)
    local.get $p1
    local.get $p2
    i32.add)
```

Quando viene chiamata la funzione $somma, viene passato il precedente valore nella pila come parametro $p1. La prima istruzione *local.get* copia il valore di $p1 nello stack, e la seconda istruzione *local.get* copia il valore di $p2 nello stack. Infine, l'istruzione *i32.add* rimuove i due valori superiori dello stack, li somma e inserisce il risultato nello stack. Alla fine dell'esecuzione della funzione, lo stack contiene il risultato della somma dei due valori passati come parametro. Come vediamo nello schema seguente:

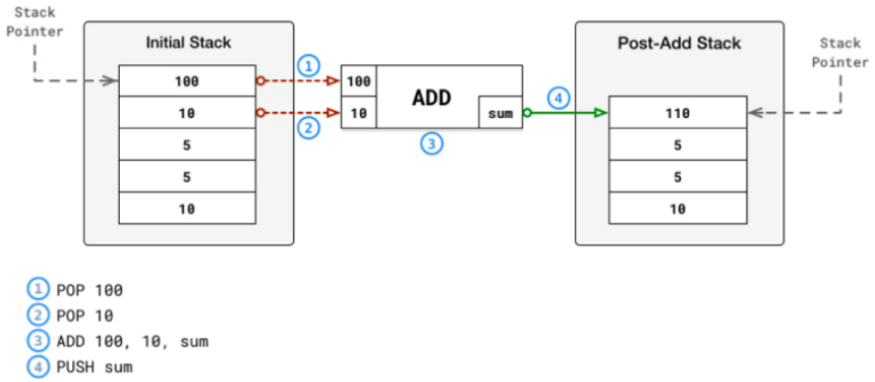

Figura 4.1: Stack Machine WASM

Un puntatore mantiene la posizione di esecuzione allínterno del codice e uno stack di controllo virtuale tiene traccia dei blocchi e dei costrutti man mano che vengono in input (operazione push) e in output (operazione pop). Le istruzioni tuttavia vengono eseguite senza alcun riferimento a un AST. Pertanto, la parte del formato delle istruzioni binarie della definizione si riferisce a una rappresentazione binaria delle istruzioni che sono in un formato leggibile dallo stack di decodifica nel browser.[19]
Per eseguire la **chiamata della funzione** 4.2, vediamo il codice seguente:

```
(func $main
    (result i32)
    i32.const 10
    i32.const 5
    call $somma)
```

La prima istruzione *i32.const* inserisce il valore constante 10 nello stack, e la seconda istruzione *call* chiama la funzione $somma. La funzione $somma viene eseguita, e il risultato viene inserito nello stack. Alla fine dell'esecuzione della funzione, lo stack contiene il risultato della somma dei due valori passati come parametro.
Dobbiamo inoltre aggiungere una dichiarazione di esportazione per fare in modo che la funzione sia visibile all'esterno del modulo(per esempio anche dal codice javascript).

```
1     (export "main"(func $main))
```

La prima stringa "main" è il nome della funzione che vogliamo esportare e che sara visibile anche all'esterno del modulo, mentre la seconda è l'identificativo della funzione a cui fa riferimento.

4.2.1 WebAssembly System Interface

Mentre il WebAssembly era inizialmente destinato all'uso nel sandbox del browser web, è progettato in modo tale da poter essere eseguito su qualsiasi piattaforma se esiste un'implementazione del runtime Wasm. Nei browser, il Wasm viene eseguito nell'interprete JavaScript e nel sandbox del browser, ma per eseguire moduli Wasm al di fuori di essi, è necessaria un'interfaccia con il sistema sottostante. Qui entra in gioco WASI (l'interfaccia di sistema WebAssembly).
WASI è un'interfaccia di sistema standardizzata per WebAssembly.
WASI si riferisce a tutti i diversi interpreti del codice Wasm al di fuori del browser. La Bytecode Alliance ha un paio di runtime Wasm di propria produzione: Wasmtime e WAMR (WebAssembly Micro Runtime). WASI promuove linee guida su come dovrebbe essere realizzato un corretto interprete Wasm in modo da mantenere le funzionalità di sicurezza sandbox che ha all'interno dei browser per impostazione predefinita. Wasmtime è un runtime di tipo just-in-time per il codice Wasm e WAMR è un interprete Wasm progettato per dispositivi embedded.[20]

4.3 Entità di WebAssembly

Le entità principali di WebAssembly sono:

- **Module**: Un modulo WebAssembly è un'unità di codice che può essere caricata e eseguita da un motore di runtime. Un modulo può contenere funzioni, variabili, tabelle e memoria, oltre a definizioni di tipi e importazioni/esportazioni.

- **Binario**: Il formato binario di WebAssembly è una rappresentazione compatta e ottimizzata del modulo Wasm. Il formato binario è progettato per essere eseguito in modo efficiente da un motore di runtime, consentendo l'esecuzione di codice Wasm in ambienti virtuali.
- **Embedder**: Un embedder è un'applicazione o un ambiente che incorpora un motore di runtime WebAssembly. Gli embedder possono includere browser web, server, dispositivi IoT e applicazioni desktop, tra gli altri.

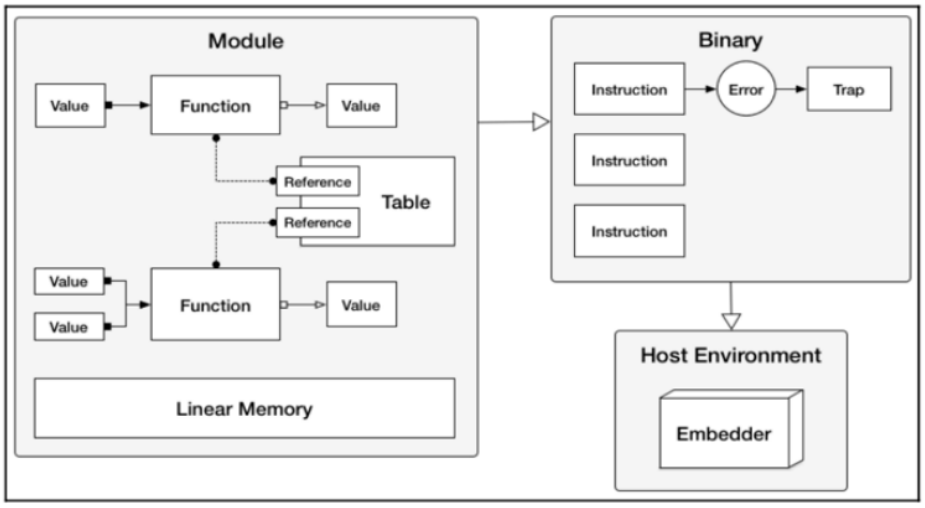

Figura 4.2: Schema Entità WebAssembly

Moduli

Un modulo WebAssembly è un'unità di codice che può essere caricata e eseguita da un motore di runtime. Un modulo può contenere funzioni, variabili, tabelle e memoria, oltre a definizioni di tipi e importazioni/esportazioni. I moduli Wasm sono progettati per essere indipendenti dalla piattaforma, consentendo l'esecuzione di codice Wasm in ambienti virtuali diversi. I moduli Wasm possono essere creati manualmente o generati da un compilatore, consentendo agli sviluppatori di scrivere codice in un linguaggio ad alto livello e di eseguirlo in un ambiente virtuale altamente performante e sicuro. I tipi di valore definiti in wasm come abbiamo visto in precedenza sono **i32**, **i64**, **f32**, **f64**, e i tipi di funzione **func**, ognuna delle quali accetta una sequenza di valori come parametri e restituisce una sequenza di valori come risultato. Un modulo interagisce con una memoria lineare, considerata come un intervallo

contiguo di indirizzi in byte che si estende dall'offset 0 fino a una dimensione variabile. La memoria può essere utilizzata per memorizzare dati e istruzioni, e può essere condivisa tra più moduli. Lo stato iniziale di una memoria lineare è definito dalla memoria lineare e dalle sezioni dei dati del modulo. La dimensione della memoria può essere aumentata dinamicamente dall'operatore grow-memory. Una memoria lineare può essere considerata una matrice non tipizzata di byte ma non è specificato come gli embedder mappino questa matrice nella propria memoria virtuale. Simile a una memoria lineare in wasm esiste il concetto di **tabelle**, che sono valori di un particolare tipo di tabella. Una tabella consente di contenere valori come puntatori nativi, a cui il codice Webassembly può accedere tramite un indice.

Formatting Binario

Le istruzioni in wasm rientrano in due categorie:**Istruzioni di controllo**, che formano costrutti di controllo e **Istruzioni di manipolazione dello stack**, che manipolano lo stack di valori. Le istruzioni di controllo sono istruzioni che modificano il flusso di controllo del programma, come ad esempio l'istruzione *if*. Le istruzioni di manipolazione dello stack sono istruzioni che manipolano lo stack di valori, come ad esempio l'istruzione *i32.const* che inserisce un valore costante nello stack. In Wasm le traps sono errori che si verificano durante l'esecuzione di un modulo Wasm, come ad esempio un'overflow aritmetico o un accesso fuori dai limiti della memoria. Le traps possono essere gestite da un motore di runtime, consentendo al modulo Wasm di continuare l'esecuzione in caso di errore.

Embedder

Un embedder è un'applicazione o un ambiente che incorpora un motore di runtime WebAssembly. Un'implementazione di webassembly viene in genere incorporata in un ambiente host. Un embedder implementa la connessione tra tale ambiente host e la semantica di webassembly e fornisce un'interfaccia per l'accesso a funzionalità come la memoria, le tabelle e le funzioni esportate da un modulo Wasm.[19][21]

4.4 CostCompiler to WAT

L'interprete che abbiamo sviluppato genera un file .wat, questo file .wat andrà poi convertito in un file .wasm, che a sua volta potrà essere eseguito da un motore di runtime. Questa conversione viene fatta tramite il tool wat2wasm

presente nella toolchain WebAssembly Binary Toolchain. [18]
Attraverso il comando:

```
wat2wasm file.wat -o file.wasm
```

Il tool wat2wasm prende in input un file di testo .wat e genera un file bytecode .wasm. Il file .wasm generato può quindi essere eseguito direttamente da un motore di runtime WebAssembly, consentendo l'esecuzione del modulo Wasm in un ambiente virtuale. Andando più nel dettaglio di come viene generato il file .wat dall'interprete, vediamo che per ogni nodo dell'ast che abbiamo parlato nei capitoli precedenti viene creata un'ulteriore funzione "codeGeneration()" che ritorna una stringa. Ricorsivamente andremo a chiamare la funzione "codeGeneration()" per ogni nodo dell'ast che ritorna una stringa che mano a mano verrà concatenata con la precedente andando a ottenere il codice wat. Andremo a vedere nello specifico due implementazioni della funzione "codeGeneration()" durante la generazione del codice wat, la codeGeneration per l'if Node e la codeGeneration per il for Node.

```
@Override
public String codeGeneration(HashMap<Node, Integer>
    offset_idx) {
    return "(local $res i32)\n" +
            "(if"+exp.codeGeneration(offset_idx)+
            "(then\n"+stmT.codeGeneration(offset_idx)+
            "(local.set $res)" +
            "\n)" +
            "(else\n"+stmF.codeGeneration(offset_idx)+
            "(local.set $res)" +
            "\n)" +
            "\n)" +
            "(local.get $res)\n";
}
```

Listing 4.3: codeGeneration() per l'if Node

Descriviamo il funzionamento della codeGeneration per l'if Node. Come vediamo nel listato 4.3 la funzione ritorna una stringa che contiene il codice wat per l'if Node e prende come parametro un *offset_idx*. *offset_idx* è un hashmap che mappa un nodo ad un intero, e ci serve per tenere traccia dell'offset delle variabili locali all'interno della memoria lineare. La prima istruzione *(local $res i32)* dichiara una variabile locale di nome $res di tipo i32, che serve da accumulatore per il risultato del ramo then e il risultato del ramo else. La seconda istruzione *if* richiama la funzione codeGeneration() dell'exp Node, che ritorna una stringa che contiene il codice wat per l'exp Node. Viene valutata l'espressione e se il risultato è 1 allora viene eseguito il ramo then,

altrimenti viene eseguito il ramo else. La terza istruzione *then* richiama la funzione codeGeneration() del ramo then, che ritorna una stringa che contiene il codice wat per il ramo then. Al termine di quella codeGeneration ci aspettiamo di avere un elemento della pila che contenga il risultato di quella espressione, con il local.set $res andiamo a salvare il risultato nella variabile locale $res, e togliendolo da quella pila, mantenendo cosi l'invariante. La quarta istruzione *else* richiama la funzione codeGeneration() del ramo else, che ritorna una stringa che contiene il codice wat per il ramo else, in maniera simmetrica a ciò che abbiamo fatto per il ramo then. La quinta istruzione *local.get* prende il valore della variabile locale $res e lo inserisce nello stack, e lo ritorna.
Andremo di seguito a vedere lo stesso ragionamento per la codeGeneration del for Node:

```
@Override
public String codeGeneration(HashMap<Node, Integer>
    offset_idx) {
     return
         "(local $"+id+" i32)\n" +
         "(local $"+id+"_max i32)\n" +
         "(local $res i32)\n" +
             exp.codeGeneration(offset_idx) +
         "(local.set $"+id+"_max)\n" +
         "(loop $for"+line+"\n" +
         "(if
           (i32.lt_u
             (local.get $"+id+")(local.get $"+id+"_max))"+
         "(then"+
             stm.codeGeneration(offset_idx) +
         "(local.set $res)\n" +
         "(local.get $"+id+"\n)" +
         "(i32.const 1)\n" +
         "(i32.add)\n" +
         "(local.set $"+id+")\n" +
         "(br $for"+line +")\n)"  +
         "(else" +
         "(local.get $"+id+"_max)\n" +
         "(local.set $"+id+")
         )\n))\n
         (local.get $res)" ;
     }
```

Listing 4.4: codeGeneration() per il for Node

La prima istruzione *(local $id i32)* dichiara una variabile locale di nome $id di tipo i32, che servirà da iteratore, e la seconda istruzione *(local $id_max i32)* dichiara una variabile locale di nome $id_max di tipo i32, che serve da limite

superiore per l'iteratore, infine viene dichiarata una variabile $res per memorizzare il risultato del corpo del ciclo, in caso di ritorno. La terza istruzione *exp.codeGeneration()* richiama la funzione codeGeneration() dell'exp Node, che ritorna una stringa che contiene il codice wat per l'exp Node. Questo valore appena valutato, verrà salvato nella variabile locale $id_max, con la quarta istruzione *(local.set $id_max)*. La quinta istruzione *(local.get $id_max)* prende il valore della variabile locale $id_max e lo inserisce nello stack, e lo ritorna. Successivamente viene eseguito un loop, che viene eseguito finchè il valore della variabile locale $id_max è minore del valore della variabile locale $id. Questo è reso possibile attraverso la definizione della label *(loop $for+line)* che ci permette di definire l'inizio del loop.
L'istruzione *if (i32.lt_u (local.get $id_max) (local.get $id))* prende i due valori $id_max e $id e li confronta, se il secondo è minore del primo allora esegue il ramo then,eseguendo il corpo del ciclo, altrimenti esce dal loop e passa al nodo successivo. Dentro il ramo then viene eseguito il corpo del ciclo, richiamando la funzione codeGeneration() del corpo del ciclo, che ritorna una stringa che contiene il codice wat per il corpo del ciclo.Inoltre verra preso il contatore $id, verrà incrementato di 1, e verrà salvato nella variabile locale $id, con l'istruzione *(local.set $id)* e salta alla label definita in precedenza*(br $for+line)*. Come ultima operazione verrà preso il valore della variabile locale $res e lo inserisce nello stack, e lo ritorna.

4.5 Esecuzione del modulo WebAssembly

Una volta generato il file .wasm, possiamo eseguirlo attraverso un motore di runtime. Nel nostro specifico caso di debug utilizziamo un file html che contiene il seguente codice javascript:

```
1  const memory = new WebAssembly.Memory({ initial: 1 });
2
3  const importObject = {
4      js: { mem: memory }
5  };
6
7  fetch("output.wasm")
8      .then(response => response.arrayBuffer())
9      .then(bytes => WebAssembly.instantiate(bytes,
       importObject))
10     .then(obj => {
11         console.log(obj.instance.exports.main(10));
12     })
13     .catch(error => console.error(error));
```

Listing 4.5: Esecuzione del modulo WebAssembly

La prima istruzione *const memory = new WebAssembly.Memory(initial: 1);* definisce un oggetto memory che rappresenta la memoria del modulo WebAssembly. In questo caso, la memoria è inizializzata con una dimensione iniziale di 1 pagina. La seconda istruzione *const importObject = { js: { mem: memory } };* definisce un oggetto di importazione che contiene la memoria del modulo WebAssembly. Questo oggetto di importazione verrà utilizzato per fornire l'accesso alla memoria del modulo WebAssembly. La terza istruzione *fetch("output.wasm")* carica il file .wasm e restituisce una Promise che contiene i byte del file .wasm.
Andando nella console di chrome è possibile vedere il file .wasm caricato, e debuggarlo come vediamo in questo snippet:

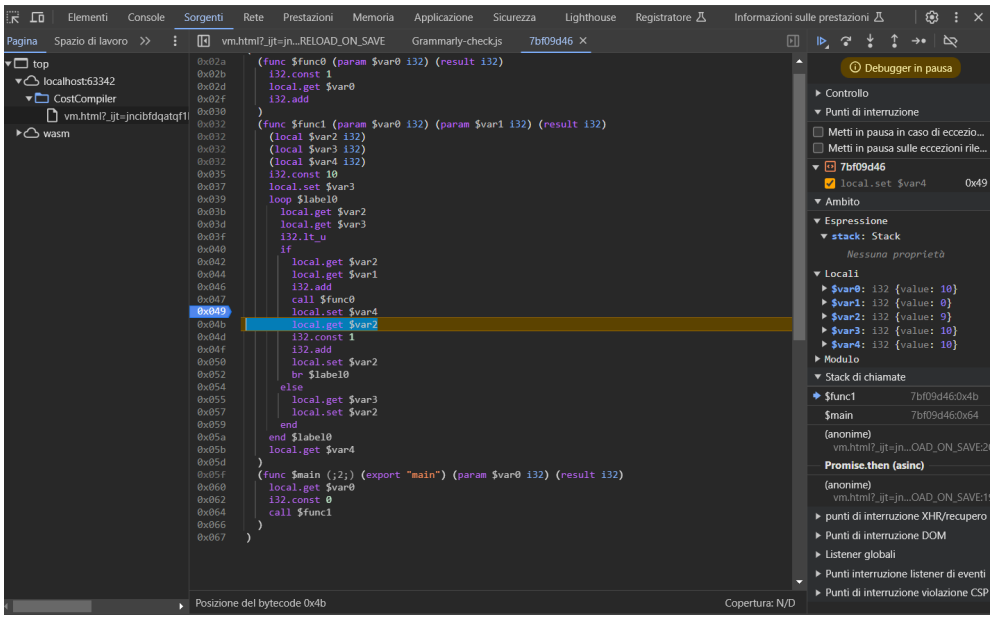

Figura 4.3: Debug del file output.wasm

Come notiamo il file .wasm è stato caricato correttamente, e possiamo andare a vedere il suo contenuto, inserire breakpoint e andare ad analizzare il runtime del nostro codice. Notiamo inoltre che il formato WAT a confronto è molto più leggibile, sopratutto per quanto riguarda il nome delle variabili, Wasm contiene il programma mappando le variabili in $var_0, \ldots, var_n$. A destra troviamo sia lo stack, mentre altre di sotto le variabili che interagiscono in un determinato scope locale, e lo stack di chiamate delle funzioni.

Capitolo 5

Conclusioni

Il progetto ha raggiunto l'obiettivo principale che ci eravamo prefissati. Abbiamo sviluppato un interprete prototipale attraverso il quale è possibile eseguire programmi scritti in HLCostLan, un linguaggio di programmazione appositamente progettato per l'analisi e l'ottimizzazione dei costi computazionali. HLCostLang offre una serie di costrutti e funzionalità che consentono agli sviluppatori di esprimere in modo chiaro e conciso il carico computazionale associato ai propri programmi, fornendo al contempo un'astrazione di alto livello che facilita la progettazione e l'implementazione del software.
L'interprete non solo è in grado di analizzare e d'interpretare i programmi scritti in HLCostLan, ma offre anche la possibilità di derivare l'equazione di costo associata a tali programmi, che viene poi calcolata attraverso il tool PUBS, per descrivere in modo formale il carico computazionale generato dal programma. Questo passaggio è di fondamentale importanza poiché fornisce agli sviluppatori una chiara comprensione del carico computazionale generato dal proprio codice, consentendo loro di valutare e ottimizzare le prestazioni in modo più accurato, ed eventualmente di prendere decisioni più accurate in merito all'allocazione delle risorse.
Inoltre, CostCompiler offre inoltre la capacità di generare automaticamente il corrispondente codice WebAssembly. Questa caratteristica rappresenta un passo significativo nella traduzione efficiente e affidabile dei programmi scritti in HLCostLan in un formato eseguibile ampiamente supportato. Tale capacità assume un'importanza fondamentale in un'epoca in cui la computazione distribuita e la scalabilità sono sempre più cruciali. Il codice WebAssembly risultante può essere facilmente incorporato in una vasta gamma di ambienti di esecuzione, consentendo una maggiore adozione e interoperabilità del linguaggio HLCostLan.

L'approccio modulare e flessibile con cui è stato sviluppato l'interprete è stato pensato per agevolare ulteriori estensioni e per semplificare le operazioni di manutenzione. Questa progettazione consente agli sviluppatori di ampliare le funzionalità dell'interprete in base alle esigenze specifiche dei loro progetti, promuovendo un'evoluzione organica e sostenibile del software nel tempo. Inoltre, abbiamo reso il codice sorgente dell'interprete liberamente accessibile su GitHub. In questo modo è possibile non solo l'esplorazione e la comprensione approfondita del funzionamento interno dell'interprete, ma anche la collaborazione e la partecipazione della comunità nell'ulteriore sviluppo e miglioramento del progetto.
In definitiva, crediamo che l'interprete per il linguaggio HLCostLan rappresenti non solo un'importante realizzazione tecnica, ma anche una risorsa preziosa per la comunità degli sviluppatori e dei ricercatori interessati alla programmazione e all'ottimizzazione dei costi computazionali. Siamo fiduciosi che il nostro lavoro servirà ad offrire una base solida per ulteriori progressi nel campo e che contribuirà a promuovere una maggiore efficienza e qualità nell'ambito dello sviluppo del software.

5.1 Sviluppi futuri

Il prototipo sviluppato, rappresenta un passo cruciale nello sviluppo dell'architettura del sistema descritto attraverso il linguaggio dichiarativo cApp (definito in 1.2.1). L'obiettivo principale di questo sistema è costruire un'infrastruttura che consenta di ottimizzare le risorse dei nodi worker in un sistema di calcolo distribuito. Attualmente, l'interprete HLCostLan costituisce un elemento fondamentale per il calcolo dei costi computazionali dei programmi, ma vi sono numerose possibilità di sviluppo future che potrebbero estendere notevolmente le funzionalità del sistema.
In particolare, l'interprete potrebbe essere integrato in un sistema di orchestrazione di container, come Kubernetes, per distribuire e gestire i programmi scritti in HLCostLan su un cluster di nodi worker. L'obiettivo sarebbe quello di sfruttare al meglio le risorse disponibili, garantendo un'allocazione efficiente e dinamica delle risorse in base al carico computazionale effettivo. cApp, essendo un linguaggio dichiarativo, può essere utilizzato per definire delle funzioni di selezione dei worker più adatti, tenendo conto di fattori come latenza, complessità computazionale e altri parametri rilevanti. Questo consentirebbe di massimizzare l'utilizzo delle risorse, migliorare le prestazioni complessive del sistema e ottimizzare i costi di esecuzione.
Alcuni possibili sviluppi futuri che potrebbero arricchire ulteriormente il nostro lavoro includono:

- Integrazione con Kubernetes: Sviluppare un'interfaccia per integrare il nostro interprete con Kubernetes, consentendo di distribuire e gestire i programmi scritti in HLCostLan su un cluster di nodi worker in modo efficiente e dinamico.
- Estendere il linguaggio HLCostLan: Aggiungere nuovi costrutti e funzionalità al linguaggio HLCostLan per consentire una rappresentazione più dettagliata e precisa del carico computazionale, consentendo agli sviluppatori di esprimere in modo più accurato le esigenze dei loro programmi.
- Implementazione di strategie di allocazione avanzate: Sviluppare algoritmi e strategie di allocazione delle risorse più sofisticati, che tengano conto di una gamma più ampia di fattori e metriche per garantire un utilizzo ottimale delle risorse del cluster.
- Supporto per l'ottimizzazione dinamica: Integrare il sistema con meccanismi di ottimizzazione dinamica, consentendo di adattare le risorse allocate in tempo reale in base alle condizioni del sistema e al carico di lavoro.
- Integrazione con sistemi di monitoraggio e gestione delle prestazioni: Collegare il sistema all'infrastruttura di monitoraggio delle prestazioni, consentendo di rilevare e rispondere dinamicamente ai cambiamenti nelle condizioni del sistema e nell'afflusso di lavoro.
- Valutazione dell'efficacia pratica: Condurre studi sperimentali e valutazioni empiriche per testare l'efficacia del sistema in scenari realistici di utilizzo, confrontandolo con altre soluzioni esistenti e analizzando le prestazioni e l'efficienza complessive.

Siamo convinti che il nostro lavoro servirà agli utenti che si troveranno ad affrontare problemi di gestione delle risorse in sistemi di calcolo distribuito, fornendo loro uno strumento potente ed efficiente per ottimizzare le prestazioni dei loro sistemi e ridurre i costi operativi.

Bibliografia

[1] Ishan Vasishta Bhatt. *A Study on the Temporal Predictability of Serverless Functions*. North Carolina State University, 2022.

[2] Matteo Trentin. "Topology-based Scheduling in Serverless Computing Platforms". URL: http://amslaurea.unibo.it/24930/.

[3] Eric Jonas et al. "Cloud Programming Simplified: A Berkeley View on Serverless Computing". In: *CoRR* abs/1902.03383 (2019). arXiv: 1902.03383. URL: http://arxiv.org/abs/1902.03383.

[4] Giuseppe De Palma et al. "Allocation Priority Policies for Serverless Function-Execution Scheduling Optimisation". In: *Service-Oriented Computing - 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14-17, 2020, Proceedings*. A cura di Eleanna Kafeza et al. Vol. 12571. Lecture Notes in Computer Science. Springer, 2020, pp. 416–430. DOI: 10.1007/978-3-030-65310-1_29. URL: https://doi.org/10.1007/978-3-030-65310-1%5C_29.

[5] Giuseppe De Palma et al. "Serverless Scheduling Policies based on Cost Analysis". In: *Proceedings of the First Workshop on Trends in Configurable Systems Analysis, TiCSA@ETAPS 2023, Paris, France, 23rd April 2023*. A cura di Maurice H. ter Beek e Clemens Dubslaff. Vol. 392. EPTCS. 2023, pp. 40–52. DOI: 10.4204/EPTCS.392.3. URL: https://doi.org/10.4204/EPTCS.392.3.

[6] Giuseppe De Palma et al. "Custom Serverless Function Scheduling Policies: An APP Tutorial". In: *Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2023.

[7] Elvira Albert et al. "Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis". In: *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*. A cura di Maria Alpuente e German Vidal. Vol. 5079.

Lecture Notes in Computer Science. Springer, 2008, pp. 221–237. DOI: 10.1007/978-3-540-69166-2_15. URL: https://doi.org/10.1007/978-3-540-69166-2%5C_15.

[8] Antonio Flores-Montoya e Reiner Hähnle. "Resource Analysis of Complex Programs with Cost Equations". In: *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*. A cura di Jacques Garrigue. Vol. 8858. Lecture Notes in Computer Science. Springer, 2014, pp. 275–295. DOI: 10.1007/978-3-319-12736-1_15. URL: https://doi.org/10.1007/978-3-319-12736-1%5C_15.

[9] Stefano Pascali. "Definizione di una grammatica per il linguaggio Jolie". Tesi di dott. URL: http://amslaurea.unibo.it/2372/.

[10] Torben Ægidius Mogensen. *Basics of compiler design*. 2010.

[11] Terence John Parr e Russell W. Quong. "ANTLR: A Predicated- *LL(k)* Parser Generator". In: *Softw. Pract. Exp.* 25.7 (1995), pp. 789–810. DOI: 10.1002/SPE.4380250705. URL: https://doi.org/10.1002/spe.4380250705.

[12] Alfred V. Aho, Ravi Sethi e Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. ISBN: 0-201-10088-6. URL: https://www.worldcat.org/oclc/12285707.

[13] Christoph Haase. "Subclasses of Presburger Arithmetic and the Weak EXP Hierarchy". In: *CoRR* abs/1401.5266 (2014). arXiv: 1401.5266. URL: http://arxiv.org/abs/1401.5266.

[14] Elvira Albert et al. "Closed-form upper bounds in static cost analysis". In: *Journal of automated reasoning* 46 (2011), pp. 161–203.

[15] Sara Bergonzoni. "Strumenti per l'analisi del tempo di esecuzione". Tesi di dott. URL: http://amslaurea.unibo.it/3135/.

[16] *WebAssembly Interface Types: Interoperate with All the Things!* URL: https://hacks.mozilla.org/2019/08/webassembly-interface-types/.

[17] *WebAssembly Doc*. URL: https://developer.mozilla.org/en-US/docs/WebAssembly.

[18] Shashank Mohan Jain e Shashank Mohan Jain. "WebAssembly Text Toolkit and Other Utilities". In: *WebAssembly for Cloud: A Basic Guide for Wasm-Based Cloud Apps* (2022), pp. 33–55.

[19] Marco Aspromonte. "Studio del WebAssembly e sperimentazioni con Emscripten". Tesi di dott. URL: http://amslaurea.unibo.it/20464/.

[20] Vili-Petteri Niemelä. "WebAssembly, Fourth Language in the Web". In: (2021).

[21] Ben L. Titzer. "A fast in-place interpreter for WebAssembly". In: *CoRR* abs/2205.01183 (2022). DOI: 10.48550/ARXIV.2205.01183. arXiv: 2205.01183. URL: https://doi.org/10.48550/arXiv.2205.01183.