Indice

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 fornite come servizi virtuali da parte di fornitori specializzati noti come provider di cloud(Amazon,Google ecc.). Questi 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, nonchè 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 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 gesti-

re 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. Alcuni esempi di FaaS li troviamo in AWS Lambda, Google Cloud Functions, Azure Functions.

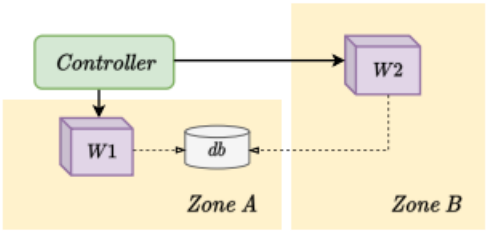
Serverless Computing

Serverless computing è un paradigma più ampio del FaaS in cui il fornitore di servizi cloud gestisce completamente l'infrastruttura, mentre FaaS è un sottoinsieme del Serverless computing che si concentra sulla gestione delle funzioni come entità indipendenti. Le funzioni vengono eseguite in modo scalabile e senza la necessità di gestire l'infrastruttura sottostante. [5]

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.

1.1.5 cApp

Partiamo da un linguaggio dichiarativo APP(*Allocation Priority Policies*) che ci permette di definire politiche di allocazione delle risorse per le funzioni serverless, derivato da un estensione dello scheduler di OpenWhisk. 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. [2]. Vediamo qui di seguito un esempio del funzionamento:




```
- db_query:
  - workers:
    - wrk: W1
    - wrk: W2
    strategy: best_first
```

Figura 1.1: Esempio di funzionamento di un sistema serverless [3]

A destra troviamo il codice dichiarativo App che fornisce le regole per allocare le risorse: 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, adottando procedure automatiche per definire le politiche di allocazione delle risorse basate sulle informazioni derivate dall'analisi statica delle funzioni che dovranno essere eseguite. Osserviamo adesso un'altro esempio andando ancor di più in profondità: Data una grammatica definità nel paper[3], 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 dovremmo assumere che il ramo condizionale prenderà il massimo della latenza tra le due chiamate, non avremo certezza che sia uno o l'altro e in questo caso dovremmo cercare di andare a ridurre entrambe.

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

```
    }
```

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

Mentre in questo caso 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*.

```
    (jobs, m, r)=>{
        for(i in range(0,m)){
            call Map(jobs, i)
            for(j in range(0,r)){
                call Reduce(jobs, i, j)
            }
        }
    }
```

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 x r volte la latenza di *Reduce*.

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 del-

l'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 puo essere ottenuta in due fasi: Calcolando i programmi di costo dal Solver e eseguendoli in ogni worker, e scegliendo il worker che ha latenza minima per contattare il *PremiumService*. Invece nel caso del Listing 1.5 invece 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.[3]

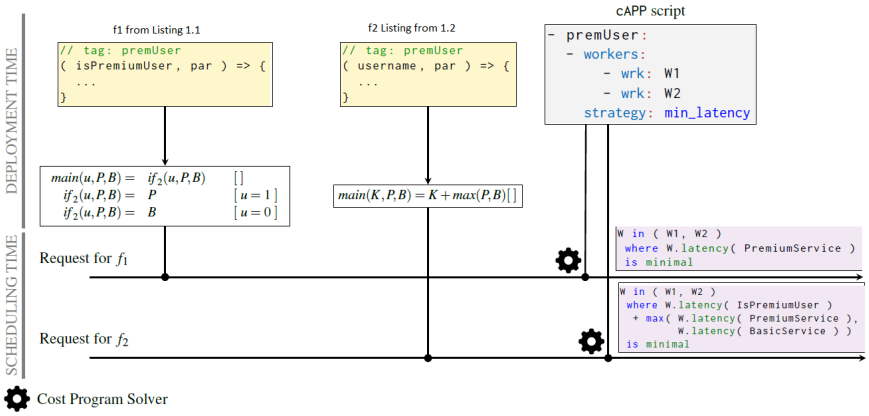


Figura 1.2: From Deploy to Scheduling for Listing 1.1 e 1.2

Obiettivo della tesi

In questa tesi ci concetreremo sulle tecniche che ci permettono, data una funzione scritta in un linguaggio con una grammatica che definiremo nei capitoli successivi, di analizzarne la semantica e andare a generare le equazioni di costo. Al fine di trarre informazioni quali la complessità (come upper bound di una funzione) al fine di poter ottimizzare lo scheduling dell'esecuzioni delle funzioni serverless una volta richieste. Procederemo andando a definire una grammatica per il linguaggio, in seguito svilupperemo un interprete per il linguaggio che analizza un programma scritto in un linguaggio ad alto livello restituendo le equazioni di costo per ogni funzione. In seguito andremo ad analizzare le equazioni di costo per ogni funzione, attraverso tool(PUBS [1] e CoFloCo[4]) specifici che data un equazione di costo ci definisce l'upper bound della una funzione.

Capitolo 2

Linguaggio e semantica

In questo capitolo introdurremo 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. Per fare un ripasso, un linguaggio viene definito da una grammatica $G = (N, T, \rightarrow, S)$ dove:

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

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

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

e T^* é la chiusura di Kleene.
Questa grammatica (che descriveremo nel dettaglio nel capitolo successivo), é riconosciuta dal plugin ANTLR che ci permette di generare un parser per il linguaggio. Il parser generato da ANTLR é un parser LL(*), ovvero un parser che riconosce linguaggi non ambigui e che non richiede backtracking. Questo ci permette di avere un parser efficiente ottimizzando il riconoscimento del linguaggio. Left to right, Leftmost derivation, * lookahead symbols

2.1 Grammatica

2.1.1 Grammatica del linguaggio

Di seguito troviamo la grammatica del linguaggio **HLCostLan** che descrive il linguaggio che andremo ad utilizzare per descrivere i programmi.
Verranno riportate le produzioni della grammatica che descrivono il linguaggio, mentre per visualizzare il file g4 nella sua totalitá si rimanda alla Repo

```
1  prg : complexType* serviceDecl* functionDecl* init;
2
3  init: '('formalParams? ')''='>' '{' stm '}';
4
5  serviceDecl:'service'ID':''('(type(','type)*)?')''→
       ''type';
6
7  functionDecl:'fn'ID'('formalParams?')''→'(type)'{'
       stm'}' ;
8
9  stm :
10     | serviceCall
11     |'if''('expOrCall')''{'stm'}''else''{'stm'}'
12     |'for''('ID'in''('NUMBER','exp ')'')''{'stm'}'
13     | letIn
14     | functionCall
15     | 'return' expPlus ;
16
17 serviceCall:'call'ID'('(exp(','exp)*)?')'(';'stm)?;
18
19 functionCall : ID'('( exp (','exp)* )? ')';
20
21 letIn: 'let' (ID '=' expPlus)+ 'in' stm;
22
23
24
```

Listing 2.1: Grammatica del linguaggio HLCostLan

Il non terminale **prg** e' il terminale iniziale della grammatica, e descrive un programma. Un programma e' 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 e' 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 e' composta da una sequenza di parametri formali, e da una istruzioni.
La **serviceDecl** descrive la dichiarazione di un servizio, che deve essere dichiarato una sola volta. Un servizio e' composto da un nome, una sequenza di parametri formali, e un tipo di ritorno.
La **functionDecl** descrive la dichiarazione di una funzione, una funzione e' 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 corretti.

2.2 Another Tool for Language Recognition

ANTLR (Another Tool for Language Recognition) è uno strumento potente e flessibile per l'analisi di 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 a un altro. Questo è utile per la migrazione del codice, la 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, che avrá estensione *.g4*, andrá a generare una cartella *gen* contenente i file necessari per generare il parser.
La cartella *gen* il codice sorgente del parser e lexel, generati automaticamente; é 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 sostanzialmente 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, andrá a riconoscere la grammatica, e generare un albero di parsing(noto anche come albero di derivazione), 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.

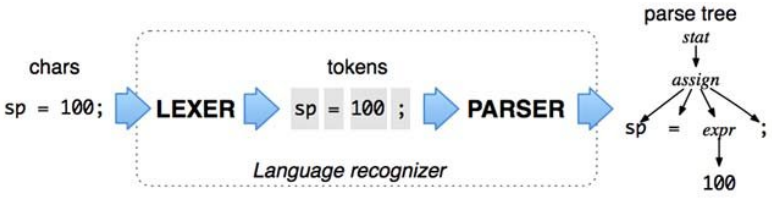


Figura 2.1: Funzione di parsing e lexing

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) é un albero che rappresenta il significato del programma, e viene utilizzato per eseguire il programma. Mentre se volessimo, come nel nostro caso andare a costruire un albero di sintassi astratta, dovremmo andare ad estendere l'interfaccia *BaseListener* e implementare i metodi che ci interessano.

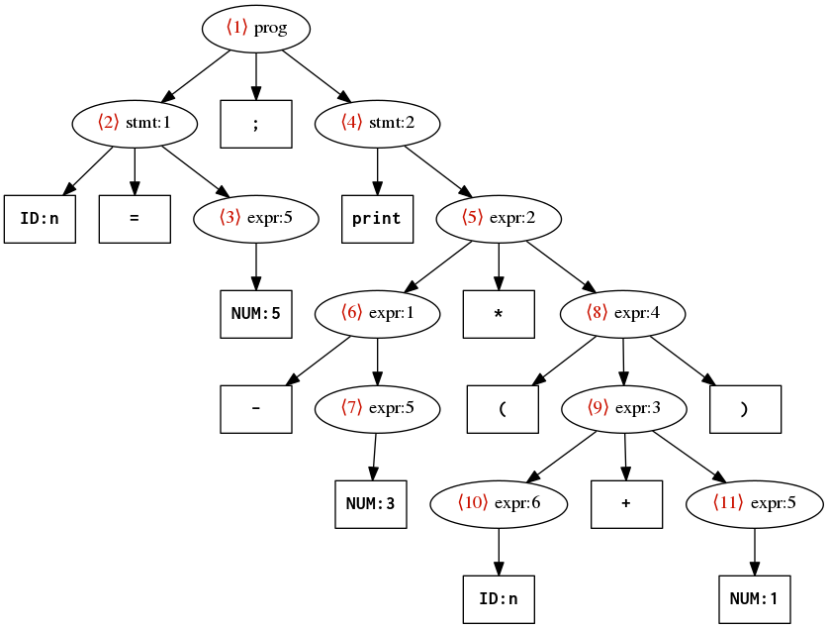


Figura 2.2: Esempio di albero di AST

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 2.1.1, il non terminale **prg** e' il terminale iniziale della grammatica, in questo caso prg, prende un parametro len di tipo intero, ed avrá 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 iterazione 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 é il nostro caso perché la grammatica HLCostLan non lo permette) dovremmo andare a controllare che la variabile assegnata sia stata dichiarata altrimenti dovrá generare un errore. Nel nostro compilatore effettuiamo altri controlli semantici come:

- 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 giá stati precedentemente dichiarati
- Controllare che i tipi di ritorno delle funzioni e dei servizi siano corretti.

Inoltre effettuiamo il type checking, che é 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.

Capitolo 3

CostCompiler

CostCompiler é un interprete per il linguaggio di programmazione definito nel capitolo precedente Grammatica 2.1.1. 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). Dopo aver generato l'albero di sintassi astratta,CostCompiler si della verifica Semantica 2.3 del linguaggio andando ad effettuare dei controlli semantici e di tipo sul programma in input, andando a garantire alcune invarianti.

Una volta effettuata la verifica semantica, CostCompiler procede con la generazione delle equazioni di costo, andando a visitare l'AST secondo determinati criteri, e che ad ogni nodo figlio verrá mandato una Map che contiene la mappatura di ogni variabile in una stringa che sará la stessa stringa che apparirá nelle equazioni di costo.
Ogni nodo figlio tornerá al padre attraverso la funzione *toEquation()* che ritorna una stringa, rappresentante l'equazione di costo del nodo figlio, e il padre andrá a concatenare le stringhe dei figli(anche in base al tipo di figlio da cui ricevere l'equazione), ad esempio la *return* $<EXP>$ sará 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.txt* inoltre lancerá il risolutore PUBS 3.3 che andrá 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:

```
1     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 verrá chiamata *if9* con parametri *ISPREMIUMUSER0,P,B*. P e B sará il costo costante delle chiamate ai servizi *isPremiumUser* e *BasicService*, mentre *ISPREMIUMUSER0* sará la valutazione del parametro *isPremiumUser* che sará 1 se sará vero, 0 altrimenti; in altri termini *ISPREMIUMUSER0* sará la valutazione della guardia del costrutto *if-then-else* e verrá eseguita la chiamata al servizio *PremiumService* se *ISPREMIUMUSER0* sará 1 con costo $nat(B)$, altrimenti verrá 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.txt*, 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 3.3) 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 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):

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

  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 *varphi* é 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 é 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 é una variabile.

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

I giudizi avranno forma:

- $\Gamma \vdash e : Nat$ che significa che il valore di E in e é rappresentato dalla costEspression E

- $\Gamma \vdash S : e; C; Q$ significa che il costo di S nell'ambiente Γ é $e + C$ dato un insieme di equazioni Q

Riassumiamo le regole descritte in precedenza:

- Regola[call] gestisce l'invocazione di un servizio; il costo della call sará il costo di S piú il costo per l'accesso al servizio h
- Regola[if] gestisce il costrutto condizionale;quando la guardi é un'espressione definita in aritmetica di Presburger e il costo verrá rappresentato da entrambi i rami con i due condizionali φ e $\not\varphi$.
  Rappresentiamo a livello di equazione if_l dove l é la linea di codice dové inizia il costrutto.
- Regola [for] descritto all'interno del rispettivo frammento di codice come for_l per lo stesso motivo citato in precedenza; Definiamo 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 é presente E e andiamo a valutare $\Gamma \vdash S$ che ritornerá un'equazione Q' con costo C.

3.2 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 conterrá il metodo *toEquation()* che prende come argomento la variabile che nostro ambiente Γ e sará appunto un dizionario. Questo dizionario di tipo $EnvVar$ é 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 peró non sempre é 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, che sará un stringa riportata all'interno dell'equazione di costo del padre.

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

    }
```

La sezione $e.get(this)$ ritorna la variabile mappata per quel determinato nodo, che sará una stringa che rappresenta la variabile all'interno dell'equazione di costo, mentre la sezione $stm.toEquation(e)$ é la chiamata sul metodo *toEquation()* del nodo figlio, che restituirá la stringa che rappresenta l'equazione di costo del nodo figlio.

Per avere una panoramica completa del processo di generazione delle equazioni di costo, riportiamo il frammento di codice di $toEquation()$ del $programNode$:

```
    public String toEquation(EnvVar e){

        for (Node n : decServices){
            n.checkVarEQ(e);
        }
        StringBuilder equ = new StringBuilder();
        for(Node n : funDec){
            equ.append(n.toEquation(e));
        }
        equ.append(main.toEquation(e));
        return equ.toString();
    }
```

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.

3.3 PUBS

Pubs é un risolutore di vincoli di costo, che prende in input un file di equazioni di costo e restituisce un file con i limiti superiori per ogni relazione di costo. PUBS (Pratical Upper Bounds Solver) ha l'obiettivo 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. 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 3.1: Esempio di output PUBS su Listing 1

Come vediamo nell'immagine sopra, PUBS ci restituisce un analisi dell'intera equazione (pub_aux_entry) e delle singole funzioni da cui essa dipende, in questo caso $pubs\_aux\_if9$ 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 sará il massimo tra i due. Inoltre non abbiamo la presenza di cicli, quindi PUBS non ci restituisce nessun invariante di ciclo.

In PUBS peró abbiamo una grammatica da rispettare, tenuta in considerazione da ogni equazione di costo generata da CostCompiler, che é la seguente:

```
1  <equation> ::= eq(Head,costExpression,[listOfCall],[
   ListOfSizeRelation]).
2  <Head> ::= Name | Name(Par).
3  <costExpression> ::= nat(<variable>)
4                     | <costExpression> + <costExpression>
5                     | <costExpression> - <costExpression>
```

```
6                     | <costExpression> * <costExpression>
7                     | max(<costExpression>,<costExpression>).
8
9     <listOfCall> ::= [] | <call> <listOfCall>.
10    <call> ::= <function>(<listOfParameters>).
11    <listOfParameters> ::= [] | <variable> <listOfParameters
    >.
```

Listing 3.3: Grammatica PUBS

Dove ¡Head¿ é il nome della entry che andremo ad analizzare insieme a i suoi parametri. CostExpression é l'espressione di costo che rappresenta il costo della entry, come notiamo rispetta la grammatica della aritmetica di Presburger. ListOfCall é la lista delle chiamate alle altre entry, che saranno rappresentate come ¡call¿ e ¡listOfCall¿ sará la lista di queste chiamate; in questo modo PUBS riesce a costruire un grafo delle dipendenze tra le entry. Infine abbiamo ¡listOfSizeRelation¿ che sará la lista delle relazioni di costo che dipendono dalla entry che stiamo analizzando, e che PUBS andrá a calcolare. Riportiamo Listing1 come esempio di equazione di costo generata da CostCompiler, dove troviamo l'analisi alla pagine precedente:

```
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 3.4: 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 andrá a chiamare la funzione $for3$ inserendo un'ulteriore parametro che sará il counter del ciclo con parametro 0 e B, che avrá costo 0 in caso $M >= 10 + 1$ altrimenti avrá costo $nat(B)$.

Capitolo 4

Conclusioni

4.1 Sviluppi futuri

Bibliografia

[1] 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 15*. Springer. 2008, pp. 221–237.

[2] 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 18*. Springer. 2020, pp. 416–430.

[3] Giuseppe De Palma et al. "Serverless Scheduling Policies based on Cost Analysis". In: ().

[4] 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, Singapore, November 17-19, 2014, Proceedings 12*. Springer. 2014, pp. 275–295.

[5] Matteo Trentin. "Topology-based Scheduling in Serverless Computing Platforms". Tesi di dott. URL: `http://amslaurea.unibo.it/24930/`.