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.
Laneve Cosimo

Correlatore:
Dott.
Matteo Trentin

Presentata da:
Simone Boldrini

Sessione III
Anno Accademico 2022-23

Indice

1 Introduzione **4**
- 1.1 Cloud Computing . . . 4
  - 1.1.1 Infrastructure as a Service . . . 4
  - 1.1.2 Platform as a Service . . . 5
  - 1.1.3 Software as a Service . . . 5
  - 1.1.4 Function As a Service . . . 5
  - 1.1.5 cApp . . . 6

2 Linguaggio e semantica **11**
- 2.1 Grammatica . . . 12
  - 2.1.1 Grammatica del linguaggio . . . 12
- 2.2 Another Tool for Language Recognition . . . 13
- 2.3 Semantica del Linguaggio . . . 16

3 CostCompiler **17**
- 3.1 Regole di Inferenza . . . 19
- 3.2 Generazione delle Equazioni di costo . . . 21
- 3.3 PUBS . . . 22
  - 3.3.1 Relazione di Costo . . . 22
  - 3.3.2 Stima del costo per Nodo . . . 24
  - 3.3.3 PUBS in pratica . . . 25

4 Generazione di WebAssembly **29**
- 4.1 Introduzione WebAssembly . . . 29
- 4.2 WebAssembly Text Format . . . 30
- 4.3 CostCompiler to WAT . . . 32
- 4.4 Esecuzione del modulo WebAssembly . . . 35

5 Conclusioni **37**
- 5.1 Sviluppi futuri . . . 38

Bibliografia **39**

Indice di Codice

1.1 La guardia della condizione é un espressione 7
1.2 La guardia della condizione é un'invocazione a un servizio esterno 7
1.3 Funzione con logica Map-Reduce 8
1.4 cAPP for Listing 1.1 e 1.2 8
1.5 cApp for Listing 1.3 8
2.1 Grammatica del linguaggio HLCostLan 12
2.2 Esempio di codice HLCostLan: example/Listing6 15
3.1 Listing8 18
3.2 Equazioni di costo per Listing8 18
3.3 toEquation() del ProgramNode 22
3.4 Listing 1 25
3.5 Grammatica PUBS 26
3.6 Listing 6 27
3.7 Equazione di costo PUBS per Listing6 27
4.1 Esempio di funzione in wat 31
4.2 Esempio di funzione in wat 31
4.3 codeGeneration() per l'if Node 32
4.4 codeGeneration() per il for Node 33
4.5 Esecuzione del modulo WebAssembly 35

Capitolo 1

Introduzione

Il cloud computing é 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 flessibilitá.

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) é 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 peculiaritá 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) é 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, cioé fuori del computer fisico o dalla LAN locale, spesso attraverso un server web; l'utente puó 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 ció che effettivamente utilizzano. Questo modello di pricing basato su abbonamento o utilizzo puó essere vantaggioso per le aziende, in quanto consentono di evitare costi iniziali elevati e di prevedere meglio i costi operativi. Alcuni esempi piú comuni di SaaS li troviamo in Google Workspace, Microsoft Office 365, Dropbox.

1.1.4 Function As a Service

Function as a Service(FaaS) é 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'attivitá specifica e puó essere attivata in risposta a eventi o richieste specifiche. Ad esempio, una funzione puó 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 é 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.

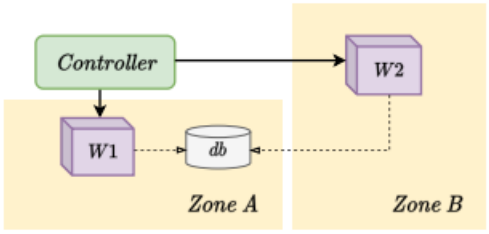
Serverless Computing

Serverless computing 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 é un sottoinsieme del Serverless computing che si concentra specificamente sulla gestione delle singole funzioni come entitá indipendenti. In questo contesto, le funzioni vengono eseguite in modo scalabile e senza che l'utente debba preoccuparsi della gestione diretta dell'infrastruttura sottostante. [7]

Nel serverless computing, il provider cloud si occuperá di allocare le risorse necessarie per eseguire il codice e di deallocarle una volta terminata l'esecuzione. Questo facilita la distribuzione e la scalabilitá del sistema, cercando di automatizzare quest'ultima fase. Focalizzandoci in questa gestione delle risorse, viene introdotto un linguaggio cApp che ci permetterá 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 é un estensione di App, dove le politiche di schedulazioni delle funzioni dipendono dai costi associati alle possibili esecuzioni delle funzioni sui worker disponibili. [5]. 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 [6]

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 peró 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 piú in profonditá: Data una grammatica definita nel paper[6], 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 é un espressione

Il parametro *isPremiumUser* é 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 verrá 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      }
```

```
    }
```

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

Nel caso sotto riportato invece, se l'utente che scatena l'evento é un membro premium, il tempo di esecuzione previsto dalla funzione é la somma delle latenze delle invocazioni dei servizi *isPremiumUser* e *PremiumService*. Quindi possiamo preventivare il ritardo atteso(come tempo di esecuzione peggiore), cioé la somma della latenza del servizio *isPremiumUser* piú 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 é indicato dal parametro m. La fase di *Map* che genera m sottotask "ridotti" é 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 é data dalla somma di m volte la latenza di *Map* e m x r volte la latenza di *Reduce*.

cApp é 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 possibilitá di allocare la funzione *premUser* su due workers, e la strategia di allocazione é quella di minimizzare la latenza, ovvero di dare prioritá al worker su cui la soluzione del-

l'espressione di costo é minima. Illustriamo peró 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, 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 é 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.[6]

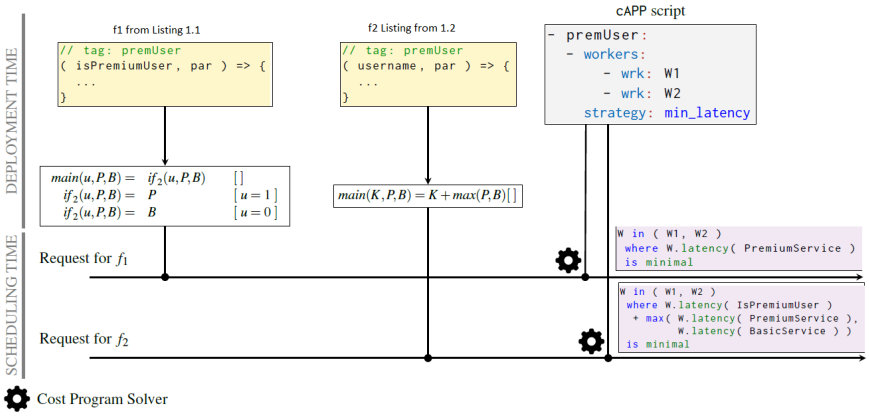


Figura 1.2: From Deploy to Scheduling for Listing 1.1 e 1.2

Obiettivo della tesi

L'obiettivo della tesi è quello di concentrarsi sullo studio delle tecniche che consentono di analizzare la semantica di una funzione scritta in un linguaggio specifico, il quale verrà definito in dettaglio nei capitoli successivi. Questa analisi della semantica è fondamentale per generare le equazioni di costo associate a ciascuna funzione. Le equazioni di costo forniscono informazioni cruciali come la complessità, che rappresenta un upper bound della funzione in questione. Questi dati sono essenziali per ottimizzare lo scheduling dell'esecuzione delle funzioni serverless una volta richieste, consentendo di allocare risorse in modo efficiente e di ridurre i tempi di esecuzione.

Il percorso di ricerca prevede diverse fasi. Inizieremo definendo una grammatica specifica per il linguaggio di programmazione considerato. Successivamente, svilupperemo un interprete per tale linguaggio, il quale analizzerà un programma scritto in un linguaggio ad alto livello e restituirà le equazioni di costo associate a ciascuna funzione. In aggiunta, sarà generato il codice WebAssembly corrispondente, che potrà essere eseguito in un ambiente serverless.

Una volta ottenute le equazioni di costo, procederemo all'analisi dettagliata di ciascuna funzione attraverso l'utilizzo di tool specifici come PUBS (Publications on Upper Bounds Synthesis) e CoFloCo (Cost Flow Complexity Analysis). Questi strumenti, citati rispettivamente in letteratura [1] e [3], sono progettati per calcolare l'upper bound delle funzioni date le loro equazioni di costo. Tale analisi ci fornirà una comprensione più approfondita delle prestazioni delle funzioni, consentendoci di ottimizzare ulteriormente l'allocazione delle risorse e migliorare il processo di scheduling delle esecuzioni.

Capitolo 2

Linguaggio e semantica

Un linguaggio di programmazione é 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 é stato pensato per essere comprensibile e allo stesso tempo espressivo, in modo da poter descrivere programmi complessi. La semantica é 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 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.
La 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, 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 utilizzeró per descrivere i programmi.
Riporto di seguito le produzioni della grammatica che descrivono il linguaggio, mentre per visualizzare il file g4 nella sua totalitá si rimanda alla Repository del progetto.

```
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) é uno strumento potente e flessibile per l'analisi di linguaggi di programmazione, linguaggi di markup e dati strutturati. é 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 é 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 é utile per la migrazione del codice, la refactoring e altre attivitá 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.
La cartella *gen* contiene 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 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.

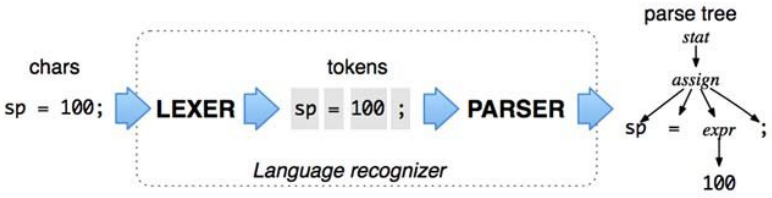


Figura 2.1: Funzione di parsing e lexing

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. 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.

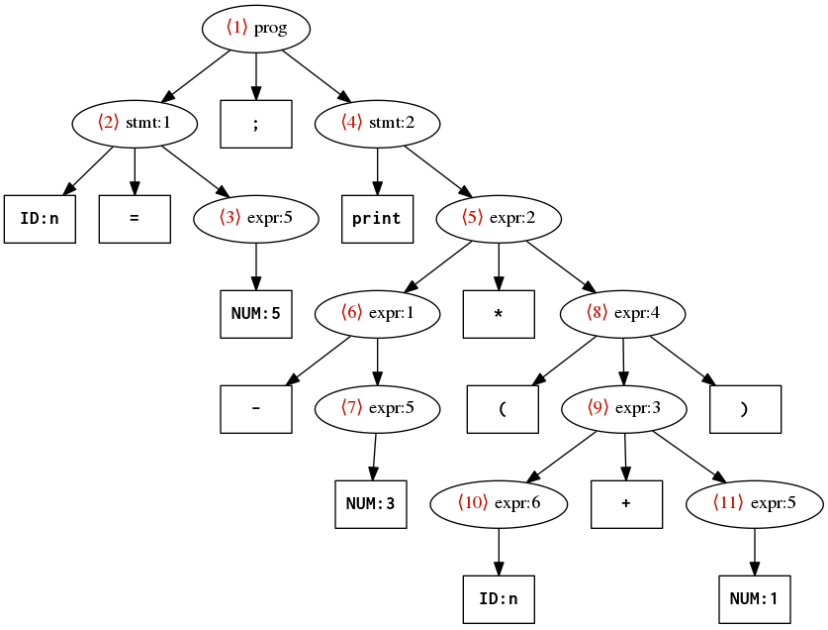


Figura 2.2: Esempio di albero di AST

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 é 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.
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 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) i rispettivi sottonodi(nel caso di *IfNode* conterrá 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 (eq. 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 verrá passata una Mappa che contiene la mappatura di ogni variabile in una stringa che sará 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* $<EXP>$ sará diverso dal *return* <function(Par) >.

Il risultato finale di questo processo di concatenazione attraverso determinati nodi dell'AST é la generazione delle equazioni di costo. Una volta generate le equazioni di costo, CostCompiler le stampa in un file *equation.txt* inoltre lancia il risolutore PUBS 3.3 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 verrá chiamata *if9* con parametri *ISPREMIUMUSER0,P,B*.
P e B sono il costo costante delle chiamate ai servizi *isPremiumUser* e *BasicService*, mentre *ISPREMIUMUSER0* sará la valutazione del parametro *isPremiumUser* che sará 1 se 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 $nat(P)$, 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) é 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 é 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 é una variabile, q é una costante intera, $e_1, \ldots, e_n$ sono espressioni aritmetiche e max é 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 hanno 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

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

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

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

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

(3.6)

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 contiene il metodo *toEquation()* che prende come argomento la variabile del 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. Questa sottostringa sará 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 $stm.toEquation(e)$ é la chiamata sul metodo *toEquation()* del nodo figlio, che restituirá 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

andrá a richiamare il metodo *toEquation*() su tutti i nodi figli e andrá a concatenare le stringhe risultanti al fine di generare l'equazione finale.

```
1     public String toEquation(EnvVar e){
2
3         for (Node n : decServices){
4             n.checkVarEQ(e);
5         }
6         StringBuilder equ = new StringBuilder();
7         for(Node n : funDec){
8             equ.append(n.toEquation(e));
9         }
10        equ.append(main.toEquation(e));
11        return equ.toString();
12    }
```

Listing 3.3: 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.

3.3 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.

3.3.1 Relazione di Costo

Un'espressione di costo di base é 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 *é un'espressione lineare,* S *é un insieme non vuoto di espressioni di costo,* $nat : \mathbb{Z} \to \mathbb{Q}^+$ *é 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 espressione di costo di base godono di due proprietá:

- Sono sempre valutate per valori non negativi
- Rimpiazzando una sottoespressione $nat(l)$ con $nat(l')$ tale che $l \geq l'$, il risultato é 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 *é 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 *é 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 é 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 supponre 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 é della forma $C(\overline{v})$ dove $C \in rel(S)$ e $\overline{v}$ sono valori interi. Le *call* sono valutate in 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* $(C(\overline{v}); e; \overline{t})$ *é un albero di evoluzione per* $C(\overline{v})$ *in* S*, indicato con* $Tree(C(\overline{v}, S))$*, se:*

1. *c'é 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* $\varphi[vars(exp)/\overline{w}, \overline{y_i}/\overline{v_i}]$ *é soddisfacibile in* $\mathbb{Z}$
3. $e = exp[vars(exp)/\overline{w}]$, T_i *é un albero di evoluzione* $tree(D_i(\overline{v_i}, S))$ *with* $i \in 0, \ldots, k$

Nel processo di risoluzione di $C(\overline{v})$ notiamo che 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 $C(\overline{v})$.

3.3.2 Stima del costo per Nodo

Notiamo che 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), contiene 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 $Loops(C)$ Logicamente, se é 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 é possibile applicare il ciclo a uno o piú 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 uno limite superiore su Ψ'. [1][2]

3.3.3 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 3.4: Listing 1

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 3.5: Grammatica PUBS

Dove <Head> é il nome della entry che andremo ad analizzare insieme ai suoi parametri. CostExpression é l'espressione di costo che rappresenta il costo della entry e rispetta la grammatica della aritmetica di Presburger. ListOfCall é 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 sará la lista delle relazioni di costo che dipendono dalla entry che stiamo analizzando, e che PUBS andrá a calcolare. Riportiamo un'altro esempio di equazione di costo generata da CostCompiler, questa volta per il programma scritto in Listing6:

```
1    service BasicService: (int) -> void;
2    fn svc(i: int) -> void{
3        for(m in (0,10)){
4            call BasicService(i)
5        }
6    }
7    (len : int) => {
8        svc(len)
9    }
```

Listing 3.6: Listing 6

Come vediamo, la funzione init chiamerá la funzione svc con parametro len, che a sua volta chiamerá la funzione BasicService per 10 volte, quindi il costo del programma sará l'invocazione della funzione $svc{+}10{\cdot}nat(B)$, dove $nat(B)$ é l'invocazione del servizio *BasicService*. L'equazione di costo risultante sará 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 3.7: 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 $nat(B)$. 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 3.2: 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 compilatore 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 compilatore 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 il compilatore 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 del compilatore, 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.

4.1 Introduzione WebAssembly

WebAssembly (Wasm) è un formato di istruzioni altamente performante, portabile e sicuro, progettato per essere eseguito in ambienti virtuali. Origi-

nariamente 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, consentendo l'esecuzione di applicazioni web e non solo in modo efficiente, sicuro e indipendente dalla piattaforma di destinazione.[8]

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. Il compilatore che abbiamo sviluppato generiamo un file .wat ed in seguito il tool wat2wasm [4] genera il file .wasm, che a sua volta potrá essere eseguito da un motore di runtime.

I tipi di dati 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 é definita in termini di Stack-Machine, dove l'idea generale é 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. Per eseguire la **chiamata della funzione** precedente, 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).

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

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

4.3 CostCompiler to WAT

Il compilatore che abbiamo sviluppato genera un file .wat, questo file .wat andrá poi convertito in un file .wasm, che a sua volta potrá essere eseguito da un motore di runtime. Questa conversione viene fatta tramite il tool wat2wasm [4]. Attraverso il comando:

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

Il tool wat2wasm prende in input un file .wat e genera un file .wasm. Andando piú nel dettaglio di come viene generato il file .wat dal compilatore, 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 verrá 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() {
```

```
        return "(local $res i32)\n" +
                "(if"+exp.codeGeneration()+
                "(then\n"+stmT.codeGeneration()+
                "(local.set $res)" +
                "\n)" +
                "(else\n"+stmF.codeGeneration()+
                "(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. 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 é 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 ció 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() {
        return  "(local $"+id+" i32)\n" +
                "(local $"+id+"_max i32)\n" +
                "(local $res i32)\n" +
                exp.codeGeneration() +
                "(local.set $"+id+"_max)\n" +
                "(loop $for"+line+"\n" +
                "(if (i32.lt_u (local.get $"+id+")(local.get
$"+id+"_max) )\n"+
```

```
10                  "(then"
11                  + stm.codeGeneration()
12                  +"(local.set $res)\n"
13                  +"(local.get $"+id+"\n)" +
14                  "(i32.const 1)\n" +
15                  "(i32.add)\n" +
16                  "(local.set $"+id+")\n" +
17                  "(br $for"+line +")\n)"  +
18                  "(else\n" +
19                  "(local.get $"+id+"_max)\n" +
20                  "(local.set $"+id+"))\n))\n +
21                  (local.get $res)" ;
22
23      }
```

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 servirá 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, verrá 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 finché il valore della variabile locale $id_max é minore del valore della variabile locale $id. Questo é 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 é 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, verrá incrementato di 1, e verrá salvato nella variabile locale $id, con l'istruzione *(local.set $id)* e salta alla label definita in precedenza*(br $for+line)*.

4.4 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 é possibile vedere il file .wasm caricato, e debuggarlo come vediamo in questo snippet:

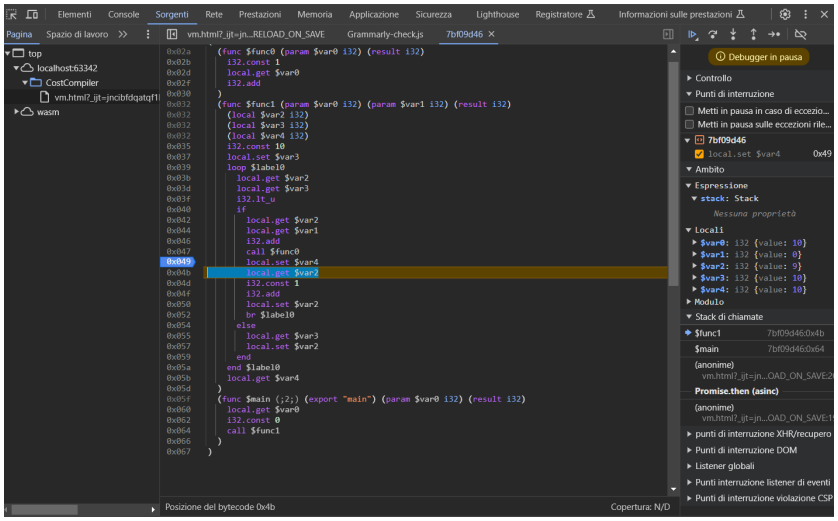


Figura 4.1: Debug del file output.wasm

Come notiamo il file .wasm é 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 é molto piú 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

In conclusione, il nostro progetto ha conseguito pienamente 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. Questo linguaggio 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.

Questo interprete non solo è in grado di analizzare e 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, il nostro interprete non si limita solo ad analizzare il linguaggio HLCostLan e a calcolare le equazioni di costo dei programmi, ma offre anche 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 critica 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 il nostro 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 del nostro interprete liberamente accessibile su GitHub. Questo rende 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 il nostro 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 continuerà a servire da 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 nostro prototipo rappresenta un passo cruciale nello sviluppo dell'architettura del sistema descritto attraverso il linguaggio dichiarativo cApp (definito in 1.1.5). 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, il nostro 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, il nostro 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 esplorando queste direzioni di sviluppo futuro, potremo contribuire in modo significativo al progresso nel campo della gestione delle risorse in sistemi di calcolo distribuito, fornendo agli utenti uno strumento potente ed efficiente per ottimizzare le prestazioni dei loro sistemi e ridurre i costi operativi.

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*. 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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