Capitolo 1

CostCompiler

CostCompiler é un interprete per il linguaggio di programmazione definito nel capitolo precedente Grammatica **??**. 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 **??** 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 *getEquationCost()* una stringa che rappresenta 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 interpretazione è la generazione delle equazioni di costo. Queste equazioni rappresentano una stima del costo computazionale del programma, permettendo così agli sviluppatori di avere un'idea del rendimento del loro codice. Riportiamo un esempio di equazione di costo generata da CostCompiler:

```
1     struct Params {
2         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 1.1: 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 1.2: 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 PUBS1.2 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.

1.0.1 Regole di Inferenza

1.1 Generazione delle Equazioni di costo

1.2 PUBS