Capitolo 1

CostCompiler

CostCompiler è 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) i rispettivi sottonodi(nel caso di *IfNode* conterrà la guardia condizionale e i due statement). Dopo aver generato l'albero di sintassi astratta, CostCompiler si occupa della verifica Semantica **??** del linguaggio andando ad effettuare i controlli semantici e di tipo sul programma in input, andando a garantire alcune invarianti (eg. le chiamate di funzioni devono rispettare i tipi di ritorno).
Una volta effettuata la verifica semantica, CostCompiler procede con la generazione delle equazioni di costo, andando a visitare l'AST secondo determinati criteri, ad ogni nodo figlio verrà passata una Mappa che contiene la mappatura di ogni variabile in una stringa che sarà la stessa stringa che compare nelle equazioni di costo.
Ogni nodo figlio invocato attraverso la funzione *toEquation()* ritorna una stringa, rappresentante l'equazione di costo del nodo figlio, e il padre va a concatenare le stringhe dei figli(anche in base al tipo di figlio da cui ricevere l'equazione), ad esempio la *return* $<EXP>$ sarà diverso dal *return* $<\text{function}(Par)>$.
Il risultato finale di questo processo di concatenazione attraverso determinati nodi dell'AST è la generazione delle equazioni di costo. Una volta generate le equazioni di costo, CostCompiler le stampa in un file *equation.ces*, inoltre

lancia il risolutore PUBS 1.4 che va a calcolare gli upper bound del programma da stampare a video. Riportiamo un esempio di equazione di costo generata da CostCompiler dato un programma scritto in HLCostLang:

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

Listing 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 verrà chiamata *if9* con parametri *ISPREMIUMUSER0,P,B*.
P e B sono il costo costante delle chiamate ai servizi *isPremiumUser* e *BasicService*, mentre *ISPREMIUMUSER0* sarà la valutazione del parametro *isPremiumUser* che sarà 1 se vero, 0 altrimenti; in altri termini *ISPREMIUMUSER0* sarà la valutazione della guardia del costrutto *if-then-else* e verrà eseguita la chiamata al servizio *PremiumService* se *ISPREMIUMUSER0* sarà 1 con costo $nat(P)$, altrimenti verrà eseguita la chiamata al servizio *BasicService* con costo $nat(B)$. Una volta avere generato l'equazioni di costo dal programma, lo stampiamo in un file *equation.ces*, cosi da poter eseguire PUBS(A Practical Upper Bounds Solver), per determinarci l'Upper Bound del programma. L'obiettivo di PUBS(come vedremo in seguito 1.4) è quello di ottenere automaticamente upper bound in forma chiusa per i sistemi di equazioni di costo, di conseguenza calcola i limiti superiori per la relazione di costo indicata come "Entry", oltre che per tutte le altre relazioni di costo di cui tale "Entry" dipende.

1.1 Implementazione

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

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

Listing 1.3: Interfaccia Node

Andando a descrivere i metodi dell'interfaccia Node:

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

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

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

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

Listing 1.4: Implementazione del Visitor AST

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

Controllo Semantico

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

```
1     HLCostLanBaseVisitorImpl visitor = new
      HLCostLanBaseVisitorImpl();
2     Node ast = visitor.visit(parser.prg());
3     Environment env = new Environment();
4     ArrayList<String> errorSemantics = ast.checkSemantics(env
      );
5     if (!errorSemantics.isEmpty()) {
6         System.err.println(errorSemantics);
7         return Results.SEMANTIC_ERROR;
8     }else{
9         //continua l'esecuzione del programma
10    }
```

Listing 1.5: Inizio Controllo semantico

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

```
1  @Override
2  public ArrayList<String> checkSemantics(Environment env) {
3      ArrayList<String> errors = new ArrayList<>();
4      if(!env.checkHeadDeclaration(id))
5          env.addDeclaration(id,new IntType());
6      else
7          errors.add("Error: Variable "+id+" already declared")
       ;
8
9      errors.addAll(exp.checkSemantics(env));
10     errors.addAll(stm.checkSemantics(env));
11     return errors;
12 }
```

Listing 1.6: Controllo semantico di ForNode

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

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

Controllo dei tipi

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

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

Listing 1.7: Controllo di tipo di CallNode

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

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

Listing 1.8: Metodo isSubtype

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

In seguito verrà specificato come vengono generati le equazioni di costo1.3 attraverso il metodo *toEquation(EnvVar e)* e come viene generato il WASM corrispondente andando a visitare l'AST con il metodo *codeGeneration*??.

1.1.1 Struttura del codice

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

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

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

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

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

Listing 1.9: Esempio Testing

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

1.2 Regole di Inferenza

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

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

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

- Un'espressione in Presburger aritmetica (PA)[4]:

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

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

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

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

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

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

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

I giudizi hanno forma:

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

[MAIN]
$$\frac{\Gamma \vdash S : e; C; Q \qquad \overline{w} = \mathrm{Var}(\overline{p}, e) \cup \mathrm{Var}(\mathrm{C})}{\Gamma \vdash \overline{p} \to \{S\} : 0; \emptyset; Q'; C} \tag{1.1}$$

[CALL]
$$\frac{\Gamma \vdash S : e; C; Q}{\Gamma \vdash \mathrm{call\ h}(\overline{E})S : e + e'; C; Q} \tag{1.2}$$

[IF]
$$\frac{\begin{array}{l}\Gamma \vdash E : \varphi \qquad \Gamma \vdash S : e'; C; Q \quad \Gamma \vdash S : e"; C'; Q' \\ W = Var(e, e', e") \cup Var(C) \qquad Q" = \left[\begin{array}{l}\mathrm{if}_l(\overline{w}) = e' + c[\varphi] \\ \mathrm{if}_l(\overline{w}) = e" + c[\neg\varphi]\end{array}\right]\end{array}}{\Gamma \vdash \mathrm{if}\ E\ \mathrm{then}\ S_1\ \mathrm{else}\ S_2 : 0; if_l(\overline{w}); Q; Q'; Q"} \tag{1.3}$$

[LET]
$$\frac{\Gamma + i : Not \qquad \overline{w} = Var(E) \cup Var(c)}{\Gamma \vdash \mathrm{let}\ i = e\ \mathrm{in}\ S : S; e; C; Q} \tag{1.4}$$

[FOR]
$$\frac{\begin{array}{l}\Gamma \vdash M : e \qquad \Gamma \vdash i : Not \qquad \Gamma \vdash S : e'; C; Q \\ \overline{w} = Var(e, e') \cup Var(C)\ i \qquad Q' = \left[\begin{array}{ll}\mathrm{for}_l(i, \overline{w}) = e + c & [i < e] \\ \mathrm{for}_l(i, \overline{w}) = 0 & [i \geq e]\end{array}\right]\end{array}}{\Gamma \vdash \mathrm{for\ i\ in\ (0,M)} \quad \mathrm{S} \quad : 0; for_l(0, \overline{w}); Q; Q'} \tag{1.5}$$

Riassumiamo le regole descritte in precedenza:

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

1.3 Generazione delle Equazioni di costo

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

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

    }
```

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

```
    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));
        }
        return main.toEquation(e) + equ;
    }
```

Listing 1.10: toEquation() del ProgramNode

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

1.4 PUBS

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

1.4.1 Analisi di costo

Come analisi statica dei costi, miriamo a ottenere risultati analitici per un dato programma P, i quali consentano di vincolare il costo dell'esecuzione di P su qualsiasi input x, senza dover effettivamente eseguire $P(x)$.[2]
Partiamo da un esempio, che ci aiuterà a capire meglio il concetto di analisi di costo, partendo da un programma java:

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

Listing 1.11: Esempio di Analisi di Costo

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

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

Figura 1.1: Relazioni di costo del programma 1.11

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

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

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

Un esempio già più completo che troviamo nel paper *Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis* [1]:

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

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

Figura 1.2: Esempio di Analisi di Costo

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

dal presupposto che tutti i valori in l siano contenuti in a o b, e il metodo *del* rimuove tutti i valori in l da a o b rispettivamente. Il metodo rm_{vec} rimuove un dato valore e da un array a di lunghezza la e ne ritorna la nuova lunghezza. Gli autori del paper [1],applicano l'analisi dei costi a questo programma, approssimando il costo del metodo *del* in termini di istruzione bytecode eseguite.
La figura 1.2 presenta i risultati dell'analisi, dopo aver effettuato una valutazione parziale. Nei risultati dell'analisi le strutture dati vengono astratte in base alle loro dimensioni:l rappresenta la lunghezza massima del percorso della corrispondente struttura dinamica, a e b sono le lunghezze degli array corrispondenti, mentre la e lb sono i valori interi delle variabili. Ci sono nove equazioni che definiscono la relazione *del* che corrisponde al costo del metodo *del* e 3 relazioni ricorsive ausiliarie C, D, E; ciascuna delle quale corrisponde a un ciclo:(C: Ciclo while in *del*, D: ciclo while in rm_{vec} e E: Ciclo for in rm_{vec}). Ogni equazioni è definita con una serie di vincoli che catturano le relazioni dimensionali tra i valori delle variabili della parte sinistra(lhs) e della parte destra(rhs). Prendiamo in esempio le equazioni per D Eq5. e Eq.6 rappresentano casi base per l'uscita dal ciclo ovvero quando $i \geq la$ e $a[i] \geq e$. Per le nostre misurazioi di costo, vengono contati 3 istruzioni bytecode in Eq.5 e 8 in Eq.6. Il costo per eseguire un iterazione del ciclo è rappresentato da Eq.7, dove la condizione $i < la$ deve essere soddisfatta e la variabili i è incrementata di uno ad ogni chiamata ricorsiva.

1.4.2 Relazione di Costo

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

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

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

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

Le espressioni di costo di base godono di due proprietà:

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

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

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

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

Semantica per CRS

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

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

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

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

1.4.3 Stima del costo per Nodo

Tutte le espressioni nei nodi sono istanze delle espressioni che compaiono nelle equazioni corrispondenti. Pertanto, il calcolo di $costr^+(\overline{x})$ e $costnr^+(\overline{x})$ può essere effettuato trovando innanzitutto un limite superiore di tali espressioni e quindi combinandoli attraverso un operatore di massimo. Prima calcoliamo gli invarianti per i valori che le variabili delle espressioni possono assumere rispetto ai valori iniziali e li utilizziamo per derivare limiti superiori per tali espressioni.
Calcolare le invarianti(in termini di vincoli lineari), in modo da raggruppare tutte le chiamate ai contesti di una relazione C, tra gli argomenti di una chiamata iniziale e ogni chiamata durante la valutazione che può essere fatta usando $Loops(C)$.
Ovvero, se è presente un vincolo lineare ψ tra gli argomenti di una chiamata iniziale $C(\overline{x_0})$, quelli di una chiamata ricorsiva $C(\overline{x})$, indicato con $\langle C(\overline{x_0}) \rightsquigarrow (\overline{x}, \psi)\rangle$, e se esiste il ciclo $C(\overline{x_0}) \rightsquigarrow (\overline{y}, \varphi) \in Loops(C)$ allora è possibile applicare il ciclo a uno o più step e prende un nuovo calling context $\langle C(\overline{x_0}) \rightsquigarrow (\overline{y}), \overline{\exists} \cup \overline{y_i}\dot{\psi} \wedge \varphi\rangle$.
Una volta che le invarianti sono state stabilite, è possibile determinare il limite superiore delle equazioni di costo massimizzando la loro parte nat indipendentemente. Questo approccio è reso possibile grazie alla proprietà di monotonia delle espressioni di costo. Considerando un'equazione di costo nella forma $\langle C(\overline{x}) = exp + \sum_{i=0}^{k} C(\overline{y_i}), \varphi\rangle$ e un invariante $C(\overline{x_0}) \rightsquigarrow C(\overline{x}, \Psi)$,una funzione può calcolare un limite superiore f' per ogni f che compare nell'operatore nat. Tale funzione sostituisce f con un limite superiore nelle espressioni exp in cui non è possibile determinare un limite superiore e la funzione tornerà ∞. Se questa funzione è completa, ovvero se i Ψ e φimplicano che esiste un limite superiore per un dato $nat(f)$, allora possiamo trovare un limite superiore su Ψ'.[3]

1.4.4 PUBS in pratica

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

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

Listing 1.12: Listing 1

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

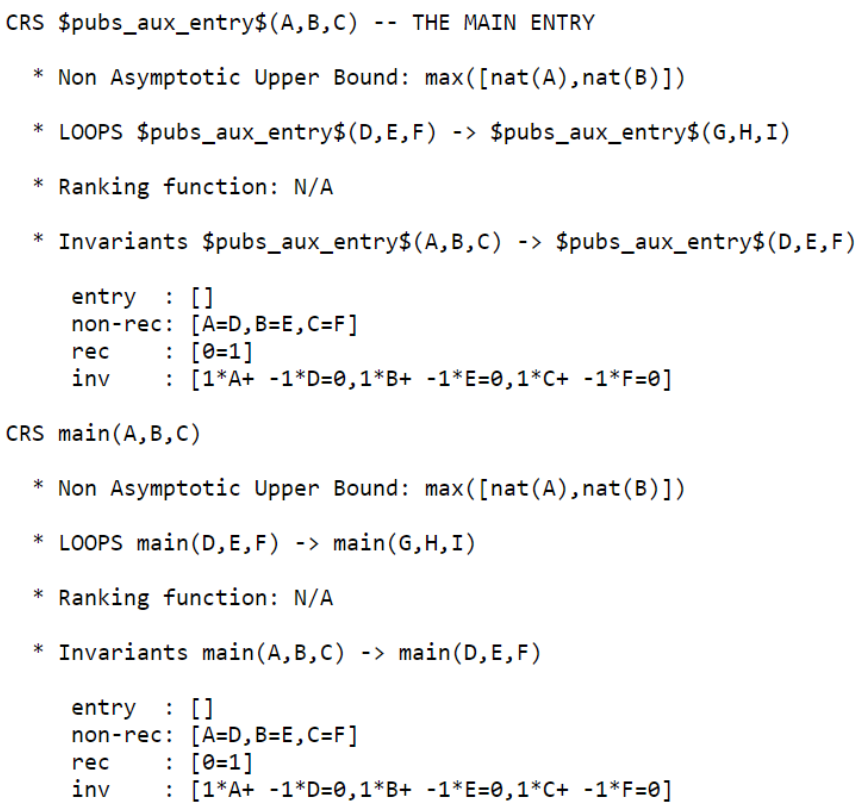

Figura 1.3: Esempio di output PUBS su Listing 1

Come vediamo nell'immagine sopra, PUBS restituisce un analisi dell'intera equazione (pub_aux_entry) e delle singole funzioni da cui essa dipende, in questo caso $pubs\_aux\_if9$ 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 sarà il massimo tra i due.
PUBS ha una grammatica che definisce l'equazione di costo che deve essere rispettata da ogni equazione di costo generata da CostCompiler, che è la seguente:

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

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

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

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

Listing 1.13: Grammatica PUBS

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

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

Listing 1.14: Listing 6

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

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

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

Listing 1.15: Equazione di costo PUBS per Listing6

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

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

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

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

  * Ranking function: N/A

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

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

Figura 1.4: Esempio di output PUBS su Listing 6