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) 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 **??** 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 1.3 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 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 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 1.3) 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.

1.1 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

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

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

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

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

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

$$\tag{1.6}$$

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.

1.2 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.

```
    @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 $stm.toEquation(e)$ 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.

```java
    public String toEquation(EnvVar e){

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

Listing 1.3: 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.

1.3 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.

1.3.1 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 $C(\overline{v})$.

1.3.2 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]

1.3.3 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 1.4: 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, 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 1.1: 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:

```
 <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 1.5: 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:

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

Listing 1.6: 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:

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

Listing 1.7: 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 1.2: Esempio di output PUBS su Listing 6