

Guaggi Formali e Traduttori

Analisi statica e traduzione di metodi

- Sommario
- Grammatica
- Note
- Esempio
- SDD per invocazione di un metodo e ritorno
- Verifica della presenza di return
- SDD per la verifica della presenza di return
- Allocazione delle variabili locali
- SDD per il calcolo degli slot di variabili locali
- Esempio
- Calcolo della dimensione massima della pila
- SDD per pila di espressioni aritmetiche
- Esempio: associatività degli operatori
- SDD per pila di comandi e metodi
- Traduzione di metodi statici
- Esercizi

È proibito condividere e divulgare in qualsiasi forma i materiali didattici caricati sulla piattaforma e le lezioni svolte in videoconferenza: ogni azione che viola questa norma sarà denunciata agli organi di Ateneo e perseguita a termini di legge.

Sommario

Problema

- La compilazione di un metodo comporta il calcolo della dimensione del suo **frame**, che comprende le **variabili locali** e la **pila degli operandi**.
- Il compilatore deve determinare questi valori **senza eseguire il codice del metodo**.
- Il compilatore deve assicurarsi che un metodo con tipo di ritorno diverso da void restituisca sempre un valore.

In questa lezione

- Presentiamo le SDD per la definizione e l'invocazione di metodi.
- Introduciamo alcune semplici forme di **analisi statica del codice** per calcolare la dimensione dei frame e individuare metodi errati.

Riferimenti esterni

- [Java Language and Virtual Machine Specifications](#)
- [JVM Instruction set](#)

Grammatica

Produzioni	Descrizione
$E \rightarrow \dots$	Come in precedenza
$E \rightarrow m(E_{list})$	Invocazione di metodo
$E_{list} \rightarrow \varepsilon$	Nessun argomento
$E_{list} \rightarrow E_{listp}$	Uno o più argomenti
$E_{listp} \rightarrow E$	Un argomento
$E_{listp} \rightarrow E, E_{listp}$	Due o più argomenti
$S \rightarrow \dots$	Come in precedenza
$S \rightarrow \text{return } E;$	Ritorno da metodo
$S_{list} \rightarrow \dots$	Come in precedenza
$S_{list} \rightarrow T \ x = E; S_{list}$	Variabile locale
$M \rightarrow T \ m(T_1 \ x_1, \dots, T_n \ x_n) \ S$	Definizione di metodo
$T \rightarrow \text{void} \mid \text{int} \mid \dots$	Tipo

Note

SDD

- Le SDD presentate vanno integrate a quelle già presentate in precedenza per la generazione del codice di **espressioni** e **comandi**.

Metodi non statici

- Consideriamo solo **metodi statici**.
- I metodi **non statici** hanno un argomento implicito `this` (l'**oggetto ricevente**).

Tipo di ritorno

- Consideriamo solo metodi che restituiscono un **valore intero**.
- I metodi con tipo di ritorno **void non possono essere invocati all'interno di un'espressione**.

Esempio

```
int fibo(int n) {  
    if (n <= 1) return n;  
    else return fibo(n-1) + fibo(n-2);  
}
```

```
int fibo(int k) {  
    int m = 0;  
    int n = 1;  
    while (k >= 0) {  
        int t = m;  
        m = n;  
        n = m + n;  
        k = k - 1;  
    }  
    return m;  
}
```

```
.method fibo(I)I  
    .limit stack 3  
    .limit locals 1  
    ...
```

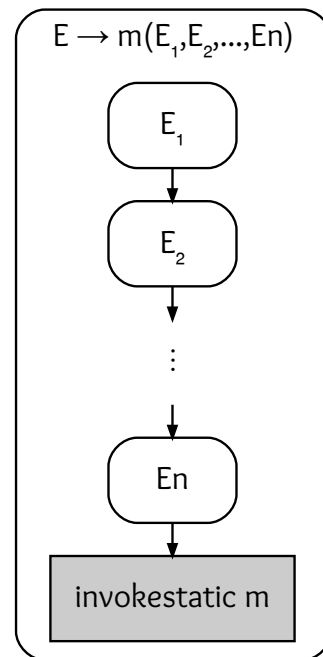
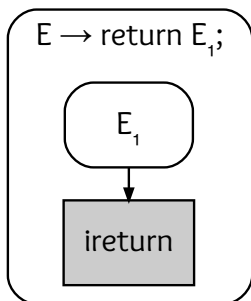
```
.method fibo(I)I  
    .limit stack 2  
    .limit locals 4  
  
    ...  
    ...  
    ...
```

Dimensione del frame di un metodo

- Numero di argomenti e variabili locali del metodo (locals=1 e locals=4).
- Dimensione massima della pila degli operandi (stack=3 e stack=2).

SDD per invocazione di un metodo e ritorno

Produzioni	Regole semantiche
$E \rightarrow m(E_{list})$	$E.code = E_{list}.code \parallel \text{invokestatic } m$
$E_{list} \rightarrow \varepsilon$	$E_{list}.code = []$
$E_{list} \rightarrow E_{listp}$	$E_{list}.code = E_{listp}.code$
$E_{listp} \rightarrow E$	$E_{listp}.code = E.code$
$E_{listp} \rightarrow E, E_{listp1}$	$E_{listp}.code = E.code \parallel E_{listp1}.code$
$S \rightarrow \text{return } E;$	$S.code = E.code \parallel \text{ireturn}$



Verifica della presenza di return

Problema

- Un metodo con tipo di ritorno diverso da void deve restituire un **risultato** al chiamante per mezzo del comando return.

Strategia per l'analisi

- Si analizza il codice del metodo per verificare che ogni cammino di esecuzione porti dall'inizio del metodo a una istruzione return.
- L'analisi è statica, dunque non tiene conto dell'effettivo flusso di esecuzione del metodo e non garantisce che return sia davvero eseguito, per esempio se l'esecuzione entra in un ciclo infinito o se viene lanciata un'eccezione (es. divisione per zero).

Approssimazioni

- Non teniamo conto del valore delle espressioni logiche in comandi condizionali e cicli, anche quando sono banali (es. true). In generale questo problema è **indecidibile**.
- Non controlliamo se il valore restituito da return è del tipo giusto. Questo controllo richiede una forma aggiuntiva di analisi statica detta **controllo dei tipi**.

SDD per la verifica della presenza di return

SDD

Attributi

Produzioni	Regole semantiche
$S \rightarrow x = E;$	$S.ret = false$
$S \rightarrow \text{if } (B) S_1$	$S.ret = false$
$S \rightarrow \text{if } (B) S_1 \text{ else } S_2$	$S.ret = S_1.ret \wedge S_2.ret$
$S \rightarrow \text{while } (B) S_1$	$S.ret = false$
$S \rightarrow \text{do } S_1 \text{ while } (B);$	$S.ret = S_1.ret$
$S \rightarrow \{S_{list}\}$	$S.ret = S_{list}.ret$
$S \rightarrow \text{return } E;$	$S.ret = true$
$S_{list} \rightarrow \epsilon$	$S_{list}.ret = false$
$S_{list} \rightarrow S S_{list1}$	$S_{list}.ret = S.ret \vee S_{list1}.ret$
$S_{list} \rightarrow T x = E; S_{list1}$	$S_{list}.ret = S_{list1}.ret$

- $S.ret$ = se l'esecuzione di S termina, è perché esegue **return**

SDD per la verifica della presenza di return

SDD

Produzioni	Regole semantiche
$S \rightarrow x = E;$	$S.ret = false$
$S \rightarrow \text{if } (B) S_1$	$S.ret = false$
$S \rightarrow \text{if } (B) S_1 \text{ else } S_2$	$S.ret = S_1.ret \wedge S_2.ret$
$S \rightarrow \text{while } (B) S_1$	$S.ret = false$
$S \rightarrow \text{do } S_1 \text{ while } (B);$	$S.ret = S_1.ret$
$S \rightarrow \{S_{list}\}$	$S.ret = S_{list}.ret$
$S \rightarrow \text{return } E;$	$S.ret = true$ <i>→ caso base</i>
$S_{list} \rightarrow \epsilon$	$S_{list}.ret = false$
$S_{list} \rightarrow S S_{list1}$	$S_{list}.ret = S.ret \vee S_{list1}.ret$
$S_{list} \rightarrow T x = E; S_{list1}$	$S_{list}.ret = S_{list1}.ret$

Attributi

- $S.ret$ = se l'esecuzione di S termina, è perché esegue **return**

Note

- In una sequenza $S S_{list}$ in cui $S.ret = true$ la continuazione S_{list} non viene mai eseguita ed è detta **codice morto**.
- La presenza di codice morto **non impedisce la compilazione** ma è probabilmente sintomo di un errore.
- Il compilatore lo segnala con un **avvertimento (warning)** o un **errore** (es. javac).

Allocazione delle variabili locali

Problema

- Determinare il **più piccolo numero di slot** necessari all'interno di un frame per la memorizzazione di argomenti e variabili locali (“più piccolo” = risparmio di memoria).

Strategia

- Determinare il **numero massimo** di variabili che sono **contemporaneamente attive**.

Esempi

```
void sequenza() {  
    {  
        int x = 42;  
    }  
    {  
        int y = 15;  
    }  
}
```

```
void alternativa() {  
    if (true) {  
        int x = 42;  
    } else {  
        int y = 15;  
    }  
}
```

- In entrambi i casi x ed y non sono **mai** attive contemporaneamente e possono **condividere lo stesso slot** nel frame del metodo.

SDD per il calcolo degli slot di variabili locali

Produzioni	Regole semantiche
$S \rightarrow x = E;$	$S.locals = 0$
$S \rightarrow \text{if } (B) S_1$	$S.locals = S_1.locals$
$S \rightarrow \text{if } (B) S_1 \text{ else } S_2$	$S.locals = \max \{S_1.locals, S_2.locals\}$
$S \rightarrow \text{while } (B) S_1$	$S.locals = S_1.locals$
$S \rightarrow \text{do } S_1 \text{ while } (B);$	$S.locals = S_1.locals$
$S \rightarrow \{S_{list}\}$	$S.locals = S_{list}.locals$
$S \rightarrow \text{return } E;$	$S.locals = 0$
$S_{list} \rightarrow \epsilon$	$S_{list}.locals = 0$
$S_{list} \rightarrow S S_{list1}$	$S_{list}.locals = \max \{S.locals, S_{list1}.locals\}$
$S_{list} \rightarrow T x = E; S_{list1}$	$S_{list}.locals = 1 + S_{list1}.locals$

in questo
sicuramente ok
b lo 0

- $S.locals$ = max numero di variabili contemporaneamente attive durante esecuzione di S

Esempio

```
void sequenza() {  
    {  
        int x = 42;  
    }  
    {  
        int y = 15;  
    }  
}
```

```
.method sequenza()V  
    .limit locals 1  
    ldc 42  
    istore 0 ; x  
    goto L1  
L1: ldc 15  
    istore 0 ; y  
    goto L0  
L0: return  
.end method
```

```
void alternativa() {  
    if (true) {  
        int x = 42;  
    } else {  
        int y = 15;  
    }  
}
```

```
.method alternativa()V  
    .limit locals 1  
    goto L3  
L3: ldc 42  
    istore 0 ; x  
    goto L2  
L4: ldc 15  
    istore 0 ; y  
    goto L2  
L2: return  
.end method
```

Nota: x ed y condividono lo stesso slot 0 nei frame dei due metodi.

Calcolo della dimensione massima della pila

Problema

- Determinare il **numero massimo di slot** occupati sulla pila degli operandi durante l'esecuzione di un metodo.

Strategia

- Tenendo conto del codice prodotto dalla traduzione di espressioni e comandi, **approssimare per eccesso** la dimensione massima della pila.

Esempio

```
int metodo() {  
    if (true) return 0;  
    else return 1 + 2 * 3;  
}
```

```
.method metodo()I  
    .limit stack 3  
    .limit locals 0  
    goto L1  
L1: ldc 0  
    ireturn  
L2: ldc 1  
    ldc 2  
    ldc 3  
    imul  
    iadd  
    ireturn  
.end method
```

SDD per pila di espressioni aritmetiche

Produzioni	Regole semantiche
$E \rightarrow E_1 + E_2$	$E.stack = \max\{E_1.stack, 1 + E_2.stack\}$
$E \rightarrow (E_1)$	$E.stack = E_1.stack$ $\epsilon_1 \quad \checkmark + E_2$
$E \rightarrow n$	$E.stack = 1$
$E \rightarrow x$	$E.stack = 1$
$E \rightarrow m(E_{list})$	$E.stack = \max\{1, E_{list}.stack\}$ $\rightarrow \epsilon_1, 0, \checkmark, \checkmark, \checkmark$
$E_{list} \rightarrow \epsilon$	$E_{list}.stack = 0$
$E_{list} \rightarrow E_{listp}$	$E_{list}.stack = E_{listp}.stack$
$E_{listp} \rightarrow E$	$E_{listp}.stack = E.stack$
$E_{listp} \rightarrow E, E_{listp1}$	$E_{listp}.stack = \max\{E.stack, 1 + E_{listp1}.stack\}$

- $E.stack$ = dimensione massima della pila durante la valutazione di E ($E.stack \geq 1$)
- $E_{list}.stack$ = dimensione massima della pila durante la valutazione cumulata di tutte le espressioni generate da E_{list} ($E_{list}.stack \geq 0$)
- Nel caso $E \rightarrow m(E_{list})$, l'1 serve per tenere conto del valore restituito dal metodo.

Esempio: associatività degli operatori

```
int sinistra() {  
    return 1 + 2 + 3 + 4 + 5;  
}
```

```
.method sinistra()I  
    .limit stack 2  
    .limit locals 0  
    ldc 1  
    ldc 2  
    iadd  
    ldc 3  
    iadd  
    ldc 4  
    iadd  
    ldc 5  
    iadd  
    ireturn  
.end method
```

```
int destra() {  
    return 1 + (2 + (3 + (4 + 5)));  
}
```

```
.method destra()I  
    .limit stack 5  
    .limit locals 0  
    ldc 1  
    ldc 2  
    ldc 3  
    ldc 4  
    ldc 5  
    iadd  
    iadd  
    iadd  
    iadd  
    ireturn  
.end method
```

Osservazione

- L'associatività a sinistra mantiene la pila piccola perché le sottoespressioni vengono valutate man mano che si incontrano, da sinistra verso destra.

SDD per pila di comandi e metodi

Produzioni	Regole semantiche
$S \rightarrow x = E;$	$S.stack = E.stack$
$S \rightarrow \text{if } (B) S_1$	$S.stack = \max \{B.stack, S_1.stack\}$
$S \rightarrow \text{if } (B) S_1 \text{ else } S_2$	$S.stack = \max \{B.stack, S_1.stack, S_2.stack\}$
$S \rightarrow \text{while } (B) S_1$	$S.stack = \max \{B.stack, S_1.stack\}$
$S \rightarrow \text{do } S_1 \text{ while } (B);$	$S.stack = \max \{S_1.stack, B.stack\}$
$S \rightarrow \{S_{list}\}$	$S.stack = S_{list}.stack$
$S \rightarrow \text{return } E;$	$S.stack = E.stack$
<hr/>	
$S_{list} \rightarrow \varepsilon$	$S_{list}.stack = 0$
$S_{list} \rightarrow S S_{list1}$	$S_{list}.stack = \max \{S.stack, S_{list1}.stack\}$
$S_{list} \rightarrow T x = E; S_{list1}$	$S_{list}.stack = \max \{E.stack, S_{list1}.stack\}$

- **$B.stack$** = dimensione massima della pila durante la valutazione di **B** (definire SDD come esercizio)
- **$S.stack$** = dimensione massima della pila durante l'esecuzione di **S**

Traduzione di metodi statici

Produzioni	Regole semantiche
$M \rightarrow \text{void } m(T_1 x_1, \dots, T_n x_n) S$	$S.next = \text{newlabel}()$ $M.code = .method\ m$.limit stack $S.stack$.limit locals $n + S.locals$ $S.code$ $S.next : \text{return}$ (se $S.ret = false$) .end method
$M \rightarrow \text{int } m(T_1 x_1, \dots, T_n x_n) S$	$S.next = \text{newlabel}()$ (etichetta inutilizzata) $M.code = .method\ m$.limit stack $S.stack$.limit locals $n + S.locals$ $S.code$.end method (se $S.ret = false$ errore)

Esercizi

1. Tradurre i seguenti metodi:

```
int min(int x, int y) {  
    if (x < y) return x;  
    else return y;  
}
```

```
int euclide(int a, int b) {  
    if (a == 0) return b;  
    while (b != 0)  
        if (a > b) a = a - b;  
        else b = b - a;  
    return a;  
}
```

```
int primo(int n) {  
    int i = 2;  
    while (i < n) {  
        if (n % i == 0)  
            return 0;  
        i = i + 1;  
    }  
    if (n >= 2) return 1;  
    else return 0;  
}
```

2. Scrivere le regole semantiche per calcolare ***E. stack*** nel caso della produzione $E \rightarrow B ? E_1 : E_2$.
3. Scrivere le regole semantiche per calcolare ***S. locals*** e ***S. stack*** per il ciclo for $S \rightarrow \text{for } (S_1; B; S_2) S_3$.