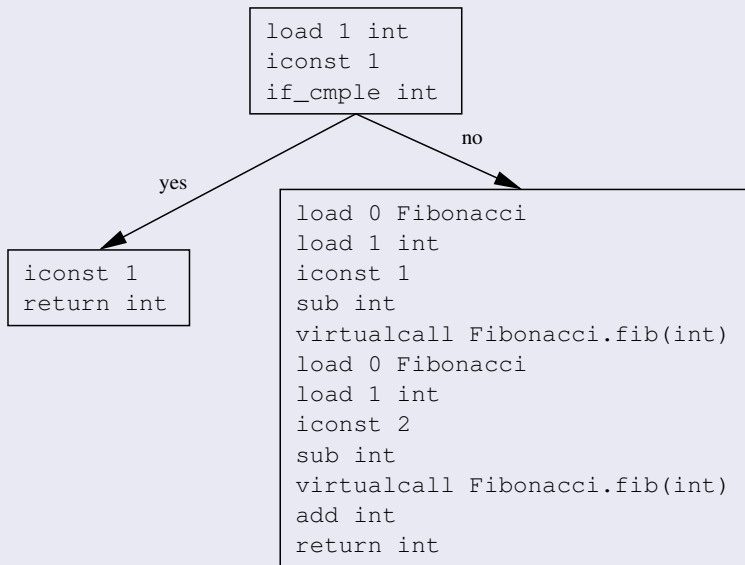


La generazione del codice *oggetto* Java bytecode per Kitten

Fausto Spoto

7 marzo 2005

Il risultato della traduzione in codice intermedio



Il risultato della generazione del Java bytecode

```
0    iload_1
1    iconst_1
2    if_icmple 21
5    aload_0
6    iload_1
7    iconst_1
8    isub
9    invokevirtual Fibonacci.fib(I)I
12   aload_0
13   iload_1
14   iconst_2
15   isub
16   invokevirtual Fibonacci.fib(I)I
19   iadd
20   ireturn
21   iconst_1
22   ireturn
```

Come avviene la traduzione

Prima fase: Traduzione dei bytecode Kitten

Ogni bytecode kitten viene tradotto in uno o più bytecode Java:

```
java.sun.com/docs/books/vmspec/2nd-edition/  
html/Instructions.doc.html
```

Seconda fase: Linearizzazione

I blocchi di base di ciascun metodo vengono disposti in un ordine arbitrario, purché il blocco iniziale sia all'inizio. Le frecce fra blocchi sono tradotte in salti espliciti a ben precisi bytecode

Terza fase: Sintesi dei file .class

La sequenza di Java bytecode per i metodi di ciascuna classe Kitten viene salvata dentro il file .class associato alla classe tramite la libreria BCEL:

```
jakarta.apache.org/bcel/apidocs/index.html
```

Prima fase: Traduzione dei bytecode Kitten

Ogni istanza di `Bytecode/Bytecode.java` avrà un metodo

```
public InstructionList JBGenerate  
    (ConstantPoolGen cpg)
```

- `InstructionList` è la classe BCEL che rappresenta una sequenza di Java bytecode
 - ha il metodo `append` che aggiunge un bytecode alla fine
 - nonché il metodo `insert` che lo aggiunge all'inizio
 - e tantissimi altri metodi. Guardate la documentazione!
- `ConstantPoolGen` è un riferimento alla tabella delle costanti del file `.class` a cui questo bytecode apparterrà

La traduzione dei singoli bytecode

Traduzione diretta

Alcuni bytecode Kitten hanno un corrispettivo diretto nel Java bytecode

Dentro `Bytecode/NOP.java`

```
InstructionList JBGenerate(ConstantPoolGen cpg) {  
    // traduciamo i bytecode che seguono  
    InstructionList il = super.JBGenerate(cpg);  
    // facciamo precedere il tutto con una NOP  
    il.insert(new org.apache.bcel.generic.NOP());  
    return il;  
}
```

La traduzione dei singoli bytecode

Traduzione con compilazione del tipo

Alcuni bytecode Kitten hanno un parametro di tipo che in Java bytecode deve essere compilato esplicitamente nell'opcode:

$$\text{add } type \Rightarrow \begin{cases} \text{iadd} & \text{se } type \in \{\text{char}, \text{int}\} \\ \text{fadd} & \text{se } type = \text{float} \end{cases}$$

Dentro `Bytecode/ADD.java`

```
InstructionList JBGenerate(ConstantPoolGen cpg) {  
    InstructionList il = super.JBGenerate(cpg);  
    if (type == TypeChecker.CHAR_TYPE ||  
        type == TypeChecker.INTEGER_TYPE)  
        il.insert(new org.apache.bcel.generic.IADD());  
    else il.insert(new org.apache.bcel.generic.FADD());  
    return il;  
}
```

A cosa serve la ConstantPoolGen ?

Alcuni bytecode Kitten hanno un parametro costante che deve essere inserito nella tabella delle costanti del file `.class` dove il bytecode occorre

Dentro `Bytecode/VIRTUALCALL.java`

```
InstructionList JBGenerate(ConstantPoolGen cpg) {  
    InstructionList il = super.JBGenerate(cpg);  
    il.insert(method.createINVOKEVIRTUAL(cpg));  
    return il;  
}
```

La segnatura del metodo è aggiunta alla tabella delle costanti

Traduzione tramite fattoria di istruzioni

Alcuni bytecode Kitten hanno più corrispondenti in Java bytecode. Ci sono metodi di ausilio dentro una `InstructionFactory` che semplificano la traduzione:

$$\text{iconst } v \Rightarrow \begin{cases} \text{iconst } v & \text{se } -1 \leq v \leq 5, \text{ super-ottimizzato} \\ \text{bipush } v & \text{se } 0 \leq v \leq 255, \text{ poco ottimizzato} \\ \text{ldc } v & \text{sempre possibile, non ottimizzato} \end{cases}$$

Dentro `Bytecode/ICONST.java`

```
InstructionList JBGenerate(ConstantPoolGen cpg) {
    InstructionFactory factory =
        new InstructionFactory(cpg);
    InstructionList il = new InstructionList
        (factory.createConstant(new Integer(value)));
    il.append(super.JBGenerate(cpg));
    return il;
}
```

Traduzioni particolarmente laboriose

Alcuni bytecode Kitten non hanno una traduzione diretta in Java bytecode, e occorre allora compilarli tramite una **sequenza** di Java bytecode:

```

                                IF_ICMPGE after
ge int                          ICONST 0
  oppure ⇒                      GOTO follow
ge char                          after:  ICONST 1
                                follow:

                                FCMPPL
ge float ⇒                      IFGE after
                                ICONST 0
                                GOTO follow
                                after:  ICONST 1
                                follow:
```

Seconda fase: Linearizzazione

- 1 I bytecode di ciascun blocco sono tradotti dalla fase 1
- 2 I blocchi vengono disposti in un ordine qualsiasi, con l'inizio del metodo in cima
- 3 Alla fine del bytecode di un blocco linkato si aggiunge un bytecode `GOTO` all'inizio del blocco a cui esso era legato
- 4 Alla fine di un blocco condizionale, che termina per esempio con `IF_CMPLT int`, si aggiungono i bytecode

```
IF_ICMPLT yesLine  
GOTO noLine
```

dove *yesLine* e *noLine* sono i numeri di linea a cui iniziano i blocchi *yes* e *no*, rispettivamente.

- 5 Si eliminano tutti i bytecode `NOP` (derivanti dai pivot)
- 6 Si eliminano i bytecode `GOTO` che saltano al bytecode immediatamente successivo

Fa tutto la libreria BCEL

- Creiamo una struttura dati `c` che descrive un file `.class`
- Aggiungiamo i campi e le costanti dichiarate per quella classe
- Aggiungiamo i costruttori e i metodi per quella classe, dandogli il codice risultate dal metodo `JBGenerate`
- Aggiungiamo la tabella delle costanti arricchita durante la generazione del codice tramite `JBGenerate`
- Chiamiamo un metodo `dump` di `c` che verifica la correttezza del codice Java bytecode e lo salva in un file `.class`

E se avessimo voluto generare assembly x86?

Molto più complicato!

- I parametri assembly sono passati sullo stack e non in variabili locali
- Lo stack assembly potrebbe implementare lo stack del bytecode Kitten, ma si mescolerebbero insieme variabili locali e stack
- Esiste un numero limitato di registri assembly, ad accesso veloce. Conviene usarli per implementare almeno una parte dello stack del bytecode Kitten
- La scelta di cosa tenere nei registri è critica: si risolve con algoritmi di colorazione di grafi
- La gestione della memoria deve essere fatta esplicitamente (`malloc/free`). Serve un supporto a run-time che implementa un garbage-collector
- Tutti i `.class` verrebbero fusi in un unico file eseguibile

Possibili ottimizzazioni del codice

- **Constant propagation**: si usano i valori delle costanti piuttosto che riferimenti alle stesse
- **Strength reduction**: calcoli interni a un ciclo potrebbero essere portati fuori e fatti una sola volta
- **Late-binding removal**: chiamate virtuali a metodi potrebbero essere trasformate in più efficienti chiamate statiche
- **Cast-removal**: cast sempre verificati potrebbero non essere controllati a tempo di esecuzione
- **Null-pointer check removal**: il controllo sul ricevitore di accessi a campi e chiamate di metodo potrebbe essere inutile
- **Stack-allocation**: alcune `new` potrebbero allocare i loro oggetti sullo stack piuttosto che nello heap