

Metody Realizacji Języków Programowania

Maszyny wirtualne: JVM, LLVM

Marcin Benke

MIM UW

20 grudnia 2010

Maszyna wirtualna Javy

- 1 Maszyna abstrakcyjna
 - ▶ pozwala wykonywać programy,
 - ▶ standaryzowany opis
 - ▶ Tim Lindholm, Frank Yellin *The Java Virtual Machine Specification*
 - ▶ wiele implementacji,
- 2 Wykonywanie programów niskopoziomowych
- 3 Zapewnia przenośność
- 4 Zapewnia pewne mechanizmy bezpieczeństwa

Maszyna wirtualna Javy

Typy danych

- Typy bazowe: całkowite (`int`, etc.), zmiennoprzecinkowe (`float`, `double`)
- Referencje do obiektów (zbędne w podstawowej wersji Javalette)

Obszary danych

- Zmienne lokalne i parametry są przechowywane na stosie.
- Stos służy też do obliczeń.
- Obiekty (w tym tablice) przechowywane na stercie. Nie potrzeba jej używać w Javalette.
- Stałe zmiennoprzecinkowe i napisowe przechowywane w obszarze stałych — nie musimy się tym przejmować jeśli używamy Jasmina.

Stos JVM

- Stos jest ciągiem *ramek*. Każda instancja metody ma swoją ramkę.
- Różne postaci wywołania:
 - ▶ `invokestatic` dla metod statycznych (dla funkcji Javalette)
 - ▶ `invokevirtual` dla metod obiektowych
 - ▶ `invokenonvirtual` np. dla konstruktorów
- JVM zajmuje się kwestiami porządkowymi, jak:
 - ▶ alokacja i zwalnianie ramek
 - ▶ przekazywanie parametrów
 - ▶ dostarczanie wyników

Struktura ramki stosu

Ramka zawiera zmienne lokalne (w tym parametry) i stos operandów (dla obliczeń). Rozmiary tych obszarów muszą być znane podczas kompilacji.

Obszar zmiennych lokalnych

Tablica słów przechowująca argumenty i zmienne lokalne

- `double` zajmują po dwa słowa, `int` — jedno.
- Dla metod instancyjnych pod indeksem 0 jest `this`, dla statycznych — pierwszy argument.

Stos operandów

- Element mieści wartość dowolnego typu.
- Przed wywołaniem kładziemy argumenty na stosie, po powrocie wynik tamże.

Przykład programu – prosta klasa

```
class Simple {  
    public static void main(String argv[]) {  
        for (int i=1;i<=10;i++){  
            System.out.println(i);  
        }  
    }  
}
```

Przykład programu – prosta klasa

```
Method Simple()
```

```
0 aload_0
```

```
1 invokespecial #1 <Method java.lang.Object()>
```

```
4 return
```

```
Method void main(java.lang.String[])
```

```
0 iconst_1
```

```
1 istore_1
```

```
2 goto 15
```

```
5 getstatic #2 <Field java.io.PrintStream out>
```

```
8 iload_1
```

```
9 invokevirtual #3 <Method void println(int)>
```

```
12 iinc 1 1
```

```
15 iload_1
```

```
16 bipush 10
```

```
18 if_icmple 5
```

```
21 return
```

Działanie maszyny

- Potrzebna klasa jest ładowana do pamięci
 - ▶ weryfikacja
 - ▶ przygotowanie
 - ▶ rozwiązywanie nazw
- Następnie wykonywana jest inicjalizacja klasy
- Wykonuje kolejne instrukcje
 - ▶ Instrukcje operują na stosie operacji
 - ▶ Metody są wywoływane na stosie wywołań
 - ▶ Przy wykonywaniu sprawdzane są różne własności

Wykonanie — instrukcje

Maszyna stosowa

Load n	załaduj n-tą zmienną lokalną (także parametry)
Store n	zapisz wartość ze stosu do zmiennej lokalnej
Push val	wstaw stałą na stos
Add, Sub, Mul, Div	operacje arytmetyczne
Ldc stała	załaduj stałą z tablicy stałych
Getfield vname cname	pobierz pole z obiektu
Getstatic vname cname	pobierz pole z klasy
Putfield vname cname	ustaw pole obiektu

Instrukcje takie jak load, store, add są prefiksowane typami, więc np. aload, istore, fadd, . . .

Wykonanie — instrukcje

Checkcast cname	sprawdź czy obiekt jest danej klasy
InvokeVirtual method	wywołanie metody
InvokeSpecial method	wywołanie inicjalizatora, metody prywatnej etc.
tReturn	powrót (prefiksowane typem)
Pop	
Goto adres	
If_icmpGe adres	weź ze stosu a i b , skocz gdy $a \geq b$; także: eq, ne, lt, gt, le
ifEq adres	weź ze stosu a (int), skocz gdy $a = 0$; także: ne, lt, gt, le, ge

Kompilacja — przekazywanie argumentów

```
int addTwo(int i, int j) {  
    return i + j;  
}
```

kompiluje się do:

```
Method int addTwo(int,int)  
  0 iload_1    // zmienna lokalna 1 na stos (i)  
  1 iload_2    // zmienna lokalna 2 na stos (j)  
  2 iadd      // dodaj; wynik na szczycie stosu  
  3 ireturn   // powrót z wynikiem
```

this w zmiennej lokalnej 0

Kompilacja – wywoływanie metod

```
int add12and13() {  
    return addTwo(12, 13);  
}
```

kompiluje się do:

```
Method int add12and13()  
  0 aload_0          // this na stos  
  1 bipush 12        // 12 na stos  
  3 bipush 13        // 13 na stos  
  5 invokevirtual addtwo(II)I // wywołanie metody  
  8 ireturn          // powrót z wynikiem, to jest  
                      // też wynik addTwo()
```

Kompilacja – switch

```
int chooseNear(int i) {  
    switch (i) {  
        case 0: return 0;  
        case 2: return 1;  
        case 3: return 2;  
        default: return -1;  
    }  
}
```

kompiluje się do...

Kompilacja – switch

Method int chooseNear(int)

```
0 iload_1    // zmienna lokalna 1 (i) na stos
1 tableswitch 0 3
           28      // gdy i to 0, skocz do 28
           34      // nie bylo 1 to default
           30      // gdy i to 2, skocz do 30
           32      // gdy i to 3, skocz do 32
   default: 34      // wpp. skocz do 34
28 iconst_0    // i to 0; stała 0 na stos
29 ireturn     // ...oraz powrót
30 iconst_1    // i to 1; stała 1 na stos
31 ireturn     // ...oraz powrót
32 iconst_2    // i to 2; stała 2 na stos
33 ireturn     // ...oraz powrót
34 iconst_m1   // wpp na stos -1...
35 ireturn     // ...oraz powrót
```

Kompilacja – switch

```
int chooseFar(int i) {  
    switch (i) {  
        case -100: return -1;  
        case 0:    return 0;  
        case 100:  return 1;  
        default:   return -1;  
    }  
}
```

kompiluje się do...

Kompilacja – switch

```
Method int chooseFar(int)
  0  iload_1
  1  lookupswitch 3:
-100: 36
  0: 38
 100: 40
default:42
 36  iconst_m1
 37  ireturn
 38  iconst_0
 39  ireturn
 40  iconst_1
 41  ireturn
 42  iconst_m1
 43  ireturn
```


Jasmin

- Java ASM
- tłumaczy tekstowy opis kodu JVM na plik .class
- `java -jar jasmin.jar hello.j`
- <http://jasmin.sourceforge.net/>

Jasmin — hello

```
.class public Hello
.super java/lang/Object

; standard initializer
.method public <init>()V
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method
```

Jasmin — hello c.d.

```
.method public static main([Ljava/lang/String;)V
.limit stack 2
  getstatic
    java/lang/System/out Ljava/io/PrintStream;
  ldc "Hello"
  invokevirtual
    java/io/PrintStream/println(Ljava/lang/String;)V
  return
.end method
```

Java – sumto

```
public class Sumto {  
    int sumto(int n)  
    {  
        int i, sum;  
        i = 0;  
        sum = 0;  
        while (i<n) {  
            i = i+1;  
            sum = sum+i;  
        }  
        return sum;  
    }  
}
```

Jasmin — sumto

```
.method public sumto(I) I  
.limit locals 4  
.limit stack 2
```

```
iconst_0  
istore_2  
iconst_0  
istore_3
```

```
L1: iload_2  
    iload_1  
    if_icmpge L2  
    iload_2  
    iconst_1  
    iadd  
    istore_2  
    iload_3  
    iload_2  
    iadd  
    istore_3  
    goto L1
```

```
L2: iload_3  
    ireturn  
.end method
```

Działanie maszyny

- Potrzebna klasa jest ładowana do pamięci
 - ▶ weryfikacja
 - ▶ przygotowanie
 - ▶ rozwiązywanie nazw
- Następnie wykonywana jest inicjalizacja klasy
- Wykonuje kolejne instrukcje
 - ▶ Instrukcje operują na stosie operacji
 - ▶ Metody są wywoływane na stosie wywołań
 - ▶ Przy wykonywaniu sprawdzane są różne własności

Ładowanie — weryfikacja

- Instrukcje mają poprawną strukturę
- Skoki sa do początków instrukcji
- Metody mają poprawne sygnatury
- Instrukcje przestrzegają typów
- i inne...

Ładowanie — przygotowanie

- Tworzenie pól statycznych klasy
- Inicjalizacja wartościami domyślnymi (zwykle 0)
- Wstępne tworzenie pomocniczych struktur danych, np.
 - ▶ tablica metod
 - ▶ tablica stałych
 - ▶ ...

Ładowanie — rozwiązywanie nazw

- Nazwy w plikach `.class` są tekstowe
`List.list [Ljava/lang/Object;`
- Możliwe opóźnione rozwiązywanie

Inicjalizacja klas

- Inicjalizacja nadklasy
- Wykonanie statycznych inicjalizatorów
- Wykonanie inicjalizatorów pól statycznych
- Inicjalizacja interfejsów niekonieczna

LLVM

- Low Level Virtual Machine, <http://llvm.org/>
- maszyna rejestrowa, nieograniczona ilość rejestrów
- generacja kodu na rzeczywisty procesor przez alokację rejestrów (następny wykład)
- biblioteka C++, ale także format tekstowy
- kod czwórkowy:

```
%t2 = add i32 %t0, %t1
```
- instrukcje są silnie typowane:

```
%t5 = add double %t4, %t3  
store i32 %t2, i32* %loc_r
```
- nowy rejestr dla każdego wyniku (tzw. Static Single Assignment, patrz następny wykład)

LLVM — przykład

```
declare void @printInt(i32) ; w innym module
define i32 @main() {
    %i1 = add i32 2, 2
    call void @printInt(i32 %i1)
    ret i32 0
}
```

```
$ llvm-as t2.ll
$ llvm-ld t2.bc runtime.bc
$ lli a.out.bc
4
```

Uwaga:

- nazwy globalne zaczynają się od @, lokalne od %
- nazwy zewnętrzne są deklarowane (@printInt)

LLVM — silnia, rekurencyjnie

```
define i32 @fact(i32 %n) {  
    %c0 = icmp eq i32 %n, 0  
    br i1 %c0, label %L0, label %L1  
  
L0:  
    ret i32 1  
  
L1:  
    %i1 = sub i32 %n, 1  
    %i2 = call i32 @fact(i32 %i1)  
    %i3 = mul i32 %n, %i2  
    ret i32 %i3  
}
```

Uwaga:

- argumenty funkcji są deklarowane
- wszystko jest typowane, nawet warunki skoków
- skoki warunkowe tylko z “else”

LLVM — typy (nie wszystkie)

- n -bitowe liczby całkowite: i_n , np.:
 - ▶ `i32` dla `int`
 - ▶ `i1` dla `bool`
 - ▶ `i8` dla `char`
- nie ma podziału na liczby ze znakiem i bez
- są natomiast operacje ze znakiem lub bez, np.
 - ▶ `sle` — signed less or equal
 - ▶ `udiv` — unsigned div
- `float` oraz `double`
- `label`
- `void`
- wskaźniki: t^*
- tablice: $[n * t]$, e.g.
`@hw = constant [13 x i8] c"hello world\0A\00"`
- struktury $\{t_1, \dots, t_n\}$

Definicja

Blok prosty jest sekwencją kolejnych instrukcji, do której sterowanie wchodzi wyłącznie na początku i z którego wychodzi wyłącznie na końcu, bez możliwości zatrzymania ani rozgałęzienia wewnątrz.

Kod LLVM:

- etykietowane bloki proste
- każdy blok kończy się skokiem (`ret` lub `br`)
- nie ma automatycznego przejścia od ostatniej instrukcji bloku do pierwszej kolejnego (kolejność bloków może być swobodnie zmieniana)
- skoki warunkowe mają dwa cele:

```
br i1 %c0, label %L0, label %L1
```
- blok wejścia do funkcji jest specjalny: nie można do niego skoczyć

LLVM — silnia, iteracyjnie

```
; r = 1
; i = n
; while (i > 1):
;   r = r * i
;   i = i - 1
; return r
define i32 @fact(i32 %n) {
entry:
; zmienne lokalne, alokowane na stosie
    %loc_r = alloca i32
    %loc_i = alloca i32
; r = 1
    store i32 1, i32* %loc_r
; i = n
    store i32 %n, i32* %loc_i
    br label %L1
```


LLVM — silnia, iteracyjnie

```
; while i > 1:
L1:    %tmp_i1 = load i32* %loc_i
        %c0 = icmp sle i32 %tmp_i1, 1
        br i1 %c0, label %L3, label %L2

; ciało pętli
L2:
; r = r * i
        %tmp_i2 = load i32* %loc_r
        %tmp_i3 = load i32* %loc_i
        %tmp_i4 = mul i32 %tmp_i2, %tmp_i3
        store i32 %tmp_i4, i32* %loc_r

; i = i-1
        %tmp_i5 = load i32* %loc_i
        %tmp_i6 = sub i32 %tmp_i5, 1
        store i32 %tmp_i6, i32* %loc_i
        br label %L1
```

LLVM — silnia, iteracyjnie

L3:

```
%tmp_i8 = load i32* %loc_r  
ret i32 %tmp_i8
```

```
}
```

W następnym odcinku: prawdziwy kod SSA

```
define i32 @fact(i32 %n) {
entry: br label %L1
L1:
    %i.1 = phi i32 [%n, %entry], [%i.2, %L2]
    %r.1 = phi i32 [1, %entry], [%r.2, %L2]
    %c0 = icmp sle i32 %i.1, 1
    br i1 %c0, label %L3, label %L2
L2:
    %r.2 = mul i32 %r.1, %i.1
    %i.2 = sub i32 %i.1, 1
    br label %L1
L3:
    ret i32 %r.1
}
```